Paradigma Calculul Functional

Cursul nr. 12

Mihai Zaharia

Cum a început.... (au fost o dată ca niciodată ...)

ipoteza Church

și conducătorii lor

de doctorat ...

ipoteza Turing

Java vs Kotlin

```
public class JavraLambda //ex 1
{   interface Test
        { void test(); }
    private static void apel_test(Test test)
        { test.test(); }
    public static void main(String[] args)
        { apel_test(()->{ System.out.println("apel metoda
apel_test()"); }); }
}
fun apel_test(test:()->Unit) //ex 2
{ test()  }
fun main(args: Array<String>)
{ apel_test({ println("apel functie apel_test()");})  }
```

Utilizarea unei funcții ca o proprietate

```
fun main(args: Array<String>)
{
    val dif_numere = { x: Int, y: Int -> x - y }
    println("Diferenta 1 este ${dif_numere(33,66)}")
    println("Diderenta 2 este ${dif_numere(77,11)}")
}
```

Sintaxa specifică funcțiilor Lambda în Kotlin

```
fun main(args: Array<String>) {
    val invers:(Int)->Int
    invers = {numar ->
        var n = numar
        var numarInvers = 0
        while (n>0) {
            val digit = n%10
            numarInvers=numarInvers*10+digit
            n/=10
        }
        numarInvers
    }
    println("inversul lui 123 ${invers(123)}")
    println("inversul lui 456 ${invers(456)}")
    println("inversul lui 789 ${invers(789)}")
}
```

si exemplul de executie

```
inversul lui 123 321
inversul lui 456 654
inversul lui 789 987

Process finished with exit code 0
```

Funcții de nivel superior

```
fun procesareNrPare(numar:Int,procesare:(Int)->Int):Int {
    if(numar%2==0) {
        return procesare(numar)
    } else {
        return numar
    }
}
fun main(args: Array<String>) {
    var nr1=4
    var nr2 = 5
    println("Apel cu ${nr1} si operatia (it*2): ${procesareNrPare(nr1,{it*2})}")
    println("Apel cu ${nr2} si operatia (it*2): ${procesareNrPare(nr2, {it*2})}")
}
```

Funcții de nivel superior

```
fun apelCuIntoarcere(n:Int):(String)->Char {
    return { it[n] }
}
fun main(args: Array<String>) {
    var pos = 4
    print("${apelCuIntoarcere(1)("abc")}\n")
    print("${apelCuIntoarcere(0)("def")}\n")
    try {
        print(apelCuIntoarcere(pos)("ghi"))
    }
    catch (e: StringIndexOutOfBoundsException) {
        print("Indexul ${pos} este in afara sirului")
    }
}
```

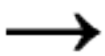
Efecte laterale

```
class operatiiAritmetice {
   var nr1:Int=0
   var nr2:Int=0
   fun suma(nr1:Int = this.nr1,nr2:Int =
this.nr2):Int {
      this.nr1 = nr1
      this.nr2 = nr2
      return nr1+nr2
   }
}
fun main(args: Array<String>) {
   var nr1 = 10
   var nr2 = 15
   val obCalcul = operatiiAritmetice()
   println("Suma dintre ${nr1} si ${nr2}
este ${obCalcul.suma(nr1,nr2)}")
}
```

```
class functionalOperatiiAritmetice {
   val sumaf: (Int, Int) -> Int = { x, y -> x + y }
   fun executaOperatia(x: Int, y: Int, op: (Int,
Int) -> Int): Int = op(x, y)
}
fun main(args: Array<String>) {
   var nr1 = 10
   var nr2 = 15
   val obFCalcul =
functionalOperatiiAritmetice()
   println("Suma dintre ${nr1} si ${nr2} este
${obFCalcul.executaOperatia(nr1,nr2,obC
alcul.sumaf)}")
 }
```

Efecte laterale

```
fun main(args: Array<String>) {
    var nr1 = 10
    var nr2 = 15
    val sumaf: (Int, Int) -> Int = { x, y -> x + y }//stil functional
    fun sumak(x: Int, y: Int) = x + y //stil kotlin
    fun executaOperatia(x: Int, y: Int, op: (Int, Int) -> Int): Int = op(x, y)
    println("Suma dintre ${nr1} si ${nr2} este ${executaOperatia(nr1,nr2,sumaf)}")
    println("Suma dintre ${nr1} si ${nr2} este ${executaOperatia(nr1,nr2,::sumak)}")
}
```



Funcții pure

```
fun suma(a:Int = 0,b:Int = 0):Int {
    return a+b
}
```

vararg

```
fun calculMedie(listaNumere: List<Int>): Float {
    var suma = 0.0f
    for (element in listaNumere) {
        suma += element
    }
    return (suma / listaNumere.size)
}
fun calculMedieParametri(vararg lista_parametri: Int):
Float {
    var suma = 0.0f
    for (element in lista_parametri) {
        suma += element
    }
    return (suma / lista_parametri.size)
}
fun <T> enumerareCaOLista(vararg parametri_intrare:
T): List<T> {
    val rezultat = ArrayList<T>()
    for (element in parametri_intrare)
        rezultat.add(element)
    return rezultat
}
```

```
fun main(args: Array<String>) {
    val tablou = arrayListOf(1, 2, 3, 4)
    val rezultat = calculMedie(tablou)
    print("\nMedia este ${rezultat}")
    val rezultat1 = calculMedieParametri(1, 2, 3)
    print("\nMedia1 este ${rezultat1}")
    val rezultat3 = enumerareCaOLista(1, 2, 3, 4, 5, 6, 7,
8, 9)
    print("\nTransformare enumerare in lista
${rezultat3}")
    print("\nMedia1 este
${calculMedieParametri(*rezultat3.toIntArray())}")
    val tablou1 = intArrayOf(1, 2, 3, 4)
    val rezultat4 = calculMedieParametri(5, 6, 7,8,9,
*tablou1)
    print("\nMedia1 este ${rezultat4}")
}
```

Parametri alias - Named parameters

```
typealias Kg = Double
typealias cm = Int
data class ClientBanca (val numeFamilie: String,
                        val numeMijlociu: String,
                        val numeMic: String,
                        val seriePasaport: String,
                        val greutate: Kg,
                        val inaltime: cm,
                        val semneParticulare: String)
fun main(args: Array<String>) {
   val client1 = ClientBanca("Mike", "Mouse", "Rabbit", "XX234837447", 82.3, 180, "nu are")
   val client2 = ClientBanca(
      numeMic = "Rabbit",
      numeMijlociu = "Mouse",
      numeFamilie = "Mike",
      seriePasaport = "xe4244rf33333",
      greutate = 100.0,
      inaltime = 180,
      semneParticulare = "nu are"
   )
print("\n${client1}")
print("\n${client2}")
}
```

alias cu vararg sau funcții de nivel superior

```
fun paramDupaVararg(nrCurs: Int, vararg lista_studenti: String, tempCamera: Double) {//ex1
    //corpul functiei
}
paramDupaVararg(688, "Gica", "Bula", "Andreea", "Veorica", tempCamera = 15.0)//apel
fun test(f: (Int, String) -> Unit) {//ex2
   f(1, "Bula")
}//cu apelul
test { q, w ->
    //procesare
}//poate fi rescrisa ca
fun test(f: (nume:String, varsta:Int) -> Unit) {
   f(“Strula", 10)
}//daca incerc insa ca mai jos
fun test(f: (nume:String, varsta:Int) -> Unit) {
   f(nume = "kati", virsta = 3, ) //eroare de compilare}
```

Funcții de extensie

```
fun String.trimitLaConsola() = println(this) // la tip
class Om(val nume: String)
fun Om.spune(): String = "${this.nume} spune Vai" //la clasa
fun main(args: Array<String>) {
    "lA examen".trimitLaConsola()
    val x=Om("Bula")
    x.spune().trimitLaConsola()
}
```

si rezultatul executiei

```
lA examen
Bula spune Vai

Process finished with exit code 0
```

Funcții de extensie și funcții membre

```
open class Canina {
   open fun vorbeste() = "Un animal din clasa Caninelor face: ham ham!"   }
fun mesajScris(canina: Canina) {
   println(canina.vorbeste())
}
class Caine : Canina() {
   override fun vorbeste() = "Un caine face: vauf vauf!" //modificare comportament in clasa derivata
}
fun mesajScris1(canina: Canina) {
   println(canina.vorbeste1())
}
fun Caine.vorbeste()="Din functia extensie Un caine face haf haf" //fiind functie de extensie nu supraincarca!!!
deci este ignorata!!!
fun Canina.vorbeste1() = "functie extensie specifica cainelui"
fun main(args: Array<String>) {
   mesajScris(Canina())
   mesajScris(Caine()) //baza polimorfismului
   mesajScris1(Caine()) //desi este definita la nivelul clasei canina prin mosternire poate fi apelata si in caine
}
```

Funcții de extensie și funcții membre

```
open class Primata(val name: String)
fun Primata.vorbeste() = "$name: uhaha uhaha"
open class MaimutaMare(name: String) : Primata(name)
fun MaimutaMare.vorbeste() = "${this.name} : urlet"
fun mesajScris(primata: Primata) {
   println(primata.vorbeste())
}
fun mesajScris1(maimutza: MaimutaMare) {
   println(maimutza.vorbeste())
}
fun main(args: Array<String>) {
   mesajScris(Primata("alex")) // apeleaza vorbeste din primata
   mesajScris(MaimutaMare("crrr")) // apeleaza vorbeste din primata
   mesajScris1(MaimutaMare("ciii")) // apeleaza vorbeste din maimuta
   mesajScris1(Primata("jiii") as MaimutaMare) // apeleaza vorbeste din maimutaeroare nu pot
converti primata la maimuta
}
```

Dispatch recv

```
open class Felina
open class Pisica():Felina()
open class Primata(val name: String)
fun Primata.vorbeste() = "$name: face uhaha uhaha"//extensie primate
open class MaimutaMare(name: String) : Primata(name)
fun MaimutaMare.vorbeste() = "${this.name} : urlet" //ignorata deoarece amm deja un vorbeste
din primata
fun mesajScris(primata: Primata) {    println(primata.vorbeste()) }
open class Ingrijitor(val name: String) {
   open fun Felina.react() = "HRRMR!!!"
   fun Primata.react() = "$name se joaca cu ${this@Ingrijitor.name}" //ditribuitor intern bagat aici
pentru ca altfel nu am acces la .name
   fun areGrija(felina: Felina) {        println("Felina face: ${felina.react()}")     }
   fun areGrija(primata: Primata){    println("Primata spune: ${primata.react()}")      }
   fun Ingrijitor.react() = "$name din afara clasei se joaca cu ${this@Ingrijitor.name}" //neglijat }
open class Vet(name: String): Ingrijitor(name) {   override fun Felina.react() = "fuge de $name"   }
```

Dispatch recv

```
fun main() {
   val om = Ingrijitor("ingrijitorul")
   val pisica = Pisica()
   val maimutaMare = Primata("maimuta")
   val femeie = Vet("corin")
   mesajScris(Primata("alex"))//apel functie din primata
   mesajScris(MaimutaMare("gibon"))//apel functie din primata
   om.areGrija(pisica)//apel functie din felina
   om.areGrija(maimutaMare)//apel dispatcher intern
   listOf(om, femeie).forEach { ingrijitor ->
      println("${ingrijitor.javaClass.simpleName} ${ingrijitor.name}")//Vet corin
      ingrijitor.areGrija(pisica)//Felina face: fuge de corin
      ingrijitor.areGrija(maimutaMare)//Primata spune: maimuta se joaca cu
corin      }
}
```

Funcții de extensie - conflict posibil de nume

```
class Sclav {
   fun munca() = "*munca la birt*"
   private fun odihna() = "*odihna la vecina*"
}

fun Sclav.munca() = "munca cu mila" //neglijata
fun <T> Sclav.munca(t:T) = "*muncesc azi? nuuuu $t*"
fun Sclav.odihna() = "odihna in sant"

fun main()
{
   val sclav = Sclav()
   println(sclav.munca()) //apel f membru
   println(sclav.munca("de maine ma apuc"))
   println(sclav.odihna()) //apel f extensie
}
```

Funcții de extensie pentru obiecte

```
object Constructor {
}

fun Constructor.casaNouaCaramida() = "casa pe pamant"

class Proiectant {
   companion object {
   }
   object Birou {
   }
}
fun Proiectant.Companion.prototipNou() = "montaj test"
fun Proiectant.Birou.mapaDeLucrari() = listOf("Proiect casa", "Proiect bloc")
fun main()
{
   println(Constructor.casaNouaCaramida())
   println(Proiectant.prototipNou())
   Proiectant.Birou.mapaDeLucrari().forEach(::println)
}
```

Funcții de extensie pentru liste

```
fun <T> List<T>.tail(): List<T> = this.drop(1)

infix fun <T> T.prependTo(list: List<T>): List<T> = listOf(this) + list

fun <T> List<T>.destructured(): Pair<T, List<T>> = first() to tail()

fun main()
{
   val intregi = listOf(11, 5, 3, 8, 1, 9, 6, 2)
   println(intregi.tail())
   println(intregi.prependTo(intregi))
   println(intregi.destructured())
}
```

Funcții infix

```
infix fun Int.sumaSmecheraCu(i: Int) = this + i //exemplul 1
fun main() {
   println(4 sumaSmecheraCu  7)
   println(2.sumaSmecheraCu(11))
}

object Toate {  // exemplul 2
   infix fun aleTale(base: Pair<Masini, Noua>) {}
}
object Masini {
   infix fun ne(apartin: Apartin) = this
}
object Apartin
object Noua
fun main() {
   println(Toate aleTale  (Masini ne Apartin to Noua))
}
```

Funcția map

```
fun main(args: Array<String>) {
    val lista1 = listOf<Int>(7,15,24,19,8,45,65,55)
    val lista2 = List(10) {
        (1..12).shuffled().first()
    }
    val lista3 = (1..15).map { it }
    val transformareLista = lista1.map { it*2 }
    println("lista 1 =${lista1}")
    println("Transformare lista1 cu map -> $transformareLista")
    println("lista 2 =${lista2}")
    println("lista 3 =${lista3}")
}
```

si rezultatul executieil

```
ista 1 =[7, 15, 24, 19, 8, 45, 65, 55]
Transformare lista1 cu map -> [14, 30, 48, 38, 16, 90, 130, 110]
lista 2 =[11, 4, 1, 9, 7, 12, 10, 7, 2, 2]
lista 3 =[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Process finished with exit code 0
```

Funcția Filtru - filter

```
import kotlin.math.*

fun main(args: Array<String>) {
    val lista = 1.until(15).toList()
    val lista1 = (1..15).map { it }
    val sublistaImpare = lista.filter { it%2==0 }
    println("Sublista care contine numai numerele pare este -> $sublistaImpare")
    val listaPatrate = lista1.filter {
        val radacinaPatrata = sqrt(it.toDouble()).roundToInt()
        radacinaPatrata*radacinaPatrata==it //conditia de filtrare
    }
    println("filteredListPSquare -> $listaPatrate")
}
```

si rezultatul executiei

```
Sublista care contine numai numerele pare este -> [2, 4, 6, 8, 10, 12, 14]
filteredListPSquare -> [1, 4, 9]

Process finished with exit code 0
```

Funcția FlatMap

```
fun main(args: Array<String>) {
    val lista1 = List(10) {
        (26..120).shuffled().first()
    }
    val listaBatutaCuCiocanul = lista1.flatMap {
        it.rangeTo(it+2*it).toList()
    }
    println("Lista dupa aplicarea flat Map este $listaBatutaCuCiocanul")
}
```

si rezultatul executiei

```
lista de intrare este [80, 35, 76, 85, 58, 42, 43, 98, 97, 110]
Lista dupa aplicarea flat Map este [80, 81, 82, 83, 84, 85, 86, 87, 88, ... mai sunt destule elemente

Process finished with exit code 0
```