Day 13: Lazy initialization + Linearization redux

This is interesting. The book talks of lazy initialization as a design pattern to help manage expensive and potentially unneeded operations. For example, you might not need to load that 500mb file into Spark's driver memory and broadcast it if the config leads to a different code path. If you had it as a `val`, it would be loaded right away; a `def`, every time! But a `lazy val` will load it once **only if / when referenced**. You can think of this as *lazy and memoized*.

But we use `lazy vals` in our code base all over the place for a different reason - *trait linearization / initialization*. Remember the order in which traits / classes are initialized - if a class `val` is initialized while referencing another class member field that is going to be injected later, that eager `val` evaluation might lead to a null pointer! This is a classic problem we deal with in our team. Example:

```
// Example 1: bad init with primitives
trait A { val x: Int; val y = x + 1 }
object B extends A { val x = 3 }
println(B.y) // prints 1 - silent error!

// Example 2: fixing example 1 with lazy val
trait A { val x: Int; lazy val y = x + 1 }
object B extends A { val x = 3 }
println(B.y) // prints 4, correctly

// Example 3: bad init with object nulls
trait A { val x: String; val y = s"$x and hello!" }
object B extends A { val x = "hi" }
println(B.y) // prints "null and hello!"

// Example 4: bad init with NullPointer in method
trait A { val x: String; val y = s"${x.toUpperCase} and hello!" }
object B extends A { val x = "hi" }
println(B.y) // throws NullPointerException!
```

Personal sidenote - I have a feeling that a lot of our null pointer problems per above come from bad design patterns. For example, a lot of our cake pattern code uses this sort of thing:

```
def _myVar: Int
lazy val myVar: Int = _myVar
```

This is actually almost error-proof. Why? Consider this slight variation on the previous examples:

```
trait A { lazy val myVar = _myVar; def _myVar: Int }

// two variations
trait B extends A {
  val myVar2 = 3 + _myVar
  println("B: constructs myVar2 using _myVar")
  println(s"_myVar side effect = ${_myVar}")
  println(s"myVar2 side effect = ${myVar2}")
}

trait B2 extends A {
  val myVar2 = 3 + myVar
  println("B2: constructs myVar2 using myVar")
  println(s"myVar side effect = ${myVar}")
  println(s"myVar2 side effect = ${myVar2}")
}

trait B3 extends A {
  val myVar2 = 3 + _myVar
  println("B3: overrides _myVar as val rather than def")
  println(s"_myVar side effect = ${_myVar}")
  println(s"myVar2 side effect = ${myVar2}")
}

trait B4 extends A {
  val myVar2 = 3 + _myVar
  println("B3: overrides _myVar as val rather than def, uses myVar to construct myVar2")
  println(s"_myVar side effect = ${_myVar}")
  println(s"myVar2 side effect = ${myVar2}")
}

object C extends B { override def _myVar = 1 }
object C2 extends B2 { override def _myVar = 1 }
object C3 extends B3 { override val _myVar = 1 }
object C4 extends B4 { override val _myVar = 1 }

println(s"final myVar2 for C = ${C.myVar2}\n")
println(s"final myVar2 for C2 = ${C2.myVar2}\n")
println(s"final myVar2 for C3 = ${C3.myVar2}\n")
println(s"final myVar2 for C4 = ${C4.myVar2}\n")
```

Here, no matter whether you refer to the lazy `myVar` or the eager `_myVar`, both will resolve correctly at the bottom, for opposite reasons. Of course, looking at the side effects in `B` and `B2`, we see the downside of referring to `_myVar` or `myVar` before it's injected - you'll get the same as the first examples.

Long story short, if you use the pattern above and lazy vals in general:

1. Always refer to the `lazy val` when using your variable, and inject the variable by overriding the `def` (not

that you have a choice).

2. Never perform side effects on class initialization.
3. If you're injecting some member field in some top-level class, but refer to it abstractly in parent classes, `lazy vals` are an easy escape hatch. Normal overrides of methods are fine because they're always `def`!