Mutable and Immutable in Scala

As so many others have said, the object assigned to a val cannot be replaced, and the object assigned to a var can. However, said object can have its internal state modified. For example:

class A(n: Int) {

var value = n

}

class B(n: Int) {

val value = new A(n)

}

object Test {

def main(args: Array[String]) {

val x = new B(5)

x = new B(6) // Doesn't work, because I can't replace the object created on the line above with this new one.

x.value = new A(6) // Doesn't work, because I can't replace the object assigned to B.value for a new one.

x.value.value = 6 // Works, because A.value can receive a new object.

}

}

So, even though we can't change the object assigned to x, we could change the state of that object. At the root of it, however, there was a var.

Now, immutability is a good thing for many reasons. First, if an object doesn't change internal state, you don't have to worry if some other part of your code is changing it. For example:

x = new B(0)

f(x)

if (x.value.value == 0)

println("f didn't do anything to x")

else

println("f did something to x")

This becomes particularly important with multithreaded systems. In a multithreaded system, the following can happen:

x = new B(1)

f(x)

if (x.value.value == 1) {

print(x.value.value) // Can be different than 1!

}

If you use val exclusively, and only use immutable data structures (that is, avoid arrays, everything in scala.collection.mutable, etc.), you can rest assured this won't happen. That is, unless there's some code, perhaps even a framework, doing reflection tricks -- reflection can change "immutable" values, unfortunately.

That's one reason, but there is another reason for it. When you use var, you can be tempted into reusing the same var for multiple purposes. This has some problems:

* It will be more difficult for people reading the code to know what is the value of a variable in a certain part of the code.
* You may forget to re-initialize the variable in some code path, and end up passing wrong values downstream in the code.

Simply put, using val is safer and leads to more readable code.

We can, then, go the other direction. If val is that better, why have var at all? Well, some languages did take that route, but there are situations in which mutability improves performance, a lot.

For example, take an immutable Queue. When you either enqueue or dequeue things in it, you get a new Queue object. How then, would you go about processing all items in it?

I'll go through that with an example. Let's say you have a queue of digits, and you want to compose a number out of them. For example, if I have a queue with 2, 1, 3, in that order, I want to get back the number 213. Let's first solve it with a mutable.Queue:

def toNum(q: scala.collection.mutable.Queue[Int]) = {

var num = 0

while (!q.isEmpty) {

num \*= 10

num += q.dequeue

}

num

}

This code is fast and easy to understand. Its main drawback is that the queue that is passed is modified by toNum, so you have to make a copy of it beforehand. That's the kind of object management that immutability makes you free from.

Now, let's covert it to an immutable.Queue:

def toNum(q: scala.collection.immutable.Queue[Int]) = {

def recurse(qr: scala.collection.immutable.Queue[Int], num: Int): Int = {

if (qr.isEmpty)

num

else {

val (digit, newQ) = qr.dequeue

recurse(newQ, num \* 10 + digit)

}

}

recurse(q, 0)

}

Because I can't reuse some variable to keep track of my num, like in the previous example, I need to resort to recursion. In this case, it is a tail-recursion, which has pretty good performance. But that is not always the case: sometimes there is just no good (readable, simple) tail recursion solution.

Note, however, that I can rewrite that code to use an immutable.Queue and a var at the same time! For example:

def toNum(q: scala.collection.immutable.Queue[Int]) = {

var qr = q

var num = 0

while (!qr.isEmpty) {

val (digit, newQ) = qr.dequeue

num \*= 10

num += digit

qr = newQ

}

num

}

This code is still efficient, does not require recursion, and you don't need to worry whether you have to make a copy of your queue or not before calling toNum. Naturally, I avoided reusing variables for other purposes, and no code outside this function sees them, so I don't need to worry about their values changing from one line to the next -- except when I explicitly do so.

Scala opted to let the programmer do that, if the programmer deemed it to be the best solution. Other languages have chosen to make such code difficult. The price Scala (and any language with widespread mutability) pays is that the compiler doesn't have as much leeway in optimizing the code as it could otherwise. Java's answer to that is optimizing the code based on the run-time profile. We could go on and on about pros and cons to each side.

Personally, I think Scala strikes the right balance, for now. It is not perfect, by far. I think both [Clojure](http://en.wikipedia.org/wiki/Clojure)and [Haskell](http://en.wikipedia.org/wiki/Haskell_%28programming_language%29) have very interesting notions not adopted by Scala, but Scala has its own strengths as well. We'll see what comes up on the future.

The difference is that a var can be re-assigned to whereas a val cannot. The mutability, or otherwise of whatever is actually assigned, is a side issue:

import collection.immutable

import collection.mutable

var m = immutable.Set("London", "Paris")

m = immutable.Set("New York") //Reassignment - I have change the "value" at m.

Whereas:

val n = immutable.Set("London", "Paris")

n = immutable.Set("New York") //Will not compile as n is a val.

And hence:

val n = mutable.Set("London", "Paris")

n = mutable.Set("New York") //Will not compile, even though the type of n is mutable.

If you are building a data structure and all of its fields are vals, then that data structure is therefore immutable, as its state cannot change.

Method def even evaluates on call and creates new function every time (new instance of Function1).

def even: Int => Boolean = \_ % 2 == 0

even eq even

//Boolean = false

val even: Int => Boolean = \_ % 2 == 0

even eq even

//Boolean = true

With def you can get new function on every call:

val test: () => Int = {

val r = util.Random.nextInt

() => r

}

test()

// Int = -1049057402

test()

// Int = -1049057402 - same result

def test: () => Int = {

val r = util.Random.nextInt

() => r

}

test()

// Int = -240885810

test()

// Int = -1002157461 - new result

val evaluates when defined, def - when called:

scala> val even: Int => Boolean = ???

scala.NotImplementedError: an implementation is missing

scala> def even: Int => Boolean = ???

even: Int => Boolean

scala> even

scala.NotImplementedError: an implementation is missing

Note that there is a third option: lazy val.

It evaluates when called the first time:

scala> lazy val even: Int => Boolean = ???

even: Int => Boolean = <lazy>

scala> even

scala.NotImplementedError: an implementation is missing

But returns the same result (in this case same instance of FunctionN) every time:

lazy val even: Int => Boolean = \_ % 2 == 0

even eq even

//Boolean = true

lazy val test: () => Int = {

val r = util.Random.nextInt

() => r

}

test()

// Int = -1068569869

test()

// Int = -1068569869 - same result

**Performance**

val evaluates when defined.

def evaluates on every call, so performance could be worse than val for multiple calls. You'll get the same performance with a single call. And with no calls you'll get no overhead from def, so you can define it even if you will not use it in some branches.

With a lazy val you'll get a lazy evaluation: you can define it even if you will not use it in some branches, and it evaluates once or never, but you'll get a little overhead from double check locking on every access to your lazy val.

As @SargeBorsch noted you could define method, and this is the fastest option:

def even(i: Int): Boolean = i % 2 == 0

But if you need a function (not method) for function composition or for higher order functions (like filter(even)) compiler will generate a function from your method every time you are using it as function, so performance could be slightly worse than with val.