Introduction to Scala: Objects, Classes, and Traits

Drew McDermott drew.mcdermott@yale.edu

Lecture notes for 2016-09-03

1 The Basics

Syntax vaguely Java-like.[1]

A typical variable declaration in Scala looks like this:

```
val cruiseControl: Int = speedLimit + 2
```

Key things to note:

- Every declaration (except for method and function parameters) begins with a reserved word of the language, such as `val`.
- Types are always capitalized. There is no distinction between "primitive" types and "object" types. Every value is an object, including numbers.
- Mnemonic variable names use camelCase. I don't like it, but the use of underscore to separate components of a name is frowned upon.
- Everything declared is declared to have a type. For a variable, that means the types of all the values it can have. However, in most cases the type can be omitted and the compiler can figure it out:

  ```
  val cruiseControl = speedLimit + 2
  ```

When I teach Intro Programming, I harp on the fact that a variable is the name of a "box" that holds its successive values. So when you write the assignment expression

```
x = x+1
```

that's interpreted to mean "Take the value out box `x`, add 1 to it, and put the result back into box `x`." In Scala, for `x` to be the name of a box, you have to declare it using reserved word `var`:

[1] This file starts off terse and then becomes increasingly verbose. I hope the information herein is a useful complement to that in *Functional Programming in Scala*. I've crammed as much possible excessive verbiage into footnotes as possible. So on first reading you may want to skip the footnotes. Except this one.

```
var x: Int = 0
while (x < n) {
  println("Now x = " x)
  x = x+1
}
```

Expressions executed purely for effect, such as the `println` expression and the assignment expression, can be separated by semicolons. Declarations, too. So we could have written the code above thus:

```
var x: Int = 0;
while (x < n) {
  println("Now x = " + x);
  x = x+1
}
```

You could put one after the assignment expression if you want. But, if you lay your code out so that immediate subexpressions of an expression E all start on a new line, indented further to the right than the column where E begins, then the semicolons can be omitted. The examples in this lecture all obey the "house style," which specifies that subexpressions are indented by two characters.[2]

I'm using the word "expression" here deliberately, to emphasize that in Scala there is no syntactic category of "statements." A statement is just an expression whose value is ignored (and hence must be evaluated for the effects it has, such as changing a variable value, mutating a data structure, doing some I/O, or throwing an exception).

Every expression must have some value; if nothing else, it can be a value of type `Unit`. There's only one value of this type, written `()` (and also pronounced *unit*).

Every `val` or `var` — anything declared — has a *scope*, the part of the program where its declaration is in force. This is the innermost brace pair the declaration is inside of (except for method and function parameters, which we'll get to).

Here's an example, a complete Scala program:

```
object Hello extends App {
  val n = 1
```

[2]It is important to present code using a fixed-width font. Most text-file editors will do this by default.

```
  var x\coln{}Int = 0
  while (x < n) {
    println("Hello, world.")
    x = x + 1
  }
}
```

Save this file with the name `Hello.scala`[3]

To run a Scala program, you have to compile it first. The compiler targets the Java Virtual Machine (JVM), so you normally pay the overhead of emulating the JVM to interpret that code.[4] The compiler is called `scalac`, which matters to you if you call it from the command line. So to compile `Hello.scala` you execute

```
cs470$ scalac Hello.scala
```

The files produced by the compiler have the extension "`.class`," because each one implements a class. (See next section.) Compiling `Hello.scala` produces three class files: `Hello.class`, `Hello$.class`, and `Hello$delayedInit$body.class`. It's the first we care about. The others are a mystery. To run the program, execute

```
cs470$ scala Hello
```

This will run the program (actually, the `main` method) associated with `Hello.class`. Using the `App` class as we did when added "`extends App`" hid the `main` method, but it's in there somewhere.

Scala produces even more classes with dollar signs in their names than Java does. It's not a good idea to use `scalac` "naked," because you will lose track of your source files in all that clutter.

I've provided two simple scripts for calling `scalac` and `scala` while tucking all the class files in a subdirectory called `bin`. (You have to create the subdirectory.) In my scenario above, I'm imagining you're in a shell (command interpreter) which prints the name of the current directory (`cs470`) followed by a dollar sign as a prompt. (That's a common arrangement in Unix systems, including OSX. Similar things can be made to happen in

[3]Unlike Java, Scala does not *require* that a file have the same name as the class it defines. You can put anything you want in a file, including multiple class definitions.

[4]Sophisticated "just in time" compilers incrementally compile to machine language pieces of JVM code that are executed many times. I doubt we will notice any resulting speedups in this course.

Windows as well.) The two scripts can be found on the Canvas system, under `Files > Code > Scripts`. In Unix they're called `scc` and `sca`. The Windows versions are `scc.bat` and `sca.bat`. They take the place of `scalac` and `scala` respectively.

To use these scripts, upload `scc` and `sca` (OSX and Unix) or `scc.bat` and `sca.bat` (Windows). The most convenient place to put them is in the directory containing your CPSC470/570 files. I'll assume this is called `cs470` for brevity; be sure to create the `cs470/bin` directory if you haven't already.

For Unix or Windows to find a file to execute, it uses the `PATH` environment variable. You should make sure that the `cs470` directory is "in" (the value of) that variable.[5]

The last issue to worry about (I hope it's the last) is that the operating system will refuse to execute any file that you don't have "execute permission" for. On Unix systems (e.g., OSX), you issue the commands

```
chmod u+x scc
chmod u+x sca
```

to any shell. On a Windows system, the equivalent is

```
icacls scc /grant U:(X)
icacls sca /grant U:(X)
```

where U is your userid, the name you're logged in as. Assuming you logged in as `wilma`, you would write

```
icacls scc /grant wilma:(X)
icacls sca /grant wilma:(X)
```

I haven't tried this; if you have trouble with it, let me know. An alternative is to replace `(X)` with `F` (*without* the parentheses), which grants `wilma` full access, which should include the right to read, execute, edit, and delete

[5] On both systems, let's assume this directory, `cs470` is under the `16f` directory of a top-level directory of yours, `classes`. If your name is Wilma van Orman Quine, then your directory might be `/Users/wilma/`. Typically you add `":/Users/wilma/classes/16f/cs470"` to the end of the `PATH` variable on an OSX or other Unix system; or add something like `";\Users\wilma\classes\16f\cs470"` to `PATH` on a Windows system, except `\Users\wilma` is not likely to work. (It won't work in every Unix system either.) In a Unix shell, `~` is shorthand for your top-level directory (which might be `/Users/whoever-you-are` or might not); in Windows, the equivalent is `%userprofile%`. So Wilma would do better to add `";%userprofile%\classes\16f\cs470"` to the end of `PATH`. (If she's on OSX or some other Unix system, she could add `":~/classes/16f/cs470"`.)

the file. Normally, if `wilma` downloaded the file, she has all of those rights already, except eXecute, which for some reason the OS demands that you declare explicitly.[6]

2 Classes and Traits

So far we have only discussed singleton objects. They are important, but not as important as classes, traits, and case classes. We discuss classes and traits here, case classes in a later lecture.

2.1 Classes

A *class* definition starts with the keyword `class`. It defines a bunch of objects that look and act similar. By that I mean that every object of the class presents the same interface to pieces of code that want to use it. *A class definition is a template for an object in the class.* It is also the code for the *constructor* of the class. It mostly consists of declarations (of `val`s, `def`s, and the occasional `var`[7]). Each such declaration establishes a member of the resulting object, public unless explicitly declared to be private.

It may seem odd that the code for a new object's constructor is also a template for that object, but consider the following example:

```
class Oddball(x: Int, y: Int) {
  val d: Int = x-y
  val s: Int = x+y
}
```

You construct a new object of class `Oddball` by executing

```
new Oddball(50, 40)
```

Inside the constructor, `x` is bound to 50 and `y` to 40. Two `val`s are declared, `d` = 10 and `s` = 60.

Any such val becomes a member of the resulting object. One might write

[6] If I had to guess, I would guess that this was originally a primitive security measure, making it slightly harder for a malicious intruder to insert a destructive file and then execute it. Just a tiny bit harder, anyway.

[7] A public `var` member of an object is a point that allows any code with a reference to the object to alter that member in an arbitrary way (as long as its type is preserved). Even a `private var` that is set by some method and read by another allows the object to be changed. Our emphasis this term will be on *immutable* objects, so expect very few `var`s with scope over the object. Declarations of `var`s *inside* methods are sometimes unavoidable and do not allow the object to be altered.

```
odd = new Oddball(50, 40)
println("odd.s = " + odd.s + " and odd.d = " + odd.d)
```

which, if executed, prints

```
odd.s = 90 and odd.d = 10
```

(assuming that `odd` is a `var` of type compatible with `Int`).

Hence it is not unnatural, in fact amazingly cool, that what look like local variables of the constructor are in fact the public members of each object of the class. (You can declare them `private` to hide them.)

You can define more complex members, the *methods* of the class, by using the keyword `def`. Suppose we add to the definition of `Oddball` this declaration:

```
def prod: BigInt = {
  d*s
}
```

Now `odd.prod` is the product of `d` and `s`. Because the product of two `Int`s can be too big to fit in the 32 bits an `Int` gets, we use the built-in class `BigInt`, which defines integers of arbitrary size. Actually, the class lives in the `scala.math` package, so we should write `scala.math.BigInt`. That soon becomes awkward. It's generally better to use `import` to declare that symbols from a package should be included in the current namespace.

Before showing how to do that, let me talk about how the body of the `prod` method works. You may have noticed that we did not write "`return d*s`"; Scala does have a `return` expression, but we won't need it. The value of any method (or function) is *the value of the last expression evaluated when it's executed.* In the case of `prod` this is `d*s`.

Most methods are defined by code blocks (a series of declarations and some expressions surrounded by braces). But if a method has no declarations and just one (simple) expression, the braces are unnecessary.[8]

So our revised class definition is

```
class Oddball(x: Int, y: Int) {
  import scala.math.Bigint
  val d: Int = x-y
  val s: Int = x+y
  def prod: BigInt = d*s
}
```

[8] Most methods have parameters, of course. But that's a totally independent issue.

The `import` declaration makes the symbol `BigInt` as defined in package `scala.math` available inside the definition of `Oddball`. Unlike many other languages, Scala allows `import` declarations to be used inside `class` definitions (and any other scope-defining context, including an `object` definition).

If a method has no parameters, it can be called with (e.g., `odd.prod()`) and without (e.g., `odd.prod`) parentheses. However, it is considered good practice to include the parens if and only if the method has side effects. Because our focus is on functional programming, almost all our methods and functions will have no side effects, so the parens will be omitted when there are no arguments.

One good thing about this convention is that the reader of `odd.prod` can't tell if `prod` is a `val` or a `def`. Indeed, we could (and, in practice, generally would) have defined it as a `val`:

```
class Oddball(x: Int, y: Int) {
  import scala.math.Bigint
  val d: Int = x-y
  val s: Int = x+y
  val prod: BigInt = d*s
}
```

In other cases, the computation of `prod` might be so expensive that one would like to delay it until the value is needed.[9]

Going back to the `def` version, could we have defined `prod` as the product of `x` and `y`? Yes; the constructor parameters' values are available as long as they are needed. However, they are not members of the object being defined.[10]

Except for method parameters, that's about it. However, method parameters take two seconds to explain. They have almost the same syntax as constructor parameters: A series (possibly empty) of comma-separated repetitions of the pattern "v`: `T `[= `d`]`", where v is an identifier, T is a type, and d is an optional default value. If a default value is supplied, then when the method is called the argument at that position may be omitted.

An argument may be omitted only if it has a default value *and* all the arguments after it are provided using *named-argument notation* or are themselves omitted. Foo,[11] I've exceeded my two seconds, so skip this paragraph on first reading. When a method or constructor is called, you don't have to

[9] Why evaluate it repeatedly? We'll talk about how to avoid that in a later lecture.

[10] Unless you put `val` in front of the parameter name, as in "`val x: Int,`...."

[11] Nerdish abbreviation for fooey?

use positional notation at all. You can write `new Oddball(y = -1, x= z/2)` if you want, so that argument `-1` is the value of `y`, and the value of `z/2` is the value of `x`. You can mix positional notation with explicit-parameter-name notation, if you're careful.

To call a method defined with parameters, you write

obj`.`*meth*`(`—*params*—`)`

The expression *obj* occupies a special position, but it's really like a zero'th parameter of *meth*. Inside the method body, you access the i'th argument by writing v_i, the name of the i'th parameter.[12] You access the zero'th argument using the reserved identifier `this`. However, it is seldom necessary (or encouraged) to write `this.`x, where x is a member of *obj* (i.e., defined in the `class` definition). That's because we're in the middle of *obj*'s class definition, inside a `def` from that definition, and *a class definition is a template for an object in the class.* (I'll keep saying this until it becomes irritating.) All the `val`s and `def`s you see in the scopes the current `def` is nested in are already "in scope." So rather than write `this.`x, *just write* x. That's the natural thing to do usually (nobody's tempted in the definition of method `prod` to write `this.d * this.s`) but in a complex class definition one can forget.

On Canvas, the definition of `Oddball` has been placed in Files > Code > Examples.[13] Download that file and you'll see the definition of an object `OddballTest` after the definition of class `Oddball`. Go ahead and compile `Oddball.scala` (using `scc`) and inspect the `bin` directory. It should contain the two class files `Oddball.class` and `OddballTest.class` (and God knows what other crap, depending on when you take a peek at it).

The script `sca` (and `scala` itself) are not interested in source files; you can delete them or archive them if you want. These scripts are interested only in `.class` files. So to run the (not very comprehensive) test for the class `Oddball`, execute

```
cs470 $ sca OddballTest
```

The output should be

[12] I'm hewing to the formal nomenclature, where a *parameter* is an identifier in a method's parameter list, and an *argument* is (the value of) the expression a parameter gets bound to on a particular call of the method. Informally, the term *argument* is used for both.

[13] Actually, there's a layer at the top, "Artificial Intelligence." I'm going to overlook this, since there's nowhere else you can get to from "Files" but "Artificial Intelligence," which gets its name from the name of the course.

```
odd.s = 90 and odd.d = 10
odd.prod = 900
cs470 $
```

Scala has relaxed a lot of Java's restrictions on what source files may contain. (Which makes one wonder why those restrictions were imposed in the first place.) You can define as many entities in a file as seem to belong together. You can go overboard with this freedom. For example, no matter how complex your application, you can define it within one source file, but it will be clearer for someone trying to read it or integrate with other things if you (a) give it its own package; and (b) split it into multiple files, and put them in their own subdirectory.[14] Scala gives you the freedom to put the source file wherever you want, independent of which subdirectory the resulting `.class` files go into.

2.2 Traits

A *trait* in Scala is a superordinate class-like object from which classes inherit various useful things. A class *inherits* from another class or trait if it is defined using the reserved word `extends`, as in

`class` N_0`(....)` `extends` N_1 `[with` N_2 `with` N_3 `...]` {

If N_1 is a class, then N_0 shares all of N_1's members (except the most private). If N_1 is a class, and its constructor takes arguments, you can't just write "`extends` N_1 `...`"; you have to write

`class` N_0`(....)` `extends` N_1`(`*–params–*`)` `[with` N_2 `with` N_3 `...]` {

[14] In this course, we won't worry about this kind of software engineering issue too much. We'll give detailed instructions on how to organize your code and hand it in. However, here's the deal on packages. Every file can optionally begin with a declaration

`package` p_1`.`p_2`.`$\cdots$`.`p_n

which places all the names defined in the file in the namespace given by the sequence of ps. When compiling, `scc` will place every `.class` file it creates in the subdirectory (using Unix notation):

`bin/`p_1`/`p_2`/.../`p_n

To execute a program (a class P with a `main` method or an object P defined by `object ... extends App`), run

`sca` p_1`.`p_2`.`$\cdots$`.`p_n`.`P

If there are other entities from which N_0 inherits stuff, they are added using the reserved word `with` as shown. At most one of the N_i ($i \geq 1$) is the name of a class; all the others are names of traits. You can think of a trait as being a bowl of "toppings" that get sprinkled on the ice cream cone specified by the class definition(s).[15]

Scala allows a class to inherit several things from a trait. Before saying what they are, let's go through the basics. A *trait definition* is exactly like a class definition except that the keyword `trait` is used in place of `class`. There are two other differences:

1. Traits do not have constructor parameters.[16] So you don't have to worry about providing them in the header of a class definition.

2. You can't directly instantiate a trait: "`new` *traitname*" is a nonstarter.

Okay, we've gone through the basics. Traits can serve as class interface descriptions, but they can do much else. Here are some of the other things a class can inherit from a trait:

- Behaviors that any class `extending` the trait must implement, in the form of *abstract* member definitions, which describe a member's type without giving it a value

- Constants

- Internal classes. If a trait contains a class definition, then it's as if that definition occured within any object or class extending the trait.

- Types, as defined by `type`. (These are just useful abbreviations, and can be used other places, too, of course.)

Traits work with *case classes,* for which see next installment of notes.

[15] They are often called *mixins*. The term comes from an ice-cream parlor in Cambridge, Massachusetts, where hackers from the MIT AI Lab hung out, the same hackers who were creating some innovative OOP concepts at the time, as part of the Flavors system, the first object-oriented add-on to Lisp. At "Steve's Ice Cream," instead of sprinkling the toppings on top, they would pound the ice cream flat, sprinkle the goodies on, then fold the ice cream over and pound it flat again. After all requested goodies were "mixed in" this way, the ice cream would be put back into a roughly spherical shape and placed in a cone or dish. Those were the days.

[16] Except type parameters, which are discussed in Lecture 5.