# Functional programming principles in Scala

## Week 2

### 2.1 – Tail recursion

* Def f(x1, ….., xn) = B. A call f(v1,……,vn) is rewritten to [x1/v1,….,/x1/v2]B using the rewriting rule, where [x1/v1,….,/x1/v2]B means the expression B with all occurences of xn replaced with vn.
* If a function calls itself as its last action, the function’s stack frame can be reused. This is called tail recursion.
* In general, if the last action of a function is a call to another function, one stack frame is sufficient for both functions. Such calls are tail calls.
* A tail-recursive function is the recursive version of a loop and just as efficient.
* Tail recursion is worth using on functions that are deeply recursive. Not doing so could cause one to exceed the allowed number of stack frames i.e. cause a stack overflow.   
  A function may be explicitly marked as tail-recursive by using the @tailrec annotation.   
  An error is generated if the implementation of such a function turns out not to be tail-recursive.
* Wherever possible, one should strive for readability over optimisation. Avoid using tail-recursive functions if there’s no real need for them and their implementation is harder to understand.

### 2.2 – Higher order functions

1. Functional languages treat functions as first-class values. This means that, like any other value, functions can be passed in as parameters or be the result of other functions. Such functions are called higher-order functions.
2. E.g. can be defined as a higher-order function in Scala:   
   def sum(f : Int => Int, a: Int, b: Int) = if(a > b) 0 else f(a) + sum(f, a+1, b)
3. Passing functions as parameters leads to the creation of many small functions. Scala permits functions to be defined as literals i.e. without an identifier, making them anonomous, just like how one might pass a string literal as an argument instead of a String reference.
4. An anonomous function *(x1 : T1, …., xn:Tn)=> E* can be expressed using def as follows:  
   *def f(x1 : T1, …., xn:Tn)=>E; f*.
5. Anonymous functions are a form of syntactic sugar – they provide a shorthand way of constructing expressions without adding to the language’s fundamental expressive power.
6. Example: sumCube(a,b) = sum((x: Int) => x\*x\*x, a, b)

### 2.3 – Currying

1. Can we eliminate redundancy in the sum function by implicitly assuming the lower/upper bound parameters, *a* and  *b*?  
   Sure, by defining a function that returns a function, as follows:  
   def sum(f: Int => Int) = {  
    def sumF(a: Int, b : Int) = {  
    if (a > b) 0 else a + SumF(a+1, b)   
    {  
   sumF  
   }
2. sumCubes(a,b) is equivalent sum(cube)(a,b).
3. The definition of functions that return functions is so useful in functional programming that there is a special syntax for it in Scala:  
   def sum(f: Int => Int)(a : Int, b: Int) : Int =  
    if (a > b) 0 else f(a) + sum(f)(a+1, b)
4. In general, a definition of a function with multiple parameter lists:  
   def f(args1)(args2)…(argsn) = E.  
   f(args1)…(argsn-1) rewrites to {def g(argsn) = E; g}, or for short, (argsn)=>E
5. By repeating the process n times, def f(args1)(args2)…(argsn) = E is shown to be equivalent to:  
   def f = (args1 =>(args2 =>…(argsn => E)…))

## Week 3

### 3.2 – Evaluation and operators

1. We previously defined the evaluation of functions using a computational model of substitution. How do we extent this model to classes and objects?
2. Instantiation of a class is evaluated no differently from any function: [v1/e1…,vn/en]new C is already a value. An infix call to a class method is evaluated using **3** substitutions. The expression  
   new C(v1,…,vm).f(w1,…,wn) becomes [w1 /y1 ,…,wn,yn] [v1 /xm ,…,vn,xm][ new C(C(v1,…,vm)/this] b
   1. The substitution of the formal parameters y1 ,…, yn of the function f with w1 ,…, wn
   2. The substitution of the formal parameters x1 ,…, xn of the class definition with w1 ,…, wn
   3. The substitution of the self reference this by the value of the object C(v1,…,vm).
3. Any method with a parameter can be used like an infix operator, so it is therefore possible to write r.add(s) simply as r add s. Secondly, any operator symbols, such as +, \*, can be overloaded and used as identifiers. Thus, the add method can have the identifier +. R add s becomes r + s.
4. The same operator can be overloaded to be used both as a prefix operator and an infix operator. E.g. In the Rational class the neg method can given the unary\_- operator identifier. This denotes a prefix, or unary, - operator. The sub operator can be given the identifier, -. Now, a-b + -b is a valid expression.
5. The precedence of an operator is determined by its character and is effectively the same as the precedence rules in Java.

### 3.5 – How classes are organised

1. Classes and objects can be imported from other packages. The syntax is akin to java, with some minor differences: To import all definitions in one class, one would use **import** Rational.\_ ;  
   several definitions can be imported inline using curly braces – import Rational.{foo,bar}
2. In scala, a class can only inherit from one base class, just like in Java. But what if a class has several natural supertypes to which it conforms to or from which it wants to inherit code?   
   Here, we can use traits. Traits are defined using the **trait** keyword, abstract classes use **abstract**  
   E.g. class Square extends Shape with Planar with Movable….
3. Traits are much like interfaces in Java, but are much more powerful because they can contain fields and concrete methods. On the other hand, traits cannot cantain (value) parameters, only classes can.
4. At the top of the type hierarchy we find
   1. *Any* – The base type of all types  
      Methods: -, +, ==, !=, equals, hashcode, toString
   2. *AnyVal* – The base type of all primitive types
   3. *AnyRef* – The base type of all reference non-primitive types (alias of Java.Lang.Object)
5. *Nothing* is at the bottom of Scala’s type hierarchy. It is a subtype of every other type. There is no value of type Nothing, it’s merely useful for signalling abnormal termination or or as an element type of empty collections.
6. Every reference class type has a null value, of type *Null.*This is a subtype of all reference types, but is incompatible with subtypes of AnyVal.

## Week 4

### 4.1 – Polymorphism

1. Prefixing the parameters of a class constructor with **val** defines them both as parameters and fields of the class. More syntactic sugar…
2. Type parameters can be used to define a class hierarchy that applies over varying types. E.g.defining a hierarchy of List classes that hold Integers is useless if we want to handle other types. This is where parameterisation of types comes in. This concept is analogous to generics in Java.
3. Like classes, functions can have type parameters.
4. Like in Java, type parameters do not affect evaluation at runtime (via type erasure) and are only used by the compiler to verify additional behavioural properties imposed on classes.
5. In functional programming, **polymorphism** means that a function comes “in many forms”.  
   This means that:
   1. The function can be applied to many types.
   2. The type can have instances of many types.
6. We have seen 2 main forms of polymorphism
   1. Subtyping: Instances of a class can be supplied to a base class.
   2. Generics: instances of a function or class created by type parameterisation.

### 4.2 – Objects everywhere

1. A pure object-oriented language is one in which every value is an object. If the language is based on classed this means that the type of every value is a class. Is Scala a pure object-oriented language?
2. Conceptually, types like *Boolean* or*Integer* do not receive any special treatment in scala. They are like other classed, defined in the package *scala*.

### 4.3 – Functions as Objects

1. We have seen that Scala’s numeric types and the Boolean type can be implemented like normal classes.
2. The function type A => B is just an abbreviation for the class scala.Function1[A, B], which is deﬁned as follows.  
    package scala
3. The function type A => B is just an abbreviation for the class scala.Function1[A, B], which is deﬁned as follows.  
   package Scala  
   trait Function1[Int, Int] { def Apply(x: Int) = x \* x}
4. There are also traits Function2, Function3, … for functions which take more parameters (currently up to 22).
5. There are also traits Function2, Function3, … for functions which take more parameters (currently up to 22).
6. An anonomous function such as (x: Int) => x\* x is expanded to:  
   {  
    class AnonFun extends Function1[Int, Int] { def Apply(x: Int) = x \* x  
   new AnonFun  
   }  
   or shorter, using anonomous class syntax:  
   new Function1[Int, Int] { def Apply(x: Int) = x \* x}
7. val f = (x: Int) => x \* x  
   f(7)

would be expanded to:  
val f = new Function1[Int, Int] { def Apply(x: Int) = x \* x}  
f.apply(7)

1. Note that a function definition is not itself a function value, but is converted automatically to the function value wherever it is used

### 4.4 – Subtyping and Generics.

1. 2 Principal forms of polymorphism: Subtyping and Generics.
2. Bound type parameters can be set using the :> (*super* in java) and <: (*extends* in java) operators.  
   E.g. def assertAllPos[S <: IntSet](r: S): S = ...
3. It is possible to mix a lower bound with an upper bound:   
   For instance, [S >: NonEmpty <: IntSet]
4. If A <: B, does this imply that SomeObject[A] <: SomeObject[B]? Where SomeObject is List, this holds. Types for which this relationship holds are *covariant*. Where the A <: B => SomeObject[A] >: SomeObject[B], SomeObject is *contravariant.*
5. In order for one type, A, to be a subtype of another type, B, the types must satisfy the Liskov Substitution principle: if a property q(x) is provable for objects x of type B, q(y) is provable for objects of type B.

### 4.5 – Variance

1. Unlike in Java, List is covariant in scala because Its immutability ensures that the Liskov Substitution Principle is satisfied.
2. Example: type A = IntSet => NonEmpty, B =NonEmpty => IntSet  
   A <: B holds.
3. Generally, we have the following rule for subtyping between function types:  
   if A2 <: A1 and B1 <: B2, then A1 => B1 <: A2 => B2.  
   So functions are contravariant in their argument types and covariant in their return types.
4. This leads to the following revised definition of the Function1 trait:  
   package scala  
   trait Function1[-T, +U]{  
    def apply(x : T) : U  
   }
5. The scala compiler will check that there are no problematic combinations when comping a class with variance annotations.  
   Roughly
   1. Covariant type parameters can only appear in method results
   2. Contravariant type paramaters can only appear in method parameters
   3. Invariant type parameters can appear anywhere.
6. Trait List[+T] {  
    def prepend(elem: T) : List[T] = new Cons(elem, this),  
   does not compile because it violates the LSP.  
   We can make prepend variance-correct by using a lower bound:  
    def prepend(U :> T) : List[U] = new Cons(elem, this).  
   This passes variance checks because
   1. Covariant type parameters may appear in lower bounds of a method.
   2. Contravariant type parameters may appear in upper bounds of a method.

### 4.6 & 4.7 – Decomposition

1. How do we find a general **and** convenient way to access objects in an extensible class hierarchy?
2. In 4.6, we tried several ways of decomposing arithmetic expressions:
   1. *Classification and access methods –* quadratic explosion
   2. *Type tests and casts –* low-level and potentially unsafe
   3. *Object-oriented decomposition* – does not always work, we needed to touch all classes to implement a show method on the arithmetic expression. This forces us to go back to a)
3. Observation: The sole purpose of test and acccessor functions is to **reverse** the construction process.
   1. Which subclass was used?
   2. What where the arguments of the constructor?
4. This situation is so common that many functional languages, Scala included, automate it
5. A *case class* definition is similar to a normal class definition, except it is preceded by the modifier *case.*
6. A case class definition also implicitly defines an object with an *apply* method.  
   E.g. *case class Number(n: Int) extends Expr* also defines *Object Number(n: Int) {def apply(n:Int) = new Number(n)}.* So we also end up with a function expression that acts as a factory for the class.
7. Pattern matching is a generalisation of *switch* to class hierarchies.  
   For example, def eval(e : expr) : Int = e **match** {  
    case Number(n) => n  
    case Sum(e1, e2) => eval(e1) + eval(e2)  
   }
8. An expression of the form e match {case p1 => e1 … case pn => en} matches the value of the selector e with the patterns pa,…,p2 in the order that they are given.
9. A pattern can contain
   1. *Constructors,* e.g. Number, Sum
   2. *Variables* e.g n, e1, e2
   3. *Wildcards*, \_
   4. *Constants,* e.g. 1, true

## Week 5

### 5.1 and 5.2 – Lists

1. There are 2 important differences between lists and arrays –
   1. Lists are immutable.
   2. Lists are recursive, arrays are flat.
2. Lists are homogenous, that is all the elements of a list must have the same type.
3. All lists are constructed from the empty list, *Nil,*  and the construction operation :: (pronounced cons) x :: xs gives a new list with x as its head and xs as its tail.
4. :: associates to the right, so 1::2::3::Nil is equivalent to Nil.::2.::2.::1
5. All operations on a list can be expressed in terms of the fundamental operations *head, tail, isEmpty.*
6. It is also possible to decompose lists with pattern matching.
   1. *Nil –* The empty list
   2. *p :: ps -*  List with a pattern matching p and a pattern matching ps
   3. *List(p1,…*,pn) – Same as p1…::pn::Nil

### 5.3 – Pairs and Tuples

1. The pair consisting of x and y is written (x,y) in Scala.  
   E.g. val pair = (“answer”, 42)
2. Pairs can also be decomposed using patterns:  
   val (label, value) = pair, assigns “answer” to label and 42 to value.
3. A tuble type (T1,…,Tn) is an abbreviation of the parameterised type  
   scala.Tuplen[T1,…,Tn]
4. All Tuplen classes are modelled on the following pattern:  
   case class Tuple2[T1, T2](\_1: +T2, \_2: +T2){  
    override def toString = “(“+ \_1 + “,” + \_2 +”)”  
   }
5. The fields of a tuple can be accessed with the the names \_1, \_2,…   
   The pattern method in 2) is preferred as it’s more concise and legible.

### 5.4 – Implicit parameters

1. If a function parameter of type T has the *implicit* modifier, then the compiler will search for an implicit definition that:
   1. Is marked *implicit*
   2. Has a type compatible with type T
   3. Is visible at the point of the function call, or is defined in a companion object associated with T.
2. If there is a single (most specific) definition, it will be taken as the actual argument for the implicit parameter. Otherwise it’s an error.
3. There is a working example that makes use of this concept in week4.MergeSort.

### 5.5 – Higher order list functions.

1. We can identify several recurring patterns like,
   1. Transforming each element in a list in a certain wwy.
   2. Returning all elements in a list that satisfy certain criteria
   3. Combining elements in a list using an operator.
2. Functional languages allow programmers to implement patterns like this using higher-order functions.
3. There are a couple of examples in week5.HigherOrderListFunctions.

### 5.6