A Brief Summary of Scala

Brian Howard

March 7, 2011

1 Introduction

This guide assumes that you are already familiar with programming in Java and/or C++. You may not be an expert, but you are comfortable with writing classes and methods. We will by no means be learning all of Scala here, and when there are multiple ways to express something in Scala we will generally just choose one. The emphasis is on becoming able to read and write simple functions to manipulate the kinds of data models talked about in the Foundations class: lists, trees, graphs, *etc.* Although Scala has extensive support for an object-oriented style of programming, we will be focusing on its support for functional programming.

Before diving into the details, here is a short sales pitch for Scala. This is a quote from the main Scala website, `http://www.scala-lang.org/`:

> Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages, enabling Java and other programmers to be more productive. Code sizes are typically reduced by a factor of two to three when compared to an equivalent Java application.

Scala is a real-world, commercial-strength language, being used at companies such as Twitter, LinkedIn, FourSquare, Novell, and many others; one of its attractions to these businesses is that it is very compatible with existing Java code, while also supporting more advanced programming techniques as found in functional languages such as Haskell, Standard ML, and F♯, and dynamic languages such as Python and Ruby.

2 Values and Expressions

The basic types of values are essentially the same as in Java, except by convention all of the type names are capitalized (*e.g.*, `Boolean` instead of `boolean`):

- Integers (`Int`): `0`, `42`, `0x2A` (hexadecimal, base-16), `052` (octal, base-8)

- Reals (`Double`): `0.`, `42.0`, `4.2e1`, `.42e+2`, `420e-1`
- Booleans (`Boolean`): `true`, `false`
- Characters (`Char`): `'*'`, `'\u002A'`, `'\52'`, `'\n'` (newline), `'\''` (single-quote)
- Strings (`String`): `"Hello World"`, `"I said, \"Hello!\""`, `"line one\nline two"`

One convenient extension in Scala is the ability to specify strings that contain special characters, such as quotes and newlines, by enclosing them in triple quotes:

```
"""This is a string with several lines.
This is the second line, which contains a quotation: "Hello World".
This is the third line."""
```

The Scala standard library provides quite a few types of collections as well; here are examples of how to create instances of some of them:

- Lists: If x_1, x_2, x_3, ... are values of some type T, then `List(`x_1`, `x_2`, `x_3`, ...)` has type `List[`T`]` (this is akin to the Java parameterized type `List<`T`>`). Another name for the empty list is `Nil`, and values may be added to the front of a list using the `::` operator, so `1 :: 2 :: 3 :: Nil` is an equivalent way of constructing `List(1, 2, 3)`, of type `List[Int]`. The front element of a (non-empty) list a may be retrieved using the `head` operation: a`.head`; the list of everything except the head is a`.tail`. For example, `List(1, 2, 3).head` is `1`, while `List(1, 2, 3).tail` is `List(2, 3)`.

- Tuples: If x_1, x_2, x_3, ... are values of types T_1, T_2, T_3, ..., respectively, then `(`x_1`, `x_2`, `x_3`, ...)` has type `(`T_1`, `T_2`, `T_3`, ...)`. A common special case of this is the *pair*; for example, `(42, "Hello World")` is a pair of type `(Int, String)`. Another special case is the type `Unit`, which has the single value `()` — that is, it is a tuple with no elements. The first element of a (non-empty) tuple a may be accessed as a`._1`; the second element is a`._2`, *etc.*

- Arrays: If x_1, x_2, x_3, ... are values of some type T, then `Array(`x_1`, `x_2`, `x_3`, ...)` has type `Array[`T`]` (this is akin to the Java type T`[]`). Just as in Java and C++, arrays are zero-indexed. Element i of array a is named by the expression a`(`i`)`, so if a`.size` is n, then the elements of a are a`(0)` through a`(`$n-1$`)`. To create an array containing n copies of some value x, use `Array.fill(`n`)(`x`)`.

- Sets: The type `Set[`T`]` is very similar to `List[`T`]`, except that, just like a mathematical set, it does not keep track of the order in which the elements were inserted, and it does not keep duplicate elements. That is, `Set(1, 2, 3)` is the same as `Set(3, 1, 1, 3, 2)`. The fundamental operation on a set is to check whether it contains a particular value: a`.contains(`x`)` returns `true` if x is an element of a.

- Maps: A common data structure in dynamic languages is the "associative array," which behaves like an array with indices more general than the integers 0 to $n-1$. For example, it is often useful to be able to index into a collection using a string: after setting `age("George")` to `13`, we may retrieve George's age by providing the index `"George"` to the map `age`. The Scala type `Map[`K`, `T`]` provides this facility: if k_1, k_2, k_3, ... are *keys* of type K, and x_1, x_2, x_3, ... are values of type T, then `Map(`k_1` -> `x_1`, `k_2` -> `x_2`, `k_3` -> `x_3`, ...)` will map each key to its corresponding value. If `age` is `Map("Alice" -> 5, "Susanna" -> 11, "George" -> 13)`, then `age("George")` will be `13`, as desired. Incidentally, the expression `"Alice" -> 5` is another way to write the pair `("Alice", 5)`.

- Options: A common source of errors in Java is the convention of returning `null` to indicate that some object was unavailable (for example, given a Java map similar to the `age` example above, `age.get("Fred")` returns `null`). Scala attempts to avoid this by providing the "option" type: a value of type `Option[`T`]` is either `Some(`x`)`, where x is a value of type T, or it is the special value `None`. The advantage over returning `null` is that `None` is an actual object; attempting to call a method on it will not result in a null pointer exception. The option type is also a signal to the programmer to be prepared for the case of a non-existent object; in Java, it is too easy to ignore this possibility because *every* object type allows `null`. To retrieve the contained value from an option object, use the `get` method: `Some(`x`).get` yields x. Attempting to call `get` on `None` will throw an exception, so a useful variant is the `getOrElse` method: a`.getOrElse(`y`)` will return the contents of a, if any, or y if a was `None`.

The expressions of Scala are carried over largely unchanged from Java and C++. For example, `(0 <= x) && (x < 10)` is true when `x` is between 0 (inclusive) and 10 (exclusive). Common mathematical functions are available in the `math` object: `math.sqrt(x)`, `math.pow(x, n)`, and `math.max(x, y)` are examples, returning $\sqrt{x}$, x^n, and the maximum of `x` and `y`, respectively. The syntax for calling methods or accessing fields of an object is the same: a`.`m`(`x`, ...)` calls method m of a with arguments x, ..., while a`.`f accesses the field of a named f. Scala provides a few nice generalizations of this: if a method takes no parameters, you may omit the parentheses around the arguments (this removes the distinction between accessing a field and calling a no-parameter method — think of the difference in Java between the `.length()` method on strings and the `.length` field on arrays; in Scala, both of these quantities may be accessed as `.length`[1]); if a method takes exactly one parameter, then the dot and the parentheses may be omitted (for example, given a set a, the expression a `contains` x is the same as a`.contains(`x`)`; Scala allows methods to be used like operators in this fashion because in fact all of the operators, such

[1] As an added convenience, all of the collections except tuples define both `.length` and `.size`, so you don't have to remember which defines which.

as `+`, are really methods: *a* `+` *b* is the same as *a*`.+(`*b*`))`. One last fact to notice about Scala expressions is that even "primitive" values are objects with methods: `42.toString` yields `"42"`.

3 Control Structures

As with expressions, the control structures such as `if` and `while` are substantially the same in Scala as in Java and C++. For example, the following code fragment works exactly the same in all three languages:

```
while (n != 1) {
  if (n % 2 == 0) {
    // n is even
    n = n / 2;
  } else {
    // n is odd
    n = 3 * n + 1;
  }
}
```

One minor difference is that Scala doesn't require the semicolons at the end of each simple statement, although they are allowed. A more significant innovation is that statements and control structures all produce values, so they may be used in expressions. The `while` loop only produces the value `()` of type `Unit` (which is equivalent to the `void` type in Java and C++), so that isn't particularly useful. However, the `if` statement evaluates to one of its two branches, depending on the test, so the above code may also be written

```
while (n != 1) {
  n = if (n % 2 == 0) n / 2 else 3 * n + 1
}
```

Java and C++ provide the operator `?:`, as in `(n % 2 == 0) ? (n / 2) : (3 * n + 1)`, for this purpose, but Scala's approach makes that unnecessary.

In a block of code surrounded by braces, the value produced is the value of the final statement. For example,

```
x = {
  while (j != 0) {
    val temp = j
    j = i % j
    i = temp
  }
  i
}
```

The value eventually assigned to `x` is the final value of `i`, after some number of times through the loop (in fact, this code assigns the GCD of `i` and `j` to `x`; the `val` declaration will be discussed below). There is a `return` statement, which breaks out of a block and returns a value from the enclosing function (see below), but it is generally used only when the normal program flow must be altered, preventing control from reaching the end of the block.

Scala does *not* provide the same `for` loop as its predecessors. Part of the reason for this is that it discourages writing code that relies on modifying, or "mutating," values bound to variables (Scala doesn't even provide the operator `++`, which is found so frequently in counting `for` loops, although it does support the shortcut assignment operators such as `+=`). More importantly, though, Scala provides a very nice generalization of the `for` loop, which can iterate through many of the collection types in a uniform way. Given a collection *a* (other than a tuple), the statement `for (`*x* `<-` *a*`)` *b* will execute the body *b* once for each value *x* taken from *a*. For example,

```
for (n <- List(3, 1, 4, 1, 5)) {
  println(n)
}
```

produces the output

```
3
1
4
1
5
```

The typical C++ or Java counting loop, written as `for (i = 0; i < N; i++)`, becomes `for (i <- 0 until N)`. The expression `0 until N` produces a "range" collection, which is a sequence where the successive values differ only by a step value; by saying `until`, the range excludes the final value, `N`; inclusive ranges are possible with the `to` operator: `1 to 3` is the sequence 1, 2, 3. The value returned from this form of the `for` loop is `()`, just as with `while`.

An additional generalization of the `for` loop can be used to construct a new collection: the expression `for (`*x* `<-` *a*`) yield` *b* evaluates the expression *b* for each element *x* in *a*, and gathers all of the results in a collection of the same kind as *a*. For example,

```
for (i <- List(3, 1, 4, 1, 5)) yield i * i
```

produces the result `List(9, 1, 16, 1, 25)`.

The remaining "control structure" to be discussed in this section is the pattern match. This generalizes the `switch` statement of Java and C++. For example,

```
name match {
  case "Alice" => 5
  case "Susanna" => 11
  case "George" => 13
  case _ => -1
}
```

Depending on the string value `name`, this expression will evaluate to `5` if it is `"Alice"`, `11` if it is `"Susanna"`, `13` if it is `"George"`, and `-1` otherwise. Many types may be matched on; one of the most useful for our purposes is the list:

```
myList match {
  case Nil => println("The list is empty")
  case h :: t => println("The head is " + h + " and the tail is " + t)
}
```

Since every list is either empty (`Nil`) or the concatenation of a head element onto a tail list, `myList` will match one of the two cases. In the second case, the names `h` and `t` (which may be arbitrary variable names) will be bound to the corresponding parts of the list. This is an instance of pattern matching over "case classes," which are described more below.

The "wildcard" pattern, `_`, matches anything (this was seen above in the default case for matching names to ages). Here is another example for lists:

```
front = myList match {
  case Nil => error("Empty list")
  case f :: _ => f
}
```

If `myList` is empty, then the `error` function will throw an exception with the given message; otherwise, the first element will be bound to `f` (which is a newly created variable, local to the match), then returned as the result of the match, and ultimately assigned to `front`.

4 Declarations

4.1 Variables

As mentioned before, Scala supports, but does not require, a functional style of programming. Part of this is that it encourages the use of variables that are never mutated. If a variable is declared with the keyword `val`, this will be enforced — the compiler will not allow you to assign a new value to such a variable later in the program (this is akin to `final` variables in Java, and `const` in C++). If you really need a variable that may vary, then it should be declared with the keyword `var`, but good style is to use `var` only when necessary.

In both cases, a variable is declared by giving the name and its initial value:

```
val x = 42
var y = "Hello World"
```

The type of the variable will be inferred from the type of the expression. This is usually the desired type, but it is possible to specify a (possibly different) type by putting it after the variable name, separated by a colon:

```
var myList: List[Int] = Nil
```

In this case, we want to specify the more general type `List[Int]`, instead of the specific type of `Nil` (which happens to be `Nil` — see below), so that later on it will be legal to assign a different list (such as `42 :: myList`) to `myList`. In this particular case, we could also have initialized `myList` to `List[Int]()`, which is an empty list of the correct type.

4.2 Functions

A function is declared much like a variable, except the keyword is `def` and the function name may be followed by a list of parameters:

```
def midpoint(a: Double, b: Double): Double = (a + b) / 2
```

Unlike a variable declaration, where the right-hand-side is evaluated right away and used to initialize the variable, the right-hand-side of a function is an expression that will be evaluated each time the function is called, with the provided arguments bound to the parameter names. For example, when we call `midpoint(3.0, 5.4)`, the expression `(a + b) / 2` will be evaluated with `a` being `3.0` and `b` being `5.4`.

A function does not need to take parameters. A useful example is a function to simulate rolling a six-sided die:

```
def roll: Int = (math.random * 6).toInt + 1
```

Each time `roll` is called, the expression will be re-evaluated; since it includes a call to a random-number generator (`math.random` returns a number between 0.0 and 1.0), the result will randomly be one of the values 1 through 6.

Scala does not always require you to specify the return type of a function, because it can frequently infer it from the right-hand-side. However, one case where it is unable to do this is when the function is recursive; since we will be frequently working with recursion, it will be a good habit to always specify the return type. There is one exception to this, however; when a function returns a value of type `Unit` — that is, when we are executing the function solely for some side-effect, such as printing a message or sorting an array in place — then we will commonly use the abbreviation

```
def myFun(...) {
  ...
}
```

instead of

```
def myFun(...): Unit = {
  ...
}
```

This matches the conventional form of a "void" function in Java or C++, and emphasizes that we are not producing a value from the function body.

Strictly speaking, the functions described so far are really methods of some object, although which object that is might be hidden by the context in which we are executing our code. As a functional language, Scala also supports true function values, which may be passed around, bound to variables, *etc.* These are introduced with syntax of the form `(`x`:` T`) =>` b, which describes an anonymous function with parameter x of type T and function body b. For example, `(n: Int) => n * n` is a function which takes an integer and returns its square. Binding it to a variable, as follows, produces essentially the same effect as a named function declaration:

```
val square = (n: Int) => n * n
square(4) // yields 16
```

4.3 Classes

As packages in which to encapsulate related fields and methods, Scala's classes are quite similar to Java's. Perhaps the most significant difference is in the form of the constructor: where Java declares a method with the same name as the class, Scala merges the constructor body with the class definition itself, which, together with other features of the language, frequently leads to a significant reduction in "boilerplate" code (that is, code which follows a standard, predictable pattern, and which contains very little new information). Here is an example:

```
// THIS IS JAVA
public class Triangle {
  private double width;
  private double height;
  private double hypotenuse;

  public Triangle(double width, double height) {
    this.width = width;
    this.height = height;
    this.hypotenuse = Math.sqrt(width * width + height * height);
  }

  public double getWidth() {
    return width;
  }

  public double getHeight() {
    return height;
  }

  public double getHypotenuse() {
    return hypotenuse;
  }
}
```

In Scala, this would typically be written as follows:

```
class Triangle(val width: Double, val height: Double) {
  val hypotenuse = math.sqrt(width * width + height * height)
}
```

The class definition header includes the constructor parameters `width` and `height`; by making them `val`s, they will be exposed as read-only fields of the object. The declarations in the body of the object are evaluated with the actual arguments bound to these parameters, so when the field `hypotenuse` is declared, it will be initialized with the appropriate value. Finally, because we have declared three `val` fields, there is no need for the getter methods which are so common in Java code; since a `val` is immutable, we do not have to worry about other code changing these fields and it is safe to make them public (which is the default in Scala).

An instance of the `Triangle` class is created just as in Java: `new Triangle(3.0, 4.0)` instantiates the class with the given constructor arguments, producing a `Triangle` object with a hypotenuse of `5.0`.

One aspect of a class which is significantly different in Scala from Java is the notion of a static member. In Java, static members are shared among all instances of the class. Scala does not use this concept; instead, it has the notion of a "companion object." First of all, Scala supports the definition of "one-off" objects — singletons, which are the only instance of their class. An example of this is the `math` object, which we may imagine being defined as follows:

```
object math {
  def max(a: Double, b: Double): Double = if (a > b) a else b
  def sqrt(x: Double): Double = ...
  ...
}
```

This directly declares an object with the given methods; there is no need to instantiate it, because there will never be more than the one `math` object. In Java, this corresponds to the `Math` class having only static members, which are called through the class name (`Math.sqrt`) rather than through an instance.

Now suppose we want to have a class of objects with a shared static member. A common example of this is a "factory" method: a method which hides the details of constructing an object (often because the actual object returned will belong to a subclass that the client might not know about). In our `Triangle` class, suppose the Java version had the following method:

```
// THIS IS JAVA
public static Triangle makeIsoceles(double width) {
  return new Triangle(width, width);
}
```

In Scala, we would put this method in the "companion" of the `Triangle` class, which is an object also named `Triangle`:

```
object Triangle {
  def makeIsoceles(width: Double): Triangle = new Triangle(width, width)
}
```

In both cases, we may call `Triangle.makeIsoceles(1.0)` to produce a new triangle with width and height both 1; the advantage in Scala is that there is a cleaner separation between the operations supported by the entire class and the operations supported by an *instance* of the class.

One more difference between Scala and Java is largely a matter of terminology, as far as it will matter to us. Where Java has "interfaces" which may be "implemented" by several classes, Scala has "traits" which may be "extended" by several classes. The effect is the same, although there are some more advanced features of traits which are difficult to achieve in Java.

For our purposes, traits are needed as part of the mechanism for defining "algebraic data types." An algebraic data type is one where there may be several ways to construct simple values, and there may be several ways to build larger, more complex values out of smaller ones. The canonical examples here are lists and trees: starting from empty lists and empty trees or individual leaves, we may grow larger structures by adding values to the front of the list, or adding new parent nodes above some number of children.

Here is the definition of the algebraic data type of expression trees, which are either leaves with a numeric value or nodes with an operator and two expression trees as operands:

```
sealed trait Expr
case class Leaf(value: Double) extends Expr
case class Node(operator: String, left: Expr, right: Expr) extends Expr
```

The first line declares the trait `Expr`, which is the base type being defined. The keyword `sealed` indicates that the following lines will specify *all* of the ways in which values of this type may be created (unlike the situation with a Java interface, which may have new implementations supplied at any time). The remaining two lines give class definitions (with no bodies, because they have no non-standard methods) for the `Leaf` and `Node` classes of expression trees. The clause `extends Expr` on each declares that they are defining specific kinds of `Expr` values (just like saying `implements` in Java).

The keyword `case` at the front of each tells the Scala compiler to generate some extra code in each class to support convenient object creation and pattern matching (for example, it creates a factory method with the name of the case class, so we may construct new objects without explicitly saying "`new`"). It also turns each of the constructor parameters into `val`s, thus making them available as fields. Note that, because the `Expr` trait is `sealed`, we know that a pattern-match on expression trees only has to cover these two cases, either `Leaf` or `Node`.

The effect of this is that we may construct an expression tree by writing code such as `Node("+", Leaf(27), Node("*", Leaf(3), Leaf(5)))`, and we can use pattern matching to pull a tree back apart:

```
def eval(e: Expr): Double = e match {
  case Leaf(v) => v
  case Node("+", left, right) => eval(left) + eval(right)
  case Node("-", left, right) => eval(left) - eval(right)
  case Node("*", left, right) => eval(left) * eval(right)
  case Node("/", left, right) => eval(left) / eval(right)
  case Node(op, _, _) => error("Invalid operator: " + op)
}
```

Here is a trace of the execution of this function:

```
eval(Node("+", Leaf(27), Node("*", Leaf(3), Leaf(5))))
= eval(Leaf(27)) + eval(Node("*", Leaf(3), Leaf(5)))
= 27 + (eval(Leaf(3)) * eval(Leaf(5)))
= 27 + (3 * 5)
= 27 + 15
= 42
```

Now we can see that the standard list type is an algebraic data type. We could define our own version as follows (specialized to integers for simplicity):

```
sealed trait MyList
case object Nil extends MyList
case class Cons(head: Int, tail: MyList) extends MyList
```

The only new wrinkle here is the `case object`, but this should make sense, since there will only ever be the one empty list (it doesn't contain any mutable data, so there is no reason not to share it with every list). The real definition of the Scala list type involves a parameterized type, plus some operator magic to allow the `Cons` operation to be called `::`, but this is the essence of it.

The final point to emphasize about these algebraic data types is that they are constructed recursively, and the natural way to define functions over them is to follow that recursion; each case class generally turns into a case in a match expression, with recursive calls corresponding to the places where smaller structures are included. The natural way to prove anything about such a type is to use structural induction, where the base cases are for the simple constructors and the induction steps are for the compound constructors. In a sense, all of these notions are equivalent ways of looking at the same thing.