

Functional Programming

Functional Programming

Week1 :

call by value vs call by name :

Conditionals :

Blocks , Scopes and first functional program :

Tail recursion :

Week 2 :

Currying :

Classes :

Substitution and Extensions :

1. Substitution :

2. Extension methods (1) :

3. Operators :

Week3 :

Abstract classes :

Object definition :

Programs :

Traits :

Cons List :

A complete definition with **Generics** :

Generic functions :

Pure Object Orientation :

Week 4 :

Decomposition :

Solution : Pattern Matching :

Lists and more pattern matching :

Enum :

Type Bounds :

Variance :

Week1:

call by value vs call by name :

Call-by-value : evaluate the parameters *then* execute the function on these values.

e.g :

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4) // evaluated 2+2 BEFORE executing sumOfSquares
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9 + 4 * 4
9 + 16
25
```

Call-by-value has the advantage that it evaluates every function argument only once.

Call-by-name : Apply the function to unreduced arguments.

e.g :

```
sumOfSquares(3, 2+2)
square(3) + square(2+2) // executing sumOfSquare before evaluating 2+2
3 * 3 + square(2+2)
9 + square(2+2)
9 + (2+2) * (2+2)
9 + 4 * (2+2)
9 + 4 * 4
```

Call-by-name has the advantage that a function argument is not evaluated if the corresponding parameter is unused in the evaluation of the function body.

e.g :

```
def loop : Int = loop
def f(a:Int , b : Int ) : Int = a // Call-by-value
f(1,loop) // does not terminate
def f(a:Int, b : => Int) : Int = a // Call-by-Name ( => )
f(1,loop) // terminates
```

Both strategies reduce to the same final values as long as

- the reduced expression consists of pure functions, and
- both evaluations terminate.

Conditionals :

```
def abs(x: Int) = if x ≥ 0 then x else -x
```

`if-then-else` in Scala is an expression (has a value) not a statement .

Blocks , Scopes and first functional program :

Sqrt with Newton's method :

```
def sqrt(x: Double) = {  
  def sqrtIter(guess: Double, x: Double): Double =  
    if isGoodEnough(guess, x) then guess  
    else sqrtIter(improve(guess, x), x)  
  def improve(guess: Double, x: Double) = (guess + x / guess) / 2  
  def isGoodEnough(guess: Double, x: Double) = abs(square(guess) - x) < 0.001  
  
  sqrtIter(1.0, x)  
}
```

`sqrtIter` , `improve` and `isGoodEnough` should not be outside sqrt because :

- they pollute the global scope.
- they should not be accessible to the user of the sqrt code given they are specific to sqrt .

the {} could be removed for more clarity.

Tail recursion :

```
def gcd(a: Int, b: Int): Int = if b == 0 then a else gcd(b, a % b)
```

```
gcd(14, 21)  
→ if 21 == 0 then 14 else gcd(21, 14 % 21)  
→ if false then 14 else gcd(21, 14 % 21)  
→ gcd(21, 14 % 21)  
→ gcd(21, 14)  
→ if 14 == 0 then 21 else gcd(14, 21 % 14)  
→ gcd(14, 7)
```

`gcd(14,21) = gcd(21,14) = gcd(14,7) = gcd(.., ..) = ... = 7` : flat structure, `gcd` only calls `gcd` => **tail recursion**.

```
def factorial(n: Int): Int = if n == 0 then 1 else n * factorial(n - 1)
```

factorial(4)

→ if 4 == 0 then 1 else 4 * factorial(4 - 1) 3-> → 4 * factorial(3)

→ 4 * (3 * factorial(2))

→ 4 * (3 * (2 * factorial(1)))

→ 4 * (3 * (2 * (1 * factorial(0))))

→ 4 * (3 * (2 * (1 * 1)))

factorial calls factorial *and multiplies*, nested structure ⇒ **not tail recursion**

To tell the compiler than a function is tail recursive:

```
import scala.annotation.tailrec
@tailrec
def gcd(a: Int, b: Int): Int = ...
```

Week 2 :

Higher order functions :

Def : Functions that take other functions as parameters and return functions.

example :

```
def sum( f : Int ⇒ Int )( a : Int , b : Int ) : Int
```

- takes as argument : the function `f : Int ⇒ Int`.
- returns : a function that has 2 arguments `a: Int, b: Int` and returns `Int`

Function types : `A ⇒ B` is the type of a function that takes an argument of type A and returns a result of type B. e.g : `Int ⇒ Int` maps integers to integers.

Anonymous Functions :

```
(x: Int, y: Int) ⇒ x + y
```

Use case :

```
def sumCubes(a: Int, b: Int) = sum(x ⇒ x * x * x, a, b) // type of function is
inferred by the compiler
```

Currying :

Suppose we have this header for `sum` : `def sum(f : Int⇒Int , a : Int , b : Int)`

```
def sumInts(a: Int, b: Int) = sum(x ⇒ x, a, b)
def sumCubes(a: Int, b: Int) = sum(x ⇒ x * x * x, a, b)
def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

Notice that `a` and `b` get passed unchanged from `sumInts` to `sum` .

Q : Can we do it shorter ?

```
def sum(f: Int ⇒ Int): (Int, Int) ⇒ Int =
  def sumF(a: Int, b: Int): Int =
    if a > b then 0
    else f(a) + sumF(a + 1, b)
  sumF

def sumInts = sum(x ⇒ x)
def sumCubes = sum(x ⇒ x * x * x)
def sumFactorials = sum(fact)
```

`sum` is now a function that takes 1 argument (`f`) and returns a function that takes 2 arguments `a` and `b` .

We can still do shorter :

```
def sum(f: Int ⇒ Int)(a: Int, b: Int): Int =
  if a > b then 0 else f(a) + sum(f)(a + 1, b)
```

Q : What is the type of `sum` ?

A : `(Int⇒Int) ⇒ ((Int,Int)⇒Int)` equivalent to

`(Int⇒Int) ⇒ (Int,Int)⇒Int`

Note that function types **associate to the right** so it is equivalent to :

`Int ⇒ Int ⇒ Int` is equivalent to `Int ⇒ (Int ⇒ Int)`

Classes :

we can define classes in Scala :

```
class Rational(x: Int, y: Int):
  def numer = x // numer and denom here are functions (recalculated each call)
  def denom = y

  // adding operations
  def addRational(r: Rational, s: Rational):
    Rational = Rational(r.numer * s.denom + s.numer * r.denom, r.denom * s.denom)
```

```
// overriding toString
override def toString = s"$numer/$denom" // s means formatted string
// what is after a $ is evaluated

val x = Rational(1, 2) // x: Rational = Rational@2abe0e27
x.numer // 1
x.denom // 2
val y = Rational(5, 7)
val z = Rational(3, 2)
x.add(y).add(z)
```

Classes are very similar to Java.

Suppose that `numer` and `denom` need more calculations, like the following:

```
class Rational(x: Int, y: Int):
  private def gcd(a: Int, b: Int): Int =
    if b == 0 then a else gcd(b, a % b)
  def numer = x / gcd(x, y) // these are recalculated at each call
  def denom = y / gcd(x, y) // BAD IDEA

  // BETTER IDEA : VARIABLES :
  val numer = x / gcd(x, y)
  val denom = y / gcd(x, y)
```

It is possible to use `this(...)` like in Java.

Substitution and Extensions:

1. Substitution:

Suppose that we have a class definition:

```
class C(x1, ..., xm){ ... def f(y1, ..., yn) = b ... }
```

Q: How is the following expression evaluated? `C(v1, ..., vm).f(w1, ..., wn)`

A: Three substitutions happen written:

```
[w1/y1, ..., wn/yn][v1/x1, ..., vm/xm][C(v1, ..., vm)/this] b
```

- Substitution of `y1, ..., yn` of `f` by the arguments `w1, ..., wn`.
- Substitution of the `x1, ..., xm` of the class `C` by the class arguments `v1, ..., vm`.
- Substitution of the self reference `this` by the value of the object `C(v1, ..., vm)`.

2. Extension methods (1):

Having to define all methods that belong to a class inside the class itself can lead to very large classes, and is not very modular.

Methods that do not need to access the internals of a class can alternatively be defined as extension methods.

For instance, we can add min and abs methods to class Rational like this:

```
extension (r: Rational):  
  def min(s: Rational): Boolean = if s.less(r) then s else r  
  def abs: Rational = Rational(r.numer.abs, r.denom)
```

- Extension methods **CANNOT** access internals (`private`) or `this(...)` .

3. Operators :

► We write `x + y` , if x and y are integers, but

► We write `r.add(s)` if r and s are rational numbers.

In Scala, we can eliminate this difference with **operators**.

```
extension (x: Rational):  
  def + (y: Rational): Rational = x.add(y)
```

Precedence rules :

```
(all letters)  
|  
^  
&  
< >  
= !  
:  
+ -  
* / %  
(all other special characters)
```

(all other special characters) having the **highest priority**.

Week3 :

Abstract classes :

```
abstract class IntSet:  
  def incl(x: Int): IntSet // ABSTRACT MEMBERS : no implementation provided  
  def contains(x: Int): Boolean  
  // incl returns the union of {x} and this set
```

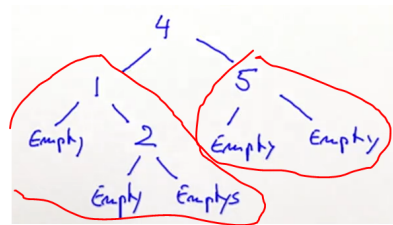
Like in Java, abstract classes cannot be instantiated.

We will try to implement a set as a binary tree.

A set can either be :

1. A tree for the **empty set**.

2. A tree consisting of **one** integer with two subtrees.



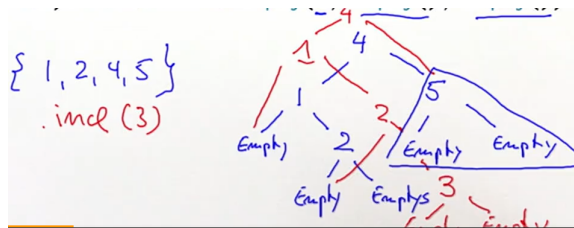
```
class Empty() extends IntSet:
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty(), Empty())

class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet:
  def contains(x: Int): Boolean =
    if x < elem then left.contains(x)
    else if x > elem then right.contains(x) else true

  def incl(x: Int): IntSet =
    if x < elem then NonEmpty(elem, left.incl(x), right)
    else if x > elem then NonEmpty(elem, left, right.incl(x))
    else this
  end NonEmpty
```

1. Persistence :

Note that we the sets are immutable, we always return a new tree while **reusing some subtrees**.(blue one in the following e.g)



A data structure that is creating by maintaining the old one is called **persistent**.

2. Dynamic binding :

```
class Empty() extends IntSet:
  ...
  def union(other : Intset): Intset = other
class NonEmpty((elem: Int, left: IntSet, right: IntSet)) extends Intset :
  ...
  def union(other : Intset ): Intset =
    left.union(right).union(that).incl(elem)

// Why does this terminate ? Union is called with strictly smaller sets each
time so union( "an empty set " ) will be called at some point.
```

Note that a call to `union` doesn't execute the same function if `s` is `Empty` or `NonEmpty` . That is called **Dynamic binding**. (Polymorphism)

Object definition :

In the `IntSet` example, one could argue that there is really **only a single** empty `IntSet` .

This defines a **singleton** object named `Empty`. No other `Empty` instance can be (or needs to be) created.

```
object Empty extends IntSet:
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty) end Empty
```

Companion object :

If a class and object with the same name are given in the same sourcefile, we call them companions. Example:

```
class IntSet ...
object IntSet:
  def singleton(x: Int) = NonEmpty(x, Empty, Empty)
```

Similar to Java, static nested class.

How is naming a class and an object the same name possible ? Scala has **two** namespaces , one for objects and the other for classes.

Programs :

Like Java , scale source files can have a main methods

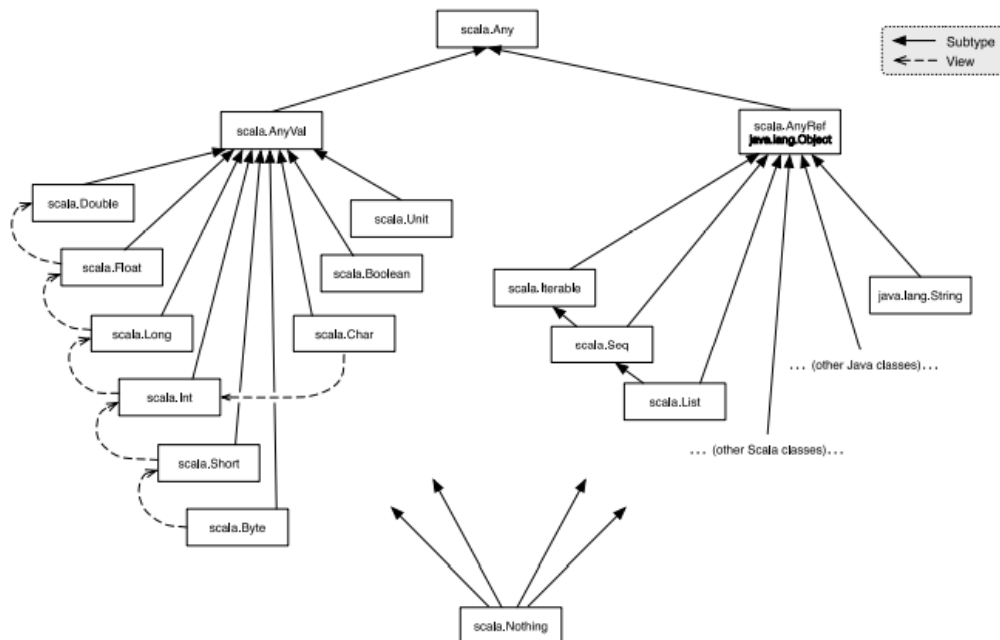
```
object Hello:
  def main(args: Array[String]): Unit = println("hello world!")
// written Shorter as the following :
@main def birthday(name: String, age: Int) =
  println(s"Happy birthday, $name! $age years old already!")
```

to run : `scala Hello`

Traits :

Java `Interface` but stronger :they can have parameters and can contain fields and concrete methods.

```
trait Planar:
  def height: Int
  def width: Int
  def surface = height * width
class Square extends Shape, Planar, Movable ...
```



- - -> : can be converted (viewed) as

Cons List :

A fundamental data structure in many functional languages is the **immutable linked list**.

It is constructed from two building blocks:

1. `Nil` :the empty list
2. `Cons` :a cell containing an element and the remainder of the list

```

trait IntList ...
class Cons(val head: Int, val tail: IntList) extends IntList ...
// NOTE THE USE OF val
// this defines at the same time a parameter and a field of a class
class Nil() extends IntList ...

```

A `IntList` is either a `Nil()` or a `Cons(x, xs)` .

A complete definition with Generics :

```

trait List[T]:
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
class Cons[T](val head: T, val tail: List[T]) extends List[T]:
  def isEmpty = false
class Nil[T] extends List[T]:
  def isEmpty = true
  def head = throw new NoSuchElementException("Nil.head")
  def tail = throw new NoSuchElementException("Nil.tail")

```

Generic functions :

```
def singleton[T](elem: T) = Cons[T](elem, Nil[T])
// We can then write:
singleton[Int](1)
singleton[Boolean](true)
```

Pure Object Orientation :

Def: A pure object-oriented language is one in which every value is an object.

Q: Is Scala a pure OO language ?

At first primitive types and functions seem like exceptions.

Primitive types are in fact not implemented as Class. (E.g `scala.Int` : 32bits , `scala.Boolean` as Java Boolean) , for reasons of efficiency.

But they are not *conceptually* treated differently. In fact one can implement them using classes :

```
package idealized.scala
abstract class Boolean extends AnyVal:
  def ifThenElse[T](t: => T, e: => T): T
  def && (x: => Boolean): Boolean = ifThenElse(x, false)
  def || (x: => Boolean): Boolean = ifThenElse(true, x)
  def unary_!: Boolean = ifThenElse(false, true)
  def == (x: Boolean): Boolean = ifThenElse(x, x.unary_!)
  def != (x: Boolean): Boolean = ifThenElse(x.unary_!, x)
  ...
end Boolean
object true extends Boolean:
  def ifThenElse[T](t: => T, e: => T) = t
object false extends Boolean:
  def ifThenElse[T](t: => T, e: => T) = e
```

[Here](#) is how to implement `Scala.Int` .

Functions : In scala functions are `objects` with `apply` methods

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[A, B]:
  def apply(x: A): B
```

Example :

```
f = (x: Int) => x * x
// is expended to
f = new Function1[Int, Int]:
  def apply(x: Int): Int = x * x
```

This anonymous class can itself expend to

```
{ class $anonfun() extends Function1[Int, Int]:  
  def apply(x: Int) = x * x  
  $anonfun()  
}  
f(7) // expends to  
f.apply(7)
```

Week 4 :

Decomposition :

Expr

Suppose you want to write a small interpreter for arithmetic

/ \

expressions.

Number

Sum

Sum(Number(1) , Number(2)).eval = 3

```
trait Expr :  
  def eval(e: Expr): Int =  
    if e.isNumber then e.numValue  
    else if e.isSum then eval(e.leftOp) + eval(e.rightOp)  
    else throw Error("Unknown expression " + e)
```

1. `isNumber` , `isSum` gets quickly gets tedious when adding other datatypes .
2. There's no static guarantee you use the right accessor functions.
You might hit an Error case if you are not careful.

Non solution :Type Tests and Type casts

USE `def isInstanceOf[T]: Boolean` to type *test* and `def asInstanceOf[T]: T` to cast.

Their use in Scala is discouraged, because there are better alternatives

Solution 1 : Object Oriented Decomposition

```
trait Expr:  
  def eval: Int  
  class Number(n: Int) extends Expr:  
    def eval: Int = n  
  class Sum(e1: Expr, e2: Expr) extends Expr:  
    def eval: Int = e1.eval + e2.eval
```

- ▶ OO decomposition mixes data with operations on the data.
- ▶ Good for encapsulation and data abstraction.
- ▶ Dependencies between classes => Increases **Complexity**.
- ▶ It makes it easy to add new kinds of data but **hard** to add new kinds of operations

OO decomposition only works well if operations are on a single object.

NOT for expressions of type `a * b + a * c -> a * (b + c)`

Solution : Pattern Matching :

```
def eval(e: Expr): Int = e match
  case Number(n) => n // pattern => expression
  case Sum(e1, e2) => eval(e1) + eval(e2)
```

A `MatchError` exception is thrown if no pattern matches the value of the selector.

To be able to do pattern matching `Number` and `Sum` should be **case classes**.

```
trait Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

Lists and more pattern matching :

► the empty list `Nil`, and

► the construction operation `::` (pronounced cons):

`x :: xs` gives a new list with the first element `x`, followed by the elements of `xs`

```
val fruit: List[String] = List("apples", "oranges", "pears")
fruit = "apples" :: ("oranges" :: ("pears" :: Nil)) // similar
```

Right associativity convention : Operators ending in “:” associate to the right.

`A :: B :: C` is interpreted as `A :: (B :: C)`.

All operations on lists can be expressed in terms of the following three:

- `head` the first element of the list
- `tail` the list composed of all the elements except the first.
- `isEmpty` true if the list is empty, false otherwise .

example of using lists and pattern matching :

```
def isort(xs: List[Int]): List[Int] = xs match
  case List() => List()
  case y :: ys => insert(y, isort(ys))

def insert(x: Int, xs: List[Int]): List[Int] = xs match
  case List() => List(x)
  case y :: ys => if x > y then y :: insert(x,ys) else x :: xs
```

Enum :

```
Here's our case class hierarchy for expressions again:
trait Expr
object Expr:
  case class Var(s: String) extends Expr
  case class Number(n: Int) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
```

This is so common in scala that there's a shorthand for that.

```
enum Expr:
  case Var(s: String)
  case Number(n: Int)
  case Sum(e1: Expr, e2: Expr)
  case Prod(e1: Expr, e2: Expr)
```

There's more to Enumeration, they can take parameters and can define methods.

```
enum Direction(val dx: Int, val dy: Int):
  case Right extends Direction( 1, 0)
  case Up extends Direction( 0, 1)
  case Left extends Direction(-1, 0)
  case Down extends Direction( 0, -1)
def leftTurn = Direction.values((ordinal + 1) % 4)
end Direction

val r = Direction.Right
val u = r.leftTurn // u = Up
val v = (u.dx, u.dy) // v = (1, 0)
```

Type Bounds :

`assertPos` should return the set itself if all elements are positive and throw
`exception` otherwise.

```
def assertAllPos(s: IntSet): IntSet
```

The above definition doesn't show that `assertAllPos` of an `Empty` returns an `Empty`
and `assertAllPos` of `NonEmpty` returns an `NonEmpty`. There's better :

```
def assertAllPos[S <: IntSet](r: S): S = ...
```

Here, `<: IntSet` is an upper bound of the type parameter S.

- ▶ `S <: T` means: S is a **subtype** of T, and
- ▶ `S >: T` means: S is a **supertype** of T, or T is a subtype of S.

We can *mix* them : `[S >: NonEmpty <: IntSet]`

The Liskov Substitution Principle

If $A \leq B$, then everything one can do with a value of type B one should also be able to do with a value of type A.

Variance :

Given: `NonEmpty <: IntSet` is `List[NonEmpty] <: List[IntSet]` ? yes.

when `A <: B \Rightarrow C[A] <: C[B]` we call `C[T]` **covariant**.

Does this work for all types ? let's consider `Arrays` (**mutable**)

```
val a: Array[NonEmpty] = Array(NonEmpty(1, Empty(), Empty()))
val b: Array[IntSet] = a // TYPE ERROR HEERE
b(0) = Empty() // CAN DO WITH ARRAY[INTSET] BUT NOT WITH ARRAY[NONEMPTY]
val s: NonEmpty = a(0)
```

Type Error Line 2 : `Array[NonEmpty]` not a subtype of `Array[IntSet]` .

Why ? Because otherwise it would contradict *Liskov Principle* **not** all you can do with `Array[IntSet]` , you can do with `Array[NonEmpty]` . (Line 3,4)

Say `C[T]` is a parameterized type and A, B are types such that $A \leq B$.

- `C[A] <: C[B]` C is **covariant** (e.g `List`)
- `C[A] >: C[B]` C is **contravariant**
- neither `C[A]` nor `C[B]` is a subtype of the other C is **nonvariant** (e.g `Array`)

```
class C[+A] { ... } C is covariant
class C[-A] { ... } C is contravariant
class C[A] { ... } C is nonvariant
```

Function types :

$A1 \Rightarrow B1 \leq A2 \Rightarrow B2$ because $A2 \leq A1$ and $B1 \leq B2$.

Covariant in return type. **Variant** in input type

```
trait Function1[-T, +U]:
  def apply(x: T): U
```