Scala Style Guide Documentation

Release 1.2.5

Scala Community

April 13, 2011

CONTENTS

In lieu of an official style guide from EPFL, or even an unofficial guide from a community site like Artima, 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.

Contents:

CHAPTER

ONE

OVERVIEW

Generally speaking, Scala seeks to mimic Java conventions to ease interoperability. When in doubt regarding the idiomatic way to express a particular concept, adopt conventions and idioms from the following languages (in this order):

- Java
- Standard ML
- Haskell
- C#
- OCaml
- Ruby
- Python

For example, you should use Java's naming conventions for classes and methods, but SML's conventions for type annotation, Haskell's conventions for type parameter naming (except upper-case rather than lower) and Ruby's conventions for non-boolean accessor methods. Scala really is a hybrid language!

CHAPTER

TWO

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.

2.1 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 or by using the pipeline operator. 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 binary function or operator 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 operator, 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.

2.2 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!

CHAPTER

THREE

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).

3.1 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.

3.2 Objects

Objects follow the class naming convention (camelCase with a capital first letter) except when attempting to mimic a package. This is a fairly rare case, but it does come up on occasion:

```
object ast {
  sealed trait Expr

  case class Plus(e1: Expr, e2: Expr) extends Expr
  ...
}
```

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

3.3 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 {
}
```

3.3.1 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.

3.4 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.

3.4.1 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 *most* boolean and non-boolean properties, the name of the method should be the name of the property
- For accessors of *some* boolean properties, the name of the method may be the capitalized name of the property with "`is`" prepended (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.

```
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;
    }
}
```

If we were to attempt to adopt this convention within Scala while observing the accessor naming conventions given above, the Scala compiler would complain about a naming collision between the `name` field and the `name` method. There are a number of ways to avoid this problem and the community has yet to standardize on any one of them. The following illustrates one of the less error-prone conventions:

```
class Company {
  private val _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` field 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 fields may actually be used in a number of situations where accessors and mutators would be required in languages like Java. Always prefer fields over methods when given the choice.

3.4.2 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 [1].

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
}
```

3.4.3 Operators

Avoid! Despite the degree to which Scala facilitates this area of API design, operator definition should not be undertaken lightly, particularly when the operator itself is non-standard (for example, >>#>>). As a general rule, operators 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, operators may be used with impunity so long as the syntax is actually beneficial. However, in the course of standard API design, operators should be strictly reserved for purely-functional operations. Thus, it is acceptable to define a >>= operator for joining two monads, but it is not acceptable to define a << operator for writing to an output stream. The former is mathematically well-defined and side-effect free, while the latter is neither of these.

Operator definition should be considered an advanced feature in Scala, to be used only by those most well-versed in its pitfalls. Without care, excessive operator overloading can easily transform even the simplest code into symbolic soup.

[1] 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.

3.5 Fields

Field names should be in camelCase with the first letter lower-case:

```
val myFairField = ...
```

3.6 Type Parameters (generics)

Type parameters are typically a single upper-case letter (from the English alphabet). Conventionally, parameters blindly start at `A` and ascend up to `Z` as necessary. This contrasts with the Java convention of using `T`, `K`, `V` and `E`. For example:

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

3.6.1 Higher-Kinds

While higher-kinds are theoretically no different from regular type parameters (except that their kind is at least `*=>*` rather than simply `*`), their naming conventions do differ somewhat. Generally, higher-kinded parameters are two upper-case characters, usually repeated. For example:

```
class HOMap[AA[_], BB[_]] { ... }
```

It is also (sometimes) acceptable to give full, descriptive names to higher-kinded parameters. In this case, use all-caps to make it clear you are not referring to a class or trait. Thus, the following would be an equally valid definition of `HOMap`:

```
class HOMap[KEY[_], VALUE[_]] { ... }
```

In such cases, the type naming conventions should be observed.

3.7 Type Aliases

Type aliases follow the same naming conventions as classes. For example:

```
type StringList = List[String]
```

3.8 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.

3.9 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 convention only applies to parameters of very simple methods (and local fields for very simply classes); everything in the public interface should be descriptive.

CHAPTER

FOUR

TYPES

4.1 Inference

Use type inference as much as possible. You should almost never annotate the type of a `val` field as their type will be immediately evident in their value:

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

However, type inference has a way of coming back to haunt you when used on non-trivial methods which are part of the public interface. Just for the sake of safety, you should annotate all public methods in your class.

4.1.1 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 =@textgreater[] 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.

4.1.2 "Void" Methods

The exception to the "annotate everything public" rule is methods which return `Unit`. *Any* method which returns `Unit` should be declared using Scala's syntactic sugar for that case:

```
def printName() {
  println("Novell")
}
```

This compiles into:

```
def printName(): Unit = {
  println("Novell")
}
```

You should prefer the former style (without the annotation or the equals sign) as it reduces errors and improves readability. For the record, it is also possible (and encouraged!) to declare abstract methods returning `Unit` with an analogous syntax:

```
def printName()          // abstract def for printName(): Unit
```

4.2 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. The other option is the "two space" syntax:

```
value : Type
```

This syntax is preferable to the prefix-style, but it is not widely adopted due to its increased verbosity.

4.3 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.

4.4 Functions

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

```
def foo(f: Int =@textgreater[] String) = ...

def bar(f: (Boolean, Double) =@textgreater[] List[String]) = ...
```

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

4.4.1 Arity-1

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

```
def map[B](f: A =@textgreater[] 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) =@textgreater[] (String) =@textgreater[] (Boolean) =@textgreater[] Double) = ...

// right!
def foo(f: Int =@textgreater[] String =@textgreater[] Boolean =@textgreater[] Double) = ...
```

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

4.5 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 =@textgreater[] String] }) = ...

// right!
private type FooParam = {
  val baz: List[String =@textgreater[] 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).

CHAPTER

FIVE

NESTED BLOCKS

5.1 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.

5.2 Parentheses

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

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

The only exception to this rule is when defining grammars using parser combinators:

```
lazy val e: Parser[Int] = (
    e @textasciitilde[] "+" @textasciitilde[] e  ^^ { (e1, _, e2) =@textgreater[] e1 + e2 }
  @textbar[] e @textasciitilde[] "-" @textasciitilde[] e  ^^ { (e1, _, e2) =@textgreater[] e1 - e2 }
  @textbar[] """\d+""".r  ^^ { _.toInt }
)
```

Parser combinators are an internal DSL, however, meaning that many of these style guidelines are inapplicable.

CHAPTER

SIX

DECLARATIONS

6.1 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 = ...
}
```

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 {
}
```

6.1.1 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() { ... }

  def add(x: Int, y: 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.

6.1.2 Methods

Methods should be declared according to the following pattern:

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

The only exceptions to this rule are methods which return `Unit`. Such methods should use Scala's syntactic sugar to avoid accidentally confusing return types:

```
def foo(bar: Baz) {        // return type is Unit
  expr
}
```

Modifiers

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

1. Annotations, *each on their own line*
2. Override modifier (`override`)
3. Access modifier (`protected`, `private`)
4. Final modifier (`final`)
5. `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) = 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]) =
  (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 unreadably 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]) = {
  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 =@textgreater[] hd + sum(tail)
  case Nil =@textgreater[] 0
}
```

Not like this:

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

1. For a fluent API

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

   ```
   def unless(exp:Boolean)(code: =@textgreater[] Unit) = if (!exp) code
   unless(x @textless[] 5) {
     println("x was not less than five")
   }
   ```

2. 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 impicit, you must use multiple parameter lists.

3. 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) =@textgreater[] B):B
List("").foldLeft(0)(_ + _.length)

// If, instead:
def foldLeft[B](z: B, op: (B, A) =@textgreater[] B): B
// above won't work, you must specify types
List("").foldLeft(0, (b: Int, a: String) =@textgreater[] 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) =@textgreater[] Any)
                         (implicit urlCreator: URLCreator, configurer: OAuthConfiguration) = {
```

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) =@textgreater[] 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.

6.1.3 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()
```

6.2 Function Values

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

1. `val f1 = { (a: Int, b: Int) => a + b }`
2. `val f2 = (a: Int, b: Int) => a + b`
3. `val f3 = (_: Int) + (_: Int)`
4. `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 curly braces (`{}`), allowing those characters to be a visual cue that a function value may be involved at some level.

6.2.1 Spacing

You will notice that both (1) and (4) insert spaces after the opening brace and before the closing brace. This extra spacing provides a bit of "breathing room" for the contents of the function and makes it easier to distinguish from the surrounding code. There are *no* cases when this spacing should be omitted.

6.2.2 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. 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) =@textgreater[]
  a + b
}
```

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

CHAPTER

SEVEN

CONTROL STRUCTURES

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

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

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

7.1 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" =@textgreater[] println("Good news!")
```

```
  case "bad" =@textgreater[] println("Bad news!")
}
```

7.2 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 @textless[]- board.rows; y @textless[]- board.files)
  yield (x, y)

// right!
for {
  x @textless[]- board.rows
  y @textless[]- 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 @textless[]- board.rows
  y @textless[]- board.files
} {
  printf("(@%d, @%d)", x, y)
}

// right!
for (x @textless[]- board.rows; y @textless[]- 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).

7.3 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.

CHAPTER

EIGHT

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()
```

8.1 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!

8.1.1 Suffix Notation

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

```
names.toList

// is the same as

names toList
```

This style should be used with great care. In order to avoid ambiguity in Scala's grammar, any method which is invoked via suffix notation must be the *last* item on a given line. Also, the following line must be completely empty, otherwise Scala's parser will assume that the suffix notation is actually infix and will (incorrectly) attempt to incorporate the contents of the following line into the suffix invocation:

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

This style should only be used on methods with no side-effects, preferably ones which were declared without parentheses (see above). The most common acceptable case for this syntax is as the last operation in a chain of infix method calls:

```
// acceptable and idiomatic
names map { _.toUpperCase } filter { _.length @textgreater[] 5 } toStream
```

In this case, suffix notation must be used with the `toStream` function, otherwise a separate value assignment would have been required. However, under less specialized circumstances, suffix notation should be avoided:

```
// wrong!
val ls = names toList

// right!
val ls = names.toList
```

The primary exception to this rule is for domain-specific languages. One very common use of suffix notation which goes against the above is converting a `String` value into a `Regexp`:

```
// tolerated
val reg = """\d+(\.\d+)?"""r
```

In this example, `r` is actually a method available on type `String` via an implicit conversion. It is being called in suffix notation for brevity. However, the following would have been just as acceptable:

```
// safer
val reg = """\d+(\.\d+)?""".r
```

8.2 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 =@textgreater[] println(n) }
names mkString ","
optStr getOrElse "@textless[]empty@textgreater[]"
```

```
// wrong!
javaList add item
```

8.2.1 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 @textgreater[] 5 }

// right!
names map { _.toUpperCase } filter { _.length @textgreater[] 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.

8.3 Operators

Symbolic methods (operators) should *always* be invoked using infix notation with spaces separated the target, the operator and the parameter:

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

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

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

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

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

Such operators 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.

CHAPTER

NINE

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.

9.1 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`).

CHAPTER

TEN

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 = {}
```

10.1 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 object 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.

10.2 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@textgreater[] 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@textgreater[] import my.package.complex.ComplexConversions._
 * scala@textgreater[] val complex = 4 + 3.i
```

```
 * complex: my.package.complex.Complex = 4 + 3i
 * }}}
 */
package complex {}
```

10.3 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`.

10.3.1 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.

10.3.2 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 {
  /** Create 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) = {}
  /** Create 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:

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

10.3.3 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.

10.4 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 to 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

ELEVEN

CHANGELOG

v1.2.5 - 4/13/2011

- Expanded multiple-parameter list styles to include formatting for long parameter lists
- Added a changelog

v1.2.4 - 2/13/2011

- Reworked the currying section to be about multiple parameter lists, based on community feedback

v1.2.3 - 1/2/2011

- Reworded bit about short parameter names based on community feedback

v1.2.2 - 12/30/2010

- Spelling mistakes

v1.2.1 - 10/2/2010

- Removing special note about IntelliJ that no longer applies
- Slight cleanup of scaladoc section and added an example for object scaladoc

v1.2.0 - 10/2/2010

- Clarified how packages work in light of Scala 2.8
- Added new section on writing scaladoc

v1.1.0 - 1/26/2010

- Clarified the use for `for` comprehensions
- Documented avoidance of `/:` and similar operators
- Possibly other changes

v1.0.0

- Similar to Daniel Spewak's initial version; possibly slight changes

CHAPTER

TWELVE

INDICES AND TABLES

- *Index*
- *Search Page*