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 $java.lang.Object$.

2 Killer features

To a modern Java programmer, Scala comes with five killer features, which make everyday programming tasks much easier.

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

3 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 {
    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 3: 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 4: Components

Without a DI framework, the work of constructing the dependencies would fall on the programmers, yielding code similar to Listing 5.

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

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

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

Listing 5: 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.

4 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 6–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 6: Fizz Buzz

The FizzBuzz from Listing 6 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 7.

```
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 7: 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 8: Fizz Buzz

In Scala, every concrete type (except $Nothing$) can have a value: for example, the type $Boolean$ is inhabited by values $\mathbf{true}$, $\mathbf{false}$; the type Int is inhabited by values such as 5, 42, -100, 0, $\ldots$; 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 $() \Rightarrow Unit$,

5 Pattern matching

sasd

6