Paradigma Programarii Generice

Cursul nr. 7

Mihai Zaharia

Clarificări cu privire la proba practică și te(R)oRetică

1. Cum se utilizează materialele de curs
- se ia fiecare slide și se înțelege (cel mai probabil necesită să citiți în plus deoarece fiecare are un set de cunoștiințe diferite.
- se citeste documentatia UML de uml.org (iar pentru fiecare program se face si diagrama de obiecte!)
- se testează fiecare exemplu din curs - pentru fiecare se face un proiect separat
- pentru acest an nu se va cere la laborator graal vm si sdl

2. Materialele de laborator se tratează într-o manieră similară

3. La proba practică problemele de pe bilet vor fi una de kotlin și una de pyton
4. Cum vor fi create aceste bilete?
- direct probleme din curs (în cazul unor aspecte mai complicate)
- combinații din exemplele parțiale prezentate la curs (e.g. tkinter)

variații la problemele de laborator și temele pe acasă
5. Cum va fi testul teoretic ?
- întrebări închise: (ca la poli-țieni) se va alege dintr-un număr de variante de răspuns și acesta va fi unic
- întrebări deschise: va trebui ca studentul să răspundă liber dar sa nu bată câmpii (ca la interviu firmă)

6. Cum vor fi întrebările?

Aceste vor urmări următoarele aspecte:
- teoretic (e.g. definiți principiul O sau cand se foloește fațada și când se folosește burlacul?)
- de limbaj - se dă o bucată de cod și fie se cere rezultatul execuției fie are variante de răspuns
- fie sunt nuanțe de utilizare concreta a unor instructiuni in anumite contexte concrete (iar stil firmă)

Funcții parametrizate

fun random(one: Any, two: Any, three: Any): Any //ex1

fun <T> random(one: T, two: T, three: T): T // ex2

val randomGreeting: String = random("hei", "hy", "comment ca va")//ex3

fun <K, V>put(key: K, value: V): Unit //ex4

Tipuri parametrizate

class Sequence<T> //ex1

- si un exemplu de utilizare

val seq = Sequence<Boolean>()

- class Dictionary<K, V>//ex2
- și un exemplu de utilizare

val dict = Dictionary<String, String>()

Polimorfism limitat superior

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

- Deoarece Comparable este o bibliotecă standard care defineste operația de compareTo rezultă că valorile lui T pot fi extinse din următoarele tipuri:

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

- deci limitarea este la nivel de submulțimea {Int, String}

Limitări multiple

```
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
}
```

Limitări multiple

class Year(valvalue: Year) : Comparable<Year>
{ override fun compareTo(other: Year): Int = this.value.compareTo(other.value) }

- linia următoare va da eroare la compilare

val a = minSerializable(Year(1969), Year(2001))

- Dar dacă extindem tipul pentru ca să suporte și serializare atunci va merge

class SerializableYear(valvalue: Int) : Comparable<SerializableYear>, Serializable
{ override fun compareTo(other: SerializableYear): Int =
this.value.compareTo(other.value) }
val b = minSerializable(SerializableYear(1969), SerializableYear(1802))

- Și clasele pot defini limitări superioare multiple de tip

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

Tipuri generice de date & modificatori

```
// ex1 - Java
interface Source<T>
{

  T nextT();

}
// ex2 - Java

void demo(Source<String> strs)
{

  Source<Object> objects = strs; // !!! NEPERIMIS in Java

  // ...

}
```

Tipuri generice de date

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

fun demo(strs: Source<String>)
{
val objects: Source<Any> = strs
// acum acest lucru este permis deoarece T este clar un parametru produs/ de iesire adica out
                // ...
}
```

Tipuri generice de date

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

fun demo(x: Comparable<Number>)
{
 x.compareTo(1.0) // 1.0 are tipul Double, care este un subtip al lui
Number

// ECI se poate asigna x unei variabile de tipul Comparable<Double>

val y: Comparable<Double> = x // OK!
}
```

Tipuri generice de date

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

SE observvă că nu poate fi nici co- nici contra-varianță în T și asta impune o serie de limitari. De exemplu fie funcția:

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

Tipuri generice de date

Aceasta ar trebui să copie elemente dintr-un tablou într-altul. Să vedem cum ar arata utilizarea ei:

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3) { "" }
copy(ints, any)
    //   ^ tipul furnizat pentru unul din parametri este Array<Int>
dar tipul Array<Any> era cel asteptat - vezi definiția


fun copy(from: Array<out Any>, to: Array<Any>) { ... }
```

Tipuri generice de date

Pentru **Foo<out T : TUpper>**, unde T este un parametru covariant mărginit superior de TUpper atunci Foo<*> este echivalentul lui **Foo<out TUpper>**. Și înseamnă că atunci când T este necunoscut se pot citi în siguranță valori ale lui TUpper din Foo<*>.

Pentru **Foo<in T>**, unde T este un parametru covariant de tip Foo<*> este echivalent cu **Foo<in Nothing>**. Și înseamnă că nu este nimic care poate fi scris într-o manieră sigură în/către Foo<*> atunci când T este necunoscut.

Pentru **Foo<T : TUpper>**, unde T este un parametru invariant de tip cu limitare superioară la TUpper atunci Foo<*> este echivalent cu **Foo<out TUpper>** pentru cazul citirii unor valori și cu **<in Nothing>** pentru cazul scrierii unor valori.

Function<*, String> echivalentă cu Function<in Nothing, String>;
Function<Int, *> echivalentă cu Function<Int, out Any?>;
Function<*, *> echivalentă cu Function<in Nothing, out Any?>.

Funcții generice

```
fun <T> singletonList(item: T): List<T> //ex1
{
    // ...
}
```

sau

```
fun <T> T.basicToString(): String // o funcție extensie
{
    // ...
}
val l = singletonList<Int>(1) //ex2
val l = singletonList(1) //ex3
```

Constrângeri Generice

fun <T : Comparable<T>> sort(list: List<T>) { ... }//ex1

sort(listOf(1, 2, 3)) // 0K. Int este un subtip al lui Comparable<Int>

sort(listOf(HashMap<Int, String>()))

// Error: HashMap<Int, String> NU este un subtip al lui Comparable<HashMap<Int, String>>

Constrângeri Generice

```
fun <T> copyWhenGreater(list: List<T>, threshold: T): List<String>

    where T : CharSequence,

          T : Comparable<T> {

    return list.filter { it > threshold }.map { it.toString() }

}
```

Ștergerea tipului (Type erasure)

- pentru intantelel lui Foo<Bar> si Foo<Baz?> în urma stergerii tipului rezultă aceași descriere și anume Foo<*> //ex1

Tipuri de date algebrice

sealed class List<out T>

- Apoi vom defini două implementări: una va conține un nod cu o valoare iar a doua un nod gol

class Node<T>(val value: T, val next: List<T>) : List<T>() object Empty : List<Nothing>()

- O listă vidă va conține numai un obiect gol (FĂRĂ NULLLLLLL)
- În acest caz vom fixa tipul acestui nod utilizând tipul Nothing

sealed class List<out T>

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

Exemplu listă algebrică

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

- Multe funcții sunt similare cu ceea ce știți de la ADT de exemplu funcția cap arată ca mai jos:

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

Exemplu listă algebrică

- O soluție ar fi să permitem ca T să fie parametru de intrare, chiar dacă anterior i-am spus compilatorului să nu-l admită, prin suprascrierea verificării de variație pentru această funcție:

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

Cealaltă posibilitate este să declarăm append ca o funcție extensie unde parametrul tip este invariant:

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

Exemplu listă algebrică

```
sealed class List<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(): List<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")}
operator fun get(pos: Int): T { require(pos>= 0, { "Position must be >=0" })
return when (this) {is Node<T> -> if (pos == 0) head() else
this.next.get(pos - 1) is Empty -> throw IndexOutOfBoundsException()}  }
fun append(t: @UnsafeVarianceT): List<T> = when (this) { is Node<T> ->
Node(this.value, this.next.append(t)) is Empty -> Node(t, Empty) }
```

Exemplu listă algebrică

```
companion object
{ operator fun <T>invoke(vararg values: T): List<T>
    { var temp: List<T> = Empty for (value in values)
                              {temp = temp.append(value)}
     return temp }
}
}
private class Node<out T>(val value: T, val next: List<T>) : List<T>()
private object Empty : List<Nothing>()
```

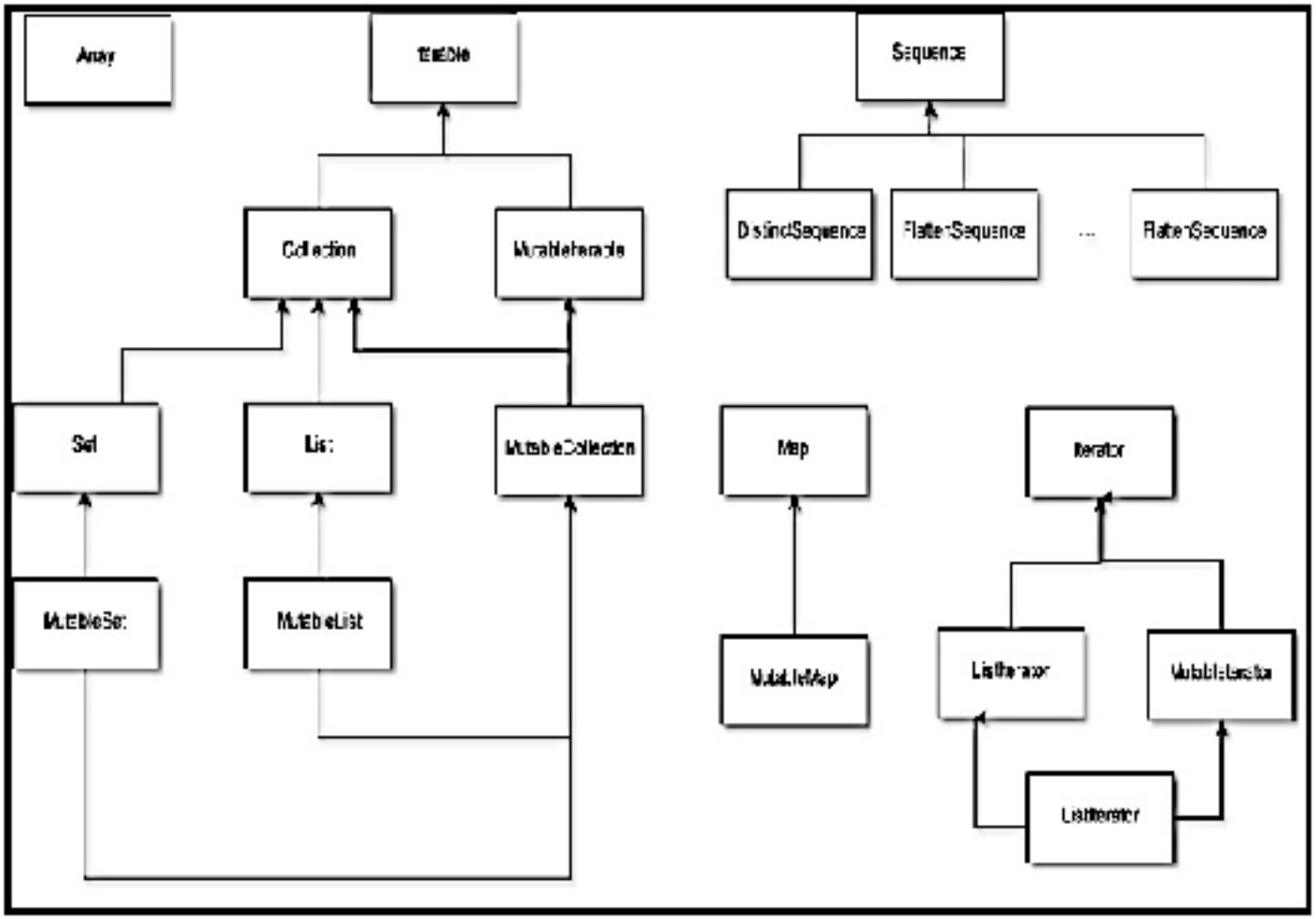
- si utilizarea :

```
val list = List("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"
```

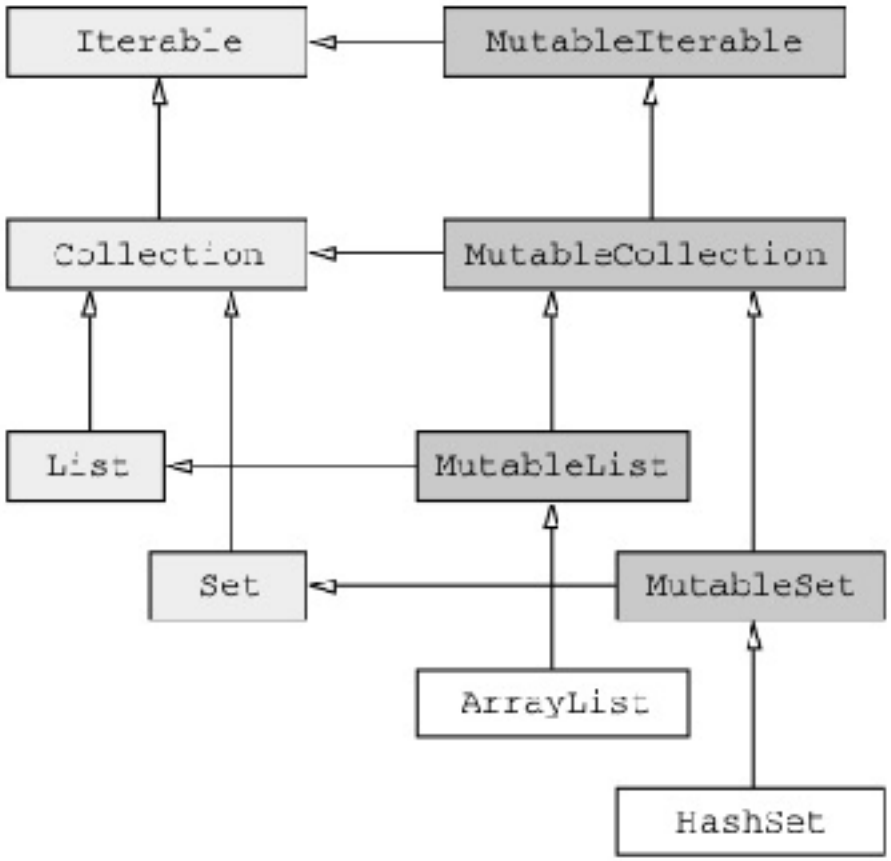
Colecții

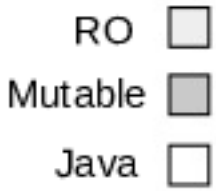
kotlin.collections namespaces

Colecții


Array
Iterable
Sequence
Collection
MutableIterable
DistinctSequence
FlattenSequence
...
FlattenSequence
Set
List
MutableCollection
Map
Iterator
MutableSet
MutableList
MutableMap
ListIterator
MutableListIterator
ListIterator

Colecții


Iterable
MutableIterable
Collection
MutableCollection
List
MutableList
Set
MutableSet
ArrayList
HashSet


RO
Mutable
Java

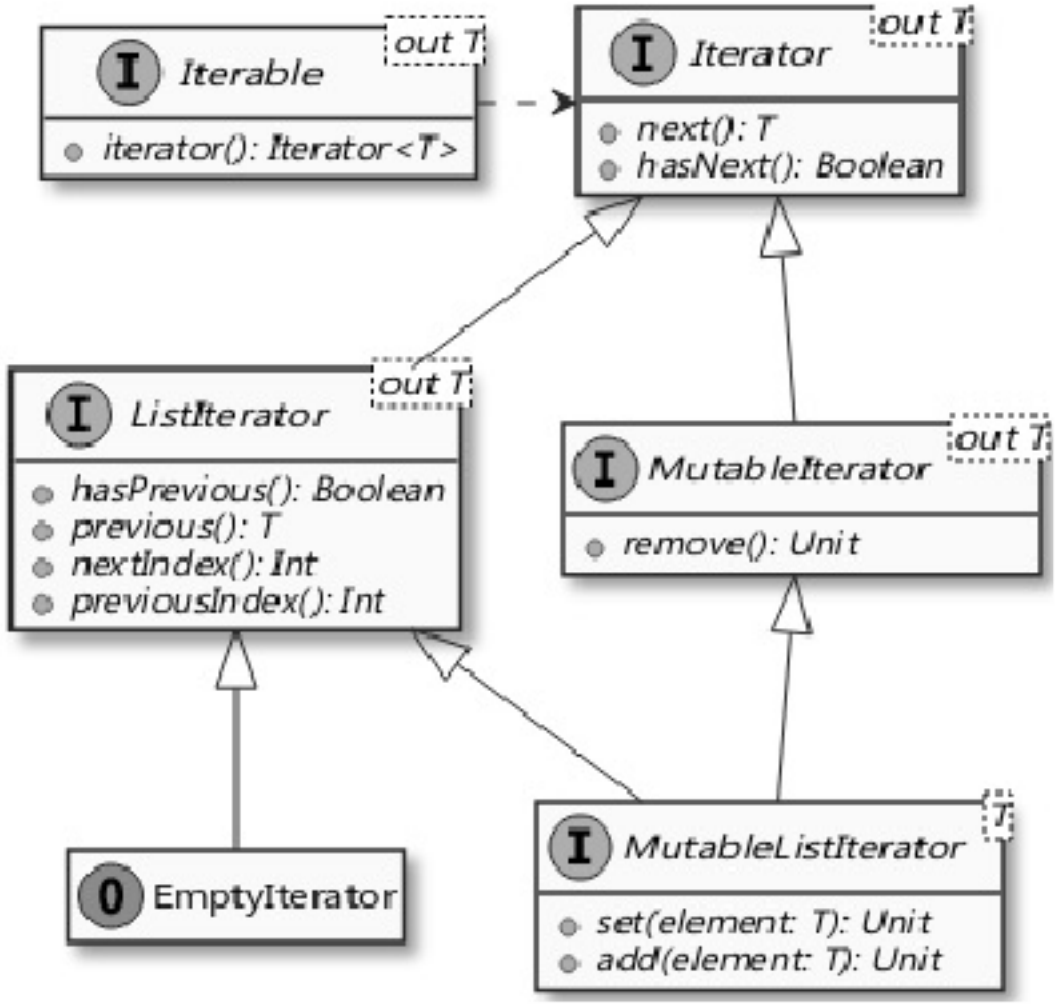
Colecții

```
public interface Iterable<out T> //ex1
{ public abstract operator fun iterator():    Iterator<T>}

public interface Collection<out E> : Iterable<E>//ex2
{
public val size: Int
public fun isEmpty(): Boolean
public operator fun contains(element: @UnsafeVariance E): Boolean
override fun iterator(): Iterator<E>
public fun containsAll(elements: Collection<@UnsafeVariance E>):
Boolean
}
```

Colecții Kotlin

List


Iterable
out T
iterator(): Iterator<T>
Iterator
out T
next(): T
hasNext(): Boolean
ListIterator
out T
hasPrevious(): Boolean
previous(): T
nextIndex(): Int
previousIndex(): Int
MutableIterator
out T
remove(): Unit
EmptyIterator
MutableListIterator
T
set(element: T): Unit
add(element: T): Unit

Colecții - List

```
public interface MutableIterable<out T> : Iterable<T> //ex1
{ override fun iterator(): MutableIterator<T> }
public interface List<out E> : Collection<E>
{
//Operatii inspecție date
override val size: Int
override fun isEmpty(): Boolean
override fun contains(element: override fun iterator(): Iterator<E>
override fun containsAll(elements: Collection<@UnsafeVariance E>): Boolean
public operator fun get(index: Int): E
public fun indexOf(element: @UnsafeVariance E): Int
public fun lastIndexOf(element: @UnsafeVariance E): Int
//Iteratori Lista
public fun listIterator(): ListIterator<E>
public fun listIterator(index:        Int): ListIterator<E>
//Vizualizare date
public fun subList(fromIndex: Int, toIndex: Int): List<E>
}
```

Liste

```
val intList: List<Int> = listOfprintln("Lista Intregi[
        ${intList.javaClass.canonicalName} ]:
                ${intList.joinToString(", ")} ")
val emptyList: List<String> = emptyList<String>()
println("Lista Vida[${emptyList.javaClass.canonicalName}]:${emptyList.joinToString(",")}")
val nonNulls: List<String> = listOfNotNull<String>(null, "a", "b","c")
println("Lista siruri nevida[${nonNulls.javaClass.canonicalName}]:${nonNulls.joinToString (",")}")
val doubleList: ArrayList<Double> = arrayListOf(84.88, 100.25, 999.99)
println("Lista Dubla:${doubleList.joinToString(",")}")
val cartoonsList: MutableList<String> = mutableListOf("Tom&Jerry",
"Dexter's Laboratory", "Johnny Bravo", "Cow&Chicken")
println("Lista DA[${cartoonsList.javaClass.canonicalName}]:
        ${cartoonsList.joinToString(",")} ")
cartoonsList.addAll(arrayOf("Ed, Edd n Eddy","Ren&Stimpy"))
println("Lista DA[${cartoonsList.javaClass.canonicalName}]:
        ${cartoonsList.joinToString(",")} ")
```

Colecții - MutableList

```
public interface MutableList<E> : List<E>, MutableCollection<E>
{ //Operații de modificare
override fun add(element: E): Boolean
override fun remove(element: E): Boolean
//Operații de modificare în masă
override fun addAll(elements: Collection<E>): Boolean
public fun addAll(index: Int, elements: Collection<E>): Boolean
override fun removeAll(elements: Collection<E>): Boolean
override fun retainAll(elements: Collection<E>): Boolean
override fun clear(): Unit
//Operatori pentru acces pozițional
public operator fun set(index: Int, element: E): E
public fun add(index: Int, element: E): Unit
public fun removeAt(index: Int): E //List Iterators
override fun listIterator(): MutableListIterator<E>
override fun listIterator(index: Int): MutableListIterator<E>
//Vizualizare
override fun subList(fromIndex: Int, toIndex: Int): MutableList<E>
}
```

Colecții - Set

```
public interface Set<out E> : Collection<E>  //ex2
{ //Operatii inspecție date
override val size: Int
override fun isEmpty(): Boolean
override fun contains(element: @UnsafeVariance E): Boolean override fun iterator(): Iterator<E>
  //Operatii de masă
override fun containsAll(elements: Collection<@UnsafeVariance E>): Boolean }
public interface MutableCollection<E> : Collection<E>, MutableIterable<E> //ex2
{  //Operatii inspecție date
override fun iterator(): MutableIterator<E>
   //Operații de modificare
public fun add(element: E): Boolean
public fun remove(element: E): Boolean
   //Operatii de masă
public fun addAll(elements: Collection<E>): Boolean
public fun removeAll(elements: Collection<E>): Boolean
public fun retainAll(elements: Collection<E>): Boolean
public fun clear(): Unit }
```

Colecții - MutableSet

```
public interface MutableSet<E> : Set<E>, MutableCollection<E>
{
//Operatii inspecție date
override fun iterator(): MutableIterator<E>
 //Operații de modificare
override fun add(element: E): Boolean
override fun remove(element: E): Boolean
//Operații de modificare în masă
override fun addAll(elements: Collection<E>): Boolean
override fun removeAll(elements: Collection<E>): Boolean
override fun retainAll(elements: Collection<E>): Boolean
override fun clear(): Unit
}
```

Set

```
fun main(args: Array<String>) {
val nums = setOf(11, 5, 3, 8, 1, 9, 6, 2)
val sortAsc = nums.sorted()
    println(sortAsc)
val sortDesc = nums.sortedDescending()
    println(sortDesc)
val revNums = nums.reversed()
    println(revNums)
val cars = setOf(Car("Mazda", 6300), Car("Toyota", 12400),
            Car("Skoda", 5670), Car("Mercedes", 18600))
val res = cars.sortedBy { car -> car.name }
    res.forEach { e -> println(e) }
    println("**********")
val res2 = cars.sortedByDescending { car -> car.name }
    res2.forEach { e -> println(e) }
}
```

LinkedHashSet

```
val set = HashSet<String>()
    set.add("a")
    set.add("b")
    set.add("c")
 val array = arrayOfNulls<String>(set.size)
 set.toArray(array)
 println("Array: ${Arrays.toString(array)}"
```

```
Set<Angajati> angajatiSet = new HashSet<>();
List<Angajati> angajatiList = new ArrayList<>();

long iterations = 1000;
Angajati angajati = new Angajati(100L, "Harry");

for (long i = 0; i < iterations; i++)
{
      angajatiSet.add(new Angajati(i, "John"));
      angajatiList.add(new Angajati(i, "John"));
  }
  angajatiList.add(angajati);
  angajatiSet.add(angajati);
```

HashSet

```
var hashSetOf1: HashSet<Int> = hashSetOf<Int>(2,6,13,4,29,15)
val mySet = setOf(6,4,29)

println("......parcurgere hashSetOf1.....")
    for (element in hashSetOf1)
        { println(element)}
println(".....dimensiunea lui hashSetOf1.....")
    println(hashSetOf1.size)
println(".....hashSetOf1 contine valoarea 13?.....")
    println(hashSetOf1.contains(13))
println("....hashSetOf1 contine toate valorile din mySet?...")
    println(hashSetOf1.containsAll(mySet))
}
```

TreeSet

```
SortedSet<String> fruits = new TreeSet<>(Comparator.reverseOrder());

    fruits.add("Banana");
    fruits.add("Apple");
    fruits.add("Pineapple");
    fruits.add("Orange");

    System.out.println("Fruits Set : " + fruits);
```

Colecții - Map

```
public interface Map<K, out V>
//Operatii inspecție date
public val size:Int
public fun isEmpty(): Boolean
public fun containsKey(key: K): Boolean
public fun containsValue(value: @UnsafeVariance V): Boolean
public operator fun get(key: K): V?
public fun getOrDefault(key: K, defaultValue: @UnsafeVariance V): V
  { //în implementarea implicită din JDK
    return null as V }
//Vizualizare
public val keys: Set<K>
public val values: Collection<V>
public val entries: Set<Map.Entry<K, V>>
public interface Entry<out K, out V>
            { public val key: K public val value: V}
}
```

Parcurgere hartă

cu forloop

```
val items = HashMap<String, Int>()
        items["A"] = 10
        items["B"] = 20

        for ((k, v) in items)
                { println("$k = $v")        }
```

cu forEach

```
        items.forEach
                { k, v ->  println("$k = $v") }
```

Colecții - MutableMap

```
public interface MutableMap<K, V> : Map<K, V>
{
 //Operații de modificare
public fun put(key: K, value: V): V?
public fun remove(key: K): V?
//Operații de modificare în masă
public fun putAll(from: Map<out K, V>): Unit
public fun clear(): Unit
//Vizualizare
override val keys: MutableSet<K>
override val values: MutableCollection<V>
override val entries: MutableSet<MutableMap.MutableEntry<K, V>>
public interface MutableEntry<K,V>: Map.Entry<K, V>
        { public fun setValue(newValue: V): V}
}
```

Map - Hărți

```
data class Customer(val firstName: String, val lastName: String, val id: Int)
val carsMap: Map<String, String> = mapOf("a" to "aston martin", "b" to "bmw", "m" to "mercedes",
                                                                                "f" to "ferrari")
println("cars[${carsMap.javaClass.canonicalName}:$carsMap]")
println("car maker starting with 'f':${carsMap.get("f")}") //Ferrari
println("car maker starting with 'X':${carsMap.get("X")}") //null
val states: MutableMap<String, String>= mutableMapOf("AL" to "Alabama", "AK" to "Alaska",
                                                                              "AZ" to "Arizona")
states += ("CA" to "California")
println("States [${states.javaClass.canonicalName}:$states")
println("States keys:${states.keys}")//AL, AK, AZ,CA
println("States values:${states.values}")//Alabama, Alaska, Arizona, California

val customers: java.util.HashMap<Int, Customer> = hashMapOf(1 to Customer("Dina", "Kreps", 1),
                                                                    2 to Customer("Andy", "Smith", 2))
val linkedHashMap: java.util.LinkedHashMap<String, String> =
linkedMapOf("red" to "#FF0000","azure" to "#F0FFFF","white" to "#FFFFFF")

val sortedMap: java.util.SortedMap<Int, String> = sortedMapOf(4 to "d", 1 to "a", 3 to "c", 2 to "b")
println("Sorted map[${sortedMap.javaClass.canonicalName}]:${sortedMap}")
```

Hash Map - immutable

```
val builder = StringBuilder()
val colors = mapOf("GOLD" to "#FFD700","YELLOW" to "#FFFF00",
                   "ALICEBLUE" to "#F0F8FF", "BISQUE" to "#FFE4C4")
//initializare
builder.append("Parcurgere\n")
colors.forEach{key,value -> builder.append("\n$key:$value")

val keys:List<String> = colors.keys.toList() //cheile trimise in lista

val values:List<String> = colors.values.toList() // valorile trimise in lista

builder.append("\n\nHashMap keys list\n") //parcurgere chei
keys.forEach{builder.append("$it,") }

builder.append("\n\nHashMap values list\n") //parcurgere valori
values.forEach{ builder.append("$it,") }
```

HasMap mutable

```
val builder = StringBuilder()
val colors = mutableMapOf<String,String>()//initializare
mutable

colors.put("INDIANRED","#CD5C5C")//adaug elem
colors.put("CRIMSON","#DC143C")
        colors.put("SALMON","#FA8072")
        colors.put("LIGHTCORAL","#F08080")

builder.append("Parcurgere")
colors.forEach{key,value ->builder.append("\n$key,$value")}

colors.remove("CRIMSON") //stergere
        builder.append("\n\n Dupa stergerea unui element")
 for ((key,value) in colors) //reafisez
          {builder.append("\n$key:$value")}

colors.put("SALMON","NEW VALUE")
//adaugare/modificare

builder.append("\n\nEste HashMap goala? :
${colors.isEmpty()}")

val value = colors.get("LIGHTCORAL") //citire
cheie
builder.append("\n\nLIGHTCORAL value $value")

val reds = mutableMapOf("RED" to "#FF0000",
"FIREBRICK" to "#B22222", "CRIMSON" to
"#DC143C")

 builder.append("\n\nParcurgere harta")
 reds.forEach{key,value->
 builder.append("\n$key : $value")

 textView.text = builder.toString()
```

LinkedHashMap

```
customers.mapKeys
{
it.toString() } // "1" =
Customer("Dina","Kreps",1),
        customers.map
                { it.key * 10 to it.value.id } // 10= 1, 20 =2
        customers.mapValues
                { it.value.lastName } // 1=Kreps, 2="Smith
        customers.flatMap
                { (it.value.firstName + it.value.lastName).toSet()}.toSet()
                                                //D, i, n, a, K, r, e, p, s, A, d, y, S, m, t, h]
linkedHashMap.filterKeys
        { it.contains("r") } //red=#FF0000,
states.filterNot
        { it.value.startsWith("C") } //AL=Alabama, AK=Alaska,AZ=Arizona
}
```

LinkedHashMap

```
interface Cache {
        val size: Int
        operator fun set(key: Any, value: Any)
        operator fun get(key: Any): Any?
        fun remove(key: Any): Any?
        fun clear() }
class LRUCache(private val delegate: Cache, private val minimalSize: Int =
DEFAULT_SIZE) : Cache  {
private val keyMap = object : LinkedHashMap<Any, Any> (minimalSize, .75f, true)
{ override fun removeEldestEntry(eldest: MutableMap.MutableEntry<Any, Any>):
        Boolean { val tooManyCachedItems = size > minimalSize
        if (tooManyCachedItems) eldestKeyToRemove = eldest.key
        return tooManyCachedItems }
}
```

TreeMap

```
SortedMap<String, String> fileExtensions = new TreeMap<>( //ex1 cu un comparator explicit
new Comparator<String>()
 {  override public int compare(String s1, String s2)
            { return s2.compareTo(s1);} }  );
fileExtensions.put("python", ".py");
fileExtensions.put("kotlin", ".kt");

SortedMap<String, String> fileExtensions = new TreeMap<>(String.CASE_INSENSITIVE_ORDER); //ex2
fileExtensions.put("PYTHON", ".py");
fileExtensions.put("KOTLIN", ".kt");

TreeMap<Integer, String> employees = new TreeMap<>(); //ex3
      employees.put(1003, "Rajeev");
      employees.put(1001, "James");
      employees.put(1002, "Sachin");
      employees.put(1004, "Chris");
print(employees.size());
      Integer id = 1004; // caut aangajat
      if(employees.containsKey(id)) {
         // Get the value associated with a given key in a TreeMap
         String name = employees.get(id);// ii aflu numele
         print(name);
      } else
         print("eroare");
```

Colecții - Array

```
public class Array<T> : Cloneable
{
public inline constructor(size:Int, init: (Int) ->T)
public operator fun get(index: Int): T
public operator fun set(index: Int, value: T): Unit
public val size: Int
public operator fun iterator(): Iterator<T>
public override fun clone(): Array<T>
}
```

Colecții - Iterator

```
public interface Iterator<out T>
{ public operator fun next(): T public operator fun hasNext(): Boolean }
public interface MutableIterator<out T> : Iterator<T>
{ public fun remove(): Unit }
public interface ListIterator<out T> : Iterator<T>
{//Operatii inspecție date
override fun next(): T
override fun hasNext(): Boolean
public fun hasPrevious(): Boolean
public fun previous(): T
public fun nextIndex(): Int
public fun previousIndex(): Int  }
public interface MutableListIterator<T> : ListIterator<T>, MutableIterator<T>
{//Operatii inspecție date
override fun next(): T
override fun hasNext(): Boolean
 //Operații de modificare
override fun remove(): Unit
public fun set(element: T): Unit
public fun add(element: T): Unit
}
```

Colecțiile Kotlin sunt deasupra celor Java

- Kotlin-->

<--Mașina Virtuală

Tipuri platformă (cochilie)

• String! sau ArrayList<Int!>! //ex1

pentru String getName() //ex2

val name=getName (IDE va afisa String! ca tip)

val name:String?= getName()

val name:String = getName().

void addFlag(String flag) //ex3

override fun addFlag(flag:String):Unit

override fun addFlag(flag:String?).

Cochilii - exemplu

```
fun <T> itWorks(list: List<T>): Unit
{
println("Java Class Type:${list.javaClass.canonicalName}")
}
val jlist = ArrayList<String>()
jlist.add("sample")
itWorks(jlist)
itWorks(Collections.singletonList(1))
```

Exemplu Array

```
val intArray = arrayOf(1, 2, 3, 4) //1
println("Int array:${intArray.joinToString(",")}")
println("Element at index 1 is:${intArray[1]}")

val stringArray = kotlin.arrayOfNulls<String>(3) //2
stringArray[0] = "a"
stringArray[1] = "b"
stringArray[2] = "c"
//stringArrays[3]="d" //genereaza eroare depasire dimensiune
println("String array:${stringArray.joinToString(",")}")

val studentArray = Array<Student>(2) { index -> //3
        when (index)
        {
        0 -> Student(1, "Alexandra", "Brook")
        1 -> Student(2, "James", "Smith")
        else ->throw IllegalArgumentException("Prea multi")
        }
}
println("Student array:${studentArray.joinToString(",")}")
println("Student at index 0:${studentArray[0]}")

val longArray = emptyArray<Long>() //4
println("Long array:${longArray.joinToString(",")}")
```

Funcții ajutătoare (helper) - ArraysKt

```
println("Primul element din IntArray:${ints.first()}")
println("Ultimul element din IntArray:${ints.last()}")
println("Ia primele trei elemente IntArray:${ints.take(3).joinToString(",")}")
println("Ia ultimele trei elemente din
IntArray:${ints.takeLast(3).joinToString(",")}")
println("Ia elementelel mai mici ca 5 din  IntArray:
        ${ ints.takeWhile { it <5 }.joinToString(",") }")
println("Ia fiecare al treilea element din IntArray:
${ints.filterIndexed { index, element -> index % 3 == 0}.joinToString(",")}")
```

Alt exemplu tratare tablouri

```
public fun IntArray.take(n: Int): List<Int>
{
require(n >= 0) { "Numarul de elemente $n este negativ" }
if (n == 0) return emptyList()
if (n >= size) return toList()
if (n == 1) return listOf(this[0])
var count = 0
val list = ArrayList<Int>(n)
for (item in this)
        {
        if (count++ == n)
        break;
        list.add(item)
        }
return list
}
```

Exemple conversie tablouri

```
val strings = ints.map { element ->"Item " + element.toString() }
println("Converteste elementele IntArray intr-un
string:${strings.joinToString(",")}")
public inline fun <R> IntArray.map(transform: (Int) ->R): List<R>
    {
    return mapTo(ArrayList<R>(size), transform)
    }
public inline fun <R, C : MutableCollection<in R>>
IntArray.mapTo(destination: C, transform: (Int) ->R): C
    {
    for (item in this)
    destination.add(transform(item))
    return destination
    }
```

Exemple conversie tablouri

```
val charArray = charArrayOf('a', 'b', 'c') //ex1
val tripleCharArray = charArray.flatMap  { c ->charArrayOf(c, c,c).asIterable() }
println("Tripleaza fiecare elemnt din charArray:    ${tripleCharArray.joinToString(",")}}")
//ex2
public inline fun <R> CharArray.flatMap(transform: (Char) ->Iterable<R>): List<R>
        { return flatMapTo(ArrayList<R>(), transform)}
public inline fun <R, C : MutableCollection<in R>>
CharArray.flatMapTo(destination: C, transform: (Char) -> Iterable<R>): C
        {for (element in this)
                {
                val list = transform(element)
                destination.addAll(list)
                }
        return destination }
```

Este immutabilul garantat?

```
(intList as AbstractList<Int>).set(0, 999999) ex1
println("Lista Intregi[${intList.javaClass.canonicalName}]:${intList.joinToString(", ")}")

(nonNulls as java.util.ArrayList).addAll(arrayOf("x", "y")) //ex2
println("lista tari [${nonNulls.javaClass.canonicalName}]:${nonNulls.joinToString( ",")}")

val hacked: List<Int>= listOfNotNull(0,1) //ex3
CollectionsJ.dangerousCall(hacked)
println("Lista modificata[${hacked.javaClass.canonicalName}]:${hacked.joinToString(",") }")

//cod Java
public class CollectionsJ
{
public static void dangerousCall(Collection<Integer> l)
        { l.add(1000);}
}
```

Exemplu de utilizare a celorlalte funcții

```
data class Planet(val name: String, val distance: Long)

val planets = listOf( Planet("Mercury", 57910000), Planet("Venus",108200000), Planet("Earth",
149600000), Planet("Mars", 227940000),Planet("Jupiter", 778330000), Planet("Saturn", 1424600000),
Planet("Uranus", 2873550000), Planet("Neptune", 4501000000), Planet("Pluto", 5945900000))

println(planets.last()) //Pluto

println(planets.first()) //Mercury

println(planets.get(4)) //Jupiter

println(planets.isEmpty()) //false

println(planets.isNotEmpty()) //true

println(planets.asReversed()) //"Pluto", "Neptune"

println(planets.elementAtOrNull(10)) //Null
```