Scala Tutorial I

Jan Macháček

June 8, 2018

Abstract

Scala is a fusion langauge that combines functional and object-oriented programming paradigms in a syntax that is similar to most other *C-like* languages. The ...

1 Syntax crash-course

Scala's syntax follows the syntax of other C-like languages, though–like Pascal–the type specification follows an identifier. Scala's ***class*** behaves exactly like Java's ***class*** , and its synax is not wildly different. (See Listing 1.)

```
Class declaration means the same thing as Java; constructor parameters are specified in the block immediately following the class name.
Note that the types follow the identifier; instead of String constructorParam1 Scala uses constructorParam1: String
class MyClass(constructorParam1: String, constuctorParam2: Int) {

  Methods begin with the keyword def, followed by name and parameters. The return type follows similar pattern; Unit means void.
  The body of the method follows the equals sign.
  def execute(methodParam1: List[Int]): Unit = {

  }

}

Constructing instances uses the typical new keyword...
new MyClass("foo", 42)
  Method invocation is exactly like Java's
  .execute(List(1, 2, 3))
```

Listing 1: Classes and methods

Most of this syntax is familiar and unsurprising; the only thing that might feel odd is the square bracket for "generics" in $List[Int]$: in Java, this would be written as $List<Int>$. This is part of Scala's legacy. A long, long time ago, XML was very exciting; and Scala allows XML literals. These XML literals use the angle brackets. This meant that a different symbol had to be used for type parameters. Because the square bracket is used for type parameters, array indexing is also done using regular parentheses[1].

Interfaces use the ***trait*** keyword in Scala; their usage and features are similar to ***interface*** in Java (particularly Java 8 which adds default implementations). It is possible to make anonymous implementations of a ***trait*** , as well as to implement it in ordinary ***class*** es. (Viz Listing 2.)

```
Apart from the trait keyword, the syntax is unremarkable
trait ReportGenerator {
  Interface methods are public and abstract; they specify parameters and return type
  def generate(userId: Int): Array[Byte]
}

A trait can be implemented in a class using the keyword extends. Additional traits to be implemented use the with keyword.
class ReportGeneratorImpl extends ReportGenerator with Cloneable {
  Instead of the @Override annotation Scala uses the override keyword.
  override def generate(userId: Int): Array[Byte] = ...
  override def clone(): AnyRef = ...
}

It is also possible to make an anonymous implementation of a trait using the new keyword.
new ReportGenerator {
  override def generate(userId: Int): Array[Byte] = ...
}
```

Listing 2: Traits

[1] I know, it looks like VisualBasic or Fortran (formerly FORTRAN)!

It is worth noting that there is no special syntax for *array of X* in Scala. Instead, it uses *Array* with the specified type parameter. (So, Java's *byte[]* becomes *Array[Byte]*, *User[]* becomes *Array[User]*, and so on.) Also notice the *AnyRef* in the implementation of the *clone()* method–it is equivalent to *java.lang.Object*.

So far, there are no major surprises: classes and interfaces work just like most other languages, constructor, method, and parameter definition also looks fairly ordinary. The syntax for functions (finally!) uses parameters (each with its type following the *:* symbol, if needed) followed by fat arrow ⇒, and the body of the function. Usage is the same as Java 8; and Scala's collections library contains concepts that are fairly similar.

```
Double every Int in the list
List(1, 2, 3, ...).map { (x: Int) ⇒ 2 * x }

Scala can infer the type of x, so there's no need to specify it
List(1, 2, 3, ...).map { x ⇒ 2 * x }

An alternative syntax uses paretheses in place of brackets
List(1, 2, 3, ...).map(x ⇒ 2 * x)

It is possible to avoid having to declare the parameter x, and use _ instead. Scala compiler replaces the every occurrence of _ with a fresh parameter
    declaration.
List(1, 2, 3, ...).map(2 * _)
```

Listing 3: Fields & variables

Finally, fields (and variables) use the keywords ***val*** and ***var***. The first declares an immutable variable (and a getter if field), the second declares a mutable variable (and a getter and setter if field); see Listing 4.

```
Field definitions have to specify initial value; use _ for default value.
class User {
  var id: Long = _
  var name: String = _
  var dob: Date = _
}

val user = new User
```

Listing 4: Fields & variables

This code is terrible! The default value for reference types (*AnyRef*s) is ***null***, and variants of *zero* for primitive types. The *name* and *dob* fields in the *user* instance are ***null***, and *id* is *0L*. What's worse, the class has setters for these fields, and it's possible to invoke them. The syntax is somewhat nicer–it looks like plain assignment using the = operator, though it is actually invoking a setter–but the semantics of the code in Listing 5 is terrible.

```
Field definitions have to specify initial value; use _ for default value.
class User {
  var id: Long = _
  var name: String = _
  var dob: Date = _
}

val user = new User
user.id = 5
user.name = "Foo Quux"
user.dob = ...
```

Listing 5: Fields & variables II

While syntactically valid code, it is very confusing. The *user* variable is declared immutable, yet it is possible to invoke its setters. The equivalent Java code would declare the *user* variable ***final***, but then still use the setters to mutate it.

1.1 Killer features

The syntax (and its application) so far looks just like Java with less typing. There must be something else that makes it worth leaving the creature comforts of Java.

- case classes
- pattern matching
- everything is an expression
- for comprehensions
- implicits
- rich type system

case classes A case class is just like a class in that it is a container for data and methods, but the fields it contains only have getters. (Immutability only goes as far as immutability of the references. Even in Scala, immutability without any additional code is equivalent to using ***final*** in Java.) Nevertheless, case classes are fantastically convenient to define data structures. Consider the *Person* case class defined in Listing 6.

```
case class Person(firstName: String, lastName: String, age: Int)
```

Listing 6: Case class *Person*

This is all it takes to define an immutable structure (with the T&Cs from above) with the fields *firstName*, *lastName*, and *age*; but also with appropriate *toString*, *hashCode*, and *equals* implementations. These automatically generated implementations delegate to the *toString*, *hashCode*, and *equals* methods of all the fields, in the order in which they are specified.

To create an instance of a case class, do not use the *new* keyword; instead, write the parameters directly after the case class name, as shown in Listing 7.

```
Notice that there is no new keyword; the parameter values are applied directly after the case class name.
val fq = Person("Foo", "Quux", 42)

To access the fields, use the familiar . notation.
fq.firstName  | "Foo"

Invoking the toString method prints a reasonable representation of the case class.
fq.toString  | "Person(Foo,Quux,42)"

It is possible to vary the order of the parameters if the parameter names are also specified. This can help readability.
val fb = Person(lastName = "Baz", firstName = "Foo", age = 50)
fb.toString  | "Person(Foo,Baz,50)"

Equality is implemented by delegation to the parameters' hashCode and equals methods.
Person("Foo", "Quux", 42) == fq    | true, even though they are different instances.
fq == Person("Foo", "Quux", 42)    | true, even though they are different instances.
```

Listing 7: Using case class *Person*

The consciseness of Scala's syntax is beginning to show. It would have been much more cumbersome to implement all this (including correct *hashCode*, *equals*, and reasonable *toString*) in Java. Even with correct implementations, it would not have been possible to use *==* to test for instance equality.

> **Try this out: Using case classes and functions**
> In *src/scala*, run *sbt console*; in the console, define a ***case class*** for an employee record (decide which fields would be useful); then define a ***case class*** for employment linking an *employee* and his or her *salary*. Then use *List.fill(N)(Employment(...))* and useful methods in *scala.util.Random* to generate *N* random *Employment* instances. Assign those to a variable.
>
> - find the top fat cat in the list. (Hint: use *map*, *max*, and *find* functions on the *List[Employment]* to find the fattest cat.)
> - find the top 10 fat cats in the list. (Hint: use *sortBy* or *sortWith*, followed by *take*)

2 Spring Framework

The Spring Framework is a dependency injection framework; it encourages composition over inheritance, it encourages expressing dependencies as interfaces rather than concrete implementations. The framework takes care of instantiating the components in the correct order; most components (the ones that fall into the *@Component* sterotype) are *singletons*. This means that it is possible to treat the *@Component*-annotated components as namespaces rather than containers of state[2]. The reason why Spring Framework encourages programming to interfaces is to make the software easily testable: there can be separate implementations or mocks for unit and integration tests.

```
interface ReportGenerator {
    Generates the PDF report for the given user,
    returns the byte array representing the PDF contents
    byte[] generate(final String user);
}

@Component
public class ReportService {
```

[2] In fact if the methods in *@Component*-annotated classes mutates & accesses its fields, it will suffer from race conditions.

```
    private final ReportGenerator reportGenerator;

    @Inject
    public ReportService(final ReportGenerator reportGenerator) {
        this.reportGenerator = reportGenerator;
    }

    public void reportAll() {
        for (final String user : Arrays.asList("a", "b", "c")) {
            final byte[] pdf = this.reportGenerator.generate(user);
            Now you're on your own...
        }
    }
}
```

Listing 8: Components

For a Spring Framework application to be able to construct the $ReportService$, it needs to be able to construct exactly one component that implements the $ReportGenerator$ interface.

```
@Component
public class JasperReportsReportGenerator implements ReportGenerator {
    public byte[] generate(final String user) {
        ...
    }
}
```

Listing 9: Components

Without a DI framework, the work of constructing the dependencies would fall on the programmers, yielding code similar to Listing 10.

```
Typically in public static void main(String[] args) or in a test:

ReportGenerator rg = new JasperReportsReportGenerator();
ReportService rs = new ReportService(rg);
```

Listing 10: Manual DI

Constructing the instances of the $JasperReportsReportGenerator$ and $ReportService$ using their constructors isn't a problem per se, but with growing number of dependencies this grows to be tedious.

3 Zero to hundred

FizzBuzz is a typical program that follows *Hello, world*, adding iteration and conditions. The Scala version of FizzBuzz is shown in Listing 11–it shows the definition of a function def, followed by name and arguments, and its implementation that follows the $=$ sign. The loop (***for***) and condition (***if*** , ***else***) keywords are the old friends from other languages.

```
def fizzBuzz = {
  for (i ← 1 to 100) {
    if (i % 15 == 0) println("FizzBuzz")
    else if (i % 3 == 0) println("Fizz")
    else if (i % 5 == 0) println("Buzz")
    else println(i)
  }
}
```

Listing 11: Fizz Buzz

The FizzBuzz from Listing 11 isn't particularly re-usable: it simply prints 100 elements to the standard output, nothing else and nothing more. There is no way, for example, to direct the output to a web socket, or to use it to determine how it maps of the value in the $integer$ domain to the "FizzBuzz domain". Hmm!–*mapping* and *domain* sound like mathematics; and functional programming is supposed to be somehow more mathematical. And mathematics is jolly wonderful.

The first step in making the $fizzBuzz$ more mathematical is to make it map an input to exactly one useful output. Right now, its return type now is $Unit$, which is a bit like $void$ in Java and C; changing its definition to $def\ fizzBuzz2(max:\ Int):\ Unit$ (and then using the max parameter in the loop) isn't particularly useful: it is a mapping from a number to $Unit$. And, if this were mathematics, there can be only one such mapping: $def\ fizzBuzz2(max:\ Int):\ Unit\ =\ ()$. Instead of printing the elements to the console, the implementation needs to return a value that can be printed. A a simple $String$ would do, but a Seq of $String$ s is better. The type becomes $Int \Rightarrow Seq[String]$, and the implementation is shown in Listing 12.

```
def fizzBuzz(max: Int): Seq[String] = {
  var result = List.empty[String]
  for (i ← 1 to max) {
    if (i % 15 == 0) result = result :+ "FizzBuzz"
    else if (i % 3 == 0) result = result :+ "Fizz"
    else if (i % 5 == 0) result = result :+ "Buzz"
    else result = result :+ i.toString
  }
  result
}
```

Listing 12: Fizz Buzz

This is a huge improvement! The *fizzBuzz* is now indeed a function: it maps input to output and its result depends only on the value of the parameter. It would even be possible to pre-compute the result for all possible values of the input and replace the function's body with a look-up in that table: the function would become just data!

Well, the outside looks great, but the implementation stinks! It uses mutation, and what about the strange *:+* operator in *result :+ "Fizz"*, never mind the ***for*** *(i ← 1 to max) {...}* nonsense!

```
def fizzBuzz(max: Int): Seq[String] = {
  def fb(i: Int): String =
    if (i % 15 == 0) "FizzBuzz"
    else if (i % 3 == 0) "Fizz"
    else if (i % 5 == 0) "Buzz"
    else i.toString

  (1 to max).map(fb)
}
```

Listing 13: Fizz Buzz

In Scala, every concrete type (except *Nothing*) can have a value: for example, the type *Boolean* is inhabited by values ***true***, ***false***; the type *Int* is inhabited by values such as *5*, *42*, *−100*, *0*, *...*; the type *String* is inhabited by values such as *"Hi"*, *":)"*, *""*; the type *Unit* is inhabited by the only value *()*. (No, really, it's perfectly good Scala syntax to write *()* as value. It's just not particularly useful.) The only type that does not have any inhabitants is *Nothing*: it represents expressions that *diverge*, for example throwing an exception.

Taking a more precise look at *def fizzBuzz* reveals its type to be *Unit*; it evaluates to only one value, namely *()*. If it were a function in the sense of strictly mapping input to output, it would be no different from any other *()* *constant*. But *fizzBuzz* does some additional work before returning *()*; this additional work is not represented by its type, even though it is its *raison d'être*.

In Java and C, there is no *value* of type *void*

As it stands, its type is *() ⇒ Unit*,

4 Pattern matching

sasd

5