CSE250 Week 3: Class Structure in Scala, Looking Toward OO Design of ADTs

These notes will cover Chapters 2--4, more quickly than Chapter 1

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 of 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`.

```
class Student (    //paren not curly brace
    val firstName: String,
    val lastName: String,
    private var _quizzes: List[Int] = Nil,
    private var _assignments: Array[Int] = Nil,
    private var _tests: List[Int] = 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.toList
    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 assingment 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.

Continuing the text's `Student` example, now on page 95:

```
class Student (    //paren not curly brace
    val firstName: String,
    val lastName: String,
    private var _quizzes: List[Int] = Nil,
    private var _assignments: Array[Int] = Nil,
    private var _tests: List[Int] = Nil,
)
object Student {
    def apply(firstName: String, lastName: String,
              quizzes: List[Int] = Nil,
              assignments: List[Int] = Nil,
              tests: List[Int] = Nil): Student = {
        new Student(firstName, lastName, quizzes, assignments, tests)
    }   //note this calls the automatic primary constructor
    /** Read records from file with scores in CSV format
     *
    def apply(fileName: String): Student = {
        val source = io.Source.fromFile(fileName)
        val lines = source.getLines()   //parens should be on p35 too
        val firstName = lines.next
        val lastName = lines.next
        val quizzes = lines.next.split(",").map(_.toInt).toList
        val assignments = lines.next.split(",").map(_.toInt).toList
        val tests = lines.next.split(",").map(_.toInt).toList
        source.close()      //IMHO should have parens since side effect
        new Student(firstName, lastName, quizzes, assignments, tests)
```

```
    }
}
```

The second `apply` creates a second constructor by calling `Student(fileName)`. The point regarding the "when does it exist" safety issue is that even if we changed the last line to

```
val ret = new Student(firstName, lastName, quizzes, assignments, tests)
```

then the object "`ret`" would be fully formed. The companion object, being "static" in C++/Java terminology, would not be able to see directly inside the individual fields of "`ret`". It would have only the ability to ask the object `ret` itself to operate on its fields or apply methods to itself. The one privilege is that the companion object can make `ret` invoke private methods of the `Student` class, but the entire-ness of the `ret` object is maintained in that process. In larger, non-toy examples this can be meaningfully safer.

[I also talked about the assumptions on the input file. The first line must have the first-name, and then the last-name must be on a new line. Then come three lines with the scores being inputted for quizzes, assignments, and tests, respectively. Speaking in general---not as a toy example---it would be better for those lines to be prefaced with labels like "Quizzes:", "Assignments:", and "Tests:" so they could be parsed and read in any order and without caring if any blank lines are before them. One "ulterior" point of the current assignment is that blank lines should not be as vital format markers as blank spaces within a line. I deviated from the text by making the numerical lines comma-separated. In both cases it is relevant that the `String.toInt` method ignores both any leading and any trailing whitespace on the number it reads.

Whether in the text it is really OK to split on a single space via `split(" ")` is a thorny issue. If there are 2 spaces between numbers, one would be left over but that would be OK if it got attached to the second number. *Oh, wait,* maybe you would get an empty string between the numbers. That empty string could then be converted to a zero, giving someone a zero on the quiz! Doing `split("\\s+")` makes sure that all the whitespace in-between is gobbled up. It and the use of a comma are generally safer.]