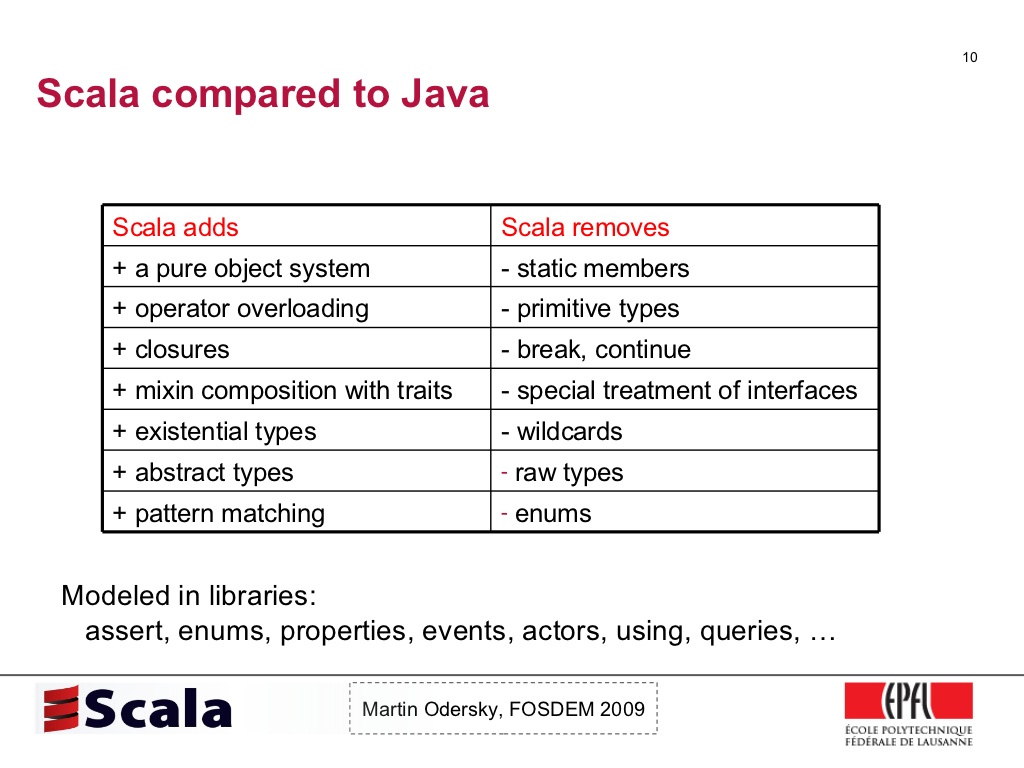
|  |
| --- |
| **Basics** |

**Scala vs. Java**





**משתנים:**

**Immutable values:**

val <name> : <type> = <value>

**Mutable Value:**

var <name> : <type> = <value>

**פונקציות:**

val add: (Int, Int) => Int = (\_ + \_)  
val add: (Int,Int) => Int = (a: Int, b:Int) => a + b

The function is converted to:

val add: (Int,Int) => Int = new Function2[Int,Int,Int] {  
 def apply(a: Int, b:Int) = a + b  
}

כל פונקציה היא בעצם אובייקט אשר מוגדר באופן הבא:

trait Function2[A,B] {  
 def apply(a: A) : B  
}

If the function has side effects, the common convention is to use “()” even though it isn’t required.

**מתודות Methods:**

def \_def(a: Int, b:Int): Int = }a + b{

מסומן ע״י def

הערה: לא ניתן להעביר מתודה כפרמטר בעוד שval- ניתן, בנוסף הערך של val מחושב בintiailize בעוד שהערך של def מחושב כל קריאה, לדוגמא:

def x1 = *println*( 1 + 1 ) // Will not print  
val x2 = *println*( 1 + 1 ) // Will print

או לדוגמא:

val Print: Unit = *println*("val Print");  
def Print2(): Unit = *println*("def Print");  
// Print val Print at **Start**  
Print // Do Nothing  
Print // Do Nothing  
  
Print2 // def Print  
Print2 // def Print  
Print2 // def Print

ניתן לאתחל ערך **דיפולטיבי** באופן הבא:

def Sum(x: Int, y: Int = 99) : Int = x + y  
*println*(Sum(1,1)); *println*(Sum(1)) // [2,100]

ניתן גם לאתחל Lazy Val

lazy val <name> : <type> = <value>

**הערה:** אין Lazy Var

**הערה:** להקפיד לרשום באופן Immutable – כלומר להשתמש ב-Val ומבנים שהם Immutable כמו List

Nested Methods:

def f(x: Int) : String = {  
 def g(x: Int) : String = "A String Example with " + x  
 g(x)  
}  
*println*(*f*(3))

**הערה:**

במקרה הבא נקבל כי def שומר מקום היות והוא לא מאותחל ישר אלא רק כאשר קוראים לו

def x = 3  
val *y* = 3

**לולאות:**

for(i <-0 until 10) // [0,10)  
for(i <- 0 to 10 ) // [0,10]

**מעבר על אוספים:**

var collection: Seq[Int] = *Array*(1,2,3)  
 val fn : Int => Unit = (a: Int) => {  
 *print*(s"a = **$**a \n")  
 }  
 collection.foreach(fn)

**לולאות פנמיות:**

for(i<-0 until 10; j<-0 until 10; k<-0 to 1 if i%2==0 && j%2==0)  
 *println*(s"(**$**i,**$**j,**$**k)")

**Yield**

val *ans* : List[Int] = for {  
 x <- *List*(1,2,3)  
 y <- *List*(1,2,3)  
 product = x \* y  
 if product%2 == 0 }  
 yield product  
*println*(*ans*) // List(2, 2, 4, 6, 6)

**Pattern Matching:**

val *x* : Int = 3  
*x* match {  
case 1 => *println*("1 Case")  
case 2 => *println*("2 Case")  
case \_ if *x*%2 == 0 => *println*("Even")  
case x: Boolean => *println*("its String")  
case \_ => *println*("None of the above")  
}

**הערה:** עובד גם על מחלקות ע״י הפעלת פונקציית unapply

**רקורסיה:**

*println*(*fib*(-1))  
def fib(n: BigInt) : BigInt = {  
 if(n < 0) throw new IllegalArgumentException("Illegal Argument")  
 if(n == 1 || n == 2) 1  
 else *fib*(n-1) + *fib*(n-2)  
}

**מערכים: (Mutable)**

val *fixed* = new Array[String](1)  
val *dynamic* = ArrayBuffer[Int]()

// scala.collection.mutable.ArrayBuffer  
*fixed*(0) = "A" // equal to fixed.apply(0)  
// fixed(1) = "B" // ArrayIndexOutOfBoundsException  
*dynamic* += 0  
*dynamic* ++= *Array*(0,1)

**מערכים דו מימדיים:**

val *multDimArray* = Array.*ofDim*[Int](10,5)  
val *sortedArray* = *multDimArray*(0).sortWith(\_>\_)

**Multiple Parameter Lists (Currying)**

def f(num: Int)(x : Int => Int)(y : Int => Int) : Int = x(y(num))  
*println*(*f*(3)(\_\*2)(\_+1)) // 2 \* ( 3 + 1)

**רשימות: (Immutable) + Map, Filter**

class List*[*+A*]* ... *{* def map*[*B*](*f: A => B*)* : List*[*B*]*

def flatMap*[*B*](*f: A => GenTraversableOnce*[*B*])*: List*[*B*]  
}*

var *\_list* : List[Int] = *List*(1,2,3,4,5,6,7)  
val *filterFunction* : (Int => Boolean) = \_ % 2 == 0  
val *mapFunction* : (Int => Int) = (x:Int) => { x\*2 }  
*\_list*.filter(*filterFunction*).map(*mapFuncton*).foreach(*println*)

**FlatMap:**

val *list*: List*[*Option*[*Int*]]* = *List(*Some*(*1*)*,None,Some*(*2*))  
list*.map*(*x => *println(*x*))* // Some(1),None,Some(2)  
var *list2* = *list*.flatMap*(*x => x*)  
list2*.foreach*(println)* // 1, 2

val *list*: List*[*String*]* = *List(*"Hey","Zvi"*)  
println(list*.map*(*\_.toUpperCase*()))* // List(HEY, ZVI)  
*println(list*.flatMap*(*x => x.toUpperCase*))* // List(H, E, Y, Z, V, I)

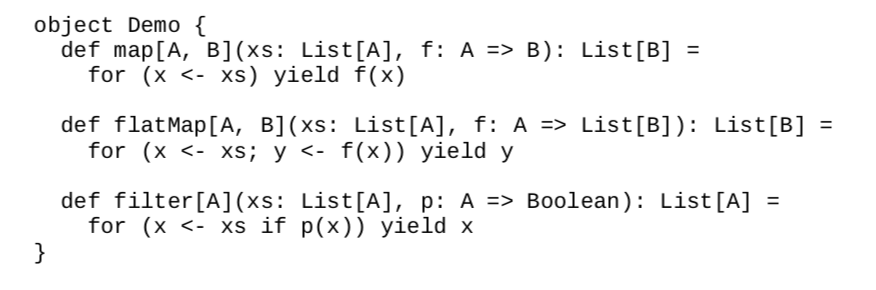
**Cons:**

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] {  
 override def isEmpty: Boolean = false  
}  
 class Nil extends List[Nothing] {  
 override def isEmpty: Boolean = true  
 override def head: Nothing = throw new

NoSuchElementException("Nil.head")  
 override def tail: List[Nothing] = throw new

NoSuchElementException("Nil.tails")  
 }

**מאחורי הקלעים:**

****

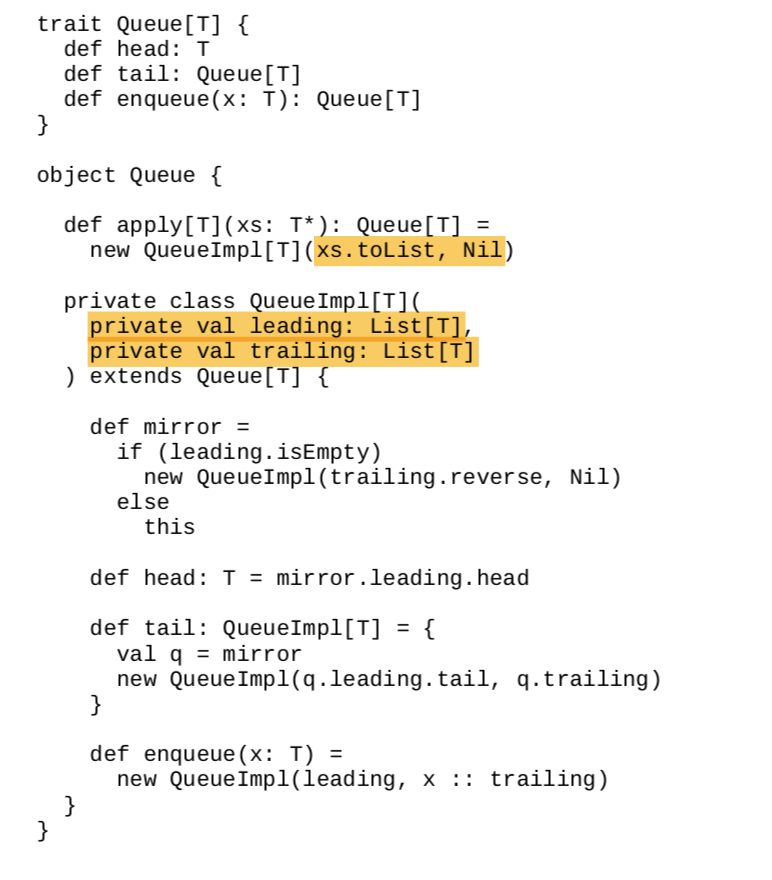
def map[A,B](list: List[A])(fn: A => B): List[B] = list match {  
 case head :: tail => fn(head) :: *map*(tail)(fn)  
 case \_ => *Nil*}  
def filter[A](list: List[A])(fn : A => Boolean): List[A] = list match {  
 case head::tail =>  
 val rest = *filter*(tail)(fn)  
 if (fn(head)) head ::rest  
 else  
 rest  
 case \_ => *Nil*}  
 def foldLeft[A,B](list: List[A], acc: B)(fn: (B,A) => B): B = list match {  
 case head :: tail => *foldLeft*(tail,fn(acc,head))(fn)  
 case \_ => acc  
}

def foreach[A](list: List[A], f: A => Unit) : Unit = list match {  
 case *Nil* => Nil  
 case head :: tail =>  
 f(head)  
 *foreach*(tail,f)  
}  
  
val *list* : List[Int] = *List*(1,2,3)  
*println*(*map*(*list*) { \_ \* 2 }) // List (2,4,6) -> Reason to Multi Parameter List

**דוגמאות נוספות:**

val *list*: List[Int] = *List*(1,2,3)  
*printAll*(*list*) // 1 \n 2 \n 3  
*println*(*length*(*list*)) // 3  
*println*(*toStringList*(*list*)) // List(1,2,3)  
  
def printAll(list: List[\_]): Unit = list match {  
 case head :: tail => *println*(head); *printAll*(tail)  
 case \_ => ()  
}  
  
def length(list: List[\_]): Int = list match {  
 case \_ :: tail => 1 + *length*(tail)  
 case \_ => 0  
}  
def toStringList(list: List[\_]): List[String] = list match {  
 case head :: tail => head.toString :: *toStringList*(tail)  
 case \_ => *Nil*}

**תור:**

****

**מפות: (Immutable)**

val *map*: Map[String,Operator] = *Map*(  
 "+" -> {\_ + \_},  
 "-" -> {\_ - \_})

|  |
| --- |
| **Apply and Unapply Methods (Extractors) ( Good Link:** [**Click**](https://www.protechtraining.com/blog/post/an-introduction-to-pattern-matching-in-scala-842) **)**  **Important Topic** |

**Introduction:**

Scala let you expend the function call syntax ***f(args1, args2, … )***

to values other than functions. if **f** is not a function or method, then this expression

is equivalent to the call ***f.apply(args1,args2….)***

unless it occurs to the left an assignment, this expression ***f(args1…) = value***

corresponds to the call ***f.update(args1,…value)***

**Example of Update, apply and unapply Method: [\*]**

class MyArray(var \_arr: Array[Int]) {  
 def update(index: Int, x: Int) = \_arr(index) = x  
 def apply(x: Int) = \_arr(x)  
}  
object MyArray {  
def apply(xs: Int\*) = new MyArray(xs.toArray)  
 def unapply(s: String): Option[Array[Int]] = try {  
 var StringArray = s.split(",")  
 var ans : Array[Int] = StringArray.map(\_.toInt)  
 Some(ans)  
 } catch {  
 case \_ => None  
 }  
}  
var *instance* = *MyArray*(1,2,3)  
*println*(*instance*.\_arr)  
*println*(*instance*(0)) // 1  
*instance*(0) = 5  
  
var *s*:String = "1,2,3"  
*s* match {  
 case *MyArray*(args) => args.foreach(x => *print*(x + " ")) // 1 2 3  
 case \_ => *println*("Nothing.")  
}

This mechanism is used in array and maps, for example:

case class Person(name: String)  
val *scores*: mutable.HashMap[Person,Int] = mutable.HashMap.*empty*val *zvi* = Person("Zvi")  
*scores*(*zvi*) = 50  
*println*(*scores*) // Map(Person(Zvi) -> 50)

Why we don’t need the new keyboard at classes?

class Person(name: String) {}  
object Person {  
 def apply(name: String): Person  
 = new Person(name)  
}

**Intoduction to unapply:**

class Fraction(var a:Int,var b: Int) {  
 def +(other: Fraction) : Fraction  
 = new Fraction(a \* other.b + other.a \* b, b \* other.b)  
}  
object Fraction {  
 def apply(a: Int, b:Int): Fraction  
 = new Fraction(a,b)  
  
 def unapply(input: Fraction): Option[(Int,Int)] = {  
 if (input.b == 0) None else Some((input.a,input.b))  
 }  
}  
var *Fraction*(*a*,*b*) = *Fraction*(1,2) + *Fraction*(1,3)  
*println*(s"a = **$***a*, b = **$***b*") // a = 5, b = 6

**דוגמא נוספת:**

class Number(a : Int)  
object Number {  
 def unapply(s: String): Option[Int] = try { Some(Integer.*parseInt*(s)) }  
 catch { case ex:NumberFormatException => None }  
}  
val *Number*(*n*) = "1512"

**דוגמא נוספת:**

class A {}  
object A {  
 def apply(s: String) : A = { *println*(s); new A() }  
 def unapply(a: A): Option[(Int,Int)] = Some(1 -> 1)  
}  
var *a* = *A*("Input to apply function")  
*a* match { // a is input [ On what we do match ]  
 case *A*(a,b) => *println*(s"a = **$**a, b = **$**b") // (a,b) is output  
}

**Unapply: (pattern match)**

type Operator = (Int,Int) => Int // Type Alias  
  
object Operator {  
 def apply(str: String) = *???* val *map*: Map[String,Operator] = *Map*(  
 "+" -> {\_ + \_},  
 "-" -> {\_ - \_})  
 def unapply(s: String): Option[Operator] = *map*.get(s)  
}  
var *inputs*: Array[String] = *Array*[String]("+","-","A")  
for (input <- *inputs*)  
input (Input from Unapply) match {  
 case *Operator*(f (OUTPUT FROM UNAPPLY) ) => *println*(f(3,3)) // Pattern Matching use unapply method  
 case \_ => *println*("None of the above")  
} // Output: 6, 0, None of the above

**דוגמא נוספת ( עם apply ):**

object HelloWorld extends App {  
 var *x* : String = *EmailAddressSplitted*("zvimints","gmail.com")  
 *println*(*x*) // zvimints@gmail.com  
 *x* match {  
 case *EmailAddressSplitted*(s) => *println*(s"(**$**{s.\_1}, **$**{s.\_2})") // (zvimints , gmail.com)  
 case \_ => *println*("Dammit.")  
 }  
 object EmailAddressSplitted {  
 def apply(name: String, domain: String): String = name + "@" + domain  
 def unapply(emailAddress: String): Option[(String,String)] = try {  
 val splitted = emailAddress split "@"  
 if(splitted.length == 2)  
 Some(splitted(0),splitted(1))  
 else  
 None  
 }  
 }  
}

**Higher-Order Function**

A **higher**-**order function** takes other **functions** as a parameter or returns a **function** as a result. This is possible because **functions** are a first-class value in **Scala**.

// Higher Order Functions  
def times3(x:Int) = x\*3  
def MultNum(f:(Int=>Int), x: Int) = f(x)  
  
*println*(*MultNum*(*times3*,10))

**Try-Catch Blocks:**

try {  
 throw new Exception()  
}  
 catch {  
 case ex: IllegalArgumentException => *println*("Shit!")  
 case \_ => *println*("Shit Shit Shit!")  
 }  
 finally {  
 *println*("Recovering...")  
 }

**Closures:**

var *more* : Int = 10  
val *add* : (Int => Int) = (x: Int) => x + *more* // more is Free variable  
*println*(*add*(3)) // 13  
*more* = 20 // The Result will change!  
*println*(*add*(3)) // 23

**Option:**

val *option*: Option[Int] = {  
 try {  
 Some("3 + 3".toInt)  
 } catch {  
 case \_ : Exception => None  
 }  
}

**שימוש:**

*println*(*option*.getOrElse(555))

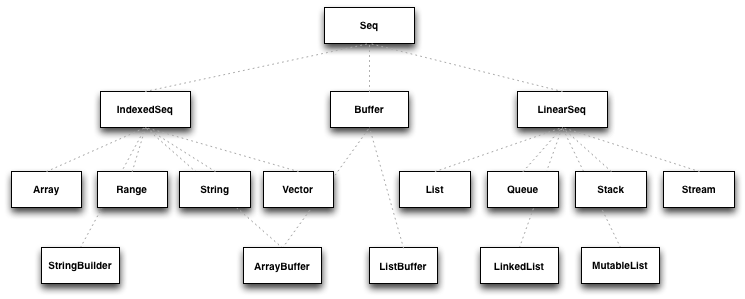
**Either:**

val *in* = Console.readLine("Type Either a string or an Int: ")  
val *result*: Either[String,Int] = try {  
 *Right*(*in*.toInt)  
} catch {  
 case e: Exception =>  
 *Left*(*in*)  
}  
  
*println*(*result* match {  
 case *Right*(x) => "You passed me the Int: " + x + ", which I will increment. " + x + " + 1 = " + (x+1)  
 case *Left*(x) => "You passed me the String: " + x  
})

**Infix notation**

class Test {  
 def fnWith1Parm(a: Int) : Unit = {}  
 def fnWith2Parm(a: Int,b: Int) : Unit = { *println*(s"a = **$**a, b = **$**b") }  
 def fnWith0OrMoreParm(args: Int\*) : Unit = for(x <- args) *println*(x)  
}  
var *test* = new Test()  
*test* fnWith0OrMoreParm (1,2,3,4,5) // Works  
*test* fnWith1Parm 1 // Works  
*test* fnWith2Parm (1,2) // Works

|  |
| --- |
| **Class Hierarchy** |

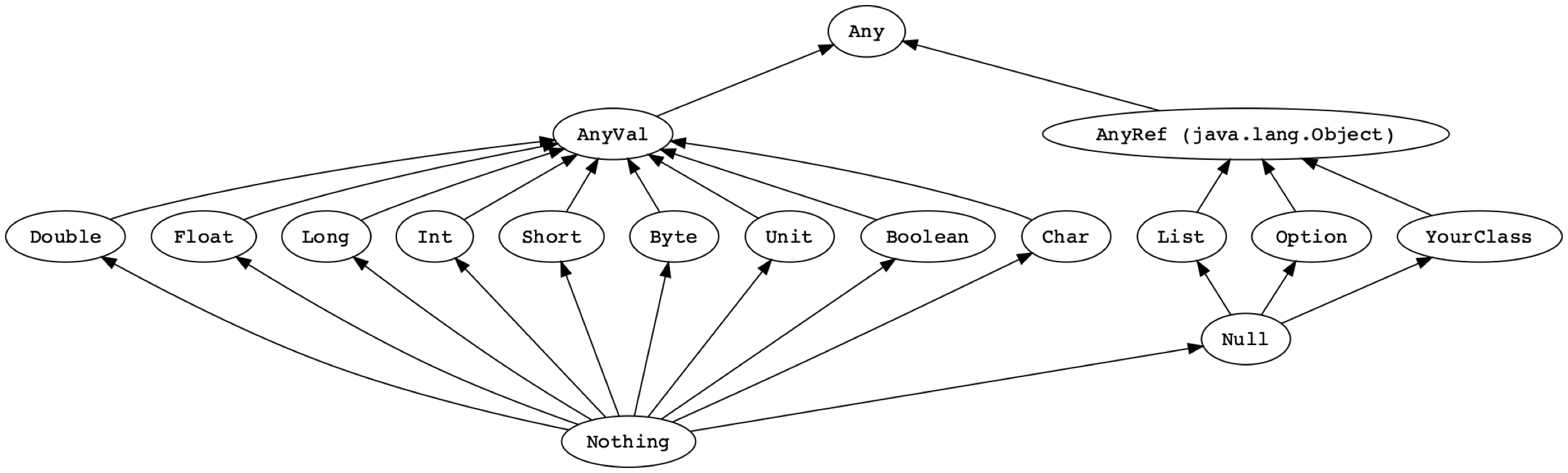


Collections in scala.collection package

Collections in scala.collection.immutable package

**Note: For more Information:** [**Click Here**](https://www.47deg.com/blog/adventures-with-scala-collections/)

|  |
| --- |
| **Unified Types** |



case class Element()  
var *Any\_list* : List[Any] = *List*("String", new Element);  
var *AnyVal\_list* : List[AnyVal] = *List*(1,1.5,true);  
var *AnyRef\_List* : List[AnyRef] = *List*("String",() => "Function",  
 new Element, null);

|  |
| --- |
| **Classes, Objects, case Classes and case Objects** |

**מחלקת נקודה:**

class Point(private var \_x: Int,  
 private var \_y: Int) {  
 private val *\_id* = Point.*getUniqueID* // "Static" Method  
 this.setX(\_x)  
 this.setY(\_y)  
 // Constructors  
 def this(x: Int) = this(x,0)  
 def this() = this(0,0)  
  
 // Getters and Setters  
 def x\_(x: Int) = { this.\_x = if(x>=0) x else 0 }  
 def x = \_x  
  
 def setX(x: Int) = { this.\_x = if(x >= 0) x else 0 }  
 def setY(y: Int) = { this.\_y = if(y >= 0) x else 0 }  
 def getX = this.\_x;  
 def getY: Int = this.\_y  
  
 // Override  
 override def toString = s"(**$**{this.*\_id*},**$**{this.\_x},**$**{this.\_y})"  
}  
  
object Point {  
 private var *count* : Int = 0  
 def getUniqueID() : Int = {  
 *count* += 1  
 *count* }  
}

|  |  |
| --- | --- |
| **Case Class**  **(Immutable)** | **Class**  **(Mutable)** |
| No need New() Since they call apply() Method | Need new() |
| **public** by default | **private** by default |
| Compare by **value** | Compare by **reference** |
| **val** by default | **var** by default |
| immutable | mutable |

**Note:** Case classes **without parameters** are meaningless and deprecated. In that situation, **case objects** are used.

**הערה:** ב-Case Class מקבלים את המתודות הבאות ״חינם״:

1. apply
2. unapply
3. toString
4. equal
5. copy
6. hashcode

**הערה:** אין צורך להגדיר field בבנאי (לרשום val או var) ב-case class

class A(x: Int) {  
 override def toString() : String = "" + x  
}  
case class B(x: Int) {  
 override def toString() : String = "" + x  
}  
var *a* = new A(3)  
// a.x No Valid cause x cant use out of the class  
*println*(*a*) // 3  
  
var *y* = B(3)  
*y*.x // Automatic make public val [Works V]  
*println*(*a*)

If you are going to write purely functional code with immutable objects, you should better try avoid using regular classes. The main idea of the functional paradigm is the separation of data structures and operations on them.

Do **not** use Case Classes if :

1. Your class carries mutable state.
2. Your class includes some logic.
3. Your class is not a data representation and you do not require structural equality.

|  |
| --- |
| **ירושה + Traits** |

trait T1  
trait T2  
abstract class abs[T](x: T) {  
 *println*("Abstract Class")  
 def fn(f: T => T , x: T) = f(x) // Implemented  
 def print() : String // Need to Implement  
}  
class A[T](x: T) extends abs(x) with T1 with T2 {  
 override def print(): String = "Its " + x  
}  
var *a* : A[Int] = new A[Int](3)  
*a*.print()  
*println*(*a*.fn( \_ + 1, 1)) // 2

|  |
| --- |
| **Tuples** |

case class Tuple2[T1,T2](\_1: T1, \_2: T2) {  
 override def toString() : String = s"(**$**\_1,**$**\_2)"  
}

val ingredient = ("Sugar" , 25)  
*println*(ingredient.\_1) // Sugar  
*println*(ingredient.\_2) // 25

ניתן לאתחל בעזרתם משתנים באופן הבא:

val (name, quantity) = ingredient  
*println*(name) // Sugar  
*println*(quantity) // 25

ניתן לעבור על Tuples באופן הבא:

ingredient.productIterator.foreach( x => *println*(x))

או לחלופין:

*println*(ingredient.toString()) // (Sugar,25)

|  |
| --- |
| **Sealed Classes (Algebric Data type)** |

Traits and classes can be marked sealed which means all subtypes must be declared in the same file. This assures that all subtypes are known.

This is useful for pattern matching because we don’t need a “catch all” case.

**sealed** **trait** Color   
case **class** Red extends Color  
case **class** Blue extends Color  
case **class** Green extends Color

**sealed** **trait** Option[+A]   
case **object** None extends Option[Nothing]   
case **class** Some[A](a: A) extends Option[A]

case class Recipe(ingredients: Map[String, Mass],

directions: List[String]) {  
 def shoppingList(kitchen: Map[String, Mass]) : List[String] = {  
 for {  
 (name, need) <- ingredients.toList  
 have = kitchen.getOrElse(name,Grams(0))  
 if have < need  
 } yield name  
 }  
}  
  
trait Measured {  
 def amount: Double // Represent the amount  
 def symbol: String // Represent the symbol s.t g,kg..  
 override def toString : String = "" + amount + symbol  
}  
// Algebraic Data Types consists of sealed abstract class and some case classes  
sealed trait Mass extends Ordered[Mass] with Measured {  
 def toGrams : Grams  
 override def compare(that: Mass): Int = (this.toGrams.amount - that.toGrams.amount).toInt  
}  
  
case class Grams(amount: Double) extends Mass {  
 override def toGrams: Grams = this  
 override def symbol: String = "g"  
}  
case class Milligrams(amount: Double) extends Mass {  
 override def toGrams: Grams = *Grams*(amount / 1000)  
 override def symbol: String = "mg"  
}  
case class Kilograms(amount: Double) extends Mass {  
 override def toGrams: Grams = *Grams*(1000 \* amount)  
 override def symbol: String = "kg"  
}

|  |
| --- |
| **Implicit** |

Implicit functions will be called automatically if the compiler thinks it’s a good idea to do so. What that means is that if your code doesn’t compile but would, if a call was made to an implicit function, Scala will call that function to make it compile. They’re typically used to create implicit conversion functions; single argument functions to automatically convert from one type to another

implicit var *value* = 3  
 *println*(*example2*) // 9  
 // println(example(1)) // Wont compile  
def example2(implicit x: Int, y: Int) : Int = x \* y /

/ x and y are implicit

*println*("Hello ".MyNewFunction())  
  
// MyNewString New Class  
case class MyNewString(s: String) {  
 def MyNewFunction() : String = "World"  
}  
// Implicit convert from String to MyNewString  
implicit def convertFunction(s: String) : MyNewString = *MyNewString*(s)

**Implicit Parameters**

****

At it’s simplest, an implicit parameter is just a function parameter annotated with the implicit keyword. It means that if no value is supplied when called, the compiler will look for an implicit value and pass it in for you.

var value = 3  
implicit val multiplier = 2  
// implicit val other = 3 // Will cause a "ambiguous implicit values"  
def multiply(implicit by: Int) = value \* by  
*println*(multiply) // 6  
*println*(multiply(0)) // 0

**Syntax**

You can only use implicit once in a parameter list and all parameters following it will be implicit. For example:

|  |
| --- |
| def example1(implicit x: Int) **// x is implicit**  def example2(implicit x: Int, y: Int) **// x and y are implicit**  def example3(x: Int, implicit y: Int) **// wont compile**  def example4(x: Int)(implicit y: Int) **// only y is implicit**  def example5(implicit x: Int)(y: Int) **// wont compile**  def example6(implicit x: Int)(implicit y: Int) **// wont compile** |

**means that if x +y not compiles, then the compiler will try compile(x) + y**

****

**מחלקה Implicit Class:**

case class Rectangle(width: Int, height: Int) {  
 override def toString : String = s"Rectangle(**$**width,**$**height)"  
}  
implicit class RectangleFrom1Dim(width: Int) {  
 def x(height: Int) : Rectangle = new Rectangle(width,height)  
}  
var *rectangle* = 3 x 4  
*println*(*rectangle*)

|  |
| --- |
| **Singleton** |

**Singleton Implementation:**

// Singleton class  
class Singleton private {  
 override def toString : String = "This is Singleton Class"  
}  
// Singleton object [ Unique ]  
object Singleton {  
 val *singleton* : Singleton = new Singleton  
 def getInstance() : Singleton = *singleton*}

var s : Singleton = Singleton.*singleton*

**לדוגמא:**

class Recipe private(  
 val ingredients: List[String] = *List*.*empty*,  
 val direction: List[String] = *List*.*empty*)  
 object Recipe {  
 def make(ing: List[String], dir: List[String]) : Recipe =  
 new Recipe(ing,dir)  
 }  
  
var x = Recipe.*make*(*List*("A","B","C"),*List*("Mix A & B & C"))

**באופן כללי:**

1. ליצור את המחלקה עם בנאי פרטי
2. ליצור את אובייקט המחלקה עם פונקציית make אשר מחזירה מופע של המחלקה

|  |
| --- |
| **Variance** |

**class** **Foo**[+**A**] **//** **A** **covariant** **class**

**class** **Bar**[-**A**] **//** **A** **contravariant** **class**

**class** **Baz**[**A**] **//** **An** **invariant** **class**

**דוגמא (תקינה):**

**הערה:** S <: T אומר שS יורש מ-T

S >: T אומר שS הוא supertype של T או לחלופין שT יורש מ-S

class Dog  
 class Puppy extends Dog // Puppy is subclass of Dog  
  
 trait PutBox[A] {  
 def put[A](a: A): Unit = *???* }  
 trait GetBox[A] {  
 def get: A = *???* }  
  
 object Boxes {  
 // putPuppy from Box that at least as Puppy  
 // for example Puppy -> [.... -> Box]  
 def putPuppy(box: PutBox[\_ >: Puppy]) : Unit =  
 box.put(new Puppy)  
  
 // getDog from box that is bounded by Dog  
 // for example [Puppy -> ...] -> Dog is valid  
 def getDog(box: GetBox[\_ <: Dog]) : Dog =  
 box.get  
  
 val *dogPutBox* = new PutBox[Dog] {}  
 val *dogGetBox* = new GetBox[Dog] {}  
  
 val *puppyPutBox* = new PutBox[Puppy] {}  
 val *puppyGetBox* = new GetBox[Puppy] {}  
  
 *putPuppy*(*puppyPutBox*) // Valid  
 *putPuppy*(*dogPutBox*) // Valid  
 *getDog*(*puppyGetBox*) // Valid   
 *getDog*(*dogGetBox*) // Vali

**אבל הדוגמא הבאה לא תקינה:**

def putPuppy(box: PutBox[Puppy]) : Unit =  
 box.put(new Puppy)  
  
def getDog(box: GetBox[Dog]) : Dog =  
 box.get  
  
val *dogPutBox* = new PutBox[Dog] {}  
val *dogGetBox* = new GetBox[Dog] {}  
  
val *puppyPutBox* = new PutBox[Puppy] {}  
val *puppyGetBox* = new GetBox[Puppy] {}  
  
*putPuppy*(*puppyPutBox*) // Valid  
// putPuppy(dogPutBox) // NOT Valid  
// getDog(puppyGetBox) // NOT Valid  
*getDog*(*dogGetBox*) // Valid

היות ונוצרת הבעיה הבא:

נניח כ S <: T

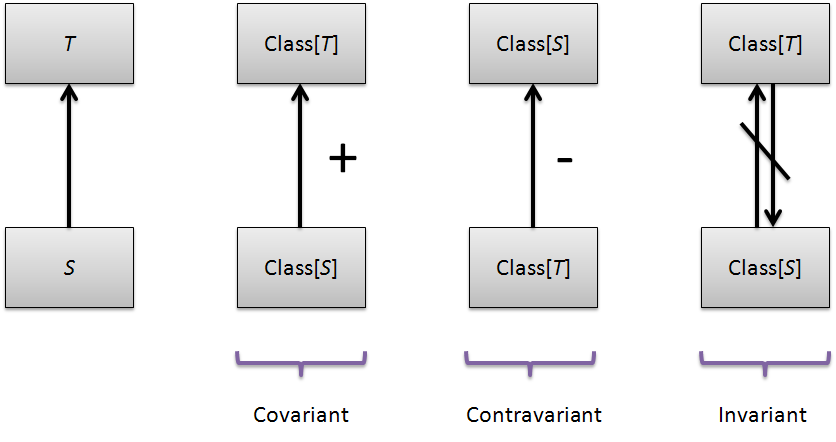
אז האם List[S] <: List[T] ? במידה וכן אז יש צורך לשים בS – Covariant (+)

האם Array[S] <: Array[T] ? אז מסתבר שזה עושה בעיות היות ו:

S[] a = new S[]{S1,S2,S3} // Fine  
T[] b = a // Fine  
b[0] = T1 // ArrayStoreException [ Its **RUN TIME** IN JAVA! ]   
T t = a[0]  
// We got an assignment of S value to T value but  
// S <- T

לדוגמא Red r = new Color()

Red <- Color



**Covariant –(Get)** אם S יורשת מ-T אז נרצה לאפשר המרה של Class[S] לירוש מ-Class[T]

**Contravariant –(Put)** אם S יורשת מ-T אז נרצה לאפשר המרה מClass[T] ל-Class[S]

**נפתור זאת ע״י:**

trait GetBox[+A] // covariant

trait PutBox[-A] // contravariant

בגלל זה בספרייה הסטנדרטית:

*List*[+A]

או לחלופין

trait Function[-I, +O] extends AnyRef

**הערה**: אם A2 <: A1 , B1 <: B2 אז A1 => B1 <: A2 => B2

**Merge Sort**

def SortList[T](list: List[T])(f: (T,T) => Boolean): List[T] = {  
 val middle: Int = list.length / 2  
 if (middle == 0) list // Means that the list length in {0,1}  
 else {  
 def merge(list1: List[T], list2: List[T]): List[T] =  
 (list1, list2) match {  
 case (list1, *Nil*) => list1  
 case (*Nil*, list2) => list2  
 case (head1 :: tail1, head2 :: tail2) =>  
 if (f(head1,head2))  
 head1 :: merge(tail1, list2)  
 else  
 head2 :: merge(list1, tail2)  
 }  
 val (firstList, secondList) = list splitAt middle  
 merge(*SortList*(firstList)(f), *SortList*(secondList)(f))  
 }  
}  
var *sortedList* = *SortList*(*List*(3, 2, 1))(\_ < \_)  
*sortedList*.foreach(x => *print*(x + " "))

או לחלופין: ( עם implicit)

def SortList[T](list: List[T])(**implicit** ord: Ordering[T]): List[T] = {  
 val middle: Int = list.length / 2  
 if (middle == 0) list // Means that the list length in {0,1}  
 else {  
 def merge(list1: List[T], list2: List[T]): List[T] =  
 (list1, list2) match {  
 case (list1, *Nil*) => list1  
 case (*Nil*, list2) => list2  
 case (head1 :: tail1, head2 :: tail2) =>  
 if (ord.lt(head1,head2))  
 head1 :: merge(tail1, list2)  
 else  
 head2 :: merge(list1, tail2)  
 }  
 val (firstList, secondList) = list splitAt middle  
 merge(*SortList*(firstList), *SortList*(secondList))  
 }  
}  
var *sortedList* = *SortList*(*List*(3, 2, 1))  
*sortedList*.foreach(x => *print*(x + “ “))

נניח שיש פונקציה *f* אשר לוקחת פרמטר implicit מטיפוס T

אזי הקומפיילר יחפש עבור הגדרה implicit ש:

1. מסומנת בimplicit
2. מתאים לפרמטר T
3. נראה בנקודת קריאה של הפונקציה או מוגדרת במחלקה אחרת אשר מתאימה ל-T

|  |
| --- |
| **Future** |

<https://www.beyondthelines.net/computing/scala-future-and-execution-context/>

<https://www.slideshare.net/javiersantospaniego/codemotion-akka-vol-sobre-el-nido-del-future-55591054>

**Example With Futures: (For Debug)**

def taskA*()*: Future*[*Unit*]* = *Future {  
 debug(*"Starting taskA"*)* Thread.*sleep(*1000*)* // wait 1secs  
 *debug(*"Finished taskA"*)  
}*def taskB*()*: Future*[*Unit*]* = *Future {  
 debug(*"Starting taskB"*)* Thread.*sleep(*2000*)* // wait 2secs  
 *debug(*"Finished taskB"*)  
}*def main*(*args: Array*[*String*])*: Unit = *{  
 debug(*"Starting Main"*)* val futureA = *taskA()* val futureB = *taskB()  
 debug(*"Finished Main"*)* Await.*result(*futureA zip futureB, Duration.*Inf)* //06:54:51.749 [main] Starting Main  
 //06:54:52.146 [scala-execution-context-global-11] Starting taskA  
 //06:54:52.147 [main] Finished Main  
 //06:54:52.147 [scala-execution-context-global-12] Starting taskB  
 //06:54:53.152 [scala-execution-context-global-11] Finished taskA  
 //06:54:54.149 [scala-execution-context-global-12] Finished taskB  
*}*

**There are 3 ways to ways to use the global execution context:**

1. import scala.concurrent.ExecutionContext.Implicits.*global*
2. implicit val *executor* = scala.concurrent.ExecutionContext.*global*
3. *Future {…}(executor)*

what is a Promise? Well it’s something that gives a Future, obviously!

**Note:** Calling

*Future {* /\* do something \*/ *}*

is actually calling the method *apply* on Future companion object:

object Future *{* def apply*[*T*](*body: => T*)(*implicit executor: ExecutionContext*)*: Future*[*T*]* =  
 unit.map*(*\_ => body*)  
}*

**Parallel Collections:**

import scala.collection.parallel.mutable.\_

import scala.collection.parallel.immutable.\_

**Example:**

def fib*(*n: Int*)*: Int = *{* if*(*n == 1 || n == 2*)* 1  
 else *fib(*n-1*)* + *fib(*n-2*)  
}*for*(*i<- *(*30 to 15 by -1*)*.par *) {  
 println(fib(*i*))* // Will Print not in order!  
*}*

**Multithread problem: ( Race Condition )**

(Since load I from memory, add I to register and store I to memory is not atomic)

var *i* : Int = 0  
var *j* : Int = 0  
for*(*index <- *(*0 to 10000*)*.par*) i* += 1  
for*(*index <- *(*0 to 10000*)) j* += 1  
*println(*s"i = **$***i*, j = **$***j*"*)* // i = 6817, j = 10001

**Scala Futures:**

A Future represents a value which may or may not **currently** be available, but will be available at some point, or an exception if that value could not be made available.

import scala.concurrent.*{*Await, Future*}*import scala.concurrent.ExecutionContext.Implicits.*global*import scala.concurrent.duration.\_

**Foreach: No-Blocking Call!**

val *s* = "Hello"  
val *f*: Future*[*List*[*String*]]* = *Future {  
 List*.fill*[*String*](*5*)((s* + " future!"*))  
}  
f* foreach *{* msg => *println(*s"List Of Messages = **$**msg"*)*// List Of Messages = List(Hello future!, Hello future!,

Hello future!, Hello future!,

Hello future!)  
*}*Thread.*sleep(*1000*)*

**Sequence:**

val *f* : Future*[*Int*]* = for *{* f1 <- *job(*1*)* f2 <- Future.*sequence(List(job(*f1*)*, *job(*f1*)))* f3 <- *job(*f2.head*)* /\*  
 Future.sequence takes a list of futures that we wish to run simultaneously.  
 So here we have f2 and f4 containing two parallel jobs.   
 \*/  
 f4 <- Future.*sequence(List(job(*f3*)*, *job(*f3*)))* f5 <- *job(*f4.head*)  
}* yield f2.size + f4.size  
*f*.foreach*(*z => *println(*s"Done. **$**z jobs run in parallel"*))*Thread.*sleep(*6000*)* // needed to prevent main thread from quitting  
// too early

**Recovering from errors:**

*Future {*"abc".toInt*}* // his will not work, as “abc” is not an int.  
 .map*(*z => z + 1*)*.foreach*(println)*

*Future {*"abc".toInt*}* .recover *{*case e => 0*}* .map*(*z => z + 1*)*.foreach*(println)*Thread.*sleep(*1000*)*

**Mapping: No-Blocking Call!**

val *fu* : Future*[*List*[*Int*]]* = for *{* f1 <- *job(*1*)* f2 <- *job(*f1*)* f3 <- *job(*f2*)* f4 <- *job(*f3*)* f5 <- *job(*f4*)  
}* yield *List(*f1, f2, f3, f4, f5*)  
fu*.map*(*z => *println(*s"Done. **$***{*z.size*}* jobs run"*))*Thread.*sleep(*6000*)* // needed to prevent main thread from quitting

MADRID Â· NOV 27-28 Â· 2015
Scala Programming @ Madrid
Future: Operations
â flatMap
def getFirstMillionOfPrimes(): Future[Li...

**Success And Failure:**

var *f*: Future*[*Int*]*= *Future {* throw new Exception*(*" Exception"*)* 1 + 1  
*}  
f*.onComplete *{* // No Blocking  
 case Success*(*n*)* => *println(*n*)* // 2  
 case Failure*(*ex*)* => *println(*"Somthing went wrong" + ex*)  
}*Thread.*sleep(*1000*)*

def f*(*n: Int*)* : Future*[*Int*]* = *Future {* n  
*}  
f(*2*)* onComplete *{* x => *println(*x*) }*// Do Heavy calculation:  
*println(*"Do Heavy Calculation"*)*Thread.*sleep(*1000*)*

**Await - Blocking Call!**

val *f* : Future*[*List*[*Int*]]* = for *{* f1 <- *job(*1*)* f2 <- *job(*2*)* f3 <- *job(*3*)* f4 <- *job(*4*)* f5 <- *job(*5*)  
}* yield *List(*f1, f2, f3, f4, f5*)  
f*.map*(*z => *println(*s"Done. **$***{*z.size*}* jobs run"*))  
println(*"Here"*)*Await.*result(f*,1.second*)  
println(*"Sup"*)*// Output:  
/\*  
 Here  
 1  
 2  
 3  
 4  
 5  
 Done. 5 Jobs [List[Int]]  
 Sup  
 \*/

**Synchronized:**

var amount: Int = 0  
def deposit*(*n: Int*)* = *{  
 amount* += n  
*}*def withdraw*(*n: Int*)* = *{  
 amount* -= n  
*}*for*(*i<- *(*1 to 100*)*.par*) {  
 deposit(*1*)  
 withdraw(*1*)  
}  
println(*s"amount = **$***amount*"*)* // **-6**

**But:**

var *amount*: Int = 0  
def deposit*(*n: Int*)* = this.synchronized*{  
 amount* += n  
*}*def withdraw*(*n: Int*)* = this.synchronized*{  
 amount* -= n  
*}*for*(*i<- *(*1 to 100*)*.par*) {  
 deposit(*1*)  
 withdraw(*1*)  
}  
println(*s"amount = **$***amount*"*)* // **0!**

|  |
| --- |
| **Dependency Injection** |

**livelessons:**

<https://drive.google.com/drive/u/0/folders/0BwYVFgjh5JF3alZBZXgwSTB3Wjg>

**Martin Odersky:**

<https://www.coursera.org/learn/progfun1/lecture/n2EWV/lecture-4-2-functions-as-objects>