תקציר לשפת Scala

נכתב ע"י צבי מינץ

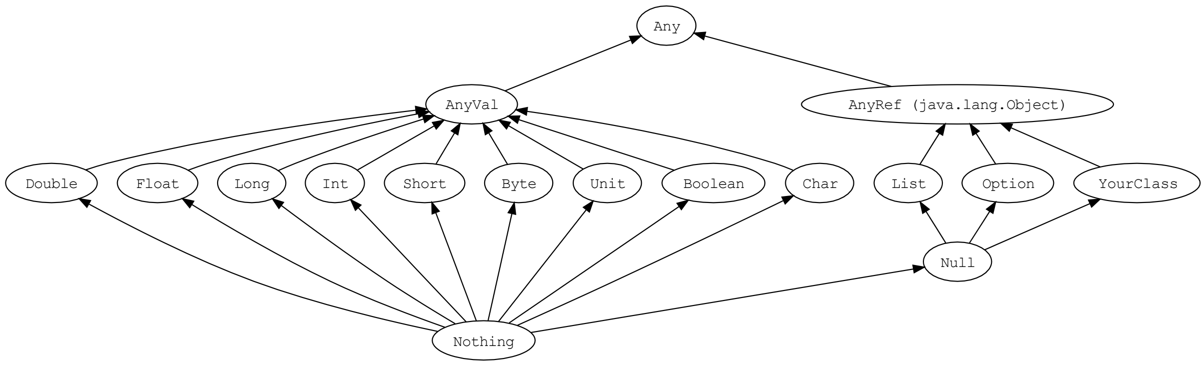
מבוסס על הקורס Functional Programming in Scala coursera

zvimints@gmail.com

```
**     ________ ___   / /  ___     Scala API                            **
**    / __/ __// _ | / /  / _ |    (c) 2003-2013, LAMP/EPFL             **
**  __\ \/ /__/ __ |/ /__/ __ |    http://scala-lang.org/               **
** /____/\___/_/ |_/____/_/ | |                                         **
**                          |/                                          **
```

https://github.com/scala/scala/tree/v2.10.1/src/library/scala
https://www.coursera.org/specializations/scala?
הועלה ל: https://github.com/ZviMints/Summaries

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);
```

משתנים

```
var x : Unit = println("Will Print") // Mutable
val y : Unit = println("Will Print") // Immutable
def z : Unit = println("Will Not Print")
lazy val w : Unit = println("Will Not Print")
```

אולם אם בהמשך נקרא ל:

```
x
y
z // Print
w // Print
w
w
```

פונקציות ו-Partial Functions

Parial Function

נסתכל על הדוגמא הבאה:

```
val f: String => String = { case "ping" => "pong" }
println(f("ping")) // "pong"
//println(f("abc")) // Exception1!!
```

נרצה לדעת אם בהינתן פונקציה כלשהי האם יש ערך ב-case עבור הקלט

```
trait PartialFunction[-I,+O] extends Function1[I,O] {
  def apply(x: I): O
  def isDefinedAt(x: I):O
}
```

ולכן:

```
val f: PartialFunction[String,String] = { case "ping" => "pong" }
println(f("ping")) // "pong"
println(f.isDefinedAt("abc")) // false
println(f.isDefinedAt("ping")) // true
```

כאשר הפונקציה isDefinedAt מוגדרת באופן הבא:

```
def isDefinedAt(s: String) x match {
  case "ping" => true
  case _ => false
}
```

פונקציה הוא ביטוי מהצורה

```
A => B
```

מבחינת הקומפיילר, כל פונקציה הופכת לאובייקט באופן הבא:

```
(a:Int, b:Int) => a + b
```

מתורגמת ל:

```
new Function2[Int, Int, Int] {
  override def apply(v1: Int, v2: Int): Int = v1 + v2
}
```

באופן כללי:

```
trait Function1[-T1,+R] extends AnyRef {
  def apply(v1: T1): R = ???
}
```

מתודות

מסומן ע"י `def`

```
def add : (Int,Int) => Int = (a,b) => a + b
def add2 : (Int,Int) => Int = (_ + _)
// def add3 = (_ + _ ) // Not Compile
def add4 = ( (_:Int) + (_:Int) )
def add5 = (a:Int, b:Int) => a + b
```

הערה: אם לפונקציה יש Side-Effect אז נהוג להשתמש ב"()" למרות שאין צורך

הגדרה: Side-Effect: נאמר שלפונקציה יש Side-Effect אם היא משנה מצב (State) מחוץ לתחום הבלוק של הפונקציה

ניתן לאתחל ערך **דיפולטיבי** באופן הבא:

```
def sum100(a: Int, b:Int = 100, c: Int = 100) = a + b + c
println(sum100(1,1)) // 102 -> c used
println(sum100(1)) // 201 -> b,c used
```

```
def isPrime(num: Int) = (2 until num).forall(d => (num%d != 0))
isPrime(17)
isPrime(15)
```

ניתן גם לעשות Nested Methods באופן הבא:

```
def f(more: Int) : (Int,Int) => Int = {
  def g(a:Int, b:Int) = a + b + more
  g
}
f(more = 5)(0,0)
```

currying

```
def fInRange(f: Int => Int) : (Int,Int) => Int = {
  def range(a: Int, b:Int) : Int = {
    if(a > b) 0
    else f(a) + range(a+1,b)
  }
  range
}
println(fInRange(_*2)(1,5)) // 2 + 4 + 6 + 8 + 10 = 30
```

זהה ל:

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

בצורה זו ניתן לממש mapReduce ב- Range

```
def mapReduceInRange(zero: Int, f: Int => Int, combiner: (Int,Int) => Int)(a: Int, b:Int) : Int = {
  if(a > b) zero
  else
    combiner(f(a),mapReduceInRange(zero,f,combiner)(a+1,b))
}
println(mapReduceInRange(0,{_*2},{_+_})(1,5)) // 30
```

require

```
val y = 0
require(y > 0, "Damn. y is lte 0")
```

Call By Name Vs Call By Value

דוגמא: (אחרת לא יעצור):

```
def loop : Boolean = loop
def and(a: Boolean, b: => Boolean) = if(a) b else false
println(and(true,false))
println(and(false,loop))
```

דוגמא נוספת:

```
def time() = System.nanoTime
def exec(t: => Long) = {
  println(s"[time: $t]: ... do calculations")
  Thread.sleep(1000)
  println(s"[time: $t]: ... do calculations")
  Thread.sleep(1000)
  t
}


println(exec(time()))
```

אולם יש בעיה אם נשתמש בפרמטר של Call By Name כמה פעמיים במהלך הבלוק של הפונקציה כי כל פעם הוא יעשה חישוב מחדש ולכן ניתן לאתחל את מה שמקבלים ע״ lazy val ולהשתמש בו במהשך וכך החישוב יתבצע פעם אחת בלבד.

לולאות

```
for(i<-0 until 10) { ??? } // [0,10)
for(j<-0 to 10) { ??? } // [0,10]
for(i<-0 until 10; j<-0 until i) { ??? }
for { i<-0 until 10
    j<-0 until i
    product = i*j
    if product%2 == 0 } { ??? }
```

כל לולאה מהצורה :

```
val ans : Unit
= for(i<-0 until 2; j<-1 until 2; k<-2 until 3) {
  println(s"i=$i, j=$j, k = $k")
}
```

מתורגמת ל:

```
val ans : IndexedSeq[IndexedSeq[IndexedSeq[Unit]]]
= (0 until 2).map(i => (1 until 2).map(j => (2 until 3).map(k =>
  println(s"i=$i, j=$j, k = $k"))))
```

Yield

```
val ans : IndexedSeq[Int]
= for(i<-0 until 2 if i%2 == 0; j<-1 until 2; k<-2 until 3) yield {
  i*j*k
}
println(ans) // Vector(0)
```

בעוד שהקומפיילר מתרגם את Yield ל-flatMap באופן הבא:

```
val ans : IndexedSeq[Int]
= (0 until 2).withFilter(i => i%2 == 0).flatMap(i => (1 until 2).flatMap(j => (2 until 3).map(k =>
  i*j*k)))
println(ans) // Vector(0)
```

לדוגמא:

```
def isPrime(num: Int) : Boolean = (2 until num) forall(d => (num%d != 0))

for {
   i<-0 until 10
   j<-0 until 10
   if isPrime(i+j)
} yield (i,j)

(0 until 10).flatMap(i => (0 until 10).filter(j => isPrime(i+j)).map(j => (i,j)))
```

Pattern Matching

```scala
val x: Int = 3
x match {
  case even if even % 2 == 0 => println(even)
  case 1 => println(x)
  case _ => println("Nothing")
}
```

אופרטור @ מתייחס למחלקה, לדוגמא:

```scala
class Person(var name: String,var id: Int)
object Person {
  def unapply(other: Person): Option[  (String,Int)  ] = {
    Some( (other.name, other.id) )
  }
}
val zvi = new Person(name = "zvi", id = 111)
zvi match {
  case p @ Person(name,id) => println(p) // Testing$Person@4e04a765
}
```

Pattern Matching עובד על המתודה unapply באופן הבא:

```scala
class MyArray(arr: Array[Int]) {
  def update(idx: Int, value: Int): Unit = arr(idx) = value
  def apply(idx: Int): Int = arr(idx)
}
object MyArray {
  def apply(xs: Int*) : MyArray = new MyArray(xs.toArray)
  def unapply(s: String): Option[Array[Int]] = {
    try {
      val StringArray = s.split(",")
      val IntArray = StringArray.map(_.toInt)
      Some(IntArray)
    } catch {
      case ex: Exception => None
    }
  }
}
val myarr : MyArray = MyArray(3) // object apply
myarr(0) = 5 // class update
println(myarr(0)) // class apply
"1,2,3" match {
  case MyArray(successParse) => successParse.foreach(println)
  case _ => println("Parse Failed")
}
```

Apply, Unapply, Update

סקלה נותנת את האפשרות **להמיר** קריאה של *f(a,b,c)* ל- *f.apply(a,b,c)*

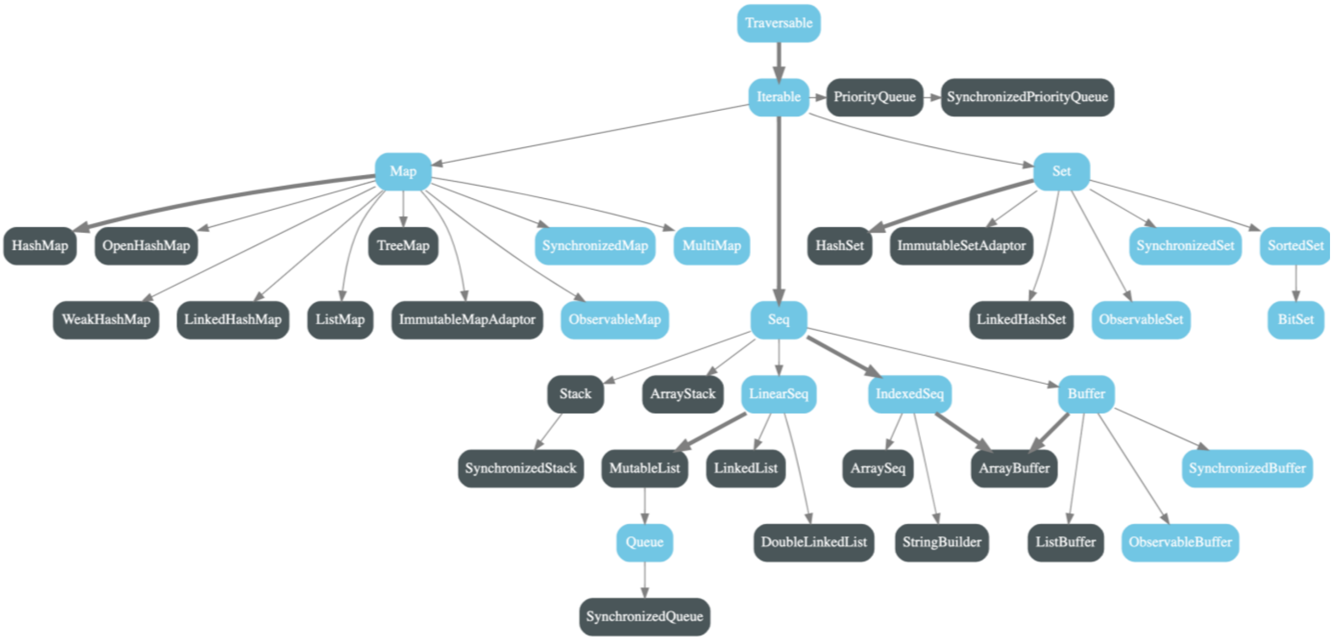
בנוסף יש את האפשרות לעשות *arr(i)=0*, זה קורה היות והקריאה ממורת ל-arr.update(I,0)

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)
}
```

אוספים Mutable

`scala.collection.mutable`



לרשת מ- Trait ה- `Traversable` ידרוש לממש את:

```
override def foreach[U](f: T => U): Unit = ???
```

מערכים דינמיים וסטטים:

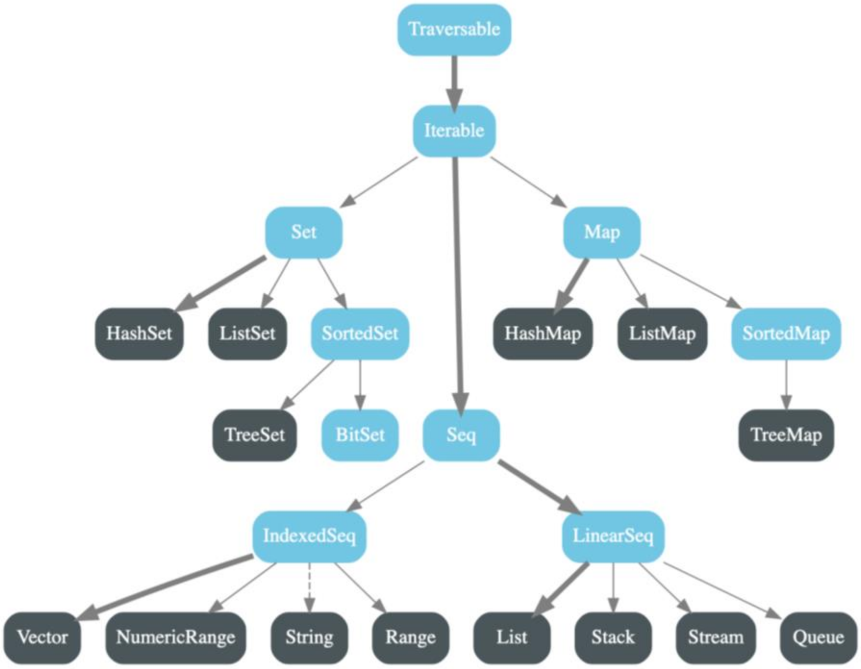
```
val fixed = new Array[String](1)
val dynamic = ArrayBuffer[Int]()
// scala.collection.mutable.ArrayBuffer
fixed(0) = "A" // equal to fixed.apply(0)
// fixed(1) = "B" // ArrayIndexOutOfBoundsException
dynamic += 0
dynamic ++= Array(0,1)
```

מערכים דו מימדיים

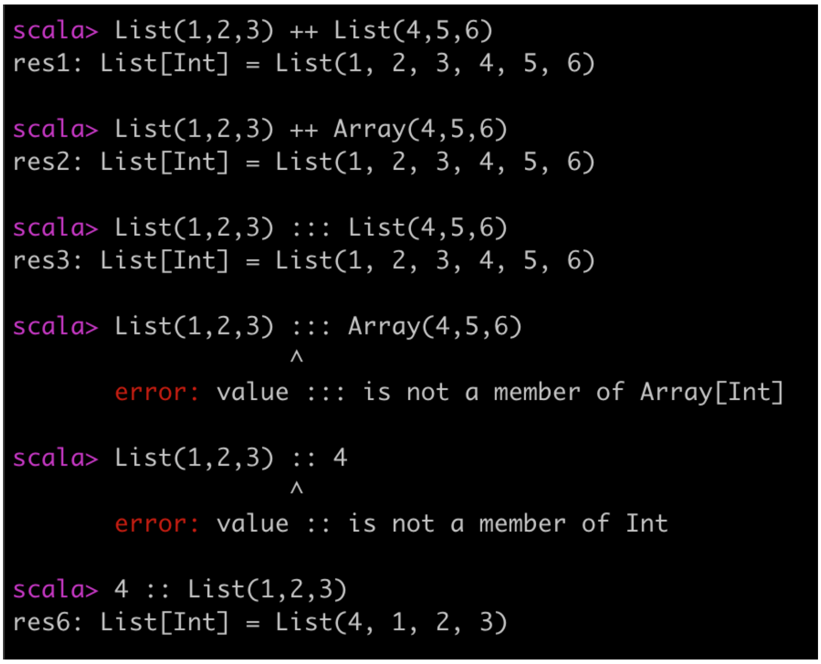
```
val multDimArray = Array.ofDim[Int](10,5)
val sortedArray = multDimArray(0).sortWith(_>_)
```

אוספים Immutable

`scala.collection.immutable`



Operators ++, ::, ::: ברשימה

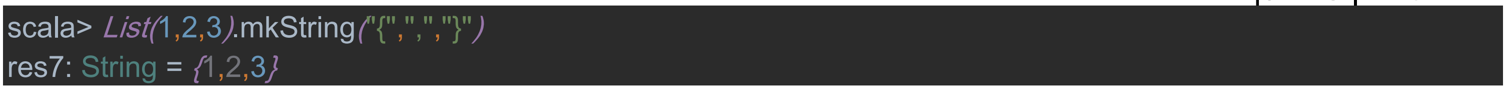


```
scala> List(1,2,3) ++ List(4,5,6)
res1: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> List(1,2,3) ++ Array(4,5,6)
res2: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> List(1,2,3) ::: List(4,5,6)
res3: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> List(1,2,3) ::: Array(4,5,6)
                   ^
       error: value ::: is not a member of Array[Int]

scala> List(1,2,3) :: 4
                   ^
       error: value :: is not a member of Int

scala> 4 :: List(1,2,3)
res6: List[Int] = List(4, 1, 2, 3)
```

לסיכום:

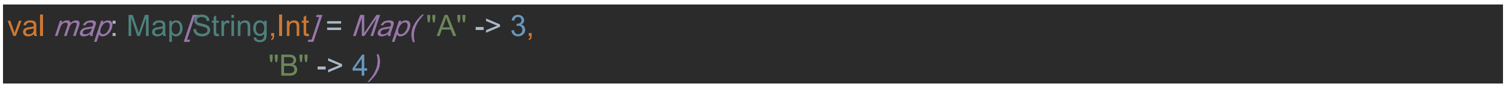
::: משמש לחיבור **רשימות**
++ משמש לחיבור IterableOnce
ולכן עדיף להשתמש ב' ::: ' בשביל efficiency and type safety

להדפיס תוכן של אוסף בתוך מחרוזת



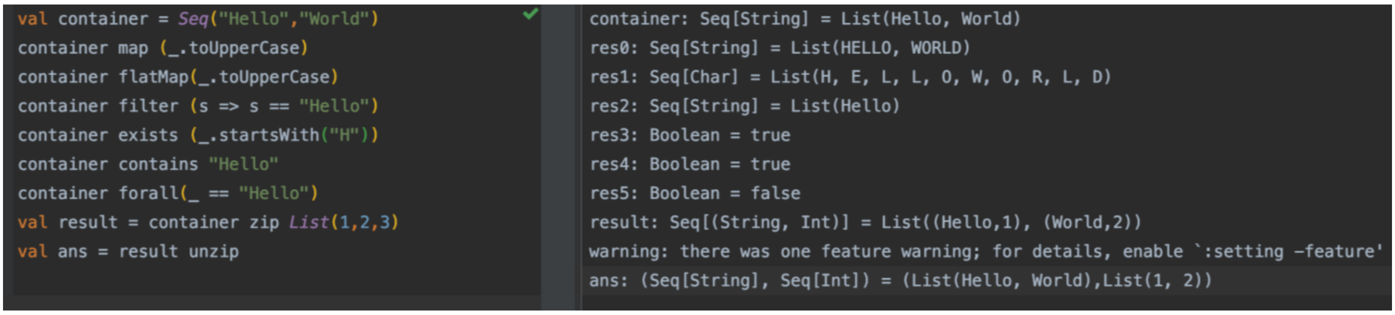
```
scala> List(1,2,3).mkString("{",",","}")
res7: String = {1,2,3}
```

מפות:



```
val map: Map[String,Int] = Map( "A" -> 3,
                                "B" -> 4)
```

```
println(map.get("A")) // Some(3)
println(map.get("C")) // None
println(map.getOrElse("A",-1)) // 3
println(map.getOrElse("C",-1)) // -1
```



```
val container = Seq("Hello","World")
container map (_.toUpperCase)
container flatMap(_.toUpperCase)
container filter (s => s == "Hello")
container exists (_.startsWith("H"))
container contains "Hello"
container forall(_ == "Hello")
val result = container zip List(1,2,3)
val ans = result unzip
```

```
container: Seq[String] = List(Hello, World)
res0: Seq[String] = List(HELLO, WORLD)
res1: Seq[Char] = List(H, E, L, L, O, W, O, R, L, D)
res2: Seq[String] = List(Hello)
res3: Boolean = true
res4: Boolean = true
res5: Boolean = false
result: Seq[(String, Int)] = List((Hello,1), (World,2))
warning: there was one feature warning; for details, enable `:setting -feature'
ans: (Seq[String], Seq[Int]) = (List(Hello, World),List(1, 2))
```

מתודות על מפות:

```
case class Person(name: String)
val map: Map[Int,Person] = Map(
  1 -> Person("zvi"),
  2 -> Person("Royi"),
  3 -> Person("Gaby"),
  4 -> Person("Dan"),
  5 -> Person("Ilan")
)
// Give Option
map get 1 // Option[Person] = Some(Person(zvi))
map get 6 // Option[Person] = None
map getOrElse (7, Person("null")) // Person = Person(null)
// GroupBy
val fruits = List("apple","banana")
val ans: Map[Char,List[String]] = fruits groupBy( _.head ) // Map(b -> List(banana), a -> List(apple))
// ++
Map(1 -> "WillDeleted") ++ Map(2 -> "Ilan", 1 -> "Zvi") //  Map(1 -> Zvi, 2 -> Ilan)
// WithDefaultValue
// (*) map(6) // Exception noSuchElementException
val MapSaved = map withDefaultValue Person("Default Value")
MapSaved(6) // Person("With Default Value")
```

Transform להמיר – Value במפה לValueאחר

```
// Transform
map transform((k,v) => Person(k.toString))
Map[Int,Person] = Map(5 -> Person(5), 1 -> Person(1), 2 -> Person(2), 3 -> Person(3), 4 -> Person(4))
```

Monads

Monads is a construction which performs successive calculations. It is an object which covers the other object

```
trait M[T] {
  def flatMap(f: T => M[T]) : M[T]
  def unit[T](x: T): M[T]
}
```

נשים לב כי:

```
map = flatMap {x => unit(f(x))}
```

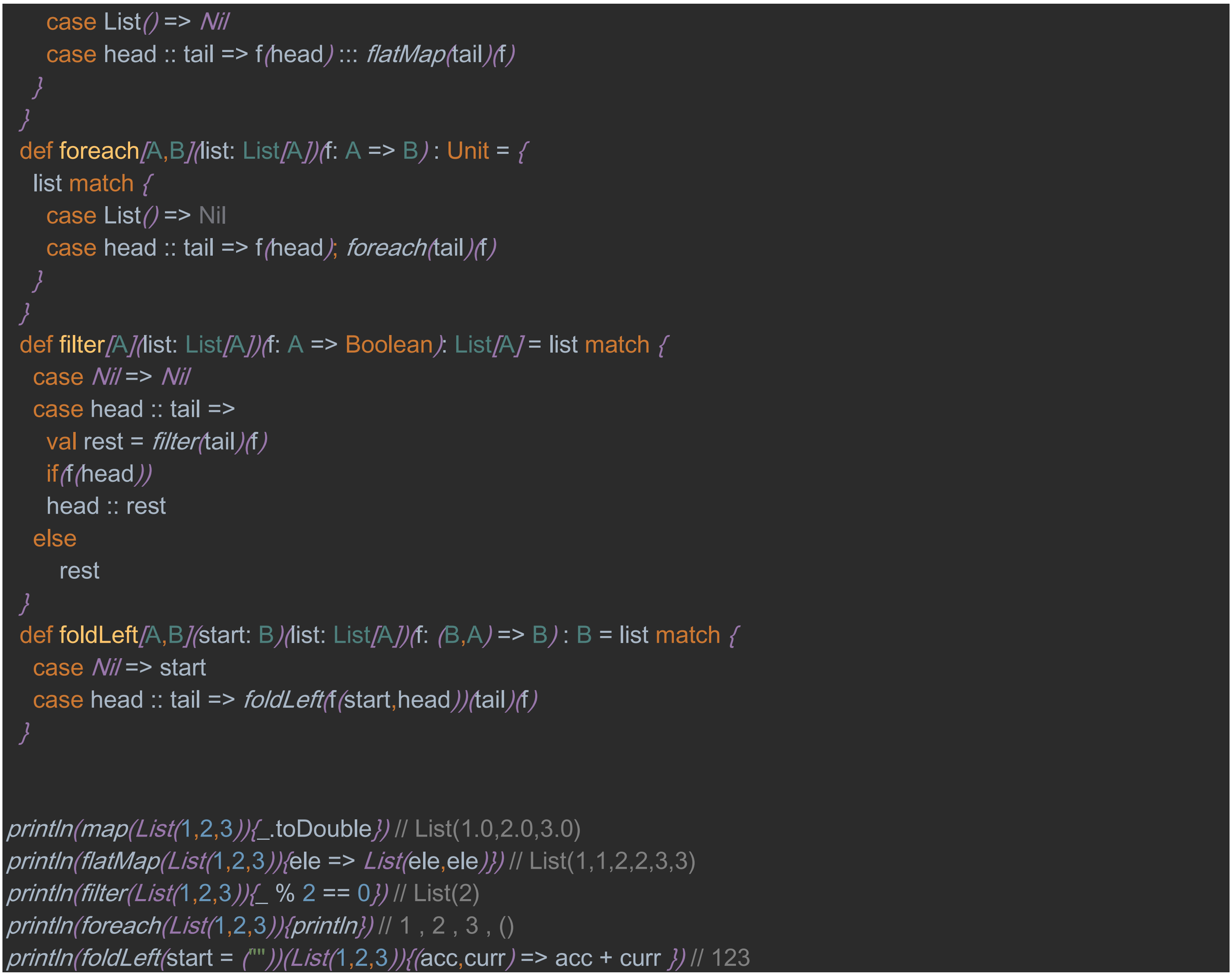
Map, flatMap, Filter, withFilter, foreach, groupBy, FoldLeft,

דוגמא לאיך המחלקה האבסטרקטית List נראית:

```
sealed abstract class  List[+T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
  def map[U](f: T => U): List[U]
  def flatMap[U](f: T => List[U]): List[U]
  def foreach[U](f: T => U) : Unit
  def filter(f : T => Boolean): List[T]
  def withFilter(f: T => Boolean) : FilterMonadic[T, Repr]
  def foldLeft[U](start: U)(f: (U,T) => U) : T
  def length
  def mkString(middle: String)
  def mkString(start: String, middle:String, end:String)
  def ::[U >: T](ele: U) : List[T]
  def :::[U >: T](list: List[U]) : List[T]
  def ++[U >: T](list: List[U]) : List[U]
}
object Nil extends List[Nothing] {
???
}
object Cons[T] extends List[T] {

}
```

מימוש מלא:

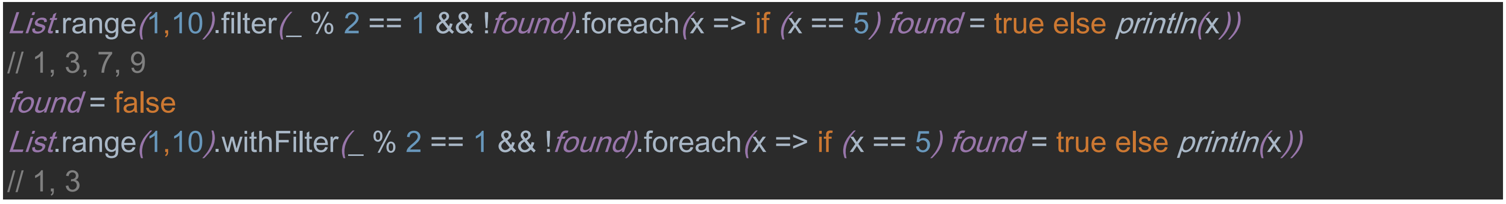
```
  def map[A,B](list: List[A])(f: A => B): List[B] = {
    val ans: List[B] = list.map(ele => f(ele))
    ans
    /* Can be implemented also as:
      for(ele <- list) yield f(ele)
              Or
       list match {
       case List() => Nil
       case head :: tail => f(head) :: map(tail)(f)
     */
  }
  def flatMap[A,B](list: List[A])(f: A => List[B]): List[B] = {
    list match {
```



```
     case List() => Nil
     case head :: tail => f(head) ::: flatMap(tail)(f)
   }
 }
 def foreach[A,B](list: List[A])(f: A => B) : Unit = {
   list match {
     case List() => Nil
     case head :: tail => f(head); foreach(tail)(f)
   }
 }
 def filter[A](list: List[A])(f: A => Boolean): List[A] = list match {
   case Nil => Nil
   case head :: tail =>
     val rest = filter(tail)(f)
     if(f(head))
     head :: rest
   else
       rest
 }
 def foldLeft[A,B](start: B)(list: List[A])(f: (B,A) => B) : B = list match {
   case Nil => start
   case head :: tail => foldLeft(f(start,head))(tail)(f)
 }


println(map(List(1,2,3)){_.toDouble}) // List(1.0,2.0,3.0)
println(flatMap(List(1,2,3)){ele => List(ele,ele)}) // List(1,1,2,2,3,3)
println(filter(List(1,2,3)){_ % 2 == 0}) // List(2)
println(foreach(List(1,2,3)){println}) // 1 , 2 , 3 , ()
println(foldLeft(start = (""))(List(1,2,3)){(acc,curr) => acc + curr }) // 123
```

הערה: כל פעם שיש Recursion אפשר לשקול להשתמש ב-FoldLeft

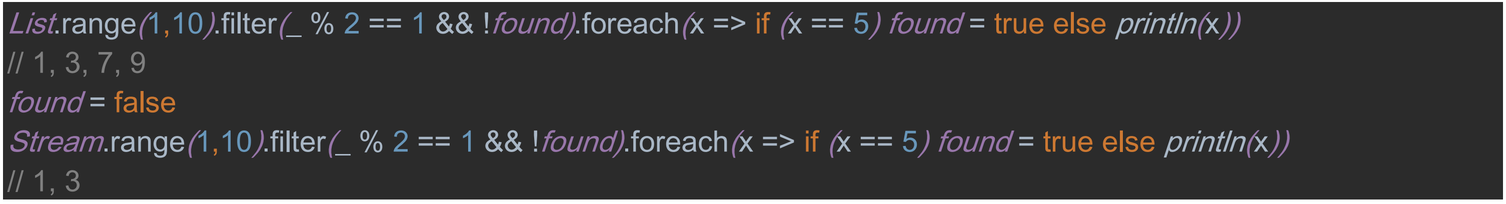
ההבדל בין **Filter** ל-**withFilter** הוא ש-Filter **מחזיר אוסף חדש** וwithFilter **לא** מחזיר, ולכן אם נסתכל על הקוד הבא נקבל את הפלט:



```
List.range(1,10).filter(_ % 2 == 1 && !found).foreach(x => if (x == 5) found = true else println(x))
// 1, 3, 7, 9
found = false
List.range(1,10).withFilter(_ % 2 == 1 && !found).foreach(x => if (x == 5) found = true else println(x))
// 1, 3
```

הסיבה לכך הוא שבפעם הראשונה אז אחרי filter קבלנו אוסף חדש ושינוי השדה הבולאני אינו משפיע בעוד שב-withFilter הוא כן משפיע

הערה: זה נכון גם לגבי Stream



```
List.range(1,10).filter(_ % 2 == 1 && !found).foreach(x => if (x == 5) found = true else println(x))
// 1, 3, 7, 9
found = false
Stream.range(1,10).filter(_ % 2 == 1 && !found).foreach(x => if (x == 5) found = true else println(x))
// 1, 3
```

Higher-Order Functions

פונקציה אשר **לוקחת** פונקציה כפרמטר ומחזירה פונקציה **כפלט**

Closures

כאשר פונקציה משתמשת בערך שמחוץ לסקופ הגדרתה לדוגמא:

```
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
```

Try and Catch

```
case class ZviMintsException(msg: String) extends Exception(msg)
try {
  throw new ZviMintsException("Sup?")
} catch {
  case ex: ZviMintsException => println("Print")
  case _ => println("Not Print")
}
```

Try

`Try[A]`
This is an Algebraic Data Type (**ADT**) composed of 2 cases: `Success[A]` and `Failure[A]`. This algebraic structure defines a lot of operations like `map`, `flatMap` and others. The fact that `Try` has a `map` and a `flatMap` plus a constructor from `A` to `Try[A]`, makes this structure a Monad.

מימוש:

```

abstract class Try[+T]{
  def flatMap[U](f: T => Try[U]): Try[U] = this match {
    case Success(x) => try f(x) catch {case NonFatal(ex) => Failure(ex) }
    case fail: Failure => fail
  }
  def map[U](f: T => U) : Try[U] = this match {
    case Success(x) => Try(f(x))
    case fail: Failure => fail
}
case class Success[T](num: T) extends Try[T]
case class Failure(ex: Throwable) extends Try[Nothing]
object Try {
  def apply[T](expr: => T) : Try[T] = {
    try {
      Success(expr)
    } catch {
      case NonFatal(ex) => Failure(ex)
    }
  }
}
```

ולכן ניתן להשתמש בfor-comprasion
שימוש:

```
case class ZviMintsException(msg: String) extends Exception(msg)
def throwRandomException() : Try[Double] = Try {
  val random: Double = Math.random()
  if(random < 0.5) throw ZviMintsException(random + "")
  else
    random
}
throwRandomException() match {
  case Success(v) => println("Success: " + v)
  case Failure(ex) => println(ex)
}

val l =  throwRandomException()
println(l.getOrElse("Error")) // Can print Error or 0.5212
```

בעזרת Try ניתן לעשות Recover אשר ממחליף ערך תקין או עם recoverWith אשר מחליף שגיאה בצורה הבאה:

```
val l =  throwRandomException()
println(l.getOrElse("Error")) // Can print Error or 0.5212
l.recover { case _ : Exception => -1.0 }.map(_*2).foreach(x => println(x)) // Can Print -2 or other
l.map(_*2).recover{ case _ : Exception => -1.0 }.foreach(x => println(x)) // Can Print -1 or other
val ans = l.map(_*2).recoverWith{ case ex:Exception => Failure(ZviMintsException("NEW")) } match {
  case Failure(ex) => println(ex) // Testing$ZviMintsException: NEW
  case _ => ()
}
```

Option

An Option[T] can be either **Some[T]** or **None** object, which represents a missing value. For instance, the get method of Scala's Map produces Some(value) if a value corresponding to a given key has been found, or **None** if the given key is not defined in the Map.

```
println({
  try {
    Some("3 + 3".toInt)
  } catch {
    case _ : Exception => None
  }
}.getOrElse(-1)) // -1
```

Either

Either is a disjoint union construct. It returns either an instance of Left[L] or an instance of Right[R]. It's commonly used for error handling, where by convention Left is used to represent failure and Right is used to represent success.

```
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
})
```

ההבדל בין Try ל-Either:

`Either[X, Y]` usage is more general. It can represent either an object of X type or of Y.
`Try[X]` has only one type and it might be either a Success[X] or a Failure (which contains a `Throwable`).

ניתן להמיר בינהם באופן הבא:

```
import scala.util.{Either, Failure, Left, Right, Success, Try }
implicit def eitherToTry[A <: Exception, B](either: Either[A, B]): Try[B] = {
  either match {
    case Right(obj) => Success(obj)
    case Left(err) => Failure(err)

  }
}
implicit def tryToEither[A](obj: Try[A]): Either[Throwable, A] = {
  obj match {
    case Success(something) => Right(something)
    case Failure(err) => Left(err)
  }
}
```

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
```

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, Public)	Class (Mutable, Private)
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 ״חינם״ מקבלים את המתודות הבאות :

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

Do **not** use **Case Classes** if :

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

(Tree) Class Hierarchies

```
abstract class IntSet {
  def incl(x: Int) : IntSet
  def contains(x: Int): Boolean
  def union(other: IntSet): IntSet
}
object Empty extends IntSet {
  override def incl(x: Int): IntSet =
    new NonEmpty(x, Empty, Empty)
  override def contains(x: Int): Boolean = false
  override def toString : String = "."
  override def union(other: IntSet): IntSet = other
}
class NonEmpty(data: Int, left: IntSet, right: IntSet) extends IntSet{
  override def incl(x: Int): IntSet = {
  if(x < data) left incl x
  else if(x > data) right incl x
  else this
  }
  override def contains(x: Int): Boolean = {
    if(x < data) left contains x
    else if(x > data) right contains x
    else true
  }
  override def toString : String =
    "{" + left + "(" + data + ")" + right + "}"

  override def union(other: IntSet): IntSet = {
    ((left union right ) union other) incl data
  }
}
```

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
```

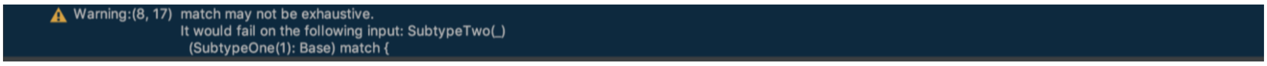
ניתן לעבור על tuples באופן הבא:

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

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 Base
final case class SubtypeOne(a: Int) extends Base
final case class SubtypeTwo(b: Option[String]) extends Base

(SubtypeOne(1): Base) match {
   case SubtypeOne(a) => println("Here")
}
```



```
⚠ Warning:(8, 17)  match may not be exhaustive.
                   It would fail on the following input: SubtypeTwo(_)
                    (SubtypeOne(1): Base) match {
```

Implicit

Implicit Parameters replace someCall(a) with someCall(a)(b),

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.

```
def example2(implicit x: Int, y: Int) : Int = x * y
implicit var value = 3
// implicit var value2 = 4 // Runtime
```

```
println(example2) // 9
// println(example(1)) // Wont compile
```

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

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

```
println("Hello ".MyNewFunction()) // World

// 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 Class SomeClass(a) with new SomeClass(a)(b)

```
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
// 3 x 4 -> RectangleFrom1Dim(3).x(4) -> Rectangle(3,4)
println(rectangle)
```

Singleton

```
// 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):

$$\begin{tikzcd} T \\ S \arrow[u] \end{tikzcd}$$

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

דוגמא:

```
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) // Valid
```

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
```

נניח כ S <: T

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

האם Array[S] <: Array[T] ? **התשובה היא לא 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

לסיכום:

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] /I/ contravariant
```

בגלל זה בספרייה הסטנדרטית:

```
List[+A]
```

או לחלופין

```
trait Function[-I, +O] extends AnyRef
```

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

MergeSort

```
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 Ordering 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 + " "))
```

InsertionSort

```
def insertionSort(list: List[Int]) : List[Int] = list match {
  case List() => List()
  case head :: tail => insert(head,insertionSort(tail))
}
def insert(x: Int, list: List[Int]) : List[Int] = list match {
  case List() => x :: Nil
  case y :: tail => if(x < y) x :: list else y :: insert(x,tail)
}
println(insertionSort(List(3,2,1)))
```

Stream

בעיה: חישובים איטיים

```
def isPrime(num: Int) : Boolean = (2 until num) forall(d => (num%d != 0))
def time = System.currentTimeMillis()
time
(1 to 10000) filter (x => isPrime(x))
time
```

יצירה:

```
(1 to 1000).toStream // Stream(1, ?)
val xs = Stream.cons(1, Stream.cons(2,Stream.empty)) //  Stream(1, ?)
```

הבדל בזמנים:

```
def isPrime(num: Int):Boolean = (2 until num) forall (d => (num % d != 0 ))
def time = System.nanoTime()
var time1 = time
(1 until 100000) filter(x => isPrime(x))
var time2 = time
println(time2 - time1) // 4044808212

time1 = time
val list : Stream[Int] = (2 until 100000).toStream filter(x => isPrime(x))
println(list) // Stream(2,?)
time2 = time
println(time2 - time1) // 21016738
```

או לחלופין:

```
val s: Stream[Int] = 3 #:: Stream.empty
```

מימוש: (הכל עם Call By Name)

```scala
trait Stream[+A] extends Seq[A] {
  def isEmpty : Boolean
  def head: A
  def tail: Stream[A]
}
object Stream {
  def cons[T](hd:T, tl: => Stream[T]) = new Stream[T] {
    override def isEmpty : Boolean = false
    override def head: T = hd
    override def tail: Stream[T] = tl
}
  val empty = new Stream[Nothing] {
    override def isEmpty : Boolean = true
    override def head = throw new NoSuchElementException("empty head")
    override def tail = throw new NoSuchElementException("empty tail")
  }
}
```

ניתן בעזרת Stream ליצור אוספים אינסופיים, לדוגמא:

```scala
def from(n: Int): Stream[Int] = n #:: from(n+1)
val natural_numbers: Stream[Int] = from(0)
natural_numbers foreach println
```

Future

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.

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

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

Note: Calling

```scala
Future { /* do something */ }
```

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

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

מתודות:

OnComplete

```scala
def onComplete(callback: Try[T] => Unit): Unit
```

שימוש:

```
val f1: Future[Int] = Future{ throw new Exception() }
val onCompleteAnwser: Unit = f1 onComplete {
  case Success(value) => println(value)
  case Failure(ex) => ()
}
```

recover

לוקחת Throwable ומחזירה T

```
def recover(f: PartialFunction[Throwable,T]) : Future[T]
```

recoverWith

לוקחת Throwable ומחזירה Future[T] כלומר ניתן לבצע שוב את אותו משימה

```
def recoverWith(f: PartialFunction[Throwable,Future[T]]): Future[T]
```

map

```
def map[U](f: T => U) : Future[U]
```

על מנת לא להפעיל Future בMap:

```
def print: Unit =  println("Here")
val list : List[Unit] = List.fill(3)(())
list.map(_ => () => print) // Will Not Print
list.map(_ => print) // Will Print
```

או לחלופין:

```
val list : List[Unit] = List.fill(3)(print) // Will Print
val list2: List[Unit] = List.fill(3)( () => print ) // Will not print
```

flatMap

```
def flatMap[U](f: T => Future[U]): Future[U]
```

שימוש:

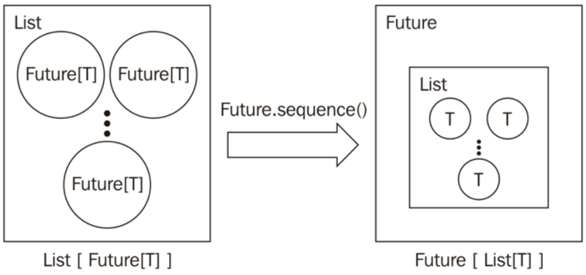
```
val confirmation: Future[Array[Byte]] =
  packet.flatMap(p => socket.sendToEur(p))
```

לקבל פקטה לוקח זמן ולשלוח לוקח זמן אז שילוב של השניים יתבצע בעזרת flatMap במקום לבצע onComplete ולבדוק אם יצא Succesfull ואז לשלוח פקטה אחרת לדחות

filter

```
def filter(f: T => Boolean) : Future[T]
```

Sequence



Scala Future API : Future.sequence()

דוגמת שימוש:

```scala
val x1 = Future {
  Thread.sleep(2000)
  println("x1 is completed")
  "Hello x1"
}

val x2 = Future {
  Thread.sleep(2000)
  println("x2 is completed")
  "Hello x2"
}

val x3 = Future {
  Thread.sleep(2000)
  println("x3 is completed")
  "Hello x3"
}

val x: Future[List[String]] = Future.sequence(List(x1, x2, x3))

x.onComplete {
  case Success(res) => println("Success: " + res) // List(Hello x1, Hello x2, Hello x3)
  case Failure(ex)  => println("Ohhh Exception: " + ex.getMessage)
}
```

Successful

Future.successful(None) just produces already completed **future**. It is more efficient. I don't think that **Future**(None) gives a big overhead, but still in it's default implementation each call to apply spawns a new task for a ForkJoin thread pool, whereas **Future.successful**(None) completes immediately

Failure

Calling Future { throw ex } and Future.failed(ex) will create an equivalent result. However, using Future.failed is more efficient.

fallBackTo

```scala
def fallbackTo[U >: T](that: Future[U]) : Future[U] = {
  this recoverWith {
    case _ =>
      that recoverWith {
        case _ => this
      }
  }
}
```

נניח שנרצה לנסות Future כלשהו, אבל אם הוא נכשל נרצה לעשות Future אחר ואם הוא נכשל אז פשוט נחזיר את הFuture המקורי.
זה fallBackTo

לדוגמא:

```
val f: Future[Int] = Future{ throw new Exception() }
val g: Future[String] = Future { "Works" }

f fallbackTo g map ( x => println(x) ) // Print Works
Thread.sleep(10)
```

בלוק:

```
val f: Future[Int] = Future{ 3 }
// Can FAIL!
val ans:Int = Await.result(f, 1 seconds) // import scala.concurrent.duration._
println(ans) // Can FAIL!
```

Synchornized

לא סינכרוני:

```
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
```

סינכרוני:

```
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!
```

Asyn

https://github.com/scala/scala-async

Dependeny Injection

https://www.playframework.com/documentation/2.6.x/ScalaDependencyInjection

קריאת קובץ JSON

```
/* JSON OBJ Example:
{ "firstName" : "Zvi",
  "lastName" : "Mints",
  "addresss" : {
            "street Name" : "Shevet reuven",
            "country" : " israel"
            },
   "phoneNumber": [
     { "type" : "home", "number" : "111" },
     { "type" : "fax", "number" : "222"
     ]
}
 */
sealed abstract class JSON
case class JSeq(list: List[JSON])       extends JSON
case class JObj(bind: Map[String,JSON])  extends JSON
case class JNum(num: Double)            extends JSON
case class JStr(s: String)              extends JSON
case class JBool(b: Boolean)            extends JSON
case object JNull                       extends JSON

def show(json: JSON) : String = json match {
  case JSeq(list) => "[" + list.foreach(ele => show(ele)) + "]"
  case JObj(bind) =>
     val ans : Iterable[String] = bind map {
       case (key,value) => "\"" + key + "\": " + show(value)
     }
    ans.mkString("{",",","}")
  case JNum(num) => num.toString
  case JStr(s) => "\"" + s + "\""
  case JBool(b) => b.toString
  case JNull => "null"
}
```