CSE250 Week 2: Collections as Data Structures (beginning section 1.7.2 on p21)

Scala provides a wealth of **container objects** for data structures. The text shows the full extent of the Scala Collections Library later at the start of chapter 6, broken down into its `mutable` and `immutable` packages. But we can see a bunch of main principles just with `Array` and `List`.

```
object ArrayAndList extends App {
   val cubesA = Array.tabulate(100)(i => i*i*i)
   var cubesL = List.tabulate(100)(i => i*i*i)
   println(s"Array item 17 cubed is ${cubesA(17)} and list item 18 cubed is ${cubesL(18)}")
```

We have used `tabulate` to initialize both the array and the list to the first hundred cubes, up to $99^3 = 970,299$. Some nontrivial function stuff is going on. The `tabulate` method takes two arguments in **curried** form, meaning one after the other with no comma, rather than in enclosing parentheses with a comma between them. The first argument is just an integer giving the range, but the second argument is a function telling how to map each index i. The function is given anonymously, in "rocket" form. (The more-formal term is *lambda* form, from the *lambda calculus,* which goes back to the 1930s. The text shows a case where you can avoid the rocket by using an underscore, but this is not one of them. Try using `(_*_*_)` instead---you'll get your head bitten off. But do note that using array parens inside the substitutable string was OK so long as we put the whole thing inside curly braces.)

Once again we have a case where **`val`** only fixes the reference to the array, not the data inside the array. But even with var for the list, the data is protected because `List` is immutable:

```
   //Try to change the first element
   cubesA(0) = -1                        //"val" doesn't stop this
   //cubesA = Array(-1,0,1,8,27,64)      //it does stop this
   //cubesL(0) = -1                      //List[Int] being immutable does stop this
```

Here's a possible gray area: Suppose we try to append one more value, 100^3. Is this OK with **`val`**? We're not changing the name of the array, and we're not changing any data in the array either. Strange to my mind, the text on pages 26--34 runs thru a gazillion array and sequence methods, many too fancy-schmantzy at this point, but doesn't give **append**. Scala uses **++** between two arrays the way Python uses **+**, but also allows **:+** to return an array with the item appended. Not only that, but "append-assign" **:+=** carries out the append on the array. So that's what we're asking: does a Scala **`val`** array allow append-assign? *No*, because of the "assign":

```
   //Try to append 100 cubed. The + goes toward the number, the : toward the array or list
   //cubesA :+= 100*100*100   //"val" stops it because this is cubesA = cubesA :+ 100*100*100
   cubesL :+= 100*100*100     //var allows this.  Note: appending is slow with a List
```

What we *can* do is create a fresh array with the extra value appended. This is where we start getting into core issues in the course. The question is: does this (have to) make a completely separate copy, going to all that work just to add one element? First we try it without appending:

```
var cubesA2 = cubesA      //OK since cubesA2 is a fresh handle. But is this a fresh copy?
//Change the first element of cubesA2 back to what it should be
cubesA2(0) = 0
println("Was this a fresh copy?  " +
   s"cubesA(0) = ${cubesA(0)}, cubesA2(0) = ${cubesA2(0)}; so ${cubesA(0) != cubesA2(0)}")
```

It is not a fresh copy. This is familiar behavior in Java and Python: the assignment `cubesA2 = cubesA` just creates a new reference to the array. But if we modify what's assigned---

```
cubesA2 = cubesA :+ 100*100*100    //Now we assign cubesA2 to be a modified copy
//Now make the first element of cubesA2 bad again.  Does cubesA go bad with it?
cubesA2(0) = -1
println("Was this a fresh copy?  " +
   s"cubesA(0) = ${cubesA(0)}, cubesA2(0) = ${cubesA2(0)}; so ${cubesA(0) != cubesA2(0)}")
```

---then it becomes a whole fresh copy. This is shown graphically by index 0 being completely on the other side of the array, but it would happen if we tried prepending while assigning the new variable instead.

Let's now see the syntax for prepending. To prepend without assigning, the operator is `+:` but with the value being prepended coming first. E.g. to prepend `-1`, we need **`-1 +: cubesA`** not **`cubesA +: -1`**. But to combine with self-assignment, we need the array coming first:

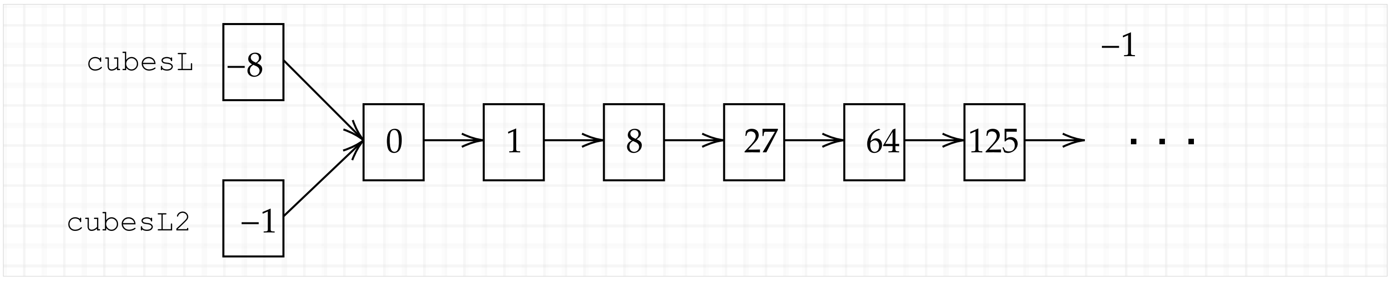
```
//Try to prepend -1, which happens to be its own cube
cubesA2 +:= -1       //OK since cubesA2 is var.  Becomes cubesA2 = -1 +: cubesA2
```

We can do this with our list, too. First we make a new list `cubesL2` with the value-prepend `+:` operator. Is this a fresh copy? Then we raise the question of whether prepending is allowed on an immutable `List` while assigning in place. Because we made `cubesL` a **`var`**, the answer is, wildly, *yes*. We could use the same `+:=` syntax as for arrays, but for lists it is more customary to combine the assignment `=` with the **cons** syntax `::` (page 24 of the text):

```
var cubesL2 = -1 +: cubesL
cubesL ::= -1 //The "cons" way of prepending to list.  Also OK: cubesL2 +:= -1
if (cubesL == cubesL2) { println("The lists are equal") }
```

In Java, comparing arrays or lists with `==` is a beginner's mistake, because you're only comparing the references. But with immutable lists in Scala, you get a real value comparison.

Now let's pose our question: Are `cubesL` and `cubesL2` independent copies? We can't test this the way we did with `cubesA` and `cubesA2` because the lists are immutable---we can't change `cubesL2(0)` after the fact. But my answer is, *I believe not*. Because the lists are immutable, they can share the elements after the first. That is, I believe what we've created is:



If so, this saves a lot of copying time. A smart compiler should be able to do this.

[*Remark:* A super-smart compiler might even avoid allocating 100 or 101 memory cells for these lists at all. Instead, it could just store a pointer to the originally-given code $i \implies i{*}i{*}i$ and just call it whenever an element `cubesL(i)` is accessed. The one niggle is that prepending -1 shifted the indices, so now given `cubesL(i)`, the system now would have to do $(i-1)^3$. This thought may reappear in this course if "**lazy** unbounded list generation" becomes a thing, but otherwise it's really fodder for CSE305.]

```
//Alas, the indices don't shift:
println(s"Now array item 17 cubed is ${cubesA(17)} and list item 18 cubed is ${cubesL(18)}")
```

Ignoring the remark, the main point is that *using immutable data can allow us to manage little modificiations to objects without having to do wholesale copying of them*. Plus it allowed us to compare two lists by-value simply using **==**. The benefit of values shows up in one more little way during the next topic.

Lists and Structural Matching (see text, page 29 middle)

We can use :: and the empty list Nil to do pattern-matching and recursion on lists. Here is a routine that takes the cube root of every element on a numerical List.

```
/** Takes cube root of every Double in the list.
    Note return type must be specified as Double to use pow function
 */
def cubeRootsL(xs: List[Int]): List[Double] = xs match {
   case Nil => Nil
   case x::xrest => scala.math.pow(x,1/3.0) :: cubeRootsL(xrest)
}
```

Our list in recursive form---which is how Scala actually treats it---is

```
-1 :: 0 :: 1 :: 8 :: 27 :: 64 :: ... :: 970299 :: 1000000 :: Nil.
```

The `cubeRoots` function recurses down to the empty list `Nil`, which as the base case goes to itself. We did have to specify the input type as `List[Int]` *and* the output type as `List[Double]`, the latter because that is the output of the `pow` function in the `scala.math` package. That function seems not to care if its first argument is an integer, but I wrote the second argument as `1/3.0` (`1.0/3` would be good too) to make clear that fractional not integer division is being used to define one-third exactly. To call it and output just ten of the roots:

```
val roots = cubeRootsL(cubesL)
println(s"Cube roots from indexes 10 to 19: ${roots.slice(10,20)}")
```

Note that owing to the limits of numerical precision, the cube roots of the cubes do not quite print as integers.

Can we recurse as easily on arrays? Scala won't treat them as lists (note how I range-comment code):

```
/*--------------Can't do this with arrays---------------------
def cubeRootsA(xs: Array[Int]): Array[Double] = xs match {
   case Nil => Nil
   case x::xrest => scala.math.pow(x,1.0/3) :: cubeRootsA(xrest)
}
*///----------------------------------------------------------
```

If we tried to recurse while pulling apart the array, we could wind up making lots of intermediate copies. But Scala has a nifty iteration feature with special for-loop syntax and **`yield`** that allows you to stream outputs without using recursion:

```
/** But we can use for and yield to do this
 */
def cubeRootsA(xs: Array[Int]): Array[Double] =
   for { x <- xs } yield scala.math.pow(x,1.0/3)
```

Scala automatically collects the `yield`ed outputs into an array of doubles, which is returned by the function. Note that the left-leaning arrow has only a single bar, not double like with the "rocket". Now we can call this on the array and output the same slice of roots:

```
val rootsA = cubeRootsA(cubesA)
//println(s"Cube roots from indexes 10 to 19: ${rootsA.slice(10,20)}")
println(s"Cube roots from indexes 10 to 19: ${rootsA.slice(10,20).toList}")
```

There's just one more surprise: If we try to print the array slice directly, what Scala gives us is a code of the array. To get the actual values to print out in bulk, we need to convert the array slice into a `List` first.

Timing Code

We will soon cover the *asymptotic* analysis of the running time of operations, plus in some cases their storage requirements. We can, however, check this analysis against concrete running times of code. Scala's nanosecond timer (derived from Java's) is said to solve issues with simple-use code timers of the past, so we will go ahead and use it simply.

Even with nanosecond resolution, measurements still jump around according to scheduling by the multithreaded OS. We need to run code on a fairly large scale both to reduce the variance and to isolate the routines being measured from ambient processing. So we replicate some of the operations in the ArrayAndList example, but with size 1,000,000 rather than 100.

```
/** File "Timing.scala" by KWR for CSE250, Spring 2022.
    Illustrates use of scala System.nanoTime
 */
object Timing extends App {
   val ms = 1000000.0
   val t1 = System.nanoTime()
   //var arr = Array.tabulate(1000000)(i => i)
   var arr = Array.fill(1000000)(7)
   val t2 = System.nanoTime()
   //var lst = List.tabulate(1000000)(i => i)
   var lst = List.fill(1000000)(7)
   val t3 = System.nanoTime()
   println(s"\nCreation times (ms): array ${(t2 - t1)/ms}, list ${(t3 - t2)/ms}")

   println("\nJust to time prepending an element to each")
   val t4 = System.nanoTime()
   arr +:= -1
   val t5 = System.nanoTime()
   lst +:= -1
   val t6 = System.nanoTime()
   println(s"Prepend times (ms): array ${(t5 - t4)/ms}, list ${(t6 - t5)/ms}")

   println("\nHow about appending an element to each---")
   val t7 = System.nanoTime()
   arr :+= 1000000
   val t8 = System.nanoTime()
   lst :+= 1000000
   val t9 = System.nanoTime()
   println(s"Append times (ms): array ${(t8 - t7)/ms}, list ${(t9 - t8)/ms}")

   println("\nNow time to prepend onto a fresh array or list:")
   val t10 = System.nanoTime()
   var arr2 = -8 +: arr
   val t11 = System.nanoTime()
   var lst2 = -8 +: lst
   val t12 = System.nanoTime()
   println(s"Update times (ms): array ${(t11 - t10)/ms}, list ${(t12 - t11)/ms}")
}
```

The really large list takes longer to create than the array. Appending to the list takes markedly more time than prepending, which shows a real difference between its ends, compared to the array. The last block gives a way to tell the time difference between creating the slightly-modified array versus the slightly-modified immutable list. The array copy takes much less time than the original array creation, but the modified list is amazingly quick.

Other Operations on Arrays and Lists (pages 26--34, esp. 30--34)

Praise of "Pythonic" coding style is discussed here and here and at length here and here. The focal point for me is:

> **Thinking at the level of data streams and bulk operations on streams, rather than individual items and steps of code on them.**

For example, instead of looping `for i = 0 to 29` to put `i*i*i` into a list one-by-one, you can get the first thirty cubes in Python by the **list comprehension** `[i*i*i for i in range(30)]`. To get the odd cubes, do `[i*i*i for i in range(30) if i%2 == 1]`.

Scala does not put the yielded value first the way Python does. Instead, there is a choice between beginning with a verb or beginning with the source object, here a range:

```
for (i <- Range(0,30)) yield i*i*i
```

or

```
Range(0,30).map(i => i*i*i)
```

The Scala REPL tells us right away that both of these give an `IndexedSeq[Int]`. If we want to **promote** the output to become a `List[Int]`, we can do the following:

```
        (for (i <- Range(0,30)) yield i*i*i).toList
```

or

```
        Range(0,30).map(i => i*i*i).toList
```

Note that the former needs the extra parentheses shown in red. We could promote to an `Array[Int]` using `.toArray` instead. The choice depends on how you want to use the resulting object---and how efficient you need it to be. To filter so we only get the odd cubes, do

```
        for (i <- Range(0,30) if i%2 == 1) yield i*i*i
```

or

```
        Range(0,30).filter(i => i%2 == 1).map(i => i*i*i)
```

Perhaps these are not as elegant as the Python syntax? Here is a natural-language analogy. The Python, with the yielded direct object first, is like the sentence

"My peace I give to you."

The latter Scala line, with Range(0,30) being like the subject of the sentence, is analogous to

"I give to you my peace."

The former, with the verb `for` coming first, is like the Yoda-esque sentence

"Give to you my peace I do."

The Scala syntax does make it easier to chain further operations on the sequence. For instance, if we want only the odd cubes that are above 1,000, we can tack on another filter:

```
    Range(0,30).filter(i => i%2 == 1).map(j => j*j*j).filter(k => k >= 1000)
```

Note how it didn't matter that we repeated the use of `i` as a "rocket variable" before, but chose j for the cube function the second time around. It doesn't matter: `i` and `j` and now `k` are all **bound variables** (older term: "dummy" variables). We can shortcut the use of the rocket to read from the variable by using an underscore in the first and third instances (but beware trying it in the second):

```
    Range(0,30).filter(_%2 == 1).map(j => j*j*j).filter(_ >= 1000)
```

The methods given by the text on pages 26--34 all work on any sequence, `Seq[A]` in Scala where `A` is the particular type of the items. If you use indexing, e.g. `cubes(0)` to get the first element, then it has to be an `IndexedSeq`. If you use the `head` function instead, then it can be any sequence again. But in practice you would usually be using one of the concrete subtypes anyway: `Array` or `List` (or in special numerical cases, `Range`).

It seems curious to me that `List` allows indexing. This doesn't mean you should *use* indexing with lists. As we saw even with the simpler *append* function, there could be a big performance penalty. You could trip yourself up by using a bad-time operation even while you were attracted to lists because of their time advantages on other methods. This is a big reason why you don't want to always use raw `List`s or `Array`s but rather package them with exactly the operations you want into an **abstract data type** (**ADT**). [Self-exercise: Try seeing if `lst.last` is as slow on the million-item `List[Int]` object `lst` in `Timing.scala` as appending is.] Judging the timing---which often requires knowing the implementation details---is a big upcoming unit.

[Footnote: The `map` higher-order function is not the same as the `Map[K,V]` ADT to come later.]

Tuples

Scala's **tuples** echo Python's tuples but have a different mix of features. They allow "Pythonic multiple assignment" but only with `val` or `var` present and not allowing to omit the parentheses:

```
  var (n,a,str,r) = (5, 3.14, "hello", new Rectangle(3.0,4.0))
```

You can't omit `var` here even when you declared the four variables as being `var`. You can, however, declare a tuple variable

```
  var t = (5, 3.14, "hello", new Rectangle(3.0,4.0))
```

and then an update it without doing a redeclaration:

```
   t = (4, 3.17, "goodbye", new Rectangle(6.0,7.0))
```

The two big differences from lists or arrays are:

- A tuple can have items of different types, without having to drop them down to a common ancestor type like `Any`.
- A tuple can have any fixed length, but the length is fixed---it can't be appended to or decreased.

Like lists, tuples have **matchable structure**. This explains the above limitation compared to Python, and gets into issues that will come up next week in chs. 2-3 of the text. (There was a good question in class about tuples being implicitly `val` that was on-point for this, but I punted answering it. There is in fact a surprising-to-me difference from how Java handles it---or how a "Pythonized Java" would handle it---but I fear going into it would be a distraction.)

The final simple point is that a tuple is an object, and can be returned as such from a function or method. This is a natural habit in Python (IMPHO), but used less in the C++/Java world (again, IMPHO); C++ and Java have a generic `Pair` class but `Triple` is not in their standard libraries (Java has it in an Apache Commons library). In Scala you can conveniently write a method to return a tuple and store the result in a variable. For instance,a call

```
var t = quadrupleWithBiggestRectangle(myList)
```

where `myList` is a list of tuples of the above form, might return the above quadruple with a 6.0 x 7.0 `Rectangle` if 42 is the biggest rectangle area on the list. Then you can read off the four individual values by using the **tuple component syntax** with dot and underscore---in this case, `t._1` would give 4, `t._2` gives 3.17, `t._3` holds "goodbye", and `t._4` holds the rectangle. (Note that the tuple fields are 1-based, not 0-based.)

In C++ or Java you can simply declare a class `Quad` with four "plain old data" fields like `int a; double b; string s; Rectangle r;` You can do this in Scala too, quite succinctly---this one line of code both declares and defines the class:

```
class Quad(var a: Integer, var b: Double, var s: String, var r: Rectangle)
```

This is also AOK for Assignment 1---you could have a triple with 2 integers and a string, and a quadruple with 3 integers and a string. *Also AOK is just using variables at the top that the for-loops modify, without returning tuples at all.* One benefit in either case is that the individual components can have more descriptive names. But in the former case, I was surprised that the use of "plain old data" objects rather than tuples made my code feel wordier---and we will carry that through to the use of 'global' variables rather than methods when comparing answers later.

[I did not cover the remaining footnote, but it's on stuff from pages 26--36 that I did not use in developing the assignment anyway. Monday's lecture will pick up in chapter 2.]

To give a few more highlights of the long list of methods in the text, `takeWhile` works like `filter` but stops on the first false element (if any). Thus if `cubes` is our list (or array) of cubes, then `cubes.takeWhile(_%10 != 9)` will take the cubes up to (not including) 729. Some of the methods require properties of the item type `A`, such as `sum` requiring it to be numeric. In the case of a list of integers, `.sum` is a special case of the call `list.foldLeft(0)(plus)` where you need to first define def `plus(a:Int,b:Int) = a+b` in order to make the sum method easier to pass as integer sum. We'll examine `foldLeft` and other higher-order function on the text's laundry-list when we do recursion. Notice, too, that `.sum` is called without parentheses. This segues into the next topic, which also takes us partwise into Chapter 2.

Fields and Arguments and public/private (including text section 2.2, esp. pp56--57)

There used to be a simple dichotomy: *fields* and *methods*. But mature software design brought more refined differences to the top level. Some have to do with data, others with system details.

★ Some data items are naturally *primary*, others *derived*. For example in baseball, a batter's `hits` and `plateAppearances` are primary data, from which `battingAverage` would be derived as `battingAverage = hits/plateAppearances`. Now the question is: do we make `battingAverage` a field or a method? Giving a field would save the time for a float division, plus whether to round it, say, to 3 decimal places. But allowing public access to the field would lock us in to our internal representation. What do we do if a batter has no plate appearances yet? Rather than mandate `battingAverage = 0.000` in this case, we might use `None` for that and `Some(ba)` otherwise. But we might not want clients to have to deal with the `Option` notation.

Scala tried to cover all bases by allowing fields to define implicit getter methods without parentheses, which the coder can explicitly `override`. Moreover, a parenthesis form like `battingAverage()` can override a non-paren form. The understanding that goes with using `()` is that the access performs some client-visible side-effect. For getting a batting average, I can't imagine any, so it would be left with field syntax. The text follows a general rule fo thumb in Scala:

- If the data item is primary and won't be changed by client or library code, make it a public `val` argument. No more getter syntax is needed.
- If it is primary but subject to internal change, make it a `private var` with a leading underscore on its name, such as _hits. Then make a public getter `def hits = _hits` with the same field syntax. No parens since there shouldn't be any side-effect from just reading the number of hits. (No need to use a `return` statement either---the value of the last statement executed becomes the return.) This still allows the `hits` field to be updated by another method that records a new hit by the batter.
- (If clients are really allowed to modify the entry, as being "plain old data" with global scope, then just use a public `var` declaration.)
- If the data item is primary but optional, keep it as an argument but give it a default value.
- If the data item is a compound object, don't just choose `val` vs. `var` but make sure the data object itself is immutable if you need it to be.

The text gives a good example for the last two points. Compared to the simple batting-average example, the text's grade-point average example has the extra wrinkle that it can come from quizzes, assignments, and/or tests. Usually the `Student` object would be created with all of these empty, but there are situations where we might need to (re-)create the student's record in mid-semester, or carry over scores from the first part of a two-course series. So the text makes each of them `private var List[Int]` constructor arguments with leading _ in their names and default value `Nil`.

Their being `List` prevents you from changing a grade once it is entered, but using `var` allows adding new grades to the list---note that the `addQuiz`, `addAssignment`, and `addTest` methods use `::=` to prepend the new grade to the list (not append!) . To allow changing grades, use `Array[Int]` instead---and the text's box on page 57 boils down to saying that so long as these fields are `private`, chaos can be avoided. To allow a student to check the entered grades, the text does one-line getter methods without praentheses:

```
    def quizzes = _quizzes
    def assignments = _assignments
    def tests = _tests
```

[It maybe should be said that these methods in practice would have package scope rather than public scope, where the package would require ID so that a student could look up only eir own grades.] Finally, how to handle the grade averages? They are derived quantities so they are not arguments. The text avoids making them fields at all. There are public methods that do the division and legislate that an empty average is returned as zero. Note the use of `toDouble` because the grades are integers but we want to avoid doing integer division.

[In Chapter 3, page 90, we will see special syntax for setter methods. The general term **property** means a getter/setter pair that looks like field syntax but gives the customizability advantage and flexibility of using methods under-the-hood. One of the languages that pioneered this idea is C#. Another innovation of C# was providing streamlined function-objects in the form of "delegates"; Scala of course implements this feature copiously in multiple ways...]

Classes and Objects and Constructors (text coverage in chs. 2--3 is diffuse, IMPHO)

★ At what point of construction does the object begin to exist? Generally speaking, construction has a "buildup" phase and then often a "modify" phase. The buildup phase is further stratified in ways that may or not be explicit to the coder:

1. Superclass constructors are called first. (In 1984, I might have inserted the words "if any", but nowadays any class you write has a "Big Brother".)
2. If this is not the **primary constructor**, then that is called next, before executing any more of the current constructor.
3. Any other constructors in the current class, that are called, must be called before doing anything else in the current constructor.
4. Then the current (i.e., top-level) buildup phase can be commenced, followed by the final modify phase. No more constructors of `this` class can be called by this point.

In Scala, **primary constructor** has a more-specific meaning: it means the constructor giving the arguments to the class, allowing arguments with default values to be omitted. Other progarmming languages have similar concepts, but they may also privilege the **default constructor** which has zero arguments, and/or force you to code a **copy constructor** (and maybe a corresponding assingmnet operator).

C++ provides separate syntax for the buildup and modify stages: a colon then comma-separated list of constructor calls and field inits for buildup, then the usual `{ ... }` with statements syntax for the body. Java doesn't use separate syntax, but forces the constructor calls to be the first statements of the body, with a call to `super` coming first (implicit if not explicit). JavaScript is not so regimented (it has other technicalities), and Python even less. Scala is more regimented---and gives even more syntax for dealing with it---but a nice principle comes out of its complexity. So let's see possible answers to the "when does the object begin to exist?" question:

1. Not until the whole construction process finishes---in particular, not until the "modify" phase of the client-invoked constructor exits.
2. When the last "modify" phase begins---which is the same as when the top-level buildup phase has finished.
3. When the last other constructor called has finished, whether another top-level constructor in this class or a superclass constructor, and only top-level fields remain to be built.
4. When some superclass constructor has finished---so that some ancestor object exists.

Option 4 can actually be broken down further, but we'll leave it at that. The previous issue about field can get involved in this. For instance, suppose we derive `LeadoffBatter` from `Batter`. The client invocation of the LeadoffBatter `constructor` will probably give the raw `plateAppearances` and `hits` values, but what if the top-level buildup wants to access the `battingAverage` value? It could do the division `plateAppearances/hits` by itself, but that would create **parallel code** with how the superclass does it. (This might not seem like an issue in our "toy example" but real life much more maintainable code could be involved.) So instead it would want to access the superlass field-or-getter for the average, but that presupposes that the ancestor object exists so the field is available.

The C++ standard seems to have settled on answer 2. Even that, however, gives a danger of "breaking encapsulation": When there is a "modify" phase, its purpose is largely to establish logical properties that the object consistently obeys (i.e., logical **invariants**). Before that phase has finished, the object may violate those properties---say because variables default to values that need to be fixed up.

Scala gives a way to have the effect of answer 1 in a natural manner:

Make the companion object be a construction factory.

This can be coupled with making the primary constructor `private`. The companion object can use special syntax with `apply` to invoke the primary constructor in ways it sees fit. Because the companion object cannot see fields of class instances (i.e., it is "static"), it cannot do anything with them until the whole object it is building is fully constructed. Then it can safely modify the object as-needed before returning it.

[still need to choose an example from the text but we are into Week 3 by now...]