Introduction to Scala Syntax

With emphasis on the functional way of doing things

Objects

- All code is in either an *Object* or a *Class*. But we will start with Objects because you always need at least one *Object* to run a Scala program:
  - An object takes no parameters and doesn’t need to be a companion to a class;
  - An object which extends *App* invokes its initialization code within the (invisible) *main* method.
  - The *main* method does not yield a value (technically, it yields *Unit*) so, unlike everywhere else in Scala, it must contain side-effects, otherwise it would do nothing..

```
object Newton extends App {
  val newton = Newton("cos(x)-x", x => math.cos(x) - x, x => -math.sin(x) - 1)
  newton.solve(10, 1E-10, 1) match {
    case Success(x) => println(s"""The solution to "$newton=0" is $x""")
    case Failure(t) => System.err.println(s"""$newton unsuccessful: $  {t.getLocalizedMessage}”"")
  }
}
```

Classes (1)

- Classes
  - Let’s think a little about what a class actually is.
  - In programming, a class represents a *category*.
  - By definition, a class has members (instances), all of which conform to the class. So, there has to be some aspect of each member which distinguishes it from the other members.
  - Let’s take an example where a class has just one field (this type of class is typically called a wrapper). For instance, in Java, the *Integer* class has just one field whose value is an *int*.
  - Different instances of the *Integer* class have different values of the *int* field, otherwise they would represent the <u>same thing</u>.
  - We could potentially have four billion (2^{32}) different instances of *Integer*.

Classes (2)

- Classes
  - Like objects, classes can have initialization code (what would be in a Java constructor or within {}). But generally, all the useful code of a class is in one of its methods.
  - All fields and methods of a class are *instance* fields/methods. If you want “class” fields/methods then you need to declare a companion object (one with the same name and in the same module).
  - A class generally takes both value parameters and type parameters; Unless it is a “case” class, you will need to invoke the constructor using the *new* keyword.

```
case class Newton(w: String, f: Double => Double, dfbydx: Double => Double) {
  override def toString: String = w
  private def step(xy: Try[Double], yy: Try[Double]) = for (x <- xy; y <- yy) yield x - y / dfbydx(x)
  def solve(tries: Int, threshold: Double, initial: Double): Try[Double] = {
    @tailrec def inner(ry: Try[Double], n: Int): Try[Double] = {
      val yy = for (r <- ry) yield f(r)
      (for (y <- yy) yield math.abs(y) < threshold) match {
        case Success(true) => ry
        case _ =>
       if (n == 0) Failure(new Exception(s"failed to converge in $tries tries, " +
         s"starting from x=$initial and where threshold=$threshold”))
          else inner(step(ry, yy), n - 1)
      }
    }
    inner(Success(initial), tries)
  }
}
```

Modules

- The code in one file (module) is treated like being in its own package.
  - Privacy rules apply at the module level.
  - A module may contain any number of traits, classes and objects.

```
sealed trait Foo {
  def a: String
  def create(a: String): Foo
}

case class Bar(a: String,b: Option[Int]) extends Foo{
  def create(a: String) = Bar(a,None)
}

case class Buzz(a: String, b: Boolean) extends Foo {
  def create(a: String) = Buzz(a,false)
}
```

Abstract methods simply lack an expression

Sealed traits can be extended only within the module

Traits

- A trait defines some behavior (something like an interface in Java):
  - Traits have type parameters (typically) but cannot have value parameters.
  - Methods and fields of traits can have concrete values.
  - A trait (usually) cannot be instantiated (but if all properties are concrete, you could write **val** *s* = **new** Silly {} or something like that).
  - A trait which may only be extended *in-module* is marked as “sealed”.

```
sealed trait TraitExample[T] extends Comparable[TraitExample[T]] {
  def name: String
  def property: T
  def compareTo(o: TraitExample[T]): Int = name.compareTo(o.name)
  def >(o: TraitExample[T]): Boolean = compareTo(o)>0
  def <(o: TraitExample[T]): Boolean = compareTo(o)<0
  def >=(o: TraitExample[T]): Boolean = compareTo(o)>=0
  def <=(o: TraitExample[T]): Boolean = compareTo(o)<=0
  def ==(o: TraitExample[T]): Boolean = compareTo(o)==0
}
case class Telephone(name: String, number: String) extends TraitExample[String] {
  override def property: String = number
}
case class Age(name: String, age: Int) extends TraitExample[Int] {
  override def property: Int = age
}
```

Expressions

- So, now we know where we can write code, what sort of code can we write?
- Basically, we will write expressions:
  - An expression yields a result (of some type, including “Unit”, a non-result);
  - An expression can be preceded by definitions of “memoizing” variables;
  - An expression can be preceded (or followed) by definitions of methods;
  - An expression can be preceded by *import* statement(s) which allow us essentially to create aliases of types;
  - An expression is a series of identifiers/literals/method invocations interspersed with operators;
  - When a method invocation takes parameters, the values of those parameters will also be expressions.

Variable definitions

- We use the word “variable” in the sense of a mathematical identifier of an expression.
- The following are examples of variable definitions:
  - `val x = Math.PI`
  - `val x: Double = Math.PI`
  - `val x = Math.PI/2 + 1`
  - `var x = 0`
  - `lazy val x = connection.get(“date”)`

Method definitions

- The following are examples of method definitions:
  - `def x = Math.PI`
  - `def x: Double = Math.PI/2 + 1`
  - `def x(s: String) = connection.get(s)`

  -
    ```
    def x(s: String) = {
      val connection = makeConnection(“myServer”)
      val r = connection.get(s)
      connection.close()
      r
    }
    ```

Val vs. Def?

- So, what’s the real difference between *val* and *def*?
  - **def** (deferred/lazy evaluation) can be parameterized therefore its “value” is really a function which will be evaluated at some later time when those parameters are actually defined (we call this a *method invocation*). Even if it doesn’t take any parameters, it still gets evaluated when invoked, not when defined.
  - **val** (eager evaluation) cannot be parameterized and its value is evaluated immediately.
  - See my answer on Quora.

Control flow?

- OK, that's great but what about control flow?
- Well, in a functional programming language, we define expressions, we don't put together a series of statements interspersed with control flows.
- But what about a simple *if*?

  -
    ```
    if (x>=0) x else -x
    ```

An "if" clause must always have an "else"

- And what about some kind of switch?

  -
    ```
    def length(xs: Seq[X]): Int = xs match {
       case Nil => 0
       case _ :: t => length(t) + 1
     }
    ```

This is is called pattern-matching and is *much* more powerful than a switch statement in Java

- And what about some kind of loop?

  -
    ```
    for (x <- xs) yield x * 2
    ```

  -
    ```
    for (x <- xs) println(x)
    ```

  -
    ```
    xs foreach println
    ```

A "for comprehension" with "yield" always returns a result of the same "shape" as its generator (*xs*)