OBJECT-ORIENTED PROGRAMMING

Modelling State - Mutable and Immutable Objects

Daniel Tattan-Birch

27 November 2017

Page count:
4 plus references and appendix

1 Introduction

The mutability of objects is an important concept to understand and consider when designing and developing object-oriented software.

In Java, objects are constructed from classes which are essentially templates with which to create instances of objects. A class generally consists of state and behaviours, represented by attributes and functions respectively.

We say that an object or class is immutable if it cannot undergo any state changes, or mutations, once it has been instantiated. Before examining the means by which this can be achieved, let's first discuss the motivation for creating immutable objects.

2 Why use immutable objects?

Immutable objects naturally produce cleaner APIs since it is guaranteed that no side effects can occur as a result of a call to an object behaviour. Additionally, control of an object's state is withheld from consumers of our object and given entirely to the class constructors. Attempting to instantiate objects with an invalid state is rejected during construction [1]. This eliminates the problem of having objects in our system with an invalid state. Together these outcomes reduce system complexity, avoiding mistakes and misuse of objects which can lead to faulty software.

A further benefit of the lack of side effects in immutable objects is that they are inherently easier to test. It enables the black box testing [2] of classes. This consists of simply testing the output from a function (in this case an object behaviour) against its input. Consequently, the clients of our objects don't have to care about the implementation details. This naturally decouples our objects from one another, allowing dependencies on abstractions such as interfaces as opposed to concrete classes. Overall this will increase the encapsulation and therefore atomicity of the objects in our system.

Another advantage of immutable objects is that they are inherently thread safe. Different threads can interact with them, but only in a *reading* capacity. In enterprise systems that require high concurrency this is paramount since data structures that mutate can produce unpredictable consequences when being modified on multiple threads. Bugs introduced in this manner tend to be subtle and difficult to diagnose.

The biggest downfall of immutable objects is that memory is reassigned much more frequently. This can have performance implications for systems that need to create large data structures that are frequently modified. However, Oracle argue that "the impact of object creation is often overestimated and can be offset by some of the efficiency associated with immutable objects" [3]. This contradicts the common assessment that the cost of instantiating a new object renders immutable architecture unfeasible.

Another potential downside is the idea that our perception of the model world is based on mutable objects [4]. This pertains to the notion that software developers generally find mutating objects a more intuitive way to model state, and therefore produce more effective code at a faster rate when following that pattern. This is particularly true in the case of game development, and GUI development in general. These are inherently coupled with real world user interaction which is inevitably based on mutable objects [5].

3 How can we eradicate state mutations?

These are the general criteria that must be met in order to transform a mutable class into an immutable one:

1. After being initialised at construction, class attributes cannot be reassigned to any other piece of memory (or must remain null in the case of a null reference).

2. The attributes that make up an object's state must themselves be prevented from modification for the lifetime of the object

Let's look at the process of removing state mutations from a class. Specifically we'll look at a Node class which is used as part of a binary search tree. Nodes are formed of keys and values in this case. I'll leave out details which aren't directly relevant for understanding the concepts of state mutations. Consider:

```
public class Node<K extends Comparable<K>, V>
{
   public K key;
   public V value;
   public Node<K, V> left;
   public Node<K, V> right;
   ...
}
```

This `Node` class directly exposes its state. Attributes could be reassigned at will by clients of the object. This is a clear violation of encapsulation. Let's make those state variables private and access them through getters and setters.

```
public class Node<K extends Comparable<K>, V>
{
   private K key;
   private V value;
   private Node<K, V> left;
   private Node<K, V> right;

   public K getKey() { ... }
   ...
   public void setKey(K key) { ... }
   ...
}
```

This is an improvement. However, we're still providing a wide API. Perhaps we cannot allow a `null` `value`, or `key` should not be modifiable at all. We have to defend against these invalid state modifications in the setters as well as the constructor. Instead, let's just remove the setters and force clients to instantiate a new instance of a `Node` if they want to change those attributes. We can achieve this by creating a new constructor, below. However, programmatically there is no guarantee that one of our class behaviours (see `insert()`) won't reassign, for example, our `left` attribute.

```
public class Node<K extends Comparable<K>, V>
{
   ...
   public Node(K key, V value, Node<K, V> left, Node<K, V> right) { ... }
   ...
   public K getKey() { ... }
   ...
   public void insert(K key, V value) { ... }
}
```

We can use the "final" keyword in Java to prevent this at compile time. These attributes can now only be set in constructors, giving full control to the object's construction phase.

It might also be desirable to prevent `Node` from being extensible to ensure subclasses cannot modify its internal state. This can be achieved by using "final" in the class declaration.

```
public final class Node<K extends Comparable<K>, V>
{
   private final K key;
   private final V value;
   private final Node<K, V> left;
   private final Node<K, V> right;
   ...
   public Node<K, V> insert(K key, V value)
   {
      ...
      Node<K, V> newLeft = left.insert(key, value);
      return new Node(this.key, this.value, newLeft, this.right)
   }
}
```

Notice that we had to change the signature of `insert()` since if it was left as void it would essentially be doing nothing. Instead of modifying `left`, a new left is instantiated and used in the construction of a new `Node`, which is returned. These kind of API changes are often necessary to achieve object immutability. A subtle bug could be introduced if someone had previously called `insert()` before its signature change. The resultant node would not be assigned to any variable, and just be garbage collected. It's preferable to create classes which are immutable from the beginning, to avoid these kinds of compatibility issues that arise from dependencies on APIs that may change.

We appear to have satisfied number (1) of our immutability criteria. However, there are still ways in which a `Node` object can be mutated. Consider:

```
public static void main(String[] args)
{
   Person personIn = new Person();
   Node<Integer, Person> node = new Node<>(123, personIn);
   personIn.setAge(35);

   node = new Node<>(456, new Person());
   Person personOut = node.getValue();
   personOut.setAge(35);
}
```

The first example demonstrates mutating the object *passed in* to `node`. The second shows mutation of the state attribute after retrieving it *back out* from `node`. One solution here would be to ensure that the `Person` class is immutable itself. In general, however, this cannot be guaranteed for all types used by our class.

The most common way of combating this problem is to defensively copy or clone state objects entering and leaving our class walls. Ideally we want a deep clone, meaning that objects referred to by our object are copied recursively. This isn't required for the `left` and `right` attributes, since they will be immutable (once we're done). It also wouldn't be required for boxed primitive types, such as `Integer`, or other immutable types, such as `String`.

There are several schools of thought on how object copying should be done in Java. These include implementing the `Cloneable` interface or creating a new `ICopyable` interface. Joshua Bloch strongly discourages the use of the `Cloneable` interface, as do many other Java developers [7, 11, 12, 13]. It is only a marker interface, meaning it doesn't *force* classes to implement `clone()`. It also doesn't call a constructor, so we have to rely on an unknown object creation mechanism. The pattern that many argue is the most widely applicable and adheres best to object-oriented principles is to use a copy constructor [7, 8]. This gives the class of the object we're trying to copy complete control

over object creation. We can also initialise final fields this way, which encourages immutable design. Here's a copy constructor for the `Person` class:

```
public class Person
{
   ...
   public Person(Person other) { ... }
   ...
}
```

In this case of generic types, as in our `Node` class, it's a bit more challenging. A copy constructor cannot be invoked without using reflection, since we can't know whether the types `K` and `V` have one. Forcing these generic types to implement a new interface is very restrictive, and would make boxed primitive types and `String` incompatible with the object. Another approach is to use an annotation such as `@Immutable` which is provided by the Java Concurrency in Practice library [14]. If the generic type has that annotation, we do not have to perform copying.

I used a combination of the above techniques. The idea was to give clients of the `Tree` every opportunity to provide a type parameter that could be either defensively copied or was marked as immutable. The `ObjectCopier` class can be found in appendix A. This approach was definitely overkill, but an interesting exercise which demonstrates the difficulty of preparing an object for every use case. In a closed system I would be inclined to carefully choose a pattern and then stick to it.

To conclude, it is evidently not trivial to design objects in an immutable fashion. The requirement of immutability forced me to come up with a much cleaner design when implementing node balancing in the tree, and also enabled me to remove the code that dealt with concurrent modification. This demonstrates how immutable architecture can drive better designed software. However, the decision to design classes in an immutable way should be made up front to avoid future work and to reduce the chance of generating undesired behavioural changes.

4 Performance implications

The biggest disadvantage of immutable design is performance. To give some idea of the cost of immutable objects, I compiled some performance data which can be seen in appendix B. There are ways to avoid excessive inefficiencies in immutable objects, such as using them together with *other* immutable objects so that defensive copying is minimised. If a whole system is built with immutable objects, classes are happy to share their state objects since they aren't worried about mutations. This could actually *increase* performance by reducing frequent garbage collection. Another pattern is to mutate an object until it is in the desired state, and then generate an immutable object from it. The Java `StringBuilder` class is an example of this pattern, which leverages both the efficiency of mutable objects and the design benefits of immutable objects.

5 Conclusion

Given a blank slate to design a system on, the advantages of immutable objects outweigh the disadvantages. Immutable architecture protects against multi-threading problems which enables extra effort to be spent on efficiency gains. For systems with high data throughput, it's a trade off between the need for high performance and the benefits of immutable data structures such as seamless concurrency and more easily testable code.

References

[1] (Online) Available from: http://www.yegor256.com/2014/06/09/objects-should-be-immutable.html (Accessed 25 November 2017).

[2] (Online) Available from: http://object-oriented-software.blogspot.co.uk/2011/10/explain-black-box-and-white-box-testing.html (Accessed 25 November 2017).

[3] (Online) Available from: https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html (Accessed 27 November 2017).

[4] (Online) Available from: https://dzone.com/articles/learning-to-use-and-abuse-mutability (Accessed 26 November 2017).

[5] (Online) Available from: https://softwareengineering.stackexchange.com/questions/151733/if-immutable-objects-are-good-why-do-people-keep-creating-mutable-objects (Accessed 25 November 2017).

[6] (Online) Available from: https://stackoverflow.com/questions/1994218/should-i-instantiate-instance-variables-on-declaration-or-in-the-constructor (Accessed 25 November 2017).

[7] (Online) Available from: https://dzone.com/articles/java-cloning-copy-constructor-vs-cloning (Accessed 25 November 2017).

[8] (Online) Available from: https://stackoverflow.com/questions/869033/how-do-i-copy-an-object-in-java (Accessed 27 November 2017).

[9] (Online) Available from: https://stackoverflow.com/questions/16257761/difference-between-red-black-trees-and-avl-trees (Accessed 25 November 2017).

[10] (Online) Available from: http://javahungry.blogspot.com/2014/04/fail-fast-iterator-vs-fail-safe-iterator-difference-with-example-in-java.html (Accessed 25 November 2017).

[11] Joshua Bloch. *Effective Java*, 2nd edition. Addison-Wesley, 2008.

[12] (Online) Available from: https://stackoverflow.com/questions/869033/how-do-i-copy-an-object-in-java (Accessed 27 November 2017).

[13] (Online) Available from: https://stackoverflow.com/questions/1106102/clone-vs-copy-constructor-vs-factory-method (Accessed 26 November 2017).

[14] (Online) Available from: http://jcip.net/annotations/doc/ (Accessed 26 November 2017).

A

ObjectCopier source code

```
package main.immutable.tree;

import java.lang.reflect.Constructor;
```

```
import java.lang.reflect.Method;
import java.util.HashSet;
import java.util.Set;
import net.jcip.annotations.Immutable;

class ObjectCopier
{
   // A set of the boxed primitive types and String, which are all already immutable.
   private final static Set<Class<?>> BOXED_PRIMITIVE_TYPES = new HashSet<>();

   static
   {
      BOXED_PRIMITIVE_TYPES.add(Boolean.class);
      BOXED_PRIMITIVE_TYPES.add(Byte.class);
      BOXED_PRIMITIVE_TYPES.add(Short.class);
      BOXED_PRIMITIVE_TYPES.add(Character.class);
      BOXED_PRIMITIVE_TYPES.add(Integer.class);
      BOXED_PRIMITIVE_TYPES.add(Long.class);
      BOXED_PRIMITIVE_TYPES.add(Float.class);
      BOXED_PRIMITIVE_TYPES.add(Double.class);
      BOXED_PRIMITIVE_TYPES.add(String.class);
   }

   static <T> T copyDefensivelyIfRequired(T obj)
   {
      if (obj == null) { return null; }

      Class<?> objClass = obj.getClass();
      if (BOXED_PRIMITIVE_TYPES.contains(obj.getClass()))
      {
         // shortcut for String and boxed primitive types - these are all immutable so just
             return the same instance
         return obj;
      }
      else if (objClass.isAnnotationPresent(Immutable.class))
      {
         // class contains the @Immutable annotation - no need to copy
         return obj;
      }
      else
      {
         try
         {
            return attemptDefensiveCopy(obj, objClass);
         }
         catch (Exception e)
         {
            // if anything unforeseen happens ensure we always return at least the object
                itself back
            return obj;
         }
```

```
    }
}

private static <T> T attemptDefensiveCopy(T obj, Class<?> objClass)
{
    // try to cast the object as an ICopyable and use that for copying
    if (obj instanceof ICopyable<?>)
    {
        // note we have no guarantee that it is an ICopyable<T> specifically - perform copy
            and look.
        ICopyable<?> asCopyable = (ICopyable<?>) obj;
        Object copy = asCopyable.copy();
        if (objClass.isAssignableFrom(copy.getClass()))
        {
            // this means the copy's type is a subtype of T for sure, we can cast it
            return (T) copy;
        }
    }

    // look for a public copy constructor through reflection
    try
    {
        Constructor<?> copyConstructor = objClass.getConstructor(objClass);
        // found a copy constructor, use this to create the copy. Can still fail if
            constructor is non-public
        return (T) copyConstructor.newInstance(obj);
    }
    catch (Exception e)
    {
        // swallow exception, no copy constructor found or the instantiation failed
    }

    // last attempt to copy, check that the object can be cast to a Cloneable
    if (obj instanceof Cloneable)
    {
        // we know the generic type will have a public override of the protected object
            clone() method
        try
        {
            Method cloneMethod = objClass.getMethod("clone");
            // invoke clone on the passed in object
            return (T) cloneMethod.invoke(obj);
        }
        catch (Exception e)
        {
            // swallow exception, something unforeseen occurred when invoking clone()
        }
    }

    return obj; // we failed all attempts to copy and accept it might be mutable
}
```

```
}
```

B

Performance of mutable and immutable AVL tree

Inserting into the immutable tree becomes much slower than the mutable tree for larger volumes of data. This is probably due to the expensive rotation operations, which cost far more for the immutable tree since it can instantiate five nodes per rotation in the worst case. A solution could be to build the tree structure in a mutable way, and then“freeze” that state into an immutable tree once it’s balanced and populated.

The searching time complexity is closer, with the mutable tree performing better by much smaller margins. Those search speeds are probably acceptable if this were to be used in a real system - only a small trade-off needed to use immutable data structures.

Table 1: Insertion (milliseconds)

Tree type	Number of elements			
	10	100	1000	10000
Mutable	0.0264	0.0934	0.860	42.4
Immutable	0.0135	0.109	3.75	530

Table 2: Searching (millseconds)

Tree type	Number of elements			
	10	100	1000	10000
Mutable	0.00240	0.0135	0.156	1.36
Immutable	0.00331	0.0261	0.197	1.92