CSci 555, Functional Programming
Notes on Scala for Java Programmers

Adapted by H. Conrad Cunningham

4 February 2016 (minor formatting udpate 3 August 2016)

Contents

Acknowledgements: In February 2016, Conrad Cunningham created these notes by adapting and expanding the web document "A Scala Tutorial for Java Programmers" by Michel Schinz and Phillipp Haller to better meet the needs of his Scala-based course. The original document can be found on the Scala Language website at A Scala Tutorial for Java Programmer.

A Scala Tutorial for Java Programmers

This is an introduction to Scala for programmers who have completed the equivalent of the Java-based Computer Science I-II-III (CSci 111-112-211) sequence at the University of Mississippi.

A First Example: Hello World

A "Hello, world!" program is the obligatory first example to give when introducing a new language. We can write a program `HelloWorld` as follows in Scala:

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

What Scala features do we use here in relation to Java?

- Keyword `object` declares a *singleton object* named `HelloWorld`. An object is essentially a class with a single instance. The body of the object is enclosed in braces following the name.
- The keyword `def` introduces a method definition.
- In a declaration, a colon (`:`) separates the name from its type.
- Method `main` takes the command line arguments as its parameter, which is an array of strings.
- The `main` method is a procedure and, hence, has no return type declared. The body of the method is enclosed in braces following the method header.
- The `main` method is not declared as `static` as in Java. Static members do not exist in Scala. We can use singleton objects instead.
- The body of `main` has a single call to predefined method `println`.

Compiling the example

We can use the `scalac` command (similar to the `javac` command) to invoke the Scala compiler. If the above Scala program is stored in file `HelloWorld.scala`, we can compile it from the command line as follows:

```
scalac HelloWorld.scala
```

The above compiles the Scala source file and generates a few class files in the current directory.

File `HelloWorld.class` contains a class that can be executed.

Running the example

We can use the `scala` command (similar to the `java` command) to execute the `main` method. Execution of the program prints the "Hello, World" string to the console.

```
> scala -classpath . HelloWorld

Hello, world!
```

Interaction with Java

Scala code can interact with Java code. Package `java.lang` is imported by default and other packages can be imported explicitly.

Consider a program to obtain and format the current date according to the conventions used in a specific country, say France.

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
  def main(args: Array[String]) {
    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format now)
  }
}
```

Java's class libraries such as `Date` and `DateFormat` can be called directly from Scala code.

The Scala `import` statement is more powerful than Java's. It can:

- Import multiple classes by enclosing names in braces.

  The first line above imports the `Data` and `Locale` classes from Java package `java.util`.

- Import everything in a package or class by using an underscore character (`_`) instead of a Java's asterisk (`*`).

  The third line imports all members of the `DateFormat` class, making static method `getDateInstance` and static field `LONG` visible.

The `main` method in the `FrenchDate` object:

- Creates an instance of Java's `Date` class, thus getting the current date

- Uses the imported static method `getDateInstance` to format the date appropriately for France using `Locale.France`
- Prints the current date formatted according to the localized `DateFormat` instance

The `main` method declares two local "variables" in its body: `now` and `df`. Note that:

- Both are declared with `val`. After initialization, Scala does not allow further assignments to a `val`.

  The alternative is `var`. Scala does allow assignments to a `var`; it is like an ordinary variable in Java.

  Although we cannot change the binding of a `val` to an object, Scala does allow the internal state of the object to be changed.

- Although Scala is statically typed like Java, neither of these variables are given explicit types. The types of the variables are *inferred* by the compiler from the type of the initializing expression.

Scala methods taking one explicit argument can be written in infix syntax such as

```
df format now
```

which is the same as the method call

```
df.format(now)
```

This feature has important consequences, as we discuss below.

It is also possible to inherit from Java classes and implement Java interfaces directly in Scala.

Everything is an Object

Scala is a pure object-oriented language. *Everything* is an object, including numbers and functions.

Scala does not distinguish primitive types (e.g., `boolean` and `int`) from reference types (e.g., objects). It enables us to manipulate functions as first-class values.

Numbers are objects

Since numbers are objects, they also have methods. For example, the expression

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

is equivalent to the expression

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

which shows the method calls explicitly.

In Scala, the "operator" symbols (e.g., `+`, `*`, `/`) are valid identifiers. The parentheses around numbers are necessary because Scala's lexical analyzer uses a longest-match rule for tokens, breaking expression

```
1.+(2)
```

into the tokens `1.`, `+`, and `2`. This results in `1.` being interpreted as a `Double`.

Functions are objects

Because Scala functions are objects, we can:

- pass a function as an argument to or return a function as a result from another function.

  That is, Scala has *higher-order* functions.

- store a function in a variable or data structure.

  That is, Scala's functions are *first-class* values.

These are key features of *functional programming*.

Consider the `Timer` program below. It includes a timer function named `oncePerSecond` that performs some action every second. The specific action performed is encoded as a *call-back* function passed into `oncePerSecond` as an argument.

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

The type of the call-back function is `() => Unit`. This means it is a function that takes no arguments and returns nothing. The type `Unit` is similar to `void` in C/C++.

Function `oncePerSecond` calls the `Thread.sleep` method (from `java.lang`) and uses an infinite `while` loop to repeat the call-back action every second.

The `main` function calls this timer function, passing a call-back function that prints the string "time flies like an arrow..." on the console. The program endlessly prints this string every second.

The program uses the predefined Scala method `println` instead the like-named Java method from `System.out`.

Anonymous functions

The `Timer` program in the subsection above can be refined by replacing the function `timeFlies` by an *anonymous function* as shown below:

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

In the anonymous function, a right arrow `=>` separates the function's argument list from its body expression.

In this example, the argument list is empty, shown by the empty pair of parentheses on the left.

Classes

Like Java, Scala is a class-based language. However, Scala classes can have parameters. Consider the following class `Complex` for representing complex numbers:

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

The `Complex` class takes two arguments, the real and imaginary parts of the complex number. In the `Complex` class, these become `val` fields of the object (i.e., class instance) that are only visible and accessible from inside the object. (This visibiliy is `private[this]`, a type of visibiliy that does not occur in Java.)

The default constructor is built into the `class` syntax. We pass values for these arguments when we create an instance of class `Complex`, as follows:

```
new Complex(1.5, 2.3)
```

This causes all statements in the class definition to be executed.

The `Complex` class defines function methods `re` and `im`, which provide access to the two parts of the complex number. That is, these are *accessor*, or *getter*, methods.

We declare a function with an `=` between the function's header (i.e., name and parameter list) and its body. The body is an expression that is evaluated when the function is called. If the body itself consists of a sequence of expressions, we enclose it in braces. The value of last expression executed is the value of the body.

In `Complex`, the compiler infers the return types for methods `re()` and `im()` by examining the right-hand sides and deducing that both return a value of type `Double`. The compiler gives an error message when it cannot infer the type of a method or variable.

Some Scala programmers suggest that the types be omitted whenever possible and only inserted when necessary.

However, your instructor considers it better software engineering practice to explicitly specify the types of all *public* features of a class or package such as the `re()` and `im()` methods.

But, for internal features of a class or method, it is convenient and safe to use type inference. For example, we did not give explicit types for the `now` and `df` values in the `FrenchDate` object shown in a previous section.

Methods without arguments

To call the methods `re` and `im`, we must put an empty pair of parentheses after their names:

```
object ComplexNumbers {
  def main(args: Array[String]) {
    val c = new Complex(1.2, 3.4)
    println("imaginary part: " + c.im())
  }
}
```

We can eliminate the empty parameter list by defining `re` and `im` as methods *without arguments* instead of *with zero arguments*. We simply omit the parentheses in the method definition, as shown below:

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

Inheritance and overriding

All classes in Scala inherit from a superclass. If no super class is given explicitly, then the superclass is `scala.AnyRef` by default.

Like Java, a Scala class inherits all methods from its superclass by default.

Like Java, a Scala class may override individual methods by giving new definitions.

Unlike Java, a Scala method that overrides a superclass method must be explicitly declared with the `override` modifier. This is to reduce the possibility of class overriding a superclass method by accident.

For example, the`Complex` class can redefine the `toString` method inherited from `AnyRef`.

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

The new definition of `toString()` overrides the definition for `Complex` objects. However, in doing so, it uses the default definitions of `toString()` for `Double` by its references to `re` and `im` in the string concatenation.

Case Classes and Pattern Matching

In programming, we often use trees and other hierarchical data structures.

We can illustrate how to implement a tree in Scala using a small calculator program for simple arithmetic expressions composed of sums, integer constants, and variables. Examples of such expressions are `1+2` and `(x+x)+(7+y)`.

We can represent expressions naturally with a tree, where nodes are operations (e.g., addition) and leaves are values (e.g., constants or variables).

In Java, we can represent such a tree using an abstract superclass for the trees and a concrete subclass for each kind of internal and leaf node. The abstract class defines the common interface to the tree and the concrete subclasses define the specific features of a node.

In functional programming languages such as Haskell, we can define an *algebraic data type* for the same purpose. These types often enable us to express programs concisely by using pattern matching constructs.

Scala combines the concepts of classes and algebraic data types into its *case classes*. Consider the 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
```

Here, we define `Sum`, `Var` and `Const` as case classes. These differ from from standard classes in several ways:

- We can create an instance without using the keyword `new` keyword, for example, we can write `Const(5)` instead of `new Const(5)`.
- The compiler automatically generates getter functions for the constructor parameters. That is, it is possible to get the value of the `v` constructor parameter of some instance `c` of class `Const` just by writing `c.v`.
- The compiler supplies default definitions for the methods `equals` and `hashCode`. These work on the *structure* of the instances and not on their identities.
- The compiler provides a default definition for method `toString`, which prints the value in a "source" form. For example, 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 see below.

These features enable us to use Scala case classes much like we would use algebraic data types in a functional language.

To explore the use of case classes, consider a function to evaluate an expression in some *environment*. The purpose of an environment is to associate values with variables.

For example, the expression `x+1` evaluated in an environment that associates the value `5` with the variable `x`, written `{ x -> 5 }`, gives `6` as result.

An environment is just a function that associates a value with a (variable) name. The environment `{ x -> 5 }` given above can be written as a Scala function as follows:

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

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

An environment is a function of type `String => Int`. To simplify our evaluation program, we define the name `Environment` to be an alias for this type using the following declaration:

```
type Environment = String => Int
```

We can now define the evaluation function in Scala as follows:

```
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 performs *pattern matching* on the tree `t` (i.e., using the `match` operator).

1. The pattern match first checks whether the tree `t` is a `Sum` object (i.e., an instance of the `Sum` case class).

   If the tree is a `Sum`, the pattern match binds the left subtree to a new variable `l` and the right subtree to a new variable `r`, then evaluates the expression following the arrow `=>`.

   This expression uses the values of the variables bound by the pattern match, i.e., `l` and `r` in the `Sum` case.

2. If the first check does not succeed, that is, the tree is not a `Sum` object, the pattern match then checks whether `t` is a `Var` object.

   If the tree is a `Var`, the pattern match binds the name contained in the `Var` node to a new variable `n` and then evaluates the right-hand-side expression.

3. If the second check also fails, that is, if `t` is neither a `Sum` nor a `Var`, the pattern match then checks whether the expression is a `Const` object.

   If the tree is a `Const`, the pattern match binds the value contained in the `Const` node to a new variable `v` and then evaluates the right-hand-side expression.

4. Finally, if all checks fail, the pattern match expression raises an exception to signal failure. In this version of `eval`, failure occurs only when there are additional subclasses of `Tree` that we have declared but not yet defined in the `eval` pattern match.

Pattern matching attempts to match a value to a series of patterns. As soon as a pattern matches (moving top to bottom, left to right in the source code), the program extracts and names various parts of the value and then evaluates the associated expression using the values of these named parts.

An object-oriented programmer might ask why we did not define `eval` as a *method* of class `Tree` and its subclasses.

We could do that, because Scala allows method definitions in case classes just as in normal classes.

Deciding whether to use pattern matching or methods is partly a matter of taste. But the choice also affects the extensibility the program.

- Using methods, we can easily add a new kind of node by defining a new subclass of `Tree`. But adding a new operation to manipulate the tree is tedious because it requires us to modify every subclass of `Tree`.
- Using pattern matching, the situation is reversed: adding a new kind of node requires us to modify all functions that do pattern matching on the tree, to take the new node into account. But adding a new operation is easy, we just define it as an independent function.

To explore pattern matching further, consider another operation on arithmetic expressions: symbolic derivation. Looking back at our calculus class, we see the following rules for differentiation:

1. The derivative of a sum is the sum of the derivatives.
2. The derivative of some variable `v` is 1 if `v` is the variable relative to which the derivation takes place, and is 0 otherwise.
3. The derivative of a constant is 0.

We can directly translate these rules into a Scala function that uses the above case classes and pattern matching, as follows:

```
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)
}
```

Function `derive` introduces two new concepts related to pattern matching.

1. 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 the guard ensures that the function returns the constant `1` only if the name of the variable being derived is the same as the derivation variable `v`.
2. A pattern can include a *wildcard*, written `_`, that matches any value, without giving it a name.

Consider an example with a simple `main` function that performs several operations on the expression `(x+x)+(7+y)`.

It first computes its value in the environment `{ x -> 5, y -> 7 }` and then computes its derivative relative to `x` and then to `y`.

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

```
  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)))
```

The result of the derivative is complex. It should be simplified before printing. Defining a basic simplification function using pattern matching is an interesting (but surprisingly tricky) problem, left as an exercise for the reader.

Traits

In addition to inheriting code from a superclass, a Scala class can also reuse code from one or several *traits*.

From a Java perpective, traits can be viewed as interfaces that can also contain code. In Scala, when a class inherits from a trait, it implements that trait's interface, and inherits all the code contained in the trait.

Note: In older Java implementations, an interface consisted of just method signatures and constants. In Java 8, interfaces can now include default definitions of methods, somewhat similar to Scala traits.

To see the usefulness of traits, consider a classic example: ordered objects. We want to compare objects of a given class among themselves. For example, we need to compare objects according to some total order to sort them.

In Java, objects that can be compared implement the `Comparable` interface.

In Scala, we can do better by defining the equivalent of `Comparable` as a trait, which we call `Ord`.

When comparing objects, six different comparison operations are useful: smaller, smaller or equal, equal, not equal, greater or equal, and greater.

But it is tedious to define all of these for every class whose instances we wish to compare.

We observe that we can define four of the six in terms of the other two. For example, given the equal and smaller comparison operators, we can define the other four comparison operators.

In Scala, we can capture this observation in the following trait declaration:

```
trait 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 generates default implementations of three comparison operators in terms of a fourth, abstract operator. All classes inherit the equality and inequality operators and, thus, those operators do not need to be defined in `Ord`.

The type `Any` used above, is the supertype of all other types in Scala. It is essentially a more general version of Java's `Object` type that is also a supertype of basic types like `Int`, `Float`, etc.

To make objects of a class comparable, it is sufficient to define equality and inferiority operators and then "mix in" the `Ord` trait.

For example, consider a `Date` class representing dates in the Gregorian calendar. Such dates are composed of a day, a month, and a year, each of which we can represent with an integer. We can start the definition of the `Date` class as follows:

```
class Date(y: Int, m: Int, d: Int) extends Ord {
  def year  = y
  def month = m
  def day   = d
  override def toString(): String = year + "-" + month + "-" + day
  // ... two methods defined below
}
```

The `extends Ord` declaration specifies that the `Date` class inherits from the `Ord` trait (in addition to the default superclass).

To make the comparisons work correctly, we redefine the `equals` method, inherited from Java's `Object` class, to compare dates by comparing their individual fields. The default implementation of `equals` does not work because it compares objects physically. In Scala, the comparison becomes:

```
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 uses the predefined methods `isInstanceOf` and `asInstanceOf`.

- `isInstanceOf` corresponds to the Java `instanceof` operator. It returns true if and only if the object to which it is applied is an instance of the given type.

- `asInstanceOf` corresponds to the Java cast operator: if the object is an instance of the given type, it is viewed as such, otherwise it throws a `ClassCastException`.

To complete our implementation, we need to define the `<` operator. It uses the predefined Scala method `error` to throw 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 the `Date` class can be seen either as dates or as comparable objects. They define all six comparison operators:

- `equals` and `<` because they appear directly in the definition of the `Date` class
- the other four because they are inherited from the `Ord` trait

Traits are useful in many other situations, as you will see as you program more in Scala.

Genericity

Scala supports generics. Java did not support generics until Java 5.

Genericity is the ability to write code parameterized by types.

For example, suppose we are writing a library for linked lists. What type do we give the elements of the list?

- If we choose a specific concrete type such as `Int`, we severely limit the usefulness of the library. It is impractical to include different implementation for every possible concrete type.
- If follow choose a default supertype like `Any`, then users of our library would need to their lace code with many type checks and type casts. (In Java, there was also the problem that basic types do not inherit from `Object`.)

Scala supports generic classes and methods to solve this problem. As an example, consider the simplest possible container class: a reference, which can either be empty or point to an object of some type.

```
class Reference[T] {
  private var contents: T = _
  def set(value: T) { contents = value }
  def get: T = contents
}
```

This example parameterizes the class `Reference` with a type `T`, which is the type of its element. The body of the class declares the `contents` variable, the argument of the `set` method, and the return type of the `get` method to have type 'T'.

This is our first example to use mutable variables declare with `var`. In this example, we initialize the `contents` variable to have the value `_`, which represents the default value for the type. 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, we need to specify what type to use for the type parameter `T`, that is, the type of the element contained in 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]) {
    val cell = new Reference[Int]
    cell.set(13)
    println("Reference contains the half of " + (cell.get * 2))
  }
}
```

This example does not need 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, because it was declared as holding an integer.

Conclusion

This document has provided a quick overview of Scala for programmers familiar with Java. To learn more, see the course textbooks and other tutorials, manuals, and books.