Scala Good Coding Practices

Spring 2020

Contents

1 Introduction

This document is the CS 18 "good" coding practices guide. Writing code that follows these practices will serve you well beyond CS 18, because when your code is well structured and well organized, it is easier to read, debug, test, optimize, etc.

Please read this guide carefully. In addition to functionality, your assignments will be graded on how well you abide by these good practices, so this guide should be a valuable tool to you while coding.

Note that this guide is an evolving document, subject to revision as the semester progresses. If we observe any particularly good or bad coding practices in the assignments you turn in, we will extend this document so that we can share those practices with other students.

2 Types

One of the reasons we love Scala is its all-powerful type inference engine. You should love it too, and will, even if only because it gives you a chance to save on some typing!

You should not annotate the type of a `val` field when its type is immediately evident from its value:

```
// Right
val name = "Alexander Hamilton"

// Wrong
val name: String = "Alexander Hamilton"
```

As you can see in this "wrong" example, type annotations should be styled without any space before the `:`, but with a space afterwards. For example:

```
value: Type
```

Here is an example of a purely functional method that is properly type-annotated:

```
def add(a: Int, b: Int): Int = a + b
```

Likewise, here is an example of a non-purely functional method that is properly type-annotated:

```
def add(a: Int, b: Int) {
  this.sum = a + b
}
```

We use this style of annotating non-purely functional methods throughout CS 18, in spite of the fact that the official Scala guide does not. The official guide (which argues in many places that saving a few keystrokes now will not prove to have been worth the trouble later) prefers:

```
def add(a: Int, b: Int): Unit = {
  this.sum = a + b
}
```

We find this verbosity makes Scala code more difficult to read, because the absence of a type annotation and an equals sign makes the non-functional nature of a method immediately obvious, while including these things obscures this point.

While Scala's type inference engine is very capable, for the sake of safety, readability, etc., you should still annotate the return types of all purely functional methods.

3 Naming

The Scala naming conventions are similar to those of Java.

```
// Package names should be all lowercase letters
package thisisapackage

// Class names should begin with an uppercase letter
class ThisIsAClass

// Method names should begin with a lowercase letter
def thisIsAMethod

// Constant names should begin with an uppercase letter
// This convention is *different from Java's*
// in which constants are in all caps
val MyConstant

// The name of everything else should begin with a lowercase letter
val thisIsNotAConstant
var thisIsAVariable
```

4 Formatting

One (of the many) important distinction(s) between Java and Scala is that the Scala compiler uses the end of a line to determine where the end of a statement is. Consequently, you can *and should* omit end-of-line semicolons in your Scala code, thereby making it more concise.

5 Declarations

Variables

All other things being equal, prefer immutability to mutability. Wherever possible, prefer `vals` to `vars`: i.e., prefer immutable variables. Likewise, wherever possible, prefer immutable data structures.

If you find that a mutable data structure is really the best solution to the problem at hand, (e.g., if you want to add and remove items from a list) you can declare a mutable data structure (e.g., `MutableList`) as a `val` or an immutable data structure (e.g., `List`) as a `var`.

Mutable `vars` *are never necessary in Scala programs, so they should be avoided.* All functionality can be achieved with either immutable `vars` or mutable `vals`.

Classes

If a class can be defined using just the constructor syntax, without any additional fields, you should not include a set of empty curly braces.

```
class Television(val screenSize: Double, val brand: String)
```

6 Statements

if, if-else, if-else-if-else Statements

In the same way that we make the distinction between purely-functional methods and methods with side effects, we can distinguish between **`if`**/**`else`** statements that are purely functional (i.e., expressions) and ones that produce side effects. Just as we omit braces in purely-functional methods and include braces in methods with side effects, we do the same for **`if`**/**`else`** expressions/statements.

Consider the following examples:

```
// Good style: this if/else expression is purely functional
if (s == "I love ice cream") "I'm glad that you like ice cream!"
else "Why don't you love ice cream?"

// Bad style: this if/else expression is purely functional
// but the braces which suggest it has side effects
if (s == "I love ice cream") {
  "I'm glad that you like ice cream!"
} else {
  "Why don't you love ice cream?"
}

// Good style: this if/else statement has side effects
if (s == "I love ice cream") {
  println("I'm glad that you like ice cream!")
} else {
  println("Why don't you love ice cream?")
```

```
}

// Bad -- this if/else statement has side effects but is
// written as if it were purely functional
if (s == "I love ice cream")
  println("I'm glad that you like ice cream!")
else
  println("Why don't you love ice cream?")
```

In the above examples, we added line breaks in the purely-functional **`if`**/**`else`** expressions, because otherwise we would have gone over our 80 character limit. However, when possible, it is better style to keep an **`if`**/**`else`** expression all on one line, like so:

```
// Good style: this if/else expression is purely functional
if (x < 10) "x is less than 10" else "x is greater than 10"
```

In summary, you should omit braces if your clauses are just one line, unless they have side effects, in which case you should surround them by curly braces, even if they are only one line.

return Statements

The `return` keyword is not generally used in Scala except to alter the flow of control, for example, to exit a loop early. Instead, any Scala method that is meant to return a value will return the value of the last line of code in the function.

More generally, in any block of code, the value of the last line of that block is considered to be the value of that block. For example, the value of this block of code is **`false`** (since 36 is greater than 33):

```
var myVar = 17
myVar += 1
myVar *= 2
myVar <= 33
```

Thus, even though `return` is a syntactically valid keyword, and could be used to return a method's value in Scala just like it is used in Java, it is not necessary to use it in this way, and it is considered poor style to do so.

Finally, where it improves readability, you should skip a line before the last line in any Scala method that returns a value, just as you (usually) did in Java. (But you need *not* do this in methods that are executed for their side effects only.)

7 Methods

Much of Scala's power derives from the fact that it supports functional, imperative, and object-oriented programming. Java does not (fully) support functional programming, and hence Java methods are not naturally purely functional. **Purely functional** functions (*a.k.a.* methods) possess two defining characteristics:

1. The function always evaluates to the same resulting value given the same argument value(s). That is, it doesn't depend on any state or on any input from I/O devices.
2. Evaluation does not cause any side effects or output anything to any I/O devices.

Hint: A good way to know if a method is purely functional is to ask yourself if it would be possible re-write this method using only the subset of Racket or OCaml that you learned in CS 17!

To simplify reasoning about your programs, you should strive to write purely functional methods whenever possible. If, however, you find yourself writing a non-purely functional method, perhaps for efficiency reasons, ideally, your method would not return a value. That is, you should strive to write methods that either return a value because they are purely functional, or do not return a value because they are instead intended to cause side effects.

One of the goals of our Scala style guidelines as they pertains to method declarations, definitions, and invocations is to enable readers of your code to tell at a glance whether a method you wrote is purely functional or not.

7.1 Syntax

Purely functional methods can be expressed on a single line if that line is sufficiently short. For example:

```
def add(a: Int, b: Int): Int = a + b
```

You can also declare methods with default parameters, as follows:

```
def add(a: Int = 17, b: Int = 18): Int = a + b
```

As always, include a space on either side of infix operators (e.g., =).

In methods with side effects, the body must be enclosed in braces:

```
def printSum(ls: List[String]) {
  val sum = ls.map(_.toInt).foldLeft(0)(_ + _)
  println(sum)
}
```

7.2 Invocation

Generally speaking, method invocation in Scala follows Java conventions. For example, if a Java class, `MyClass`, has a method, `myMethod`, that takes as input two integers, invoking this method looks like this:

```
  MyClass.myMethod(17, 18);
```

The Scala equivalent looks the same (without the semi-colon):

```
  MyClass.myMethod(17, 18)
```

However, method invocation looks a little different in Scala when it comes to methods without any arguments, and those with exactly one.

Arity-0 Methods

An **arity-0** method has no arguments. Scala attaches significance to whether an arity-0 method is declared with or without parentheses:

```
def foo1() = ...
def foo2 = ...
```

Only methods that have side effects should be declared with parentheses. Purely functional methods should be declared without parentheses.

Furthermore, method invocations should conform with their declarations: if a method was declared with parentheses (because it has side effects), it should be called with parentheses; likewise, methods declared without parentheses should be called without parentheses. That way, you can determine from just a quick glance whether a method has side effects or is purely functional.

While there may sometimes be a temptation to save a few (only two, actually!) characters, resist! Instead, opt in favor of writing code that is more readable and maintainable. You will thank yourself (and perhaps even us) later.

Note: As constructors cause side effects—they create objects—constructors that do not take any arguments should be declared with parentheses.

Arity-1 Methods

Scala allows you to use infix notation (e.g., addition in the expression `1 + 2`) even for methods you write on your own (e.g., `1 myAdd 2`). Specifically, if an **arity-1** method (i.e., one with exactly one argument) can be invoked an infix style, like this:

```
MyClass myMethod 18
```

instead of like this:

```
MyClass.myMethod.(18)
```

Infix notation is preferred in two cases: 1) for purely functional methods, and 2) for methods with functions as parameters. For example:

```
// Good style: example of an arity-1 method that is purely functional
cs18StudentList mkString ","

// Good style: example of an arity-1 method that takes as input a function
myList foreach (listElement => println(listElement))
```

The use of infix notation is especially important when invoking higher-order functions and chains of higher-order functions, as in the following example:

```
// Good style
myData map (_ => _ + 1) filter (_ > 10)

// Bad style: does not use infix notation
myData.map (_ => _ + 1).filter (_ > 10)
```

In the second of these function applications, the style is "bad" because one might misinterpret this code as saying that the object `(_ => _ + 1)` is invoking its method `filter`, when in fact it is the object that results from applying `map` to `myData` and `(_ => _ + 1)` that is invoking `filter`.

Note: This style guideline only applies to higher-order functions of arity-1. It does not apply to `foldLeft`, for example, a higher-order method of arity-2:

```
// Not a syntactically valid expression
map (_.toInt) foldLeft (0) (_ + _)


// A syntactically valid expression
map(_.toInt).foldLeft(0)(_ + _)
```

8 Options vs. Exceptions

In CS 18, where we are first learning imperative and object-oriented programming principles, we strive first for correctness, and only second for efficiency. A type checker, which finds bugs during compile rather than run time, is an invaluable aid in writing correct programs. So, while options and exceptions are both ways of handling failure in our programs, options are usually preferable, because they can be type checked, and hence mostly debugged during compile time.

But there is (at least) one exception to this general rule. If failure is expected to be a rare occurrence, then there may be unnecessary overhead in using options (verbose code, and loss of run time efficiency), so exceptions might be preferable. For example, if you have implemented a class `BrownStudent` with a field `bannerID`, then it might make sense to raise an exception if ever a Brown student does not have a banner ID, since all Brown students should have a banner ID.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: `https://cs.brown.edu/courses/cs018/feedback`.