Scala Documentation

Release 2.11.6

Various

Sep 30, 2017

Contents

Contents:

CHAPTER 1

Getting Started

Scala for Programming Beginners

Your First Lines of Code

The “Hello, world!” Program

Run it Interactively!

Compile it!

Execute it!

Script it!

CHAPTER 2

Tutorials

Contents:

FAQ

Frequently Asked Questions (and their answers!)

A Tour of Scala

Contents:

Introduction

Abstract Types

Annotations

Classes

Case Classes

Compound Types

Sequence Comprehensions

Extractor Objects

Generic Classes

Implicit Parameters

Inner Classes

Mixin Class Composition

Nested Functions

Anonymous Function Syntax

Currying

Automatic Type-Dependent Closure Construction

Operators

Higher-order Functions

Pattern Matching

Polymorphic Methods

Regular Expression Patterns

Traits

Upper Type Bounds

Lower Type Bounds

Explicitly Typed Self References

Local Type Inference

Unified Types

Variances

CHAPTER 3

Cheatsheet

About

Thanks to Brendan O'Connor, this cheatsheet aims to be a quick reference of Scala syntactic constructions. Licensed by Brendan O'Connor under a CC-BY-SA 3.0 license.

Variables

variable `var x = 5`

constant

> **GOOD** `val x = 5`
> *BAD* `x = 6`

explicit type `var x: Double = 5`

Functions

define function

> **GOOD** `def f(x: Int) = { x*x }`
> *BAD* `def f(x: Int) { x*x }`[1]

define function

> **GOOD** `def f(x: Any) = println(x)`
> *BAD* `def f(x) = println(x)`[2]

type alias `type R = Double`

[1] Hidden error: without = it's a unit-returning procedure; causes havoc
[2] Syntax error: need types for every arg

call-by-value `def f(x: R)`

call-by-name `def f(x: => R)`

anonymous function `(x:R) => x*x`

anonymous function: underscore is positionally matched arg

```
(1 to 5).map(_*2)
(1 to 5).reduceLeft( _+_ )
```

anonymous function: to use an arg twice, have to name it `(1 to 5).map( x => x*x )`

anonymous function: bound infix method.

GOOD `(1 to 5).map(2*)`
BAD `(1 to 5).map(*2)`[3]

anonymous function: block style returns last expression `(1 to 5).map { val x=_*2; println(x); x }`

anonymous functions: pipeline style. (or parens too) `(1 to 5) filter {_%2 == 0} map {_*2}`

anonymous functions: to pass in multiple blocks, need outer parens

```
def compose(g:R=>R, h:R=>R) = (x:R) => g(h(x))
val f = compose({_*2}, {_-1})
```

currying, obvious syntax `val zscore = (mean:R, sd:R) => (x:R) => (x-mean)/sd`

currying, obvious syntax `val zscore = (mean:R, sd:R) => (x:R) => (x-mean)/sd`

currying, sugar syntax

`def zscore(mean:R, sd:R)(x:R) = (x-mean)/sd`
`val normer = zscore(7, 0.4)_` *need trailing underscore to get the partial, only for the sugar version*

generic type `def mapmake[T](g:T=>T)(seq: List[T]) = seq.map(g)`

infix sugar

```
5.+(3); 5 + 3
(1 to 5) map (_*2)
```

varargs `def sum(args: Int*) = args.reduceLeft(_+_)`

Packages

wildcard import `import scala.collection._`

selective import

```
import scala.collection.Vector
import scala.collection.{Vector, Sequence}
```

renaming import `import scala.collection.{Vector => Vec28}`

import all from `java.util` except `Date` `import java.util.{Date => _, _}`

declare a package

`package pkg` *at start of file*

[3] Use `2*` for sanity's sake instead

```
package pkg { ... }
```

Data Structures

tuple literal `(1,2,3)`

destructing bind: tuple unpacking via pattern matching

> **GOOD** `var (x,y,z) = (1,2,3)`
>
> *BAD* `var x,y,z = (1,2,3)`[4]

list (immutable) `var xs = List(1,2,3)`

paren indexing (slides) `xs(2)`

cons `1 :: List(2,3)`

range sugar

> `1 to 5` same as `1 until 6`
>
> `1 to 10 by 2`

sole member of the unit type (like C/Java void) `()` (*empty parens*)

Control Constructs

conditional `if (check) happy else sad`

conditional sugar `if (check) happy` *same as* `if (check) happy else ()`

while loop `while (x < 5) { println(x); x += 1}`

do while loop `do { println(x); x += 1} while (x < 5)`

break (slides)

```
import scala.util.control.Breaks._
breakable {
    for (x <- xs) {
        if (Math.random < 0.1) break
    }
}
```

for comprehension: filter/map `for (x <- xs if x%2 == 0) yield x*10` same as `xs.filter(_%2 == 0).map(_*10)`

for comprehension: destructing bind `for ((x,y) <- xs zip ys) yield x*y` same as `(xs zip ys) map { case (x,y) => x*y }`

for comprehension: cross product `for (x <- xs; y <- ys) yield x*y` same as `xs flatMap {x => ys map {y => x*y}}`

for comprehension: imperative-ish sprintf-style

```
for (x <- xs; y <- ys) {
    println("%d/%d = %.1f".format(x,y, x*y))
}
```

[4] Hidden error: each assigned to the entire tuple

for comprehension: iterate including the upper bound

```
for (i <- 1 to 5) {
    println(i)
}
```

for comprehension: iterate omitting the upper bound

```
for (i <- 1 until 5) {
    println(i)
}
```

Pattern Matching

use case in function args for pattern matching

GOOD `(xs zip ys) map { case (x,y) => x*y }`

BAD `(xs zip ys) map( (x,y) => x*y )`

`v42` is interpreted as a name matching any Int value, and "42" is printed *BAD*

```
val v42 = 42
Some(3) match {
    case Some(v42) => println("42")
    case _ => println("Not 42")
}
```

`` `v42` `` with backticks is interpreted as the existing val `v42`, and "Not 42" is printed GOOD

```
val v42 = 42
Some(3) match {
    case Some(`v42`) => println("42")
    case _ => println("Not 42")
}
```

`UppercaseVal` is treated as an existing val, rather than a new pattern variable, because it starts with an uppercase letter. Thu

GOOD

```
val UppercaseVal = 42
    Some(3) match {
    case Some(UppercaseVal) => println("42")
    case _ => println("Not 42")
}
```

Object Orientation

constructor params - private

`class C(x: R)` same as `class C(private val x: R)`

`var c = new C(4)`

constructor params - public

`class C(val x: R)`

```
var c = new C(4)
c.x
```

constructor is class body, declare a public member, declare a gettable but not settable member, declare a private member, altern

```
class C(var x: R) {
    assert(x > 0, "positive please")
    var y = x
    val readonly = 5
    private var secret = 1
    def this = this(42)
}
```

anonymous class `new{ ... }`

define an abstract class. (non-createable) `abstract class D { ... }`

define an inherited class `class C extends D { ... }`

inheritance and constructor params. (wishlist: automatically pass-up params by default)

```
class D(var x:  R)
class C(x:  R) extends D(x)
```

define a singleton. (module-like) `object O extends D { ... }`

traits[5]

```
trait T { ... }
class C extends T { ... }
class C extends D with T { ... }
```

multiple traits

```
trait T1; trait T2
class C extends T1 with T2
class C extends D with T1 with T2
```

must declare method overrides `class C extends D { override def f = ...}`

create object `new java.io.File("f")`

callable factory shadowing the type

GOOD `List(1,2,3)`
BAD `new List[Int]`[6]

class literal `classOf[String]`

type check (runtime) `x.isInstanceOf[String]`

type cast (runtime) `x.asInstanceOf[String]`

ascription (compile time) `x:  String`

[5] Interfaces-with-implementation. no constructor params. mixin-able
[6] Type error: abstract type

CHAPTER 4

Style Guide

Contents:

Overview

This document is intended to outline some basic Scala stylistic guidelines which should be followed with more or less fervency. Wherever possible, this guide attempts to detail why a particular style is encouraged and how it relates to other alternatives. As with all style guides, treat this document as a list of rules to be broken. There are certainly times when alternative styles should be preferred over the ones given here.

Indentation

Indentation should follow the "2-space convention." Thus, instead of indenting like this:

```
// wrong!
class Foo {
    def bar = ...
}
```

You should indent like this:

```
// right!
class Foo {
  def bar = ..
}
```

The Scala language encourages a startling amount of nested scopes and logical blocks (function values and such). Do yourself a favor and don't penalize yourself syntactically for opening up a new block. Coming from Java, this style does take a bit of getting used to, but it is well worth the effort.

Line Wrapping

There are times when a single expression reaches a length where it becomes unreadable to keep it confined to a single line (usually that length is anywhere above 80 characters). In such cases, the *preferred* approach is to simply split the expression up into multiple expressions by assigning intermediate results to values. However, this is not always a practical solution.

When it is absolutely necessary to wrap an expression across more than one line, each successive line should be indented two spaces from the first. Also remember that Scala requires each "wrap line" to either have an unclosed parenthetical or to end with an infix method in which the right parameter is not given:

```
val result = 1 + 2 + 3 + 4 + 5 + 6 +
  7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 +
  15 + 16 + 17 + 18 + 19 + 20
```

Without this trailing method, Scala will infer a semi-colon at the end of a line which was intended to wrap, throwing off the compilation sometimes without even so much as a warning.

Methods with Numerous Arguments

When calling a method which takes numerous arguments (in the range of five or more), it is often necessary to wrap the method invocation onto multiple lines. In such cases, put all arguments on a line by themselves, indented two spaces from the current indent level:

```
foo(
  someVeryLongFieldName,
  andAnotherVeryLongFieldName,
  "this is a string",
  3.1415)
```

This way, all parameters line up, but you don't need to re-align them if you change the name of the method later on.

Great care should be taken to avoid these sorts of invocations well into the length of the line. More specifically, such an invocation should be avoided when each parameter would have to be indented more than 50 spaces to achieve alignment. In such cases, the invocation itself should be moved to the next line and indented two spaces:

```
// right!
val myOnerousAndLongFieldNameWithNoRealPoint =
  foo(
    someVeryLongFieldName,
    andAnotherVeryLongFieldName,
    "this is a string",
    3.1415)
// wrong!
val myOnerousAndLongFieldNameWithNoRealPoint = foo(someVeryLongFieldName,
                                                    andAnotherVeryLongFieldName,
                                                    "this is a string",
                                                    3.1415)
```

Better yet, just try to avoid any method which takes more than two or three parameters!

Naming Conventions

Generally speaking, Scala uses "camelCase" naming conventions. That is, each word (except possibly the first) is delimited by capitalizing its first letter. Underscores (_) are heavily discouraged as they have special meaning within the Scala syntax. Please note that there are a few important exceptions to this guideline (as given below).

Classes/Traits

Classes should be named in the camelCase style with the very first letter of the name capitalized:

```
class MyFairLady
```

This mimics the Java naming convention for classes.

Objects

Objects follow the class naming convention (camelCase with a capital first letter) except when attempting to mimic a package or a function. These situations don't happen often, but can be expected in general development:

```
object ast {
  sealed trait Expr
  case class Plus(e1: Expr, e2: Expr) extends Expr
  ...
}
object inc {
  def apply(x: Int): Int = x + 1
}
```

In *all* other cases, objects should be named according to the class naming convention.

Packages

Scala packages should follow the Java package naming conventions:

```
// wrong!
package coolness

// right!
package com.novell.coolness

// right, for package object com.novell.coolness
package com.novell
/**
 * Provides classes related to coolness
 */
package object coolness {
}
```

Versions Prior to 2.8

Scala 2.8 changes how packages worked. For 2.7 and earlier, please note that this convention does occasionally lead to problems when combined with Scala's nested packages feature. For example:

```
import net.liftweb._
```

This import will actually fail to resolve in some contexts as the net package may refer to the java.net package (or similar). To compensate for this, it is often necessary to fully-qualify imports using the `_root_` directive, overriding any nested package resolves:

```
import _root_.net.liftweb._
```

Do not overuse this directive. In general, nested package resolves are a good thing and very helpful in reducing import clutter. Using `_root_` not only negates their benefit, but also introduces extra clutter in and of itself.

Methods

Textual (alphabetic) names for methods should be in the camelCase style with the first letter lower-case:

```
def myFairMethod = ...
```

This section is not a comprehensive guide to idiomatic methods in Scala. Further information may be found in the method invocation section.

Accessors/Mutators

Scala does not follow the Java convention of prepending set/get to mutator and accessor methods (respectively). Instead, the following conventions are used:

- For accessors of properties, the name of the method should be the name of the property.
- In some instances, it is acceptable to prepend `is` on a boolean accessor (e.g. `isEmpty`). This should only be the case when no corresponding mutator is provided. Please note that the Lift convention of appending _? to boolean accessors is non-standard and not used outside of the Lift framework.
- For mutators, the name of the method should be the name of the property with _= appended. As long as a corresponding accessor with that particular property name is defined on the enclosing type, this convention will enable a call-site mutation syntax which mirrors assignment. Note that this is not just a convention but a requirement of the language.

  ```
  class Foo {
    def bar = ...

    def bar_=(bar: Bar) {
       ...
    }
    def isBaz = ...
   }

  val foo = new Foo
  foo.bar             // accessor
  foo.bar = bar2      // mutator
  foo.isBaz           // boolean property
  ```

Quite unfortunately, these conventions fall afoul of the Java convention to name the private fields encapsulated by accessors and mutators according to the property they represent. For example:

```
public class Company {
  private String name;

  public String getName() {
    return name;
```

```
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

In Scala, there is no distinction between fields and methods. In fact, fields are completely named and controlled by the compiler. If we wanted to adopt the Java convention of bean getters/setters in Scala, this is a rather simple encoding:

```
class Company {
  private var _name: String = _

  def name = _name

  def name_=(name: String) {
    _name = name
  }
}
```

While Hungarian notation is terribly ugly, it does have the advantage of disambiguating the `_name` variable without cluttering the identifier. The underscore is in the prefix position rather than the suffix to avoid any danger of mistakenly typing name _ instead of `name_`. With heavy use of Scala's type inference, such a mistake could potentially lead to a very confusing error.

Note that the Java getter/setter paradigm was often used to work around a lack of first class support for Properties and bindings. In Scala, there are libraries that support properties and bindings. The convention is to use an immutable reference to a property class that contains its own getter and setter. For example:

```
class Company {
  val string: Property[String] = Property("Initial Value")
```

Parentheses

Unlike Ruby, Scala attaches significance to whether or not a method is declared with parentheses (only applicable to methods of arity-0). For example:

```
def foo1() = ...

def foo2 = ...
```

These are different methods at compile-time. While `foo1` can be called with or without the parentheses, `foo2` *may not* be called with parentheses.

Thus, it is actually quite important that proper guidelines be observed regarding when it is appropriate to declare a method without parentheses and when it is not.

Methods which act as accessors of any sort (either encapsulating a field or a logical property) should be declared *without* parentheses except if they have side effects. While Ruby and Lift use a `!` to indicate this, the usage of parens is preferred (please note that fluid APIs and internal domain-specific languages have a tendency to break the guidelines given below for the sake of syntax. Such exceptions should not be considered a violation so much as a time when these rules do not apply. In a DSL, syntax should be paramount over convention).

Further, the callsite should follow the declaration; if declared with parentheses, call with parentheses. While there is temptation to save a few characters, if you follow this guideline, your code will be *much* more readable and maintainable.

```
// doesn't change state, call as birthdate
def birthdate = firstName

// updates our internal state, call as age()
def age() = {
  _age = updateAge(birthdate)
  _age
}
```

Symbolic Method Names

Avoid! Despite the degree to which Scala facilitates this area of API design, the definition of methods with symbolic names should not be undertaken lightly, particularly when the symbols itself are non-standard (for example, >>#>>). As a general rule, symbolic method names have two valid use-cases:

- Domain-specific languages (e.g. `actor1 ! Msg`)
- Logically mathematical operations (e.g. `a + b` or `c :: d`)

In the former case, symbolic method names may be used with impunity so long as the syntax is actually beneficial. However, in the course of standard API design, symbolic method names should be strictly reserved for purely-functional operations. Thus, it is acceptable to define a >>= method for joining two monads, but it is not acceptable to define a << method for writing to an output stream. The former is mathematically well-defined and side-effect free, while the latter is neither of these.

As a general rule, symbolic method names should be well-understood and self documenting in nature. The rule of thumb is as follows: if you need to explain what the method does, then it should have a real, descriptive name rather than a symbols. There are some very rare cases where it is acceptable to invent new symbolic method names. Odds are, your API is not one of those cases!

The definition of methods with symbolic names should be considered an advanced feature in Scala, to be used only by those most well-versed in its pitfalls. Without care, excessive use of symbolic method names can easily transform even the simplest code into symbolic soup.

Constants, Values, Variables, and Methods

Constant names should be in upper camel case. That is, if the member is final, immutable and it belongs to a package object or an object, it may be considered a constant (similar to Java's `static final` members):

```
object Container {
    val MyConstant = ...
}
```

The value: `Pi` in `scala.math` package is another example of such a constant.

Method, Value and variable names should be in lower camel case:

```
val myValue = ...
def myMethod = ...
var myVariable
```

Type Parameters (Generics)

For simple type parameters, a single upper-case letter (from the English alphabet) should be used, starting with `A` (this is different than the Java convention of starting with `T`). For example:

```
class List[A] {
  def map[B](f: A => B): List[B] = ...
}
```

If the type parameter has a more specific meaning, a descriptive name should be used, following the class naming conventions (as opposed to an all-uppercase style):

```
// Right
class Map[Key, Value] {
  def get(key: Key): Value
  def put(key: Key, value: Value): Unit
}

// Wrong; don't use all-caps
class Map[KEY, VALUE] {
  def get(key: KEY): VALUE
  def put(key: KEY, value: VALUE): Unit
}
```

If the scope of the type parameter is small enough, a mnemonic can be used in place of a longer, descriptive name:

```
class Map[K, V] {
  def get(key: K): V
  def put(key: K, value: V): Unit
}
```

Higher-Kinds and Parameterized Type Parameters

Higher-kinds are theoretically no different from regular type parameters (except that their kind is at least `*=>*` rather than simply `*`). The naming conventions are generally similar, however it is preferred to use a descriptive name rather than a single letter, for clarity:

```
class HigherOrderMap[Key[_], Value[_]] { ... }
```

The single letter form is (sometimes) acceptable for fundamental concepts used throughout a codebase, such as `F[_]` for Functor and `M[_]` for Monad.

In such cases, the fundamental concept should be something well known and understood to the team, or have tertiary evidence, such as the following:

```
def doSomething[M[_]: Monad](m: M[Int]) = ...
```

Here, the type bound `: Monad` offers the necessary evidence to inform the reader that `M[_]` is the type of the Monad.

Annotations

Annotations, such as `@volatile` should be in camel-case, with the first letter being lower case:

```
class cloneable extends StaticAnnotation
```

This convention is used throughout the Scala library, even though it is not consistent with Java annotations.

Note: This convention applied even when using type aliases on annotations. For example, when using JDBC:

```
type id = javax.persistence.Id @annotation.target.field
@id
var id: Int = 0
```

Special Note on Brevity

Because of Scala's roots in the functional languages, it is quite normal for local field names to be extremely brief:

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

While this would be bad practice in languages like Java, it is *good* practice in Scala. This convention works because properly-written Scala methods are quite short, only spanning a single expression and rarely going beyond a few lines. Very few local fields are ever used (including parameters), and so there is no need to contrive long, descriptive names. This convention substantially improves the brevity of most Scala sources. This in turn improves readability, as most expressions fit in one line and the arguments to methods have descriptive type names.

This convention only applies to parameters of very simple methods (and local fields for very simply classes); everything
in the public interface should be descriptive. Also note that the names of arguments are now part of the public API of a class, since users can use named parameters in method calls.

Types

Inference

Use type inference where possible, but put clarity first, and favour explicitness in public APIs.

You should almost never annotate the type of a private field or a local variable, as their type will usually be immediately evident in their value:

```
private val name = "Daniel"
```

However, you may wish to still display the type where the assigned value has a complex or non-obvious form.

All public methods should have explicit type annotations. Type inference may break encapsulation in these cases, because it depends on internal method and class details. Without an explicit type, a change to the internals of a method or val could alter the public API of the class without warning, potentially breaking client code. Explicit type annotations can also help to improve compile times.

Function Values

Function values support a special case of type inference which is worth calling out on its own:

```
val ls: List[String] = ...
ls map (str => str.toInt)
```

In cases where Scala already knows the type of the function value we are declaring, there is no need to annotate the parameters (in this case, `str`). This is an intensely helpful inference and should be preferred whenever possible. Note that implicit conversions which operate on function values will nullify this inference, forcing the explicit annotation of parameter types.

Annotations

Type annotations should be patterned according to the following template:

```
value: Type
```

This is the style adopted by most of the Scala standard library and all of Martin Odersky's examples. The space between value and type helps the eye in accurately parsing the syntax. The reason to place the colon at the end of the value rather than the beginning of the type is to avoid confusion in cases such as this one:

```
value :::
```

This is actually valid Scala, declaring a value to be of type `::`. Obviously, the prefix-style annotation colon muddles things greatly.

Ascription

Type ascription is often confused with type annotation, as the syntax in Scala is identical. The following are examples of ascription:

- `Nil: List[String]`
- `Set(values: _*)`
- `"Daniel": AnyRef`

Ascription is basically just an up-cast performed at compile-time for the sake of the type checker. Its use is not common, but it does happen on occasion. The most often seen case of ascription is invoking a varargs method with a single `Seq` parameter. This is done by ascribing the `_*` type (as in the second example above).

Ascription follows the type annotation conventions; a space follows the colon.

Functions

Function types should be declared with a space between the parameter type, the arrow and the return type:

```
def foo(f: Int => String) = ...
def bar(f: (Boolean, Double) => List[String]) = ...
```

Parentheses should be omitted wherever possible (e.g. methods of arity-1, such as `Int => String`).

Arity-1

Scala has a special syntax for declaring types for functions of arity-1. For example:

```
def map[B](f: A => B) = ...
```

Specifically, the parentheses may be omitted from the parameter type. Thus, we did *not* declare `f` to be of type `(A) => B`, as this would have been needlessly verbose. Consider the more extreme example:

```
// wrong!
def foo(f: (Int) => (String) => (Boolean) => Double) = ...

// right!
def foo(f: Int => String => Boolean => Double) = ...
```

By omitting the parentheses, we have saved six whole characters and dramatically improved the readability of the type expression.

Structural Types

Structural types should be declared on a single line if they are less than 50 characters in length. Otherwise, they should be split across multiple lines and (usually) assigned to their own type alias:

```
// wrong!
def foo(a: { def bar(a: Int, b: Int): String; val baz: List[String => String] }) = ...

// right!
private type FooParam = {
  val baz: List[String => String]
  def bar(a: Int, b: Int): String
}

def foo(a: FooParam) = ...
```

Simpler structural types (under 50 characters) may be declared and used inline:

```
def foo(a: { val bar: String }) = ...
```

When declaring structural types inline, each member should be separated by a semi-colon and a single space, the opening brace should be *followed* by a space while the closing brace should be *preceded* by a space (as demonstrated in both examples above).

Structural types are implemented with reflection at runtime, and are inherently less performant than nominal types. Developers should prefer the use of nominal types, unless structural types provide a clear benefit.

Nested Blocks

Curly Braces

Opening curly braces ({) must be on the same line as the declaration they represent:

```
def foo = {
  ...
}
```

Technically, Scala's parser does support GNU-style notation with opening braces on the line following the declaration. However, the parser is not terribly predictable when dealing with this style due to the way in which semi-colon inference is implemented. Many headaches will be saved by simply following the curly brace convention demonstrated above.

Parentheses

In the rare cases when parenthetical blocks wrap across lines, the opening and closing parentheses should be unspaced and generally kept on the same lines as their content (Lisp-style):

```
(this + is a very ++ long *
  expression)
```

Parentheses also serve to disable semicolon inference, and so allow the developer to start lines with operators, which some prefer:

```
(   someCondition
||  someOtherCondition
||  thirdCondition
)
```

A trailing parenthesis on the following line is acceptable in this case, for aesthetic reasons.

Declarations

Classes

Class/Object/Trait constructors should be declared all on one line, unless the line becomes "too long" (about 100 characters). In that case, put each constructor argument on its own line, indented four spaces:

```
class Person(name: String, age: Int) {
}
class Person(
    name: String,
    age: Int,
    birthdate: Date,
    astrologicalSign: String,
    shoeSize: Int,
    favoriteColor: java.awt.Color) {
  def firstMethod: Foo = ...
}
```

If a class/object/trait extends anything, the same general rule applies, put it one one line unless it goes over about 100 characters, and then indent four spaces with each item being on its own line and two spaces for extensions; this provides visual separation between constructor arguments and extensions.:

```
class Person(
    name: String,
    age: Int,
    birthdate: Date,
    astrologicalSign: String,
    shoeSize: Int,
    favoriteColor: java.awt.Color)
  extends Entity
  with Logging
  with Identifiable
  with Serializable {
}
```

Ordering of Class Elements

All class/object/trait members should be declared interleaved with newlines. The only exceptions to this rule are `var` and `val`. These may be declared without the intervening newline, but only if none of the fields have ScalaDoc and if all of the fields have simple (max of 20-ish chars, one line) definitions:

```
class Foo {
  val bar = 42
  val baz = "Daniel"
  def doSomething(): Unit = { ... }
  def add(x: Int, y: Int): Int = x + y
}
```

Fields should *precede* methods in a scope. The only exception is if the `val` has a block definition (more than one expression) and performs operations which may be deemed "method-like" (e.g. computing the length of a `List`). In such cases, the non-trivial `val` may be declared at a later point in the file as logical member ordering would dictate. This rule only applies to `val` and `lazy val`! It becomes very difficult to track changing aliases if `var` declarations are strewn throughout class file.

Methods

Methods should be declared according to the following pattern:

```
def foo(bar: Baz): Bin = expr
```

Methods with default parameter values should be declared in an analogous fashion, with a space on either side of the equals sign:

```
def foo(x: Int = 6, y: Int = 7): Int = x + y
```

You should specify a return type for all public members. Consider it documentation checked by the compiler. It also helps in preserving binary compatibility in the face of changing type inference (changes to the method implementation may propagate to the return type if it is inferred).

Local methods or private methods may omit their return type:

```
private def foo(x: Int = 6, y: Int = 7) = x + y
```

Procedure Syntax

Avoid the procedure syntax, as it tends to be confusing for very little gain in brevity.

```
// don't do this
def printBar(bar: Baz) {
  println(bar)
}

// write this instead
def printBar(bar: Bar): Unit = {
  println(bar)
}
```

Modifiers

Method modifiers should be given in the following order (when each is applicable):

- Annotations, each on their own line
- Override modifier (`override`)
- Access modifier (`protected`, `private`)

- Final modifier (`final`)
- `def`

```
@Transaction
@throws(classOf[IOException])
override protected final def foo() {
  ...
}
```

Body

When a method body comprises a single expression which is less than 30 (or so) characters, it should be given on a single line with the method:

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

When the method body is a single expression *longer* than 30 (or so) characters but still shorter than 70 (or so) characters, it should be given on the following line, indented two spaces:

```
def sum(ls: List[String]): Int =
  ls.map(_.toInt).foldLeft(0)(_ + _)
```

The distinction between these two cases is somewhat artificial. Generally speaking, you should choose whichever style is more readable on a case-by-case basis. For example, your method declaration may be very long, while the expression body may be quite short. In such a case, it may be more readable to put the expression on the next line rather than making the declaration line too long.

When the body of a method cannot be concisely expressed in a single line or is of a non-functional nature (some mutable state, local or otherwise), the body must be enclosed in braces:

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

Methods which contain a single match expression should be declared in the following way:

```
// right!
def sum(ls: List[Int]): Int = ls match {
  case hd :: tail => hd + sum(tail)
  case Nil => 0
}
```

Not like this:

```
// wrong!
def sum(ls: List[Int]): Int = {
  ls match {
    case hd :: tail => hd + sum(tail)
    case Nil => 0
  }
}
```

Multiple Parameter Lists

In general, you should only use multiple parameter lists if there is a good reason to do so. These methods (or similarly declared functions) have a more verbose declaration and invocation syntax and are harder for less-experienced Scala developers to understand.

There are three main reasons you should do this:

- For a fluent API

  Multiple parameter lists allow you to create your own "control structures":

```
def unless(exp: Boolean)(code: => Unit): Unit = if (!exp) code
  unless(x < 5) {
    println("x was not less than five")
  }
```

- Implicit Parameters

  When using implicit parameters, and you use the `implicit` keyword, it applies to the entire parameter list. Thus, if you want only some parameters to be implicit, you must use multiple parameter lists.

- For type inference

  When invoking a method using only some of the parameter lists, the type inferencer can allow a simpler syntax when invoking the remaining parameter lists. Consider fold:

```
def foldLeft[B](z: B)(op: (A,B) => B): B
List("").foldLeft(0)(_ + _.length)

// If, instead:
def foldLeft[B](z: B, op: (B, A) => B): B
// above won't work, you must specify types
List("").foldLeft(0, (b: Int, a: String) => a + b.length)
List("").foldLeft[Int](0, _ + _.length)
```

For complex DSLs, or with type-names that are long, it can be difficult to fit the entire signature on one line. In those cases, alight the open-paren of the parameter lists, one list per line (i.e. if you can't put them all on one line, put one each per line):

```
protected def forResource(resourceInfo: Any)
                         (f: (JsonNode) => Any)
                         (implicit urlCreator: URLCreator, configurer:␣
↪OAuthConfiguration): Any = {
  ...
}
```

Higher-Order Functions

It's worth keeping in mind when declaring higher-order functions the fact that Scala allows a somewhat nicer syntax for such functions at call-site when the function parameter is curried as the last argument. For example, this is the `foldl` function in SML:

```
fun foldl (f: ('b * 'a) -> 'b) (init: 'b) (ls: 'a list) = ...
```

In Scala, the preferred style is the exact inverse:

```
def foldLeft[A, B](ls: List[A])(init: B)(f: (B, A) => B): B = ...
```

By placing the function parameter *last*, we have enabled invocation syntax like the following:

```
foldLeft(List(1, 2, 3, 4))(0)(_ + _)
```

The function value in this invocation is not wrapped in parentheses; it is syntactically quite disconnected from the function itself (`foldLeft`). This style is preferred for its brevity and cleanliness.

Fields

Fields should follow the declaration rules for methods, taking special note of access modifier ordering and annotation conventions.

Lazy vals should use the `lazy` keyword directly before the `val`:

```
private lazy val foo = bar()
```

Function Values

Scala provides a number of different syntactic options for declaring function values. For example, the following declarations are exactly equivalent:

- `val f1 = ((a: Int, b: Int) => a + b)`
- `val f2 = (a: Int, b: Int) => a + b`
- `val f3 = (_: Int) + (_: Int)`
- `val f4: (Int, Int) => Int = (_ + _)`

Of these styles, (1) and (4) are to be preferred at all times. (2) appears shorter in this example, but whenever the function value spans multiple lines (as is normally the case), this syntax becomes extremely unwieldy. Similarly, (3) is concise, but obtuse. It is difficult for the untrained eye to decipher the fact that this is even producing a function value.

When styles (1) and (4) are used exclusively, it becomes very easy to distinguish places in the source code where function values are used. Both styles make use of parentheses, since they look clean on a single line.

Spacing

There should be no space between parentheses and the code they contain.

Curly braces should be separated from the code within them by a one-space gap, to give the visually busy braces "breathing room".

Multi-Expression Functions

Most function values are less trivial than the examples given above. Many contain more than one expression. In such cases, it is often more readable to split the function value across multiple lines. When this happens, only style (1) should be used, substituting braces for parentheses. Style (4) becomes extremely difficult to follow when enclosed in large amounts of code. The declaration itself should loosely follow the declaration style for methods, with the opening brace on the same line as the assignment or invocation, while the closing brace is on its own line immediately following the last line of the function. Parameters should be on the same line as the opening brace, as should the "arrow" (=>):

```
val f1 = { (a: Int, b: Int) =>
  val sum = a + b
  sum
}
```

As noted earlier, function values should leverage type inference whenever possible.

Control Structures

All control structures should be written with a space following the defining keyword:

```
// right!
if (foo) bar else baz
for (i <- 0 to 10) { ... }
while (true) { println("Hello, World!") }

// wrong!
if(foo) bar else baz
for(i <- 0 to 10) { ... }
while(true) { println("Hello, World!") }
```

Curly-Braces

Curly-braces should be omitted in cases where the control structure represents a pure-functional operation and all branches of the control structure (relevant to if/else) are single-line expressions. Remember the following guidelines:

- `if` - Omit braces if you have an else clause. Otherwise, surround the contents with curly braces even if the contents are only a single line.
- `while` - Never omit braces (while cannot be used in a pure-functional manner).
- `for` - Omit braces if you have a yield clause. Otherwise, surround the contents with curly-braces, even if the contents are only a single line.
- `case` - Omit braces if the case expression fits on a single line. Otherwise, use curly braces for clarity (even though they are not required by the parser).

```
val news = if (foo)
  goodNews()
else
  badNews()

if (foo) {
  println("foo was true")
}

news match {
  case "good" => println("Good news!")
  case "bad" => println("Bad news!")
}
```

Comprehensions

Scala has the ability to represent for-comprehensions with more than one generator (usually, more than one `<-` symbol). In such cases, there are two alternative syntaxes which may be used:

```
// wrong!
for (x <- board.rows; y <- board.files)
  yield (x, y)

// right!
for {
  x <- board.rows
  y <- board.files
} yield (x, y)
```

While the latter style is more verbose, it is generally considered easier to read and more "scalable" (meaning that it does not become obfuscated as the complexity of the comprehension increases). You should prefer this form for all `for`-comprehensions of more than one generator. Comprehensions with only a single generator (e.g. `for (i <- 0 to 10) yield i`) should use the first form (parentheses rather than curly braces).

The exceptions to this rule are `for`-comprehensions which lack a `yield` clause. In such cases, the construct is actually a loop rather than a functional comprehension and it is usually more readable to string the generators together between parentheses rather than using the syntactically-confusing `} {` construct:

```
// wrong!
for {
  x <- board.rows
  y <- board.files
} {
  printf("(%d, %d)", x, y)
}

// right!
for (x <- board.rows; y <- board.files) {
  printf("(%d, %d)", x, y)
}
```

Finally, `for`-comprehensions are preferred to chained calls to `map`, `flatMap`, and `filter`, as this can get difficult to read (this is one of the purposes of the enhanced for comprehension).

Trivial Conditionals

There are certain situations where it is useful to create a short `if`/`else` expression for nested use within a larger expression. In Java, this sort of case would traditionally be handled by the ternary operator (`?`/`:`), a syntactic device which Scala lacks. In these situations (and really any time you have a extremely brief `if`/`else` expression) it is permissible to place the `then` and `else` branches on the same line as the `if` and `else` keywords:

```
val res = if (foo) bar else baz
```

The key here is that readability is not hindered by moving both branches inline with the `if`/`else`. Note that this style should never be used with imperative `if` expressions nor should curly braces be employed.

Method Invocation

Generally speaking, method invocation in Scala follows Java conventions. In other words, there should not be a space between the invocation target and the dot (`.`), nor a space between the dot and the method name, nor should there be any space between the method name and the argument-delimiters (parentheses). Each argument should be separated by a single space following the comma (`,`):

```
foo(42, bar)
target.foo(42, bar)
target.foo()
```

As of version 2.8, Scala now has support for named parameters. Named parameters in a method invocation should be treated as regular parameters (spaced accordingly following the comma) with a space on either side of the equals sign:

```
foo(x = 6, y = 7)
```

While this style does create visual ambiguity with named parameters and variable assignment, the alternative (no spacing around the equals sign) results in code which can be very difficult to read, particularly for non-trivial expressions for the actuals.

Arity-0

Scala allows the omission of parentheses on methods of arity-0 (no arguments):

```
reply()

// is the same as

reply
```

However, this syntax should only be used when the method in question has no side-effects (purely-functional). In other words, it would be acceptable to omit parentheses when calling `queue.size`, but not when calling `println()`. This convention mirrors the method declaration convention given above.

Religiously observing this convention will dramatically improve code readability and will make it much easier to understand at a glance the most basic operation of any given method. Resist the urge to omit parentheses simply to save two characters!

Suffix Notatoin

Scala allows methods of arity-0 to be invoked using suffix notation:

```
names.toList

// is the same as

names toList // Unsafe, don't use!
```

This style is unsafe, and should not be used. Since semicolons are optional, the compiler will attempt to treat it as an infix method if it can, potentially taking a term from the next line.

```
names toList
val answer = 42        // will not compile!
```

This may result in unexpected compile errors at best, and happily compiled faulty code at worst. Although the syntax is used by some DSLs, it should be considered deprecated, and avoided.

As of Scala 2.10, using suffix operator notation will result in a compiler warning.

Arity-1

Scala has a special syntax for invoking methods of arity-1 (one argument):

```
names.mkString(",")

// is the same as

names mkString ","
```

This syntax is formally known as "infix notation". It should only be used for purely-functional methods (methods with no side-effects) - such as `mkString` - or methods which take functions as parameters - such as `foreach`:

```
// right!
names foreach (n => println(n))
names mkString ","
optStr getOrElse "<empty>"

// wrong!
javaList add item
```

Higher-Order Functions

As noted, methods which take functions as parameters (such as `map` or `foreach`) should be invoked using infix notation. It is also *possible* to invoke such methods in the following way:

```
names.map (_.toUpperCase)     // wrong!
```

This style *is not* the accepted standard! The reason to avoid this style is for situations where more than one invocation must be chained together:

```
// wrong!
names.map (_.toUpperCase).filter (_.length > 5)

// right!
names map (_.toUpperCase) filter (_.length > 5)
```

Both of these work, but the former exploits an extremely unintuitive wrinkle in Scala's grammar. The sub-expression `(_.toUpperCase).filter` when taken in isolation looks for all the world like we are invoking the filter method on a function value. However, we are actually invoking filter on the result of the `map` method, which takes the function value as a parameter. This syntax is confusing and often discouraged in Ruby, but it is shunned outright in Scala.

Symbolic Methods / Operators

Methods with symbolic names should always be invoked using infix notation with spaces separating the target, the symbolic method and the parameter:

```
// right!
"daniel" + " " + "Spiewak"

// wrong!
"daniel"+" "+"spiewak"
```

For the most part, this idiom follows Java and Haskell syntactic conventions.

Symbolic methods which take more than one parameter (they do exist!) should still be invoked using infix notation, delimited by spaces:

```
foo ** (bar, baz)
```

Such methods are fairly rare, however, and should be avoided during API design.

Finally, the use of the `/:` and `:\` should be avoided in preference to the more explicit `foldLeft` and `foldRight` method of `Iterator`. The right-associativity of the `/:` can lead to extremely confusing code, at the benefit of saving a few characters.

Files

As a rule, files should contain a single logical compilation unit. By "logical" I mean a class, trait or object. One exception to this guideline is for classes or traits which have companion objects. Companion objects should be grouped with their corresponding class or trait in the same file. These files should be named according to the class, trait or object they contain:

```
package com.novell.coolness

class Inbox { ... }

// companion object
object Inbox { ... }
```

These compilation units should be placed within a file named `Inbox.scala` within the `com/novell/coolness` directory. In short, the Java file naming and positioning conventions should be preferred, despite the fact that Scala allows for greater flexibility in this regard.

Multi-Unit Files

Despite what was said above, there are some important situations which warrant the inclusion of multiple compilation units within a single file. One common example is that of a sealed trait and several sub-classes (often emulating the ADT language feature available in functional languages):

```
sealed trait Option[+A]

case class Some[A](a: A) extends Option[A]

case object None extends Option[Nothing]
```

Because of the nature of sealed superclasses (and traits), all subtypes *must* be included in the same file. Thus, such a situation definitely qualifies as an instance where the preference for single-unit files should be ignored.

Another case is when multiple classes logically form a single, cohesive group, sharing concepts to the point where maintenance is greatly served by containing them within a single file. These situations are harder to predict than the aforementioned sealed supertype exception. Generally speaking, if it is *easier* to perform long-term maintenance and development on several units in a single file rather than spread across multiple, then such an organizational strategy should be preferred for these classes. However, keep in mind that when multiple units are contained within a single file, it is often more difficult to find specific units when it comes time to make changes.

All multi-unit files should be given camelCase names with a lower-case first letter. This is a very important convention. It differentiates multi- from single-unit files, greatly easing the process of finding declarations. These filenames may be based upon a significant type which they contain (e.g. `option.scala` for the example above), or may be descriptive of the logical property shared by all units within (e.g. `ast.scala`).

Scaladoc

It is important to provide documentation for all packages, classes, traits, methods, and other members. ScalaDoc generally follows the conventions of Javadoc, however there are many additional features to make writing scaladoc simpler.

In general, you want to worry more about substance and writing style than in formatting. ScalaDocs need to be useful to new users of the code as well as experienced users. Achieving this is very simple: increase the level of detail and explanation as you write, starting from a terse summary (useful for experienced users as reference), while providing deeper examples in the detailed sections (which can be ignored by experienced users, but can be invaluable for newcomers).

The general format for a ScalaDoc comment should be as follows:

```
/** This is a brief description of what's being documented.
  *
  * This is further documentation of what we're documenting.  It should
  * provide more details as to how this works and what it does.
  */
def myMethod = {}
```

For methods and other type members where the only documentation needed is a simple, short description, this format can be used:

```
/** Does something very simple */
def simple = {}
```

Note, especially for those coming from Java, that the left-hand margin of asterisks falls under the *third* column, not the second, as is customary in Java.

See the <AuthorDocs https://wiki.scala-lang.org/display/SW/Writing+Documentation>_ on the Scala wiki for more technical info on formatting ScalaDoc.

General Style

It is important to maintain a consistent style with ScalaDoc. It is also important to target ScalaDoc to both those unfamiliar with your code and experienced users who just need a quick reference. Here are some general guidelines:

- Get to the point as quickly as possible. For example, say "returns true if some condition" instead of "if some condition return true".
- Try to format the first sentence of a method as "Returns XXX", as in "Returns the first element of the List", as opposed to "this method returns" or "get the first" etc. Methods typically return things.
- This same goes for classes; omit "This class does XXX"; just say "Does XXX"

- Create links to referenced Scala Library classes using the square-bracket syntax, e.g. `[[scala.Option]]`
- Summarize a method's return value in the @return annotation, leaving a longer description for the main ScalaDoc.
- If the documentation of a method is a one line description of what that method returns, do not repeat it with an `@return` annotation.
- Document what the method does do not what the method should do. In other words, say "returns the result of applying f to x" rather than "return the result of applying f to x". Subtle, but important.
- When referring to the instance of the class, use "this XXX", or "this" and not "the XXX". For objects, say "this object".
- Make code examples consistent with this guide.
- Use the wiki-style syntax instead of HTML wherever possible.
- Examples should use either full code listings or the REPL, depending on what is needed (the simplest way to include REPL code is to develop the examples in the REPL and paste it into the ScalaDoc).
- Make liberal use of `@macro` to refer to commonly-repeated values that require special formatting.

Packages

Provide ScalaDoc for each package. This goes in a file named package.scala in your package's directory and looks like so (for the package `parent.package.name.mypackage`):

```
package parent.package.name
/** This is the ScalaDoc for the package. */
package object mypackage {
}
```

A package's documentation should first document what sorts of classes are part of the package. Secondly, document the general sorts of things the package object itself provides.

While package documentation doesn't need to be a full-blown tutorial on using the classes in the package, it should provide an overview of the major classes, with some basic examples of how to use the classes in that package. Be sure to reference classes using the square-bracket notation:

```
package my.package
/** Provides classes for dealing with complex numbers.  Also provides
  * implicits for converting to and from `Int`.
  *
  * ==Overview==
  * The main class to use is [[my.package.complex.Complex]], as so
  * {{{
  * scala> val complex = Complex(4,3)
  * complex: my.package.complex.Complex = 4 + 3i
  * }}}
  *
  * If you include [[my.package.complex.ComplexConversions]], you can
  * convert numbers more directly
  * {{{
  * scala> import my.package.complex.ComplexConversions._
  * scala> val complex = 4 + 3.i
  * complex: my.package.complex.Complex = 4 + 3i
  * }}}
  */
package complex {}
```

Classes, Objects, and Traits

Document all classes, objects, and traits. The first sentence of the ScalaDoc should provide a summary of what the class or trait does. Document all type parameters with `@tparam`.

Classes

If a class should be created using it's companion object, indicate as such after the description of the class (though leave the details of construction to the companion object). Unfortunately, there is currently no way to create a link to the companion object inline, however the generated ScalaDoc will create a link for you in the class documentation output.

If the class should be created using a constructor, document it using the `@constructor` syntax:

```
/** A person who uses our application.
  *
  * @constructor create a new person with a name and age.
  * @param name the person's name
  * @param age the person's age in years
  */
class Person(name: String, age: Int) {
}
```

Depending on the complexity of your class, provide an example of common usage.

Objects

Since objects can be used for a variety of purposes, it is important to document how to use the object (e.g. as a factory, for implicit methods). If this object is a factory for other objects, indicate as such here, deferring the specifics to the ScalaDoc for the `apply` method(s). If your object doesn't use apply as a factory method, be sure to indicate the actual method names:

```
/** Factory for [[mypackage.Person]] instances. */
object Person {
  /** Creates a person with a given name and age.
    *
    * @param name their name
    * @param age the age of the person to create
    */
  def apply(name: String, age: Int) = {}
  /** Creates a person with a given name and birthdate
    *
    * @param name their name
    * @param birthDate the person's birthdate
    * @return a new Person instance with the age determined by the
    *         birthdate and current date.
    */
  def apply(name: String, birthDate: java.util.Date) = {}
}
```

If your object holds implicit conversions, provide an example in the ScalaDoc:

```
/** Implicit conversions and helpers for [[mypackage.Complex]] instances.
  *
  * {{{
  * import ComplexImplicits._
  * val c: Complex = 4 + 3.i
  * }}}
  */
object ComplexImplicits {}
```

Traits

After the overview of what the trait does, provide an overview of the methods and types that must be specified in classes that mix in the trait. If there are known classes using the trait, reference them.

Methods and Other Members

Document all methods. As with other documentable entities, the first sentence should be a summary of what the method does. Subsequent sentences explain in further detail. Document each parameter as well as each type parameter (with `@tparam`). For curried functions, consider providing more detailed examples regarding the expected or idiomatic usage. For implicit parameters, take special care to explain where these parameters will come from and if the user needs to do any extra work to make sure the parameters will be available.

CHAPTER 5

Authors

Alphabetical by last name

I

- Oleg Ilyenko (The awesome icon)

O

- Brendan O'Connor (Cheatsheet)

CHAPTER 6

Indices and tables

- genindex
- modindex
- search