A Scala Tutorial

for Java programmers
July 27, 2004

Michel Schinz

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

1 Introduction

This document gives a quick introduction to the Scala language and compiler. It is intended for people who already have some programming experience and want an overview of what they can do with Scala. A basic knowledge of object-oriented programming, especially in Java, is assumed.

2 A first example

As a first example, we will use the standard *Hello world* program. It is not very fascinating but makes it easy to demonstrate the use of the Scala tools without knowing too much about the language. Here is how it looks:

```
object HelloWorld {
  def main(args: Array[String]): unit = {
    System.out.println("Hello, world!");
  }
}
```

The structure of this program should be familiar to Java programmers: it consists of one method called `main` which takes the command line arguments, an array of strings, as parameter; the body of this method consists of a single call to the `println` method of the object representing the standard output, with the friendly greeting as argument. The `main` method is declared as returning a value of type `unit`, which for now can be seen as similar to Java's `void` type.

What is less familiar to Java programmers is the **`object`** declaration containing the `main` method. Such a declaration introduces what is commonly known as a *singleton object*, that is a class with a single instance. The declaration above thus declares both a class called `HelloWorld` and an instance of that class, also called `HelloWorld`. This instance is created on demand, the first time it is used.

The astute reader might have noticed that the `main` method is not declared as `static` here. This is because static members (methods or fields) do not exist in Scala. Rather than defining static members, the Scala programmer declares these members in singleton objects.

2.1 Compiling the example

To compile the example, we use `scalac`, the Scala compiler. `scalac` works like most compilers: it takes a source file as argument, maybe some options, and produces one or several object files. The object files it produces are standard Java class files.

If we save the above program in a file called `HelloWorld.scala`, we can compile it by issuing the following command (the greater-than sign '>' represents the shell prompt and should not be typed):

```
> scalac HelloWorld.scala
```

This will generate a few class files in the current directory. One of them will be called `HelloWorld.class`, and contains a class which can be directly executed using the `scala` command, as the following section shows.

2.2 Running the example

Once compiled, a Scala program can be run using the `scala` command. Its usage is very similar to the `java` command used to run Java programs, and accepts the same options. The above example can be executed using the following command, which produces the expected output:

```
> scala -classpath . HelloWorld

Hello, world!
```

3 Interaction with Java

One of the strength of Scala is that it makes it very easy to interact with Java code. Actually, the example of the previous section showed this: to print the message on screen, we simply used a call to Java's `println` method on the (Java) object `System.out`.

All Java code is accessible as easily from Scala. Like in Java, all classes from the `java.lang` package are imported by default, while others need to be imported explicitly.

Let's look at another example to see this. The aim of this example is to compute and print the factorial of 100 using Java big integers (i.e. the class `BigInteger` in package `java.math`), since the result does not fit in a Java integer. This program looks like this:

```
object BigFactorial {
  import java.math.BigInteger, BigInteger._;

  def fact(x: BigInteger): BigInteger =
    if (x == ZERO) ONE
    else x multiply fact(x subtract ONE);

  def main(args: Array[String]): unit =
    System.out.println("fact(100) = "
                       + fact(new BigInteger("100")));
}
```

Scala's **`import`** statement looks very similar to Java's equivalent, but an important difference appears here: to import all the names of a package or class, one uses the underscore (`_`) character instead of the asterisk (`*`). That's because the asterisk is actually a valid Scala identifier, as we will see later.

The **import** statement above therefore starts by importing the class `BigInteger`, and then all the names it contains. This makes the static fields `ZERO` and `ONE` directly visible.

While we're talking about `ZERO`, something must be said about the condition of the **if** expression in method `fact`. Is it really correct to check that `x` is zero by using the == operator? A Java programmer would say no, because in that language the == operator compares objects by physical equality, and this is not what we want here. What we want to know is whether `x` is *some* big integer object representing zero, and there might be several of them. So a Java programmer would use the `equals` method to perform the comparison.

The Scala programmer, on the other hand, can use == here because that operator compares objects according to the `equals` method. Is == just an alias for `equals` then? Well, almost, but == has one advantage over `equals` in that it works also when the selector is the **null** constant.

The `fact` method also shows some characteristics of Scala's syntax. The first one is that the method body does not have to be surrounded by curly braces if it consists of a single expression. The second one is that methods taking one argument can be used with an infix syntax. That is, the expression

```
x subtract ONE
```

is just another, slightly less verbose way of writing the expression

```
x.subtract(ONE)
```

This might seem like a minor syntactic detail, but it has important consequences, one of which will be explored in the next section.

To conclude this section about integration with Java, it should be noted that it is also possible to inherit from Java classes and implement Java interfaces directly in Scala.

4 Everything is an object

Scala is a pure object-oriented language in the sense that *everything* is an object, including numbers or functions. It differs from Java in that respect, since Java distinguishes numeric types from objects, and does not enable one to manipulate functions as values.

4.1 Numbers are objects

Since numbers are objects, they also have methods. And in fact, an arithmetic expression like the following:

```
1 + 2 * 3 / x
```

consists exclusively of method calls, because it is equivalent to the following expression, as we saw in the previous section:

```
1.+(2.*(3./(x)))
```

This also means that `+`, `*`, etc. are valid identifiers in Scala.

4.2 Functions are objects

Perhaps more surprising for the Java programmer, functions are also objects in Scala. It is therefore possible to pass functions as arguments, to store them in variables, and to return them from other functions. This ability to manipulate functions as values is one of the cornerstone of a very interesting programming paradigm called *functional programming*.

As a very simple example of why it can be useful to use functions as values, let's consider a timer function whose aim is to perform some action every second. How do we pass it the action to perform? Quite logically, as a function. This very simple kind of function passing should be familiar to many programmers: it is often used in user-interface code, to register call-back functions which get called when some event occurs.

In the following program, the timer function is called `oncePerSecond`, and it gets a call-back function as argument. The type of this function is written `() => unit` and is the type of all functions which have no arguments and return a value of type `unit`. The main function of this program simply calls this timer function with a call-back which prints a sentence on the terminal. In other words, this program endlessly prints the sentence *time flies like an arrow* every second.

```
object Timer {
  def oncePerSecond(callback: () => unit): unit =
    while (true) { callback(); Thread sleep 1000 };

  def timeFlies(): unit =
    Console.println("time flies like an arrow...");

  def main(args: Array[String]): unit =
    oncePerSecond(timeFlies);
}
```

We note that in order to print the string, we used method `println` of class `Console` instead of using the one from `System.out`. For now, `Console` can be seen as a Scala equivalent of Java's `System.out`.

4.2.1 Anonymous functions

While this program is easy to understand, it can be refined a bit. First of all, notice that the function `timeFlies` is only defined in order to be passed later to the `oncePerSecond` function. Having to name that function, which is only used once, might seem unnecessary, and it would in fact be nice to be able to construct this function just as it is passed to `oncePerSecond`. This is possible in Scala using *anonymous functions*, which are exactly that: functions without a name. The revised ver-

sion of our timer program using an anonymous function instead of `timeFlies` looks like that:

```
object TimerAnonymous {
  def oncePerSecond(callback: () => unit): unit =
    while (true) { callback(); Thread sleep 1000 };

  def main(args: Array[String]): unit =
    oncePerSecond(() =>
      Console.println("time flies like an arrow..."));
}
```

The presence of an anonymous function in this example is revealed by the right arrow '=>' which separates the function's argument list from its body. In this example, the argument list is empty, as witnessed by the empty pair of parenthesis on the left of the arrow. The body of the function is the same as the one of `timeFlies` above.

5 Classes

As we have seen above, Scala is an object-oriented language, and as such it has a concept of class.[1] Classes in Scala are declared using a syntax which is close to Java's syntax. One important difference is that classes in Scala can have parameters. This is illustrated in the following definition of complex numbers.

```
class Complex(real: double, imaginary: double) {
  def re() = real;
  def im() = imaginary;
}
```

This complex class takes two arguments, which are the real and imaginary part of the complex. These arguments must be passed when creating an instance of class `Complex`, as follows: `new Complex(1.5, 2.3)`. The class contains two methods, called `re` and `im`, which give access to these two parts.

It should be noted that the return type of these two methods is not given explicitly. It will be inferred automatically by the compiler, which looks at the right-hand side of these methods and deduces that both return a value of type `double`.

The compiler is not always able to infer types like it does here, and there is unfortunately no simple rule to know exactly when it will be, and when not. In practice, this is usually not a problem since the compiler complains when it is not able to infer a type which was not given explicitly. As a simple rule, beginner Scala programmers should try to omit type declarations which seem to be easy to deduce from the context, and see if the compiler agrees. After some time, the programmer should get a good feeling about when to omit types, and when to specify them explicitly.

[1] For the sake of completeness, it should be noted that some object-oriented languages do not have the concept of class, but Scala is not one of them.

5.1 Methods without arguments

A small problem of the methods `re` and `im` is that, in order to call them, one has to put an empty pair of parenthesis after their name, as the following example shows:

```
val c = new Complex(1.2, 3.4);
Console.println("imaginary part: " + c.im());
```

It would be nicer to be able to access the real and imaginary parts like if they were fields, without putting the empty pair of parenthesis. This is perfectly doable in Scala, simply by defining them as methods *without arguments*. Such methods differ from methods with zero arguments in that they don't have parenthesis after their name, neither in their definition nor in their use. Our `Complex` class can be rewritten as follows:

```
class Complex(real: double, imaginary: double) {
  def re = real;
  def im = imaginary;
}
```

5.2 Inheritance and overriding

All classes in Scala inherit from a super-class. When no super-class is specified, as in the `Complex` example of previous section, `scala.Object` is implicitly used.

It is possible to override methods inherited from a super-class in Scala. It is however mandatory to explicitly specify that a method overrides another one using the **`override`** modifier, in order to avoid accidental overriding. As an example, our `Complex` class can be augmented with a redefinition of the `toString` method inherited from `Object`.

```
class Complex(real: double, imaginary: double) {
  def re = real;
  def im = imaginary;
  override def toString() =
    "" + re + (if (im < 0) "-" else "+") + im + "i";
}
```

6 Case classes and pattern matching

A kind of data structure that often appears in programs is the tree. For example, interpreters and compilers usually represent programs internally as trees; XML documents are trees; and several kinds of containers are based on trees, like red-black trees.

We will now examine how such trees are represented and manipulated in Scala through a small calculator program. The aim of this program is to manipulate very

simple arithmetic expressions composed of sums, integer constants and variables. Two examples of such expressions are $1+2$ and $(x+x)+(7+y)$.

We first have to decide on a representation for such expressions. The most natural one is the tree, where nodes are operations (here, the addition) and leaves are values (here constants or variables).

In Java, such a tree would be represented using an abstract super-class for the trees, and one concrete sub-class per node or leaf. In a functional programming language, one would use an algebraic data-type for the same purpose. Scala provides the concept of *case classes* which is somewhat in between the two. Here is how they can be used to define the type of the trees for our example:

```
abstract class Tree;
case class Sum(l: Tree, r: Tree) extends Tree;
case class Var(n: String) extends Tree;
case class Const(v: int) extends Tree;
```

The fact that classes `Sum`, `Var` and `Const` are declared as case classes means that they differ from standard classes in several respects:

- the **`new`** keyword is not mandatory to create instances of these classes (i.e. one can write `Const(5)` instead of **`new`** `Const(5)`),
- getter functions are automatically defined for the constructor parameters (i.e. it is possible to get the value of the `v` constructor parameter of some instance `c` of class `Const` just by writing `c.v`),
- default definitions for methods `equals` and `hashCode` are provided, which work on the *structure* of the instances and not on their identity,
- a default definition for method `toString` is provided, and prints the value in a "source form" (e.g. the tree for expression $x+1$ prints as `Sum(Var(x),Const(1))`),
- instances of these classes can be decomposed through *pattern matching* as we will see below.

Now that we have defined the data-type to represent our arithmetic expressions, we can start defining operations to manipulate them. We will start with a function to evaluate an expression in some *environment*. The aim of the environment is to give values to variables. For example, the expression $x+1$ evaluated in an environment which associates the value 5 to variable x, written $\{x \rightarrow 5\}$, gives 6 as result.

We therefore have to find a way to represent environments. We could of course use some associative data-structure like a hash table, but we can also directly use functions! An environment is really nothing more than a function which associates a value to a (variable) name. The environment $\{x \rightarrow 5\}$ given above can simply be written as follows in Scala:

```
{ case "x" => 5 }
```

This notation defines a function which, when given the string "x" as argument, returns the integer 5, and fails with an exception otherwise.

Before writing the evaluation function, let us give a name to the type of the environments. We could of course always use the type `String => int` for environments, but it simplifies the program if we introduce a name for this type, and makes future changes easier. This is accomplished in Scala with the following notation:

```
type Environment = (String => int);
```

From then on, the type `Environment` can be used as an alias of the type of functions from `String` to `int`.

We can now give the definition of the evaluation function. Conceptually, it is very simple: the value of a sum of two expressions is simply the sum of the value of these expressions; the value of a variable is obtained directly from the environment; and the value of a constant is the constant itself. Expressing this in Scala is not more difficult:

```
def eval(t: Tree, env: Environment): int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

This evaluation function works by performing *pattern matching* on the tree `t`. Intuitively, the meaning of the above definition should be clear:

1. it first checks if the tree `t` is a `Sum`, and if it is, it binds the left sub-tree to a new variable called `l` and the right sub-tree to a variable called `r`, and then proceeds with the evaluation of the expression following the arrow; this expression can (and does) make use of the variables bound by the pattern appearing on the left of the arrow, i.e. `l` and `r`,
2. if the first check does not succeed, that is if the tree is not a `Sum`, it goes on and checks if `t` is a `Var`; if it is, it binds the name contained in the `Var` node to a variable `n` and proceeds with the right-hand expression,
3. if the second check also fails, that is if `t` is neither a `Sum` nor a `Var`, it checks if it is a `Const`, and if it is, it binds the value contained in the `Const` node to a variable `v` and proceeds with the right-hand side,
4. finally, if all checks fail, an exception is raised to signal the failure of the pattern matching expression; this could happen here only if more sub-classes of `Tree` were declared.

We see that the basic idea of pattern matching is to attempt to match a value to a series of patterns, and as soon as a pattern matches, extract and name various parts of the value, to finally evaluate some code which typically makes use of these named parts.

A seasoned object-oriented programmer might wonder why we did not define `eval` as a *method* of class `Tree` and its subclasses. We could have done it actually, since Scala allows method definitions in case classes just like in normal classes. Deciding whether to use pattern matching or methods is therefore a matter of taste, but it also has important implications on extensibility:

- when using methods, it is easy to add a new kind of node as this can be done just by defining the sub-class of `Tree` for it; on the other hand, adding a new operation to manipulate the tree is tedious, as it requires modifications to all sub-classes of `Tree`,
- when using pattern matching, the situation is reversed: adding a new kind of node requires the modification of all functions which do pattern matching on the tree, to take the new node into account; on the other hand, adding a new operation is easy, by just defining it as an independent function.

To explore pattern matching further, let us define another operation on arithmetic expressions: symbolic derivation. The reader might remember the following rules regarding this operation:

1. the derivative of a sum is the sum of the derivatives,
2. the derivative of some variable v is one if v is the variable relative to which the derivation takes place, and zero otherwise,
3. the derivative of a constant is zero.

These rules can be translated almost literally into Scala code, to obtain the following definition:

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

This function introduces two new concepts related to pattern matching. First of all, the **case** expression for variables has a *guard*, an expression following the **if** keyword. This guard prevents pattern matching from succeeding unless its expression is true. Here it is used to make sure that we return the constant 1 only if the name of the variable being derived is the same as the derivation variable v. The second new feature of pattern matching used here is the *wild-card*, written _, which is a pattern matching any value, without giving it a name.

We did not explore the whole power of pattern matching yet, but we will stop here in order to keep this document short. We still want to see how the two functions above perform on a real example. For that purpose, let's write a simple `main` function which performs several operations on the expression $(x+x)+(7+y)$: it first computes its value in the environment $\{x \to 5, y \to 7\}$, then computes its derivative relative to x and then y.

```
  def main(args: Array[String]): Unit = {
    val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")));
    val env: Environment = { case "x" => 5 case "y" => 7 };
    Console.println("Expression: " + exp);
    Console.println("Evaluation with x=5, y=7: " + eval(exp, env));
    Console.println("Derivative relative to x:\n " + derive(exp, "x"));
    Console.println("Derivative relative to y:\n " + derive(exp, "y"));
  }
```

Executing this program, we get the expected output:

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

By examining the output, we see that the result of the derivative should be simplified before being presented to the user. Defining a basic simplification function using pattern matching is an interesting (but surprisingly tricky) problem, left as an exercise for the reader.

7 Mixins

Apart from inheriting code from a super-class, a Scala class can also import code from one or several *mixins*.

Maybe the easiest way for a Java programmer to understand what mixins are is to view them as interfaces which can also contain code. In Scala, when a class inherits from a mixin, it implements that mixin's interface, and inherits all the code contained in the mixin.

To see the usefulness of mixins, let's look at a classical example: ordered objects. It is often useful to be able to compare objects of a given class among themselves, for example to sort them. In Java, objects which are comparable implement the `Comparable` interface. In Scala, we can do a bit better than in Java by defining our equivalent of `Comparable` as a mixin, which we will call `Ord`.

When comparing objects, six different predicates can be useful: smaller, smaller or equal, equal, not equal, greater or equal, and greater. However, defining all of them is fastidious, especially since four out of these six can be expressed using the remaining two. That is, given the equal and smaller predicates (for example), one can express the other ones. In Scala, all these observations can be nicely captured by the following mixin declaration:

```
abstract class Ord {
  def < (that: Any): boolean;
  def <=(that: Any): boolean = (this < that) || (this == that);
```

```
  def > (that: Any): boolean = !(this <= that);
  def >=(that: Any): boolean = !(this < that);
}
```

This definition both creates a new type called `Ord`, which plays the same role as Java's `Comparable` interface, and default implementations of three predicates in terms of a fourth, abstract one. The predicates for equality and inequality do not appear here since they are by default present in all objects.

The type `Any` which is used above is the type which is a super-type of all other types in Scala. It can be seen as a more general version of Java's `Object` type, since it is also a super-type of basic types like `int`, `float`, etc.

To make objects of a class comparable, it is therefore sufficient to define the predicates which test equality and inferiority, and mix in the `Ord` class above. As an example, let's define a `Date` class representing dates in the Gregorian calendar. Such dates are composed of a day, a month and a year, which we will all represent as integers. We therefore start the definition of the `Date` class as follows:

```
class Date(y: int, m: int, d: int) with Ord {
  def year = y;
  def month = m;
  def day = d;

  override def toString(): String = year + "-" + month + "-" + day;
```

The important part here is the **`with`** `Ord` declaration which follows the class name and parameters. It declares that the `Date` class inherits from the `Ord` class as a mixin.

Then, we redefine the `equals` method, inherited from `Object`, so that it correctly compares dates by comparing their individual fields. The default implementation of `equals` is not usable, because as in Java it compares object physically. We arrive at the following definition:

```
  override def equals(that: Any): boolean = {
    that.isInstanceOf[Date] && {
      val o = that.asInstanceOf[Date];
      o.day == day && o.month == month && o.year == year
    }
  }
```

This method makes use of the predefined methods `isInstanceOf` and `asInstanceOf`. The first one, `isInstanceOf`, corresponds to Java's `instanceof` operator, and returns true if and only if the object on which it is applied is an instance of the given type. The second one, `asInstanceOf`, corresponds to Java's cast operator: If the object is an instance of the given type, it is viewed as such, otherwise a `ClassCastException` is thrown.

Finally, the last method to define is the predicate which tests for inferiority, as follows. It makes use of another predefined method, `error`, which throws an exception with the given error message.

```
    def <(that: Any): boolean = {
      if (!that.isInstanceOf[Date])
        error("cannot compare " + that + " and a Date");

      val o = that.asInstanceOf[Date];
      (year < o.year)
        || (year == o.year && (month < o.month
                               || (month == o.month && day < o.day)))
    }
  }
```

This completes the definition of the `Date` class. Instances of this class can be seen either as dates or as comparable objects. Moreover, they all define the six comparison predicates mentioned above: `equals` and `<` because they appear directly in the definition of the `Date` class, and the others because they are inherited from the `Ord` mixin.

Mixins are useful in other situations than the one shown here, of course, but discussing their applications in length is outside the scope of this document.

8 Genericity

The last characteristic of Scala we will explore in this tutorial is genericity. Java programmers should be well aware of the problems posed by the lack of genericity in their language, a shortcoming which is addressed in Java 1.5.

Genericity is the ability to write code parametrised by types. For example, a programmer writing a library for linked lists faces the problem of deciding which type to give to the elements of the list. Since this list is meant to be used in many different contexts, it is not possible to decide that the type of the elements has to be, say, `int`. This would be completely arbitrary and overly restrictive.

Java programmers resort to using `Object`, which is the super-type of all objects. This solution is however far from being ideal, since it doesn't work for basic types (`int`, `long`, `float`, etc.) and it implies that a lot of dynamic type casts have to be inserted by the programmer.

Scala makes it possible to define generic classes (and methods) to solve this problem. Let us examine this with an example of the simplest container class possible: a reference, which can either be empty or point to an object of some type.

```
class Reference[a] {
  private var contents: a = _;

  def set(value: a): Unit = { contents = value; }
  def get: a = contents;
}
```

The class `Reference` is parametrised by a type, called `a`, which is the type of its ele-

ment. This type is used in the body of the class as the type of the `contents` variable, the argument of the `set` method, and the return type of the `get` method.

The above code sample introduces variables in Scala, which should not require further explanations. It is however interesting to see that the initial value given to that variable is `_`, which represents a default value. This default value is 0 for numeric types, **`false`** for the boolean type, `()` for the unit type and **`null`** for all object types.

To use this `Reference` class, one needs to specify which type to use for the type parameter `a`, that is the type of the element contained by the cell. For example, to create and use a cell holding an integer, one could write the following:

```
object IntegerReference {
  def main(args: Array[String]): Unit = {
    val cell = new Reference[Int];
    cell.set(13);
    Console.print("Reference contains the half of " + (cell.get * 2));
  }
}
```

As can be seen in that example, it is not necessary to cast the value returned by the `get` method before using it as an integer. It is also not possible to store anything but an integer in that particular cell, since it was declared as holding an integer.

9 Conclusion

This document gave a quick overview of the Scala language and presented some basic examples. The interested reader can go on by reading the companion document *Scala By Example*, which contains much more advanced examples, and consult the *Scala Language Specification* when needed.