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>()
```

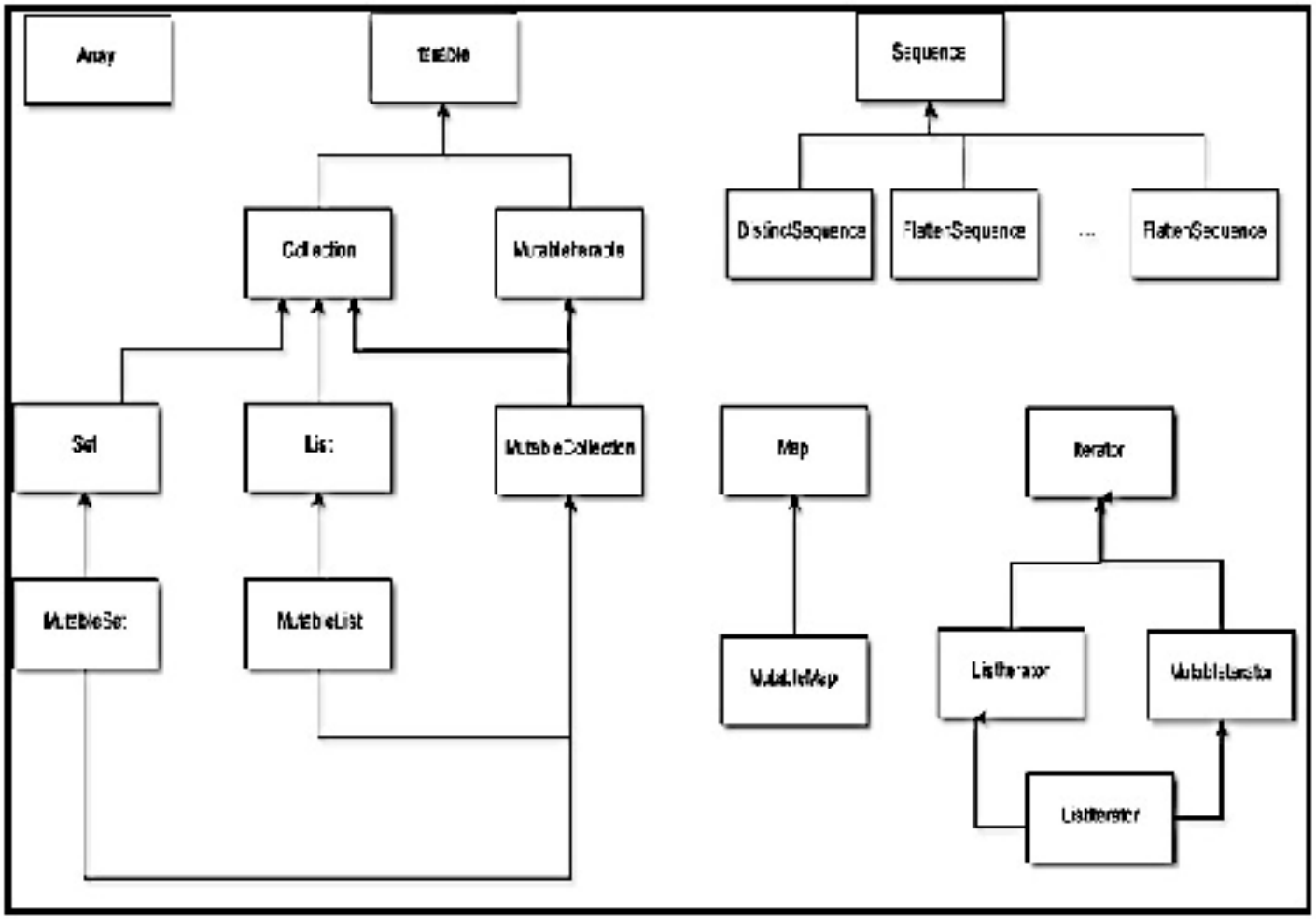
- si utilizarea :

```
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"
```

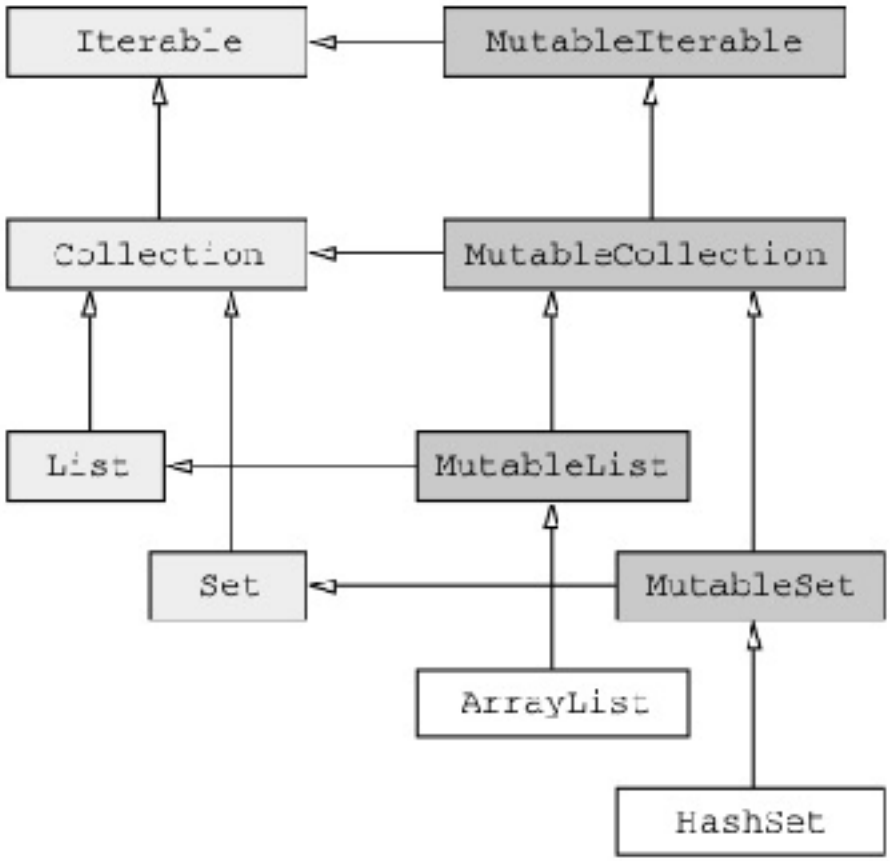
Colecții

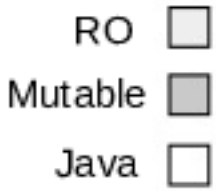
kotlin.collections namespaces

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

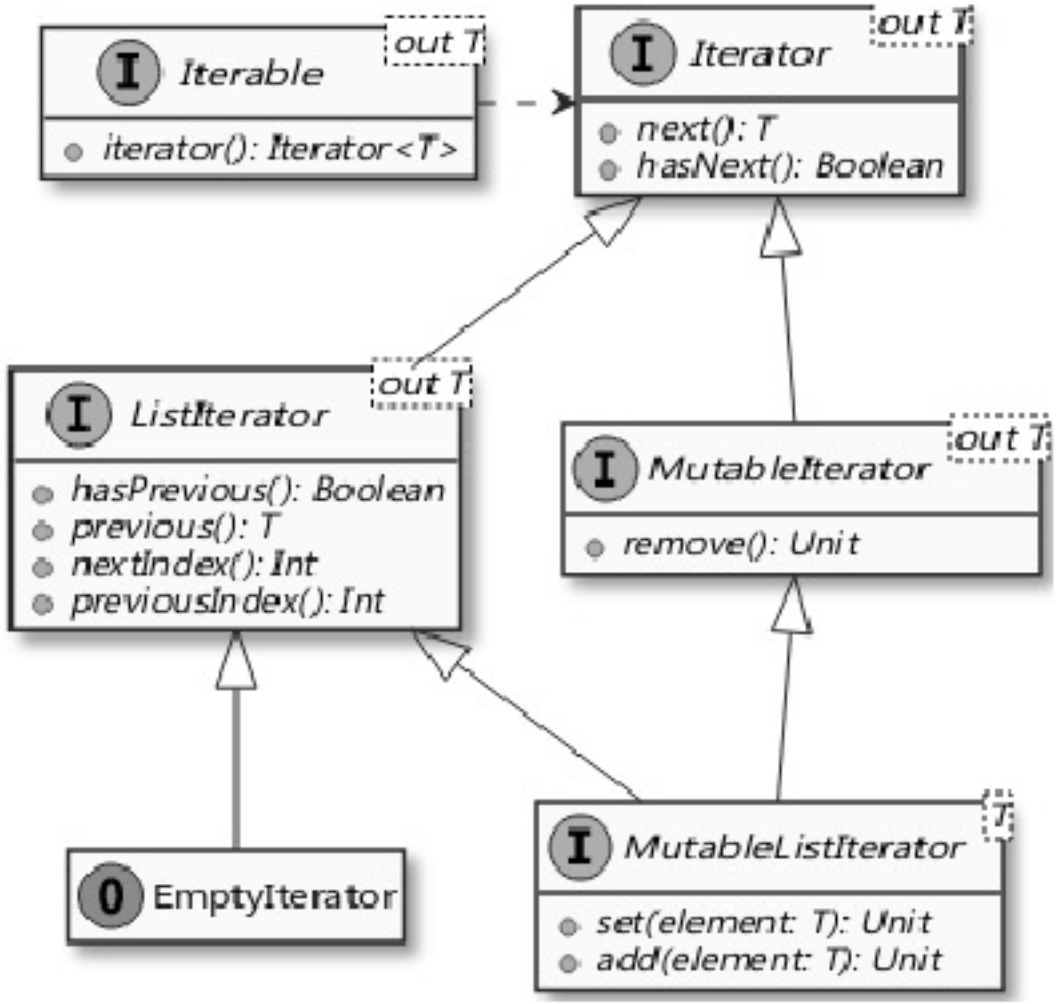
Colecții

```
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