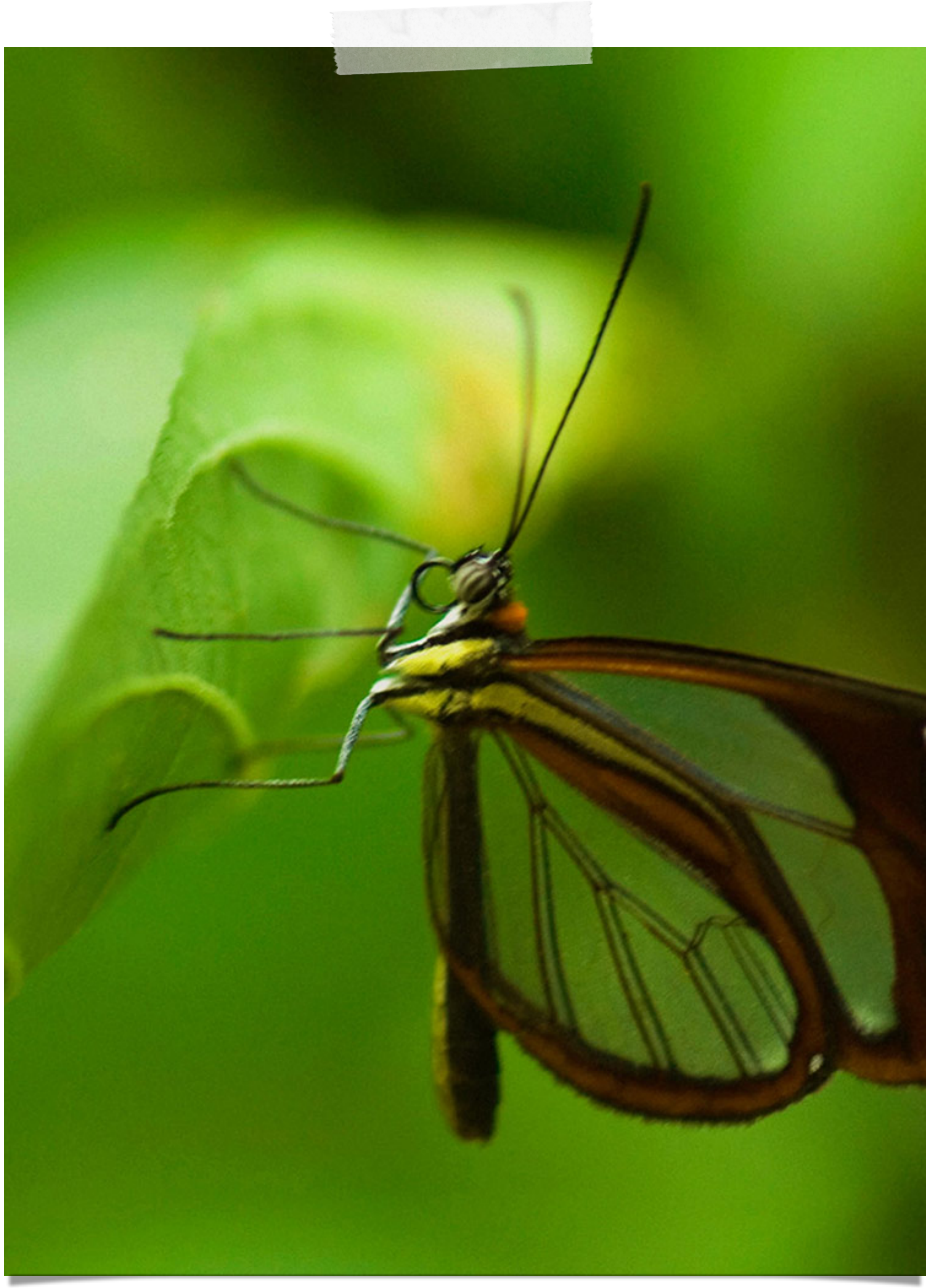
Colecții

Laborator 08

15 - Apr - 2019



Generics

Operatorul Diamond a apărut în java 1.5 / 5

- List<String> names = new ArrayList<String>();

În 1.7 apare simplificarea, deși nu pare are un impact foarte mare în claritatea codului

- List<String> names = new ArrayList<>();

Generics

- Verificarea tipului la compilare
- Eliminarea cast-ului
- Oferă posibilitatea implementării unor algoritmi generici

Generics

Example de clasă generică

```
public class Crate<T> {
    private T contents;
    public T emptyCrate() {
        return contents;
    }

    public void packCrate(T contents) {
        this.contents = contents;
    }
}
```

Generics

Type erasure

- În clasa definită anterior avem un tip T ce este verificat la compilare, însă odată cu realizarea build-ului, tipul T va fi înlocuit cu Object precum mai jos.

```
public class Crate {
    private Object contents;
    public Object emptyCrate() {
        return contents;
    }
    public void packCrate(Object contents) {
        this.contents = contents;
    }
}
```

Generics

Un parametru de tip type poate fi notat cum vrem noi. Convenția spune să folosim uppercase.

- E pentru un element
- K pentru o valoare a unei key
- V pentru o valoare din map
- N pentru un număr
- T pentru un tip generic de data
- S, U, V ș.a.m.d. dacă avem multiple valori generice

Generics

Există un număr de limitări legate de utilizarea generics, majoritatea vin din Type Erasure.

- Apelarea constructorului .new T(), la runtime vom avea new Object().
- Crearea unui array folosind acest tip, deoarece va rezulta un obiect de Objects.
- Apelul lui instanceOf. List<Integer> și List<String> vor arăta la fel la runtime.
- Nu se poate folosi o primitivă ca obiect T, însă se poate folosi wrapper-ul Integer etc.
- Crearea unei variabile statice ca și generic. Nu este acceptat pentru ca tipul este legat de o instanță a clasei

Generics

Bounding

Type of bound	Syntax	Example
Unbounded wildcard	?	List<?> l =new ArrayList<String>();
Wildcard with an upper bound	? extends type	List<? extends Exception> l =new ArrayList<RuntimeException>();
Wildcard with a lower bound	? super type	List<? super Exception> l =new ArrayList<Object>();

Colecții

O colecție este un obiect ce conține, la rândul său, un număr de obiecte. Fiecare obiect este denumit “Element”.

Arhitectura framework-ului conține trei grupe mari de “componente”

- Interfețe
- Clase implementate
- Clase algoritmi

Colecții

Utilizarea de interfețe în loc de clase implementate vine cu un număr de avantaje

- Codul implementat ce utilizează interfețe nu este legat specific de o implementare. Are o mare flexibilitate la schimbare.
- Clasele specifice ce utilizează o anumită implementare pot fi schimbate cu ușurință, fără a modifica prea mult codul existent.
- Rămâne deschisă posibilitatea de a realiza propria implementare a unei colecții

Colecții - iterator

Interfața `Iterator<E>` conține o multitudine de metode ce permit manipularea:

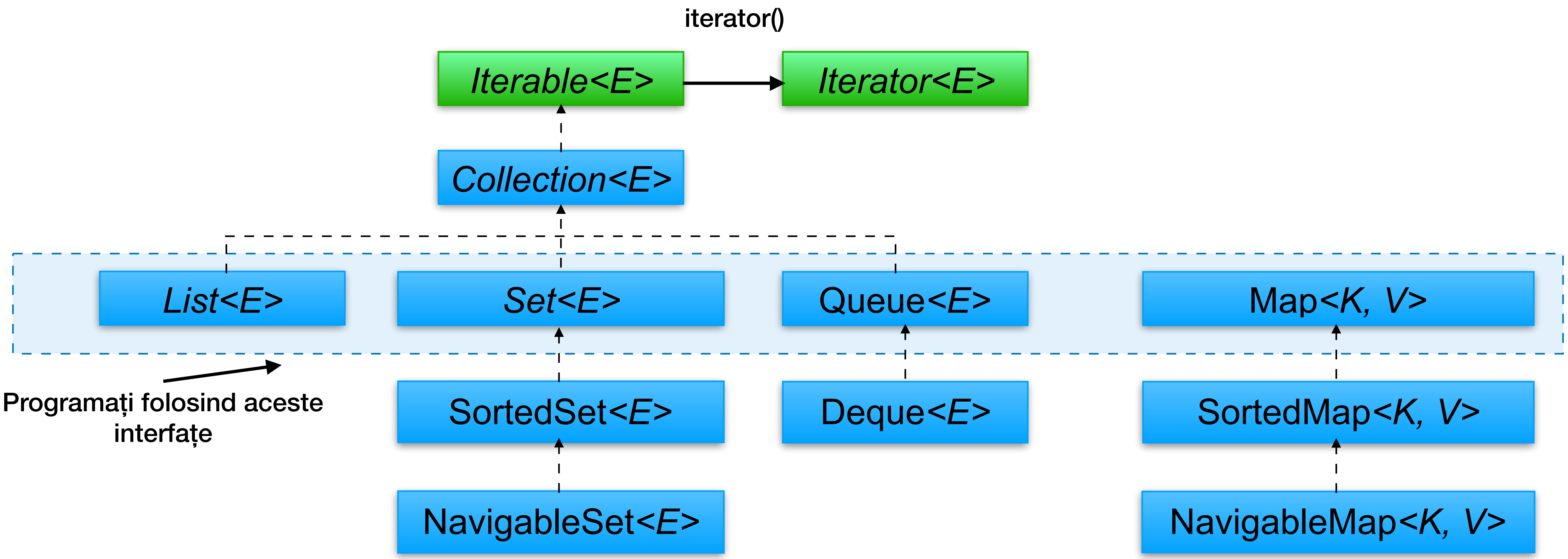
- boolean hasNext()
- E next()
- default void remove()
- default void forEachRemaining(Consumer<? super E> action)

Colecții - iterator

Un exemplu de parsare a unei colecții de tip list

```
List<String> list = new ArrayList<>();

list.add("first item");
list.add("second item");
list.add("third item");

for(String item : list) {
    System.out.println(item);
}

System.out.println("now with iterator!!!");
list.iterator().forEachRemaining(System.out::println);
// another iterator approach

Iterator<String> it = list.iterator();

while (it.hasNext()) {
    System.out.println(it.next());
    it.remove();
}
```

Collection framework



iterator()

Iterable<E>

Iterator<E>

Collection<E>

List<E>

Set<E>

Queue<*E*>

Map<*K, V*>

Programați folosind aceste interfațe

SortedSet<*E*>

Deque<*E*>

SortedMap<*K, V*>

NavigableSet<*E*>

NavigableMap<*K, V*>

Clase ce implementează aceste interfețe

Array*List*
LinkedList
Stack
Vector(sync)

HashSet
LinkedHashSet
TreeSet(SortedSet)

PriorityQueue
ArrayDeque
LinkedList

HashMap
LinkedHashMap
HashtTable(sync)
TreeMap(sorted)

BOXING / UNBOXING

Boxing / unboxing

Primitive type	Wrapper class	Example of initializing
boolean	Boolean	new Boolean(true)
byte	Byte	new Byte((byte) 1)
short	Short	new Short((short) 1)
int	Integer	new Integer(1)
long	Long	new Long(1)
float	Float	new Float(1.0)
double	Double	new Double(1.0)
char	Character	new Character('c')

Comparator vs. Comparable

- Comparable - este o interfață cu o singură metodă

```
public interface Comparable<T> {

   public int compareTo(T o);

}
```

- Comparator - este interfata cu o metodă

  compare(T o1, T o2)

  Nota

  Implementarea metodei compareTo trebuie să fie în respect cu metoda equals.

Comparator vs. Comparable

Difference	Comparable	Comparator
Package name	java.lang	java.util
Interface must be implemented by class comparing?	Yes	No
Method name in interface	compareTo	compare
Number of parameters	1	2
Common to declare using a lambda	No	Yes

NAVIGABLESET

The ***java.util.NavigableSet*** interface is a subtype of the java.util.SortedSet interface. It behaves like a TreeSet and in addition it has navigation methods.

- **descendingSet() –** returns a NavigableSet in which the order of the elements is reversed. The changes to the descending set are also reflected in the original set.

- **descendingIterator() –** allows you to iterate the elements in reverse order.

- **headset()** – returns of a view of the original NavigableSet which only contains elements that are “less than” the given element.

- **tailSet()** - returns all elements that are **higher** than the given parameter element.

- **subset()** - pass two parameters demarcating the boundaries of the view set to return.

NAVIGABLESET

- **higher() –** returns the least (smallest) element in the set that is greater than (not equal too) the element passed as parameter to the method.

- **lower() –** opposite of higher() method.

- **ceiling() –** returns the least (smallest) element in the set that is greater than or equal to the element passed as parameter to the method.

- **floor() –** opposite of ceiling() method.

- **pollFirst()** – returns and removes the "first" element (smallest element) in the set or null if the set is empty.

- **pollLast()** – returns and removes the "last" element (largest)in the set or null if the set is empty.

Queue vs Deque

Queue	Deque	Note
add(e)	addLast(e)	Adaugă un element
offer(e)	offerLast(e)	Adaugă element, respectiv adaugă pe ultima poziție
remove()	removeFirst()	Obține și șterge
poll()	pollFirst()	Obține primul element
element()	getFirst()	Obține dar nu șterge
peek()	peekFirst()	Obține și șterge

Există și addFirst, offerFirst, poolLast, getLast, peekFirst pentru Deque

Priority queue

A priority queue retrieves elements in sorted order after they were inserted in arbitrary order. This is, whenever you call remove method, you get the smallest element currently in the priority queue. However the priority queue does not sort all its elements. It stores the data in a heap, a self-organizing binary tree in which the add and remove operations cause the smallest element to gravitate to the root, without wasting time on sorting all elements.

Just like a TreeSet, a priority queue can either hold elements of a class that implements the Comparable interface or a Comparator object you supply in the constructor.

Method	Description
int **compareTo**(T o) ~ interface **Comparable**	Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
int **compare**(T o1, T o2) ~ interface **Comparator**	Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

SET - HashSet - TreeSet

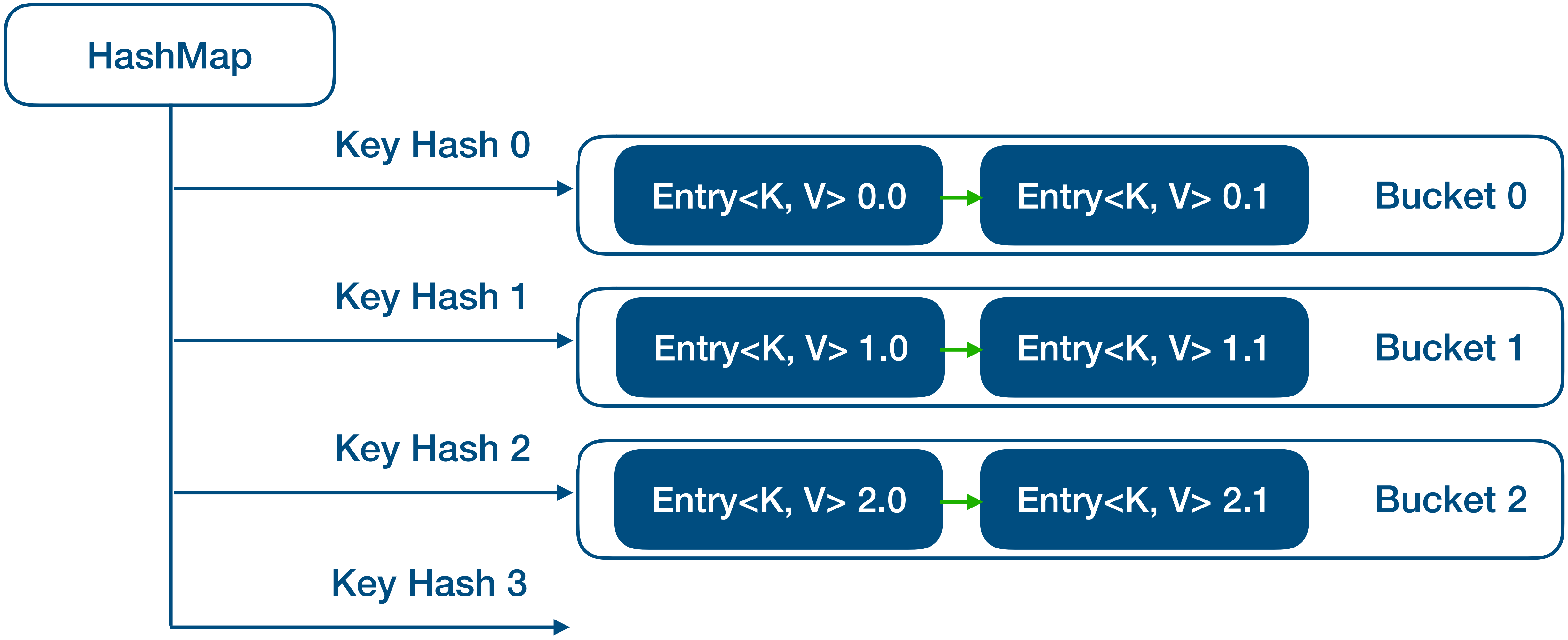
HashSet

- Salvează datele într-un hashtable
- Folosește hashCode pentru a îmbunătății căutarea în Set
- Benefit: adăugarea și căutarea uni element au un time de rezolvare constant
- Tradeoff: Ordinea de inserare este pierdută

TreeSet

- Salvează datele într-o structură sortatată (necesită comprator sau implementarea Comparable)
- Implementează interfața NavigableSet
- Benefit: Setul este mereu sortat
- Tradeoff: adăugarea sau verificarea dacă un element este present are performanța O(log n)

Map - HashMap



HashMap

Key Hash 0

Entry<K, V> 0.0

Entry<K, V> 0.1

Bucket 0

Key Hash 1

Entry<K, V> 1.0

Entry<K, V> 1.1

Bucket 1

Key Hash 2

Entry<K, V> 2.0

Entry<K, V> 2.1

Bucket 2

Key Hash 3

Colecții sincronizate

// get a sync collection from a list

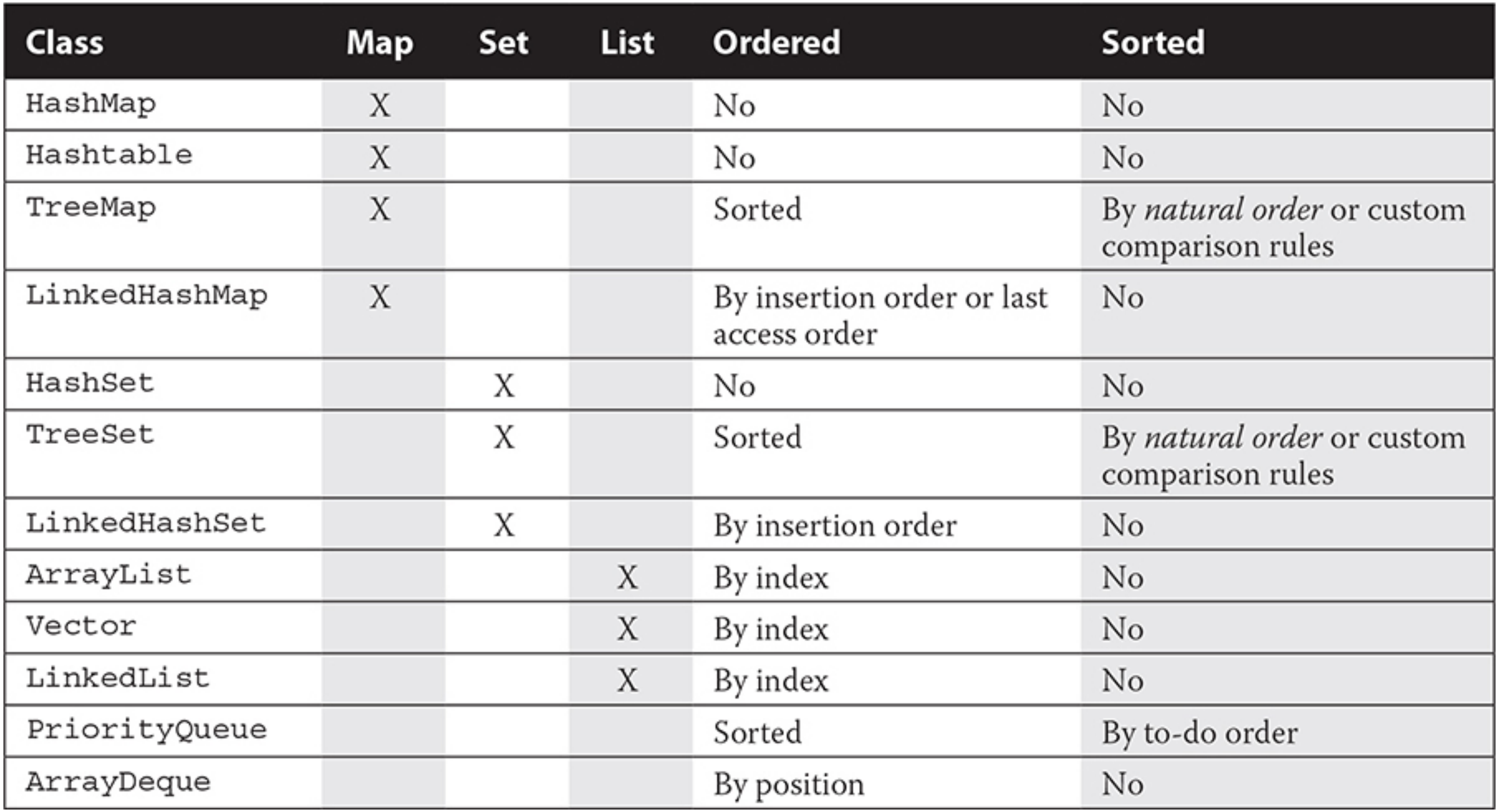
Collection<String> myList = Collections.*synchronizedCollection*(list);

De fapt este un SynchronizedCollection ce implementeaza Collection

Sorted and Ordered

Ordered When a collection is ordered, it means you can iterate through the collection in a specific (not random) order.

Sorted A sorted collection means that the order in the collection is determined according to some rule or rules, known as the “sort order.” A sort order has nothing to do with when an object was added to the collection or when it was last accessed or at what “position” it was added.



Class	Map	Set	List	Ordered	Sorted
`HashMap`	X			No	No
`Hashtable`	X			No	No
`TreeMap`	X			Sorted	By *natural order* or custom comparison rules
`LinkedHashMap`	X			By insertion order or last access order	No
`HashSet`		X		No	No
`TreeSet`		X		Sorted	By *natural order* or custom comparison rules
`LinkedHashSet`		X		By insertion order	No
`ArrayList`			X	By index	No
`Vector`			X	By index	No
`LinkedList`			X	By index	No
`PriorityQueue`				Sorted	By to-do order
`ArrayDeque`				By position	No

COMPARING LIST IMPLEMENTATIONS

All List implementations are ordered and allow duplicates.
Items can be retrieved and inserted at specific positions in the list based on an int index

Big O:

- We use big O notation to talk about the performance of algorithms
- Compares the order of magnitude of performance rather than the exact performance.
- Assumes the worst-case response time and uses an n to reflect the number of elements

- *O(1)—constant time*: It doesn’t matter how large the collection is, the answer will always take the same time to return. Ex: Returning the last element of an array.
- *O(log n)—logarithmic time*: A logarithm is a mathematical function that grows much more slowly than the data size. The point is that logarithmic time is better than linear time. *Divide Et Impera* doesn’t look at the majority of the elements for large collections.
- *O(n)—linear time*: The performance will grow linearly with respect to the size of the collection. Looping through a list and returning the number of elements matching a particular value will take linear time.
- *O(n^2)—n squared time*: Code that has nested loops where each loop goes through the data takes n squared time. Whenever you nave nested for or while loops.

COMPARING LIST IMPLEMENTATIONS

- **ArrayList**

➢Like a resizable array -> the ArrayList automatically grows when elements are added.
➢The look up any element in constant time -> O(1)
➢Adding or removing an element is slower than accessing an element -> O(n-i), where *n* is the number of elements and *i* is the index of the element added or removed
➢Good when you are reading more often than writing to the ArrayList

- **LinkedList**

➢A LinkedList is special because it implements both List and Queue.
➢The main benefits of a LinkedList are that you can access, add, and remove from the beginning and end of the list in constant time -> O(1)
➢The tradeoff is that dealing with an arbitrary index takes linear time. -> O(n)
➢This makes a LinkedList a good choice when you'll be using it as Queue .

COMPARING LIST IMPLEMENTATIONS

Vector

- Very old -> Ages and ages old
- Vector does the same thing as ArrayList except more slowly.
- It is thread-safe -> better ways of doing things with other classes

Stack

- Very old -> Ages and ages old
- Data structure where you add and remove elements from the top of the stack. Think about a stack of paper as an example.
- In fact, Stack extends Vector (thread-safe). If you need a stack, use an ArrayDeque instead.