Scala Tutorial I

Jan Macháček

June 7, 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 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[1]. 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 User user);
}

@Component
public class ReportService {
    private final ReportGenerator reportGenerator;
    private final CrudRepository<User, Long> userRepository;

    @Inject
    public ReportService(final ReportGenerator reportGenerator,
                         final CrudRepository<User, Long> userRepository) {
        this.reportGenerator = reportGenerator;
        this.userRepository = userRepository;
    }

    public void reportAll() {
        for (final User user : userRepository.findAll()) {
            final byte[] pdf = this.reportGenerator.generate(user);
            Now you're on your own...
        }
    }
}
```

Listing 1: 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 User user) {
        ...
    }
}
```

Listing 2: Components

Without a DI framework, the work of constructing the dependencies would fall on the programmers, yielding code similar to Listing 3.

[1] 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();
CrudRepository<User, Long> ur = ...;
ReportService rs = new ReportService(rg, ur);
```

Listing 3: Manual DI

2 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 4–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 4: Fizz Buzz

The FizzBuzz from Listing 4 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 5.

```
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 5: 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 6: 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 $() \Rightarrow Unit$,

3 Pattern matching

sasd

4