

## Laboratorul 12: Paradigma calculului funcțional în Kotlin

### 1. Introducere

Pentru o imagine de ansamblu asupra paradigmei de calcul funcțional în Kotlin, se recomandă parcurgerea următoarelor resurse:

- „**Functional Kotlin**“ - **Mario Arias și Rivu Chakraborty**
- **Cursul 12** de la disciplina *Paradigme de programare*

Programarea funcțională este o paradigmă în care accentul este pus pe transformarea datelor cu expresii (ideal, aceste expresii ar trebui să nu aibă *side effects*).

#### 1.1 Expresii lambda și funcții anonime

Expresiile lambda și funcțiile anonime sunt funcții care nu sunt declarate, dar sunt folosite imediat ca o expresie.

```
val capitalize = { str: String -> str.capitalize() }

fun main() {
    println(capitalize("hello world!"))
}
```

Funcția *capitalize* este o **funcție lambda** a cărei tip este `(String) -> String` (syntactic sugar). Cu alte cuvinte, primește un `String` ca parametru și returnează un `String`. Acest tip de date este o scurtătură pentru `Function1<String, String>`, unde `Function1<P1, R>` este o interfață definită în biblioteca standard Kotlin și are o singură metodă, `invoke(P1): R` marcată ca operator.

Funcția de mai sus este echivalentă cu:

```
val capitalize = object : Function1<String, String> {
    override fun invoke(p1: String): String {
        return p1.capitalize()
    }
}
```

##### 1.1.1 Număr variabil de argumente

```
fun <T>varargFun(vararg items: T) {
    items.forEach(::println)
}

fun main() {
    varargFun(1)
    varargFun(1.1, 2.2)
    varargFun("Cristina", "Andrei", "Roxana")
}
```

##### 1.1.2 Parametru vararg pentru funcții lambda

```
fun <T> emit(t: T, vararg listeners: (T) -> Unit) = listeners.forEach { listener ->
    listener(t)
}
```

```

fun main() {
    emit(1, ::println, {i -> println(i * 2)})
    emit(
        listOf('a', 'b', 'c'),
        ::println,
        {list -> println(list.joinToString(prefix="<",
                                          postfix=">",
                                          separator=""))})
    }
}

```

## 1.2 Funcții de ordin superior (higher-order function)

Funcția de ordin superior<sup>1</sup> este o funcție care primește funcții ca parametri sau returnează o funcție.

Funcțiile lambda pot fi utilizate ca parametri în alte funcții:

```

fun <T> transform(t: T, fn: (T) -> T): T {
    return fn(t)
}

fun main() {
    println(transform("kotlin", { str: String -> str.capitalize() }))
    println(transform("kotlin", { it.capitalize() }))
}

```

În exemplul de mai sus, *it* este un parametru implicit (nu trebuie declarat) care poate fi utilizat în funcțiile lambda cu un singur parametru.

**ATENȚIE:** parametrul *it* ar trebui folosit doar în cazul în care este clar ce tip de date se folosește.

De asemenea, dacă se dorește ca o funcție clasică să fie trimisă ca parametru, se poate utiliza *double colon (::)*

```

fun reverse(str: String): String {
    return str.reversed()
}

fun main() {
    println(transform("kotlin", ::reverse))
}

```

Pentru a trimite funcții dintr-o clasă ca parametru în alte funcții:

```

class Transformer {
    fun upperCased(str: String): String {
        return str.toUpperCase()
    }
    companion object {
        fun lowerCased(str: String): String {
            return str.toLowerCase()
        }
    }
}

```

<sup>1</sup> <https://kotlinlang.org/docs/reference/lambdas.html>

```

fun main() {
    val transformer = Transformer()
    println(transform("kotlin", transformer::upperCased))
    println(transform("kotlin", Transformer.Companion::lowerCased))
}

```

Utilizând funcții de ordin superior, se pot aplica operații dacă sunt îndeplinite anumite condiții:

```

fun performOperationOnEven(number:Int, operation:(Int)->Int):Int {
    if(number%2==0) {
        return operation(number)
    } else {
        return number
    }
}
fun main() {
    println("Called with 4, (it*2): ${performOperationOnEven(4, {it*2})}")
    println("Called with 5, (it*2): ${performOperationOnEven(5, {it*2})}")
}

```

Un exemplu de funcție de ordin superior care returnează o altă funcție:

```

fun getAnotherFunction(n:Int):(String)->Unit {
    return {
        println("n:$n it:$it")
    }
}
fun main() {
    getAnotherFunction(0) ("laborator")
    getAnotherFunction(2) ("PP")
    getAnotherFunction(3) ("Functional Kotlin")
}

```

Funcția `getAnotherFunction()` primește un număr întreg și returnează o funcție care primește un `String` și nu returnează nimic (doar afișează). În `main`, se apelează totodată și funcția returnată de `getAnotherFunction()`, la consolă afișându-se:

```

n:0 it:laborator
n:2 it:PP
n:3 it:Functional Kotlin

```

## 1.3 Funcții pure și efecte secundare (side effects)

### 1.3.1 Efecte secundare

Într-un program, când o funcție modifică orice obiect/date din afara scopului propriu, acest lucru se numește **efect secundar**.

O funcție care modifică o proprietate statică sau globală, modifică un argument, generează o excepție, scrie output-ul la consolă / în fișier, sau apelează o altă funcție, **are un efect secundar**.

```

class Calc {
    var a:Int=0
    var b:Int=0
}

```

```

    fun addNumbers(a:Int = this.a, b:Int = this.b): Int {
        this.a = a
        this.b = b
        return a + b
    }
}
fun main() {
    val calc = Calc()
    println("Result is ${calc.addNumbers(10,10)}")
}

```

În exemplul de program orientat pe obiecte de mai sus, se observă aceste efecte secundare, întrucât funcția `addNumbers()` modifică starea clasei `Calc`, ceea ce nu e indicat în programarea funcțională. Deși nu se pot evita întotdeauna efectele secundare (spre exemplu în cazul accesării IO și/sau bazei de date), aceste efecte trebuie îndepărtate oricând este posibil.

### 1.3.2 Funcții pure

O funcție pură este o funcție a cărei rezultat returnat este complet dependent de parametrii acesteia. Pentru fiecare apel al unei funcții pure cu același parametru, aceasta va produce mereu același rezultat.

De asemenea, o funcție pură **NU** trebuie să cauzeze *efecte secundare*, sau să apeleze alte funcții.

Eliminând efectele secundare din funcția anterioară, se obține:

```

fun addNumbers(a:Int = 0,b:Int = 0):Int {
    return a+b
}
fun main() {
    println("Result is ${addNumbers(10,10)}")
}

```

### 1.4 Funcții extensie

Funcțiile extensie permit modificarea tipurilor existente cu funcții noi. Sintaxa pentru adăugarea unei funcții extensie la un tip existent: `NumeClasă.funțieExtensie(listăParametri)`

```

fun String.toPascalCase0(): String {
    val components = this.split(" ")
    var result: String = ""
    for(component in components) {
        result += component.capitalize()
    }
    return result
}

fun String.toPascalCase1(): String = this.split(" ")
    .map{it.capitalize()}.joinToString(separator="")

fun String.toPascalCase2(): String = this.split(" ")
    .joinToString(separator = "") { it.capitalize() }

val toPascalCase0 = {str: String -> str.split(" ")
    .map{it.capitalize()}.joinToString(separator="")}

```

```

val toPascalCase1 = {str: String -> str.split("
").joinToString(separator = "") { it.capitalize() } }

fun main() {
    println("Functii extensie: ")
    println("whatever name you want".toPascalCase0())
    println("whatever name you want".toPascalCase1())
    println("whatever name you want".toPascalCase2())
    println("Expresii lambda stocate in variabile: ")
    println(toPascalCase0("whatever name you want"))
    println(toPascalCase1("whatever name you want"))
}

```

## 1.5 Operații pe datele dintr-o colecție

```

fun main() {
    val list = 1.until(5).toList() // [1, 2, 3, 4]
    // filter - returnează doar elementele care îndeplinesc condiția
    // din predicat
    list.filter{ it % 2 == 0 } // [2, 4]

    // map - returnează elementele unei colecții după prelucrarea
    // acestora
    list.map{ it * 2 } // [2, 4, 6, 8]

    // flatMap - returnează rezultatul concatenării mai multor colecții
    list.flatMap{ listOf(it, it+10) } // [1, 11, 2, 12, 3, 13, 4, 14]

    // fold / reduce - acumularea elementelor
    list.fold(0.0) {acc, i -> acc + i } // 10
    list.reduce{ acc, i -> acc * i } // 24

    // forEach / onEach - realizează o acțiune pe fiecare element din
    // colecție
    list.forEach{ println(it) } // afișează 1234, returnează Unit
    //list.forEach(::println)
    list.onEach{ println(it) } // afișează 1234, returnează [1, 2, 3,
4]
    //list.onEach(::println)

    // partition - împarte într-o pereche de liste
    val (even, odd) = list.partition{ it % 2 == 0 }
    println(even) // [2, 4]
    println(odd) // [1, 3]

    // min, max
    list.min() // 1
    list.max() // 4

    // first, firstBy
    list.first() // 1
    list.first{ it % 2 == 0 } // first even: 2

    // count - numără elementele care îndeplinesc condiția din predicat

```

```

list.count{ it % 2 == 0 } // 2

// sorted, sortBy - returnează colecția sortată
listOf(2, 3, 1, 4).sorted() //returnează o nouă listă, sortată:
[1, 2, 3, 4]
list.sortedBy{ it % 2 } // [2, 4, 1, 3]

// groupBy - grupează elementele unei colecții după cheie
list.groupBy{ it% 2 } // Map: {1=[1, 3], 0=[2, 4]}

// distinct, distinctBy - returnează doar elementele unice
listOf(1, 1, 1, 2, 2, 3).distinct() // [1, 2, 3]

// drop, dropLast - elimină primele/ultimele n elemente din
colecție
val longList = 1.until(50).toList()
longList.drop(25) // [26, 27, 28, ..., 49]
longList.dropLast(25) // [1, 2, 3, ..., 24]

// take, takeLast, takeWhile, takeLastWhile - selectează
primele/ultimele n elemente din colecție
longList.take(25) // [1, 2, 3, ..., 25]
longList.takeLast(25) // [25, 26, 27, ..., 49]
longList.takeWhile { it <= 10 } // [1, 2, 3, ..., 10]
longList.takeLastWhile { it >= 40 } // [40, 41, 42, ..., 49]

// zip - funcția fermoar (zip) ia câte un element din fiecare
colecție și formează o pereche
val list1 = listOf(1,2,3,4,5,6,7,8)
val list2 = listOf(
    "Item 1", "Item 2", "Item 3", "Item 4", "Item 5")
list1.zip(list2) // [(1, Item 1), (2, Item 2), (3, Item 3), (4,
Item 4), (5, Item 5)]
list1.zipWithNext() // [(1, 2), (2, 3), (3, 4), (4, 5), (5, 6),
(6, 7), (7, 8)]
}

```

## 1.6 Functori, funcții ca functori și delegați

### 1.6.1 Functori

Un functor este un tip care definește o modalitate de a transforma (transform/map) conținutul lui.

```

fun inc(value: Int): Int = value + 1

class ValueFunctor<T>(val value: T) {
    fun map(function: (T) -> T): ValueFunctor<T> {
        return ValueFunctor(function(value))
    }
}

class CollectionFunctor<T>(val list: List<T>) {
    fun map(function: (T) -> T): CollectionFunctor<T> {
        var result = mutableListOf<T>()
        for(item in list) {

```

```

        result.add(function(item)
        }
        return CollectionFunctor(result)
    }
}

fun main() {
    println(ValueFunctor(1).map(::inc).map(::inc).map(::inc).value)
    println(CollectionFunctor(listOf(1, 2, 3, 4)).map{ it * 2 }.map{
it + 3}.list)
}

```

## 1.6.2 Delegați

### 1.6.2.1 Funcția Delegates.notNull și lateinit

```

import kotlin.properties.Delegates

var notNullStr:String by Delegates.notNull<String>()
lateinit var notInit: String

fun main(args: Array<String>) {
    notNullStr = "Initial value"
    println(notNullStr)
    println(notInit)
}

```

Delegatul `Delegates.notNull` permite continuarea programului fără inițializarea proprietății `notNullStr`. Dacă acea variabilă este utilizată înainte de a fi inițializată, va genera o excepție.

Funcția `lateinit` este o variantă mai simplă pentru a obține același comportament.

**Observație:** `Delegates.notNull` și `lateinit` funcționează numai pentru proprietăți declarate cu `var`.

### 1.6.2.2 Funcția lazy

Spre deosebire de `lateinit` și `Delegates.notNull`, la funcția `lazy` trebuie specificată variabila de inițializare în momentul declarării, avantajul fiind că inițializarea nu va fi executată până când variabila este folosită.

```

fun main(args: Array<String>) {
    val i by lazy {
        println("Lazy evaluation")
        123
    }
    println("before using i")
    println(i)

    // the following throws ArithmeticException
    val size0 = listOf(2+1, 3*2, 1/0, 5-4).size

    // the following is a lazy evaluation (1 / 0 is not executed)
    val size1 = listOf({ 2+1 }, { 3*2 }, { 1/0 }, { 5-4 }).size
}

```

### 1.6.2.3 Funcția Delegates.observable

Funcția `Delegates.observable` are nevoie de doi parametri pentru a crea delegatul: valoarea inițială a proprietății și o funcție lambda care să fie executată de fiecare dată când se schimbă valoarea proprietății. Funcția lambda primește 3 parametri:

- o instanță de `KProperty<out R>` (o proprietate - `val` sau `var`)
- valoarea veche a proprietății
- valoarea nou asignată

```
import kotlin.properties.Delegates

var myStr:String by Delegates.observable("<Initial Value>") {
    property, oldValue, newValue ->
    println("Property `${property.name}` changed value from $oldValue
to $newValue")
}
fun main() {
    myStr = "Change Value"
    myStr = "Change Value again"
}
```

### 1.6.2.4 Funcția Delegates.vetoable

Dreptul de *veto* permite o verificare logică la fiecare asignare a unei proprietăți, unde se poate decide dacă se continuă cu asignarea sau nu.

```
import kotlin.properties.Delegates

var myIntEven:Int by Delegates.vetoable(0) {
    property, oldValue, newValue ->
    println("${property.name} $oldValue -> $newValue")
    newValue%2==0
}
fun main() {
    myIntEven = 6
    myIntEven = 3
    println("myIntEven:$myIntEven") // 6
}
```



## Exemple

### Exemplu de utilizare ale secvențelor și funcțiilor de prelucrare a datelor



```
import java.time.LocalDate
import java.time.Period
import java.time.format.DateTimeFormatter

data class Person(val firstName: String, val lastName: String, val
dateOfBirth: LocalDate, val emailAddress: String) {
    override fun toString(): String {
        return firstName + " " + lastName + ", " +
dateOfBirth.format(DateTimeFormatter.ofPattern("d MMM yyyy"))
    }
}

fun main() {
    var persons = listOf(
        Person("John", "Doe", LocalDate.of(1960, 11, 3),
"jdoe@example.com"),
        Person("Ellen", "Smith", LocalDate.of(1992, 5, 13),
"ellensmith@example.com"),
        Person("Jane", "White", LocalDate.of(1986, 2, 1),
"janewhite@example.com"),
        Person("Bill", "Jackson", LocalDate.of(1999, 11, 6),
"bjackson@example.com"),
        Person("John", "Smith", LocalDate.of(1975, 7, 14),
"johnsmith@example.com"),
        Person("Jack", "Williams", LocalDate.of(2005, 5, 28), "")
    )

    // sort by age and find youngest and oldest person
    val youngest = persons.asSequence().sortedByDescending { person:
Person -> person.dateOfBirth }.first()
    val oldest = persons.asSequence().sortedByDescending { person:
Person -> person.dateOfBirth }.first()
    println("youngest person is: $youngest")
    println("oldest person is: $oldest\n\n")

    // filter under age
    val underage = persons.asSequence()
        .filter { person: Person ->
            Period.between(
                person.dateOfBirth,
                LocalDate.now()
            ).years < 18
        }
        .toList()
    println("underage: $underage\n\n")
}
```

```

    // list of emails
    val emails = persons.asSequence().map { obj: Person ->
obj.emailAddress }.toList()
    println("emails: $emails\n\n")

    // map of name and email
    val emailsMap = persons.asSequence().map{ person: Person ->
(person.firstName + " " + person.lastName) to person.emailAddress
}.toMap()
    println("emails: $emailsMap\n\n")

    // map of email and person
    val emailPersonMap = persons.asSequence().map { person: Person ->
person.emailAddress to person }.toMap()
    emailPersonMap.forEach(::println)

    // group by month of birthday
    val peopleToCelebrateEachMonth = persons.asSequence().groupBy {
person: Person -> person.dateOfBirth.month }
    println("birthdays each month: $peopleToCelebrateEachMonth\n\n")

    // partition
    val mapByBirthYear = persons.asSequence()
        .partition { person: Person -> person.dateOfBirth.year <= 1980
}
    println("born before / after 1980 : $mapByBirthYear\n\n")

    // distinct first names
    val names = persons.asSequence()
        .map { obj: Person -> obj.firstName }
        .distinct().joinToString(separator=", ")
    println("first names: $names\n\n")

    // average age
    val averageAge = persons.asSequence().map { person: Person ->
Period.between(person.dateOfBirth, LocalDate.now()).years.toDouble()
}.average()
    println("Average age: $averageAge\n\n")

    // count
    val smiths = persons.asSequence()
        .filter { person: Person -> person.lastName == "Smith" }
        .count()
    println("number of people called Smith: $smiths\n\n")

    // find any with optional result
    val optional: Person? = persons.asSequence()
        .filter { person: Person -> person.firstName == "John"
}.firstOrNull()
    optional?.let {
        println(optional!!)
    } ?: run {
        println("No one named John was found")
    }

    // find any with optional and alternative result

```

```
val searchResult = persons.asSequence()
    .filter { person: Person -> person.firstName == "Thomas" }
    .map { obj: Person -> obj.lastName }.elementAtOrElse(0) { _ ->
"No one named Thomas was found."}
println(searchResult)

// check any with missing email
val noEmail = persons.asSequence().any { person: Person ->
person.emailAddress.isEmpty() }
println("any with missing email: $noEmail")
}
```

## Aplicații și teme

### Aplicații de laborator:

1. Să se implementeze o funcție extensie pe clasa `Int` care să determine dacă numărul este prim sau nu
2. Să se implementeze o extensie de convertit o dată calendaristică din `String` în `Date`. Aceasta va primi ca parametru formatter-ul.
3. Să se inverseze cheile cu valorile dintr-un map utilizând expresii lambda și funcția `map`. Spre exemplu pentru `{1: "abc", 2: "def", 3: "ghi"}` se va returna `{"abc": 1, "def": 2, "ghi": 3}`
4. Să se inițializeze o variabilă care conține un număr prim folosind `Delegates.vetoable` și o funcție extensie de verificare dacă numărul este prim
5. Să se citească de la tastatură un număr `n` care reprezintă de câte ori se va replica fiecare element dintr-o lista. Spre exemplu, pentru `n = 3` și lista `[1, 2, 3]` se va afișa `[1, 1, 1, 2, 2, 2, 3, 3, 3]`
6. Utilizând secvențe, să se elimine caracterele duplicate dintr-un string citit de la tastatură (`aaaabbbcc` devine `abc`)
7. Utilizând secvențe, să se micșoreze un string citit de la tastatură astfel: `aaaabbbccd` devine `a4b3c2d`. Dacă caracterul apare o singură dată, nu se va adăuga count-ul.

### Tema pe acasă:

1. Utilizând programarea funcțională, să se efectueze următoarele operații pe lista `[1, 21, 75, 39, 7, 2, 35, 3, 31, 7, 8]`:
  - Eliminarea oricărui număr `< 5` --> `[21, 75, 39, 7, 35, 31, 7, 8]`
  - Gruparea numerelor în perechi --> `[(21, 75), (39, 7), (35, 31), (7, 8)]`
  - Multiplicarea numerelor din perechi --> `[1575, 273, 1085, 56]`
  - Sumarea rezultatelor --> `1575 + 273 + 1085 + 56 = 2989`
2. Să se citească dintr-un fișier text conținutul și să fie prelucrat astfel încât toate cuvintele cu lungime cuprinsă între 4 și 7 caractere să fie criptate cu cifra Caesar cu offset dat.
3. Fiind dat un număr `n` care reprezintă numărul de puncte ce formează un poligon, utilizând funcțiile `zip` și `zipWithNext`, să se calculeze perimetrul poligonului.

Exemplu de input:

```
4
0 0
0 1
1 0
1 1
```

Se va returna 4.

**Atenție:** `zipWithNext` nu va crea și perechea dintre primul și ultimul punct.

4. Să se creeze un functor pentru o colecție de tip `MutableMap` care să prelucreze valorile. Pentru testare, se va utiliza un `MutableMap` în care cheile sunt valori întregi, iar valorile sunt string-uri ce conțin mai multe cuvinte separate prin spațiu. Apelurile funcției `map` din functor vor prelucra `MutableMap`-ul astfel încât:
  - primul `map` va adăuga prefixul „Test” la fiecare valoare
  - al doilea `map` va apela funcția extensie `toPascalCase`