Java Collections Framework

Materialul folosește

The Java™ Tutorials

https://docs.oracle.com/javase/tutorial/collections/TOC.html

și

Slides by Donald W. Smith
TechNeTrain.com

Collection

C*ollection = u*n container

Un obiect care grupează mai multe obiecte într-o singură unitate.

În mod obișnuit o colecție este formată din elemente care pot forma un grup în mod natural (un set de cărți, emailurile primite, trimise, numerele de telefon ale prietenilor).

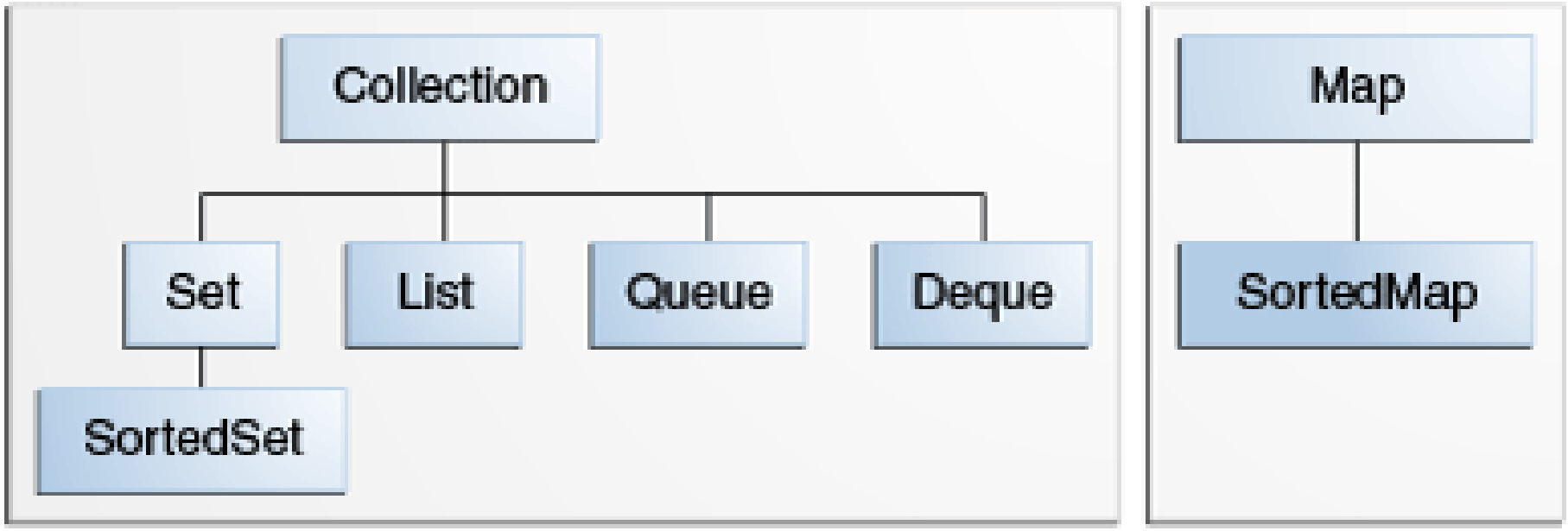
Java Collections Framework

O arhitectură unificată pentru a reprezenta și prelucra colecțiile.

Conține:

- Interfețe
- Implementări (structuri de date reutilizabile)
- Algoritmi – metode ce realizează căutarea, sortarea elementelor colecții; algoritmii sunt polimorfici

Collection- Interfețe



Toate interfețele sunt generice

public interface Collection<E>...

Collection	Nu are o implementare directă în JDK. Reprezintă cel mai general tip de colecţie fiind utilizat în special pentru transferul colecţiilor ca argumente ale unor metode.
List	o colecţie secvenţială în care elementele sunt ordonate după indecşi. Se admit duplicate ale elementelor, stocarea şi regăsirea lor fiind posibilă datorită indecşilor (similar cu clasa Vector).
Set	implementează o ***mulţime*** (o colecţie care conţine numai elemente unice)
SortedSet	un obiect **set** care garantează traversarea de către iteratorul propriu în ordinea crescătoare a elementelor. Sortarea va fi făcută conform ordinii naturale a cheilor (pe baza interfeţii Comparable din java.lang) sau a unui comparator furnizat ca argument constructorului.
Map	reprezintă un obiect care stochează perechi (cheie, valoare). Mulţimea cheilor nu poate conţine duplicate, o cheie fiind asociată cel mult unei valori.
SortedMap	un obiect **map** care garantează că elementele vor fi menţinute în ordinea crescătoare a cheilor. Sortarea va fi făcută conform ordinii naturale a cheilor (pe baza interfeţii Comparable din java.lang) sau a unui comparator furnizat ca argument constructorului.

Collection

Queue	Reprezintă o coadă; accesul poate fi FIFO dacă este o coadă simplă, sau pe baza unui comparator al priorităților dacă este o PriorityQueue
Deque	o structură abstractă care permite inserările și eliminările la ambele capete; accesul poate fi FIFO sau LIFO; o structură mai bogată decît coada și stiva deoarece înglobează funcționalitățile pentru Queue și Stack în același timp

Iterator	`public interface Iterator<E> {`
	`  boolean hasNext();`
	`  E next();`
	`  void remove(); //optional`
	`}`
	
	for-each ascunde iteratorul, desi îl folosește
	`for (Object o : collection)`
	`  System.out.println(o);`
	Nu se poate folosi **remove()**

Collection

Constructorul de conversie

List<String> lista = new ArrayList<> (c);

Metode ale interfeței Collection

- int **size(),**
- boolean **isEmpty(),**
- boolean **contains(Object element),**
- boolean **add(E element),**
- boolean **remove(Object element),**
- Iterator<E> **iterator().**

Metode ce operează asupra întregii colecții

- boolean **containsAll(Collection<?> c),**
- boolean **addAll(Collection<? extends E> c),**
- boolean **removeAll(Collection<?> c),**
- boolean **retainAll(Collection<?> c),**
- void **clear().**

Metode pentru operații cu vectori

- Object[] **toArray()**
- **<T> T[] toArray(T[] a)**

Iterator în Collection

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

Interfața List

- O listă este o secvență *ordonată* de elemente

  interface List<E> extends Collection, Iterable
- Metode importante List:
  - void add(int index, E element)
  - E remove(int index)
  - boolean remove(Object o)
  - E set(int index, E element)
  - E get(int index)
  - int indexOf(Object o)
  - int lastIndexOf(Object o)
  - ListIterator<E> listIterator()
- A ListIterator este similar cu Iterator, în plus are metodele hasPrevious și previous

Implementări ***List***

ArrayList

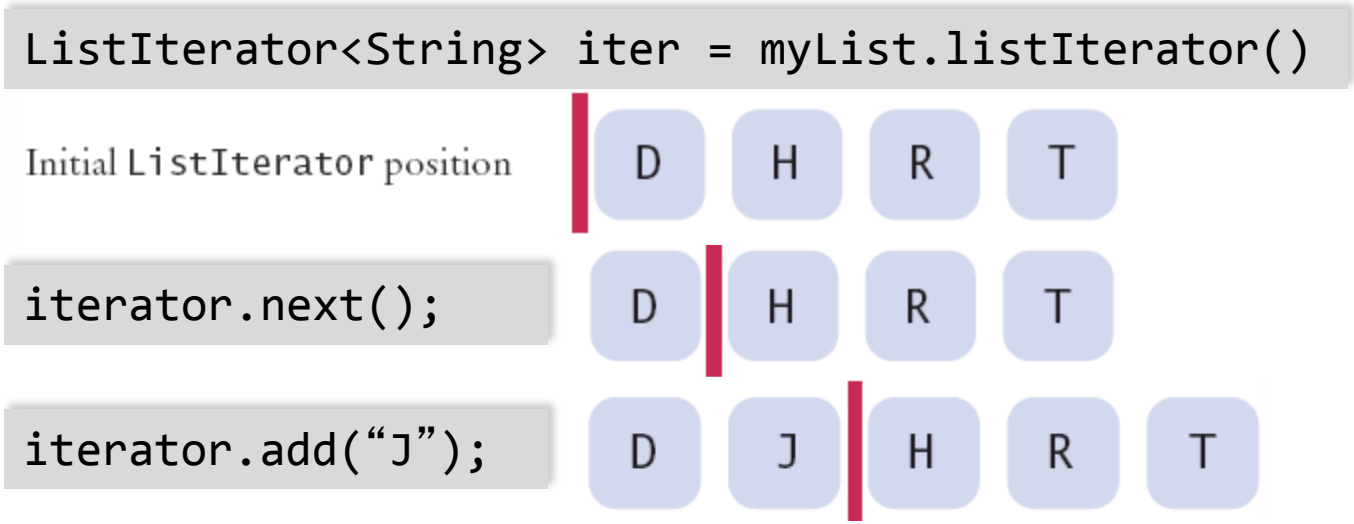
List<String> numeAn3 = new ArrayList<>(numeAn2);

LinkedList

```
LinkedList<String> numeAngajati =...;
ListIterator<String> iter =
  numeAngajati.listIterator()
```

Iteratori

- Imaginați-vă un pointer între 2 elemente = cursorul



  - Tipul generic pentru `listIterator` trebuie să coincidă cu tipul generic al `LinkedList`

ListDemo.java (1)

- Exemplifică add, remove și print

```
1  import java.util.LinkedList;
2  import java.util.ListIterator;
3
4  /**
5     This program demonstrates the LinkedList class.
6  */
7  public class ListDemo
8  {
9     public static void main(String[] args)
10    {
11       LinkedList<String> staff = new LinkedList<String>();
12       staff.addLast("Diana");
13       staff.addLast("Harry");
14       staff.addLast("Romeo");
15       staff.addLast("Tom");
16
17       // | in the comments indicates the iterator position
18
19       ListIterator<String> iterator = staff.listIterator(); // |DHRT
20       iterator.next(); // D|HRT
21       iterator.next(); // DH|RT
22
```

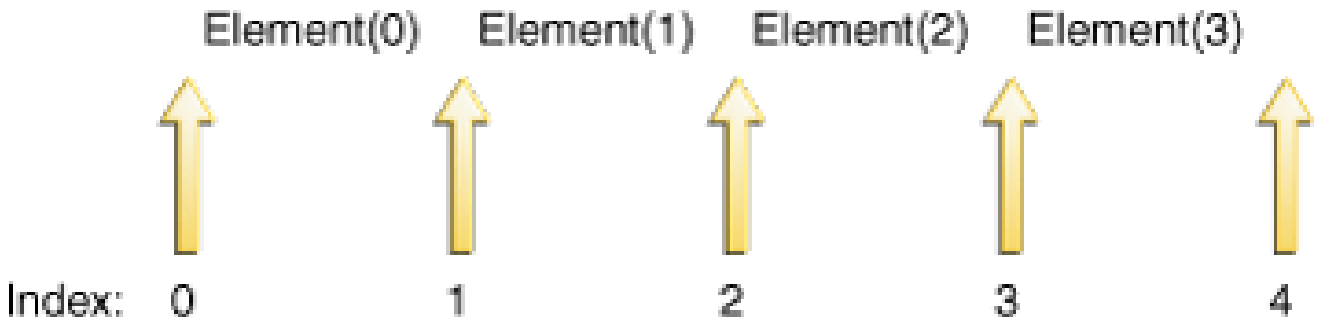
ListDemo.java (2)

```
23         // Add more elements after second element
24
25         iterator.add("Juliet"); // DHJ|RT
26         iterator.add("Nina"); // DHJN|RT
27
28         iterator.next(); // DHJNR|T
29
30         // Remove last traversed element
31
32         iterator.remove(); // DHJN|T
33
34         // Print all elements
35
36         System.out.println(staff);
37         System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
38      }
39   }
```

Program Run

```
[Diana, Harry, Juliet, Nina, Tom]
Expected: [Diana, Harry, Juliet, Nina, Tom]
```

Indexul unui Iterator



hasPrevious() și **previous()** se referă la elementul de dinaintea cursorului

hasNext() și **next()** se referă la elementul de după cursor

```
public int indexOf(E e) {
   for (ListIterator<E>   it = listIterator();   it.hasNext();    )
      if (e == null ? it.next() == null : e.equals(it.next()))
         return it.previousIndex();   //s-a găsit elementul e
   return -1;  //element negăsit
}
```

Metoda set() a unui Iterator

- set() – înlocuiește ultimul element returnat de next() sau previous()

```
public static <E> void replace(List<E> lst, E val, E newVal) {
    for (ListIterator<E> it = lst.listIterator();  it.hasNext();  )
        if (val == null ? it.next() == null : val.equals(it.next()))
            it.set(newVal);
}
```

Înlocuirea unui element cu o listă

```
public static <E>  void replace (
            List<E> list, E val, List<? extends E> newVals) {

   for (ListIterator<E> it = list.listIterator(); it.hasNext(); ){

      if (val == null ? it.next() == null : val.equals(it.next()))
      {
          it.remove();
          for (E e : newVals)
              it.add(e);
       }
   }
}
```

Interschimbă și amestecă

```
public static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}

public static void shuffle(List<?> list, Random rnd) {
    for (int i = list.size(); i > 1; i--)
        swap(list, i - 1, rnd.nextInt(i));
}
```

Un algoritm polimorfic, fair (permutările se realizează de la sfârșit către început cu aceeași probabilitate) **și rapid** (list.size() operații)

Amestecarea șirurilor din linia de c-dă

```
import java.util.*;

public class Shuffle {
   public static void main(String[] args) {
      List<String> list = new ArrayList<String>();
      for (String a : args)
            list.add(a);
      Collections.shuffle(list, new Random());
      System.out.println(list);
   }
}
```

Metoda Arrays.asList()

```
import java.util.*;

public class Shuffle {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }
}
```

`Arrays.asList(vector)` dă o vedere (view) a vectorului.
Aici shuflle folosește o metodă implicită de randomizare.

Sub-liste

List subList(int fromIndex, int toIndex)

Este identică cu

for (int i = fromIndex; i < toIndex; i++) { ... }

Sub-liste

```
lst.subList(fromIndex, toIndex).clear();
```

- Se elimină toate elementele din sublistă de la indicele fromIndex inclusiv la indicele toIndex exclusiv

```
int i = lst.subList(fromIndex, toIndex).indexOf(ob);
int j = lst.subList(10, 20).lastIndexOf(ob);
```

- Sunt căutate doar în sub-listă nu și în listă

Sub-liste

```
public static <E> List<E> dealHand(List<E> deck, int n) {
   int deckSize = deck.size();
   List<E> handView =
                        deck.subList(deckSize - n, deckSize);
   List<E> hand = new ArrayList<E>(handView);
   handView.clear();
   return hand;
}
```

Eliminarea ultimelor elemente din ArrayList este mai rapidă decât eliminarea primelor elemente.

Exemplu complet la
https://docs.oracle.com/javase/tutorial/collections/interfaces/list.html

Algoritmi din List

- sort — sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
- shuffle — randomly permutes the elements in a List.
- reverse — reverses the order of the elements in a List.
- rotate — rotates all the elements in a List by a specified distance.
- swap — swaps the elements at specified positions in a List.
- replaceAll — replaces all occurrences of one specified value with another.
- fill — overwrites every element in a List with the specified value.
- copy — copies the source List into the destination List.
- binarySearch — searches for an element in an ordered List using the binary search algorithm.
- indexOfSubList — returns the index of the first sublist of one List that is equal to another.
- lastIndexOfSubList — returns the index of the last sublist of one List that is equal to another.