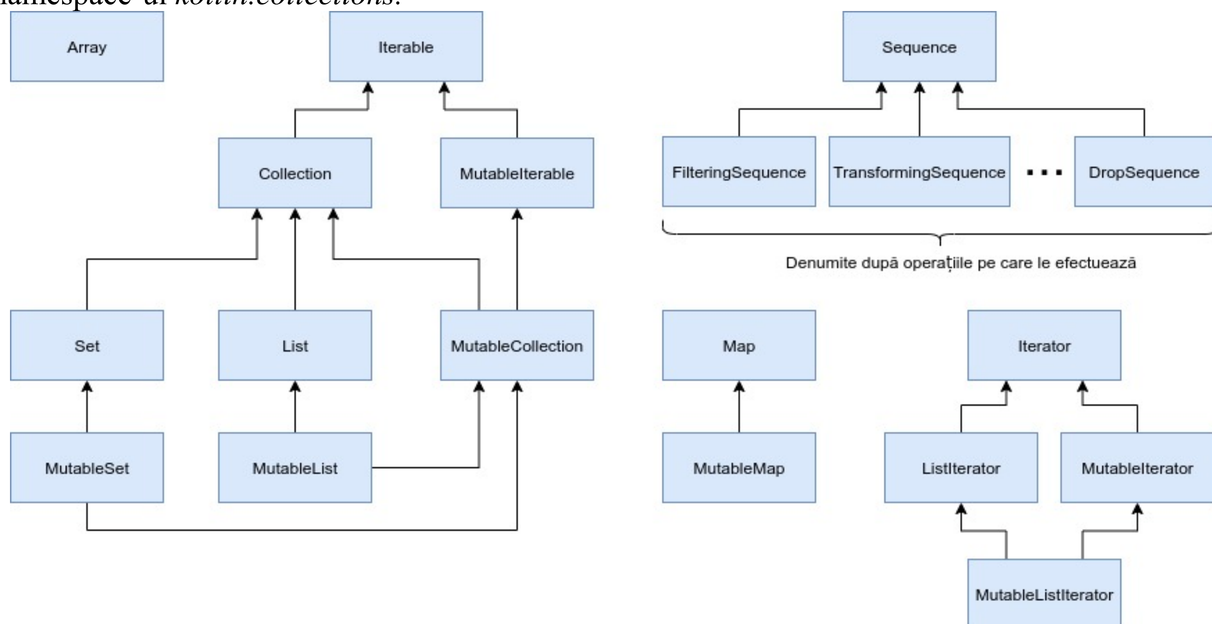
Laboratorul 7: Colecții și generice în Kotlin

Introducere

1. Colecții

Kotlin face distincție între colecțiile **mutabile** și cele **imutabile**. O colecție **mutabilă** poate fi actualizată pe loc prin adăugarea, ștergerea sau înlocuirea unui element. O colecție **imutabilă**, deși oferă aceleași operații (adăugare, ștergere, înlocuire) prin funcțiile operator, va crea o nouă colecție, lăsând-o pe cea veche neschimbată. Toate colecțiile se regăsesc în namespace-ul *kotlin.collections.*



Ierarhia de clase a colecțiilor

Pentru o imagine de ansamblu asupra tuturor colecțiilor din Kotlin, vezi:

- **Cursul 7 de la disciplina *Paradigme de programare***
- Cartea **„Hands-on data structures and algorithms with Kotlin”** (2019) scrisă de Chandra Sekhar și Rivu Chakraborty
- https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/
- https://kotlinlang.org/docs/reference/collections-overview.html

Iterable vs Sequence vs Java Stream

```
interface Iterable<out T> {
    operator fun iterator(): Iterator<T>
}

interface Sequence<out T> {
    operator fun iterator(): Iterator<T>
}
```

Întrucât singura diferență care se observă la prima vedere între ***Iterable*** și ***Sequence*** este denumirea interfeței, în cele ce urmează se vor evidenția diferențele și cazurile de utilizare.

O **secvență** este un tip iterabil[1] pe care se pot efectua operații fără crearea de colecții intermediare inutile, prin executarea tuturor operațiilor aplicabile pe fiecare element înainte de trecerea la următorul. Pentru clarificări, vezi exemplul cu creioane de colorat: https://typealias.com/guides/kotlin-sequences-illustrated-guide/

Secvențele sunt leneșe (lazy), funcțiile intermediare corespunzătoare pentru procesarea secvențelor nu fac calcule, ci returnează o nouă secvență ce o decorează pe cea anterioară cu o nouă operație. Toate calculele sunt evaluate în timpul operației terminale (cum ar fi *toList* sau *count*).

```
fun main(args: Array<String>) {
    val seq = sequenceOf(1,2,3)
    print(seq.filter { it % 2 == 1 })
    // Prints: kotlin.sequences.FilteringSequence@XXXXXXXX
    print(seq.filter { it % 2 == 1 }.toList()) // Prints: [1, 3]
    val list = listOf(1,2,3)
    print(list.filter { it % 2 == 1 }) // Prints: [1, 3]
}
```

Procesarea secvențelor este în general mai rapidă decât procesarea directă a colecțiilor unde există mai mult de un pas de procesare.

ATENȚIE: Există cazuri în care secvențele nu sunt recomandate. Spre exemplu, în cazul sortării prin apelul funcției ***sorted***, întrucât este necesară parcurgerea întregii colecții.

```
generateSequence(0) { it + 1 }.sorted().take(10).toList()
// Infinitive calculation time

generateSequence(0) { it + 1 }.take(10).sorted().toList()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

O soluție posibilă pentru sortare cu secvențe este utilizarea funcției ***sortedBy:***

```
// Took around 150 482 ns
fun productsSortAndProcessingList(): Double {
    return productsList
        .sortedBy { it.price }
        .filter { it.bought }
        .map { it.price }
        .average()
}

// Took around 96 811 ns
fun productsSortAndProcessingSequence(): Double {
    return productsList.asSequence()
        .sortedBy { it.price }
        .filter { it.bought }
        .map { it.price }
        .average()
}
```

În ceea ce privește stream-urile din Java, acestea sunt tot „leneșe”, fiind colectate în pasul final de procesare. De asemenea, stream-urile Java sunt mult mai eficiente pentru procesarea colecțiilor decât funcțiile de procesare corespunzătoare din Kotlin

[1] În cadrul secvenței, „iterabil” nu se referă la *Iterable* ci la abilitatea de a itera peste o secvență (cum ar fi cu o buclă *for*)

```
productsList.asSequence()
     .filter { it.bought }
     .map { it.price }
     .average()

productsList.stream()
     .filter { it.bought }
     .mapToDouble { it.price }
     .average()
     .orElse(0.0)
```

Diferențe între stream-urile Java și secvențele Kotlin:

- secvențele Kotlin au mult mai multe funcții de procesare (fiind definite ca funcții extensie)
- Stream-urile Java pot fi pornite în mod paralel, utilizând o funcție *parallel*

Se recomandă utilizarea stream-urilor Java doar pentru procesări computaționale grele, unde se poate beneficia de modul paralel. Pentru mai multe detalii, vezi: https://blog.kotlin-academy.com/effective-kotlin-use-sequence-for-bigger-collections-with-more-than-one-processing-step-649a15bb4bf

2. Generice

Programarea generică (sau genericele) este o tehnică prin care funcțiile pot fi scrise cu tipuri care nu sunt specificate în momentul scrierii și sunt folosite mai târziu cu diferite tipuri de date.

Pentru mai multe detalii, vezi cartea „Programming Kotlin" (2017) scrisă de Stephen Samuel și Ștefan Bocuțiu.

```
fun <T> random(one: T, two: T, three: T): T = TODO()

fun <K, V> put(key: K, value: V): Unit = TODO()

fun main() {
    val str: String = random("hello", "hola", "bonjour")
    val any: Any = random("a", 1, false)
    put(0, "value0")
}
```

După cum se observă în funcția ***random*** avem un singur parametru tip (T) care este folosit pentru toți cei trei parametri și pentru tipul returnat. Se remarcă faptul că funcțiile generice au un parametru tip introdus prin paranteze unghiulare **<T>.**

În funcția ***put*** au fost incluși mai mulți parametri tip **<K, V>**. Corpul celor două funcții trebuie implementat, apelul funcției TODO() generând o eroare de tipul NotImplemented.

Ca și în Java, clasele din Kotlin pot avea parametri tip:

```
class Box<T>(t: T) {
    var value = t
}
```

Pentru a crea o instanță a unei asemenea clase, trebuie precizat tipul argumentelor:

```
val box: Box<Int> = Box<Int>(1)
```

2.1. Polimorfism mărginit

Funcțiile care sunt generice pentru **orice** tip sunt utile, dar cumva limitate. Adesea, va fi nevoie de scrierea unor funcții care sunt generice pentru **unele** tipuri care au o caracteristică comună. Spre exemplu, definirea unei funcții care returnează minimul dintre două valori, pentru oricare valori ce suportă noțiunea de comparare.

Pentru a forța valorile să aibă acea noțiune de comparare, trebuie restricționate tipurile generice la cele care suportă funcțiile care trebuie invocate. Cu alte cuvinte, mărginim funcția polimorfică (generică), acest lucru fiind numit **polimorfism mărginit**.

2.1.1. Mărginiri superioare

Kotlin suportă un tip de mărginire a polimorfismului, mai precis mărginirea superioară. Din denumire, se remarcă că tipurile generice sunt restricționate la cele care sunt subclase ale mărginirii. Mărginirea se declară împreună cu parametrul tip:

```
fun <T : Comparable<T>>min(first: T, second: T): T {
    val k = first.compareTo(second)
    return if (k <= 0) first else second
}
```

Comparable este un tip din biblioteca standard care definește metoda *compareTo* ce returnează o valoare mai mică decât 0 dacă primul element este mai mic, mai mare decât 0 dacă al doilea element este mai mic, egală cu 0 dacă elementele sunt egale.

```
val a: Int = min(4, 5)
val b: String = min("e", "c")
```

Observație: Dacă nu se specifică o mărginire superioară pentru parametrul tip, compilatorul va folosi tipul *Any* ca mărginire superioară implicită.

2.1.2. Mărginiri multiple

Uneori, este necesară declararea de mărginiri superioare multiple. Spre exemplu, dacă se dorește extinderea funcției *min()* pentru a funcționa pe valori care sunt de asemenea serializabile, se mută declararea mărginirii superioare într-o clauză *where* separată:

```
fun <T>minSerializable(first: T, second: T): T
where T : Comparable<T>,T : Serializable {
    val k = first.compareTo(second)
    return if (k <= 0) first else second
}
```

Observație: Toate mărginirile superioare sunt scrise ca și clauze where și formează o **uniune** de mărginire superioară.

```
class SerializableYear(val value: Int) : Comparable<SerializableYear>,
Serializable {
    override fun compareTo(other: SerializableYear): Int =
this.value.compareTo(other.value)
}
```

```
val b = minSerializable(SerializableYear(1969),
                        SerializableYear(1802))
```

Clasele pot defini de asemenea mărginiri superioare multiple:

```
class MultipleBoundedClass<T>where T : Comparable<T>, T : Serializable
```

2.2. Varianța (Variance)

Una dintre cele mai complicate părți ale sistemului de tipuri Java sunt tipurile **wildcard**. Pentru mai multe detalii, se recomandă parcurgerea documentației oficiale: https://kotlinlang.org/docs/reference/generics.html, precum și următoarele resurse:

- https://typealias.com/guides/ins-and-outs-of-generic-variance/
- https://typealias.com/guides/star-projections-and-how-they-work/

Kotlin nu are asemenea tipuri, dar oferă:

- varianța la momentul declarării (declaration-site variance)
- proiecțiile de tip
- proiecțiile stea

PENTRU A ÎNȚELEGE MAI BINE COVARIANȚA ȘI CONTRAVARIANȚA, SE RECOMANDĂ PARCURGEREA EXEMPLULUI CU LENEȘI:
https://medium.com/kotlin-thursdays/introduction-to-kotlin-generics-9d18d3719e1d

2.2.1. Varianța la momentul declarării

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs
    // asignarea anterioară este ok, T fiind un parametru de ieșire
    // ...
}
```

Regulă: Când un parametru tip T este declarat ca **out** al unei clase, atunci acest tip T poate fi folosit doar ca tip de return în membrii clasei respective.

Modificatorul **out** se numeste ***adnotare de varianță*** (variance annotation) și fiind vorba despre declararea parametrului tip, se numesște *varianța la momentul declarării.* Despre interfața Source se spune că e **covariantă** în parametrul tip T.

Pe lângă adnotarea de varianță (**out**), Kotlin oferă și o **adnotare complementară de varianță**, **in**. Această adnotare face un parametru tip să fie **contravariant**: acel tip poate fi doar consumat (parametru de intrare), nu și produs (tip returnat).

Un exemplu de contravarianță este interfața *Comparable:*

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 (Double) este un subtip al Number
    // Deci se poate asigna x unei variabile de tip Comparable<Double>
    val y: Comparable<Double> = x // OK!
}
```

2.2.2. Proiecțiile de tip: varianța la momentul utilizării (use-site variance)

Unele clase nu pot fi constrânse sa returneze un singur parametru tip T. Spre exemplu, clasa *Array*:

```
class Array<T>(val size: Int) {
    fun get(index: Int): T { ... }
    fun set(index: Int, value: T) { ... }
}
```

Observație: clasa Array nu poate fi nici *covariantă* nici *contravariantă.*

Exemplu de proiecție cu **out:**

```
fun copy(from: Array<out Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

În funcția *copy* de mai sus, tipurile elementelor din Array-ul *from* sunt declarate cu **out** pentru a interzice modificarea acestora. Acesta este un exemplu de varianță la momentul utilizării. Cu alte cuvinte array-ul *from* este un array restricționat.

Exemplul de proiecție cu **in:**

```
fun fill(dest: Array<in String>, value: String) { ... }
```

2.2.3. Proiecțiile stea

Uneori este necesară utilizarea unui argument tip necunoscut. Pentru a rezolva această problemă printr-o <u>*modalitate sigură*</u>, Kotlin a introdus așa numitele *proiecții stea*. Trebuie definită o proiecție a unui tip generic pentru care fiecare instanțiere concretă a acelui tip generic să fie un subtip al proiecției.

Sintaxa Kotlin pentru proiecțiile stea:

- **Foo<out T: TUpper>** unde T este un parametru tip **covariant** cu mărginirea superioară (upper bound) TUpper, Foo<*> este echivalent cu Foo<out TUpper>. Cu alte cuvinte, când T este necunoscut, se poate citi în mod *sigur* valori TUpper din Foo<*>

- **Foo<in T>**, unde T este un parametru tip **contravariant,** Foo<*> este echivalent cu Foo<in Nothing>. Cu alte cuvinte, nu se poate scrie nimic în Foo<*> într-un mod sigur, când T este necunoscut
- **Foo<T: TUpper>** unde T este un parametru tip **invariant** cu mărginirea superioară TUpper, Foo<*> este echivalent cu Foo<out TUpper> pentru citirea valorilor și echivalent cu Foo<in Nothing> pentru scrierea valorilor

Dacă un tip generic are mai mulți parametri tip, fiecare poate fi proiectat independent. Spre exemplu, pentru:

```
interface Function<in T, out U>
```

se pot defini următoarele proiecții stea:

- **Function<*, String>** echivalent cu **Function<in Nothing, String>**
- **Function<Int, *>** echivalent cu **Function<Int, out Any?>**
- **Function<*, *>** echivalent cu **Function<in Nothing, out Any?>**

Exemple

1. Exemple cu colecții

1.1. Crearea colecțiilor

```
/**
 * Arrays
 */
val intArray: Array<Int> = arrayOf(1, 2, 3)
val primitiveIntArray: IntArray = intArrayOf(1, 2, 3)
val copyOfArray: Array<Int> = intArray.copyOf()
val partialCopyOfArray: Array<Int> = intArray.copyOfRange(0, 2)

/**
 * Lists
 */
val intList: List<Int> = listOf(1, 2, 3) //Or arrayListOf(1,2,3)
val emptyList: List<Int> = emptyList() // Or listOf()
val listWithNonNullElements: List<Int> = listOfNotNull(1, null, 3)
// --> List(1,3)

/**
 * Sets
 */
val aSet: Set<Int> = setOf(1) // Or hashSetOf(1) / linkedSerOf(1)
val emptySet: Set<Int> = emptySet()
// Or setOf() / hashSetOf() / linkedSetOf()

/**
 * Maps
 */
val aMap: Map<String, Int> = mapOf("hi" to 1, "hello" to 2)
// Or mapOf(Pair("hi", 1) / hashMapOf("hi" to 1) / linkedMapOf("hi" to 1)
val emptyMap: Map<String, Int> = emptyMap()
// Or mapOf()/hashMapOf()/linkedMapOf()
val defaultingMap: Map<String, Int> = aMap.withDefault { key -> if
(key == "no") 1 else 999 }


/**
 * Black sheep, mutables
 */
val mutableList: MutableList<Int> = mutableListOf(1, 2, 3)
val mutableSet: MutableSet<Int> = mutableSetOf(1)
var mutableMap: MutableMap<String, Int> = mutableMapOf("hi" to 1,
"hello" to 2)
```

1.2 Operații pe colecții

```
/**
 * Operates on all iterables
 */
fun operate(): Unit {
    assert(listOf(1, 2, 3, 1) == intList + 1) // [1, 2, 3, 4]
```

```
    assert(listOf(2, 3) == intList - 1) // [2, 3]

    assert(listOf(1, 2, 3, 1, 2, 3) == intList + listOf(1, 2, 3)) //
[1, 2, 3, 1, 2, 3]
    assert(listOf(3) == intList - listOf(1, 2)) // [3]

    assert(mapOf("hi" to 1, "hello" to 2, "Goodbye" to 3) == aMap +
Pair("Goodbye", 3)) // {hi=1, hello=2, Goodbye=3}
    assert(mapOf("hi" to 1) == aMap - "hello") // Takes in a key and
removes if found

    assert(mapOf("hi" to 1, "hello" to 2, "Goodbye" to 3) == aMap +
mapOf(Pair("hello", 2), Pair("Goodbye", 3))) // {hi=1, hello=2,
Goodbye=3}
    assert(emptyMap<String, Int>() == aMap - listOf("hello", "hi")) //
Takes in an iterable of keys and removes if found

    /**
     * For Black sheep - mutants
     */
    mutableList -= 2
    println(mutableList) // [1, 3]
    mutableList += 2
    println(mutableList) // [1, 3, 2]

    mutableMap.minusAssign("hello") // {hi=1} same as -=
    println(mutableMap) //
    mutableMap.plusAssign("Goodbye" to 3) // {hi=1, Goodbye=3} same as
+=
    println(mutableMap)
}
```

2. Exemple cu generice

2.1. Tipuri parametrizate

```
class Sequence<T>

class Dictionary<K, V>

fun example() {
    val seq = Sequence<Boolean>()
}
```

2.2 Proiecția tipurilor

```
fun isSafe(crate: Crate<out Fruit>): Boolean = crate.elements.all {
it.isSafeToEat() }

fun usingTypeProjection() {
    val oranges = Crate(mutableListOf(Orange(), Orange()))
    isSafe(oranges)
}
```

```
interface Listener2<T> {
    fun onNext(t: T): Unit
}

class EventStream2<T>(val listener: Listener2<in T>) {
    fun start(): Unit = TODO()
    fun stop(): Unit = TODO()
}
```

2.3 Tipuri de date algebrice

```
sealed class LinkedList<out T> {

    fun isEmpty() = when (this) {
        is Empty -> true
        is Node -> false
    }

    fun size(): Int = when (this) {
        is Empty -> 0
        is Node -> 1 + this.next.size()
    }

    fun tail(): LinkedList<T> = when (this) {
        is Node -> this.next
        is Empty -> this
    }

    fun head(): T = when (this) {
        is Node<T> -> this.value
        is Empty -> throw RuntimeException("Empty list")
    }

    fun drop(k: Int): LinkedList<T> = when (this) {
        is Empty -> Empty
        is Node -> if (k <= 0) this else this.next.drop(k - 1)
    }

    operator fun get(pos: Int): T {
        require(pos >= 0, { "Position must be greater than or equal to
zero" })
        return when (this) {
            is Node<T> -> if (pos == 0) head() else this.next.get(pos
- 1)
            is Empty -> throw IndexOutOfBoundsException()
        }
    }

    fun append(t: @UnsafeVariance T): LinkedList<T> = when (this) {
        is Node<T> -> Node(this.value, this.next.append(t))
        is Empty -> Node(t, Empty)
    }

    companion object {
        operator fun <T> invoke(vararg values: T): LinkedList<T> {
            var temp: LinkedList<T> = Empty
```

```
            for (value in values) {
                temp = temp.append(value)
            }
            return temp
        }
    }
}

private class Node<out T>(val value: T, val next: LinkedList<T>) :
LinkedList<T>()
private object Empty : LinkedList<Nothing>()

fun <T> LinkedList<T>.append(t: T): LinkedList<T> = when (this) {
    is Node<T> -> Node(this.value, this.next.append(t))
    is Empty -> Node(t, Empty)
}

fun main(args: Array<String>) {
    val list =
LinkedList("this").append("is").append("my").append("list")
    println(list.size()) // prints 4
    println(list.head()) // prints "this"
    println(list[1]) // prints "is"
    println(list.drop(2).head()) // prints "my"
}
```

Aplicații și teme

Pentru o imagine de ansamblu asupra colecțiilor, vezi cheat sheet-ul: https://jussi.hallila.com/Kollections/

Aplicații de laborator:

- Să se proceseze toate intrările din /var/log/syslog din ultimele 30 de minute, aplicând următoarele operațiuni:
  - să se citească înregistrările împărțind textul linie cu linie şi stocând elementele într-o secvență de obiecte (se va crea o clasă *SyslogRecord* conform formatului general de mai jos)
  - parcurgând secvența, să se identifice aplicația care a cauzat intrarea respectivă şi să se grupeze în map-uri toate intrările în funcție de aplicațiile găsite (cheia fiind aplicația, iar valoarea fiind un obiect SyslogRecord)
  - pentru fiecare aplicație în parte, să se sorteze crescător descrierile intrărilor din log corespunzătoare acesteia (LOG_ENTRY)
  - utilizând funcțiile de filtrare ale colecțiilor, să se afişeze doar înregistrările care specifică un PID (Process ID) pentru aplicația țintă

  La modul general, formatul unei înregistrări în syslog este:

```
<TIMESTAMP> <HOSTNAME> <APPLICATION_NAME>[PID]: LOG_ENTRY
#(unde PID este opțional)
```

Observație: pentru a deschide fișierul respectiv, sunt necesare drepturi de administrator. În directorul proiectului, se vor executa următoarele comenzi:

```
sudo cp /var/log/syslog .
sudo chown <username>:<username> syslog
chmod a+rw syslog
```

Tema pe acasă:

- Să se proceseze ultimele 50 de intrări din fișierul /var/log/apt/history.log. Fișierul *history.log* are următoarea structură generală:

```
Start-Date: <start_date>
Commandline: <command>
Requested-By: <user> (<user_id>)
<action>: <description>
End-Date: <end_date>
```

- Se vor extrage metadatele *start-date* și *CommandLine* și se va transforma start-date într-un timestamp;

- Timestamp-ul și comanda vor fi stocate într-o clasă numită HistoryLogRecord ce implementează interfața Comparable;

- **[OPȚIONAL]:** Se pot stoca și celelalte metadate utilizând colecțiile după necesitate;

- Se va crea un MutableHashMap cu elemente de forma: <timestamp, HistoryLogRecord> (deci cheile vor fi timestamp-uri, iar valorile vor fi obiecte HistoryLogRecord);

- După cum se intuiește, va trebui suprascrisă metoda ***compareTo*** care să utilizeze timestamp-urile pentru a compara obiectele.

- Se va crea o funcție polimorfică (generică) **mărginită superior** (interfața Comparable) care va calcula maximul dintre două obiecte HistoryLogRecord ca fiind obiectul cu cel mai recent timestamp.

- Utilizând o proiecție de tip **out** (covarianța), să se implementeze o funcție polimorfică (generică) de căutare și înlocuire, care să primească trei parametri: obiectul căutat (un HistoryLogRecord), obiectul cu care să fie înlocuit (alt HistoryLogRecord), și obiectul în care se realizează căutarea și înlocuirea (MutableHashMap-ul). Observație: se poate folosi și o proiecție stea.