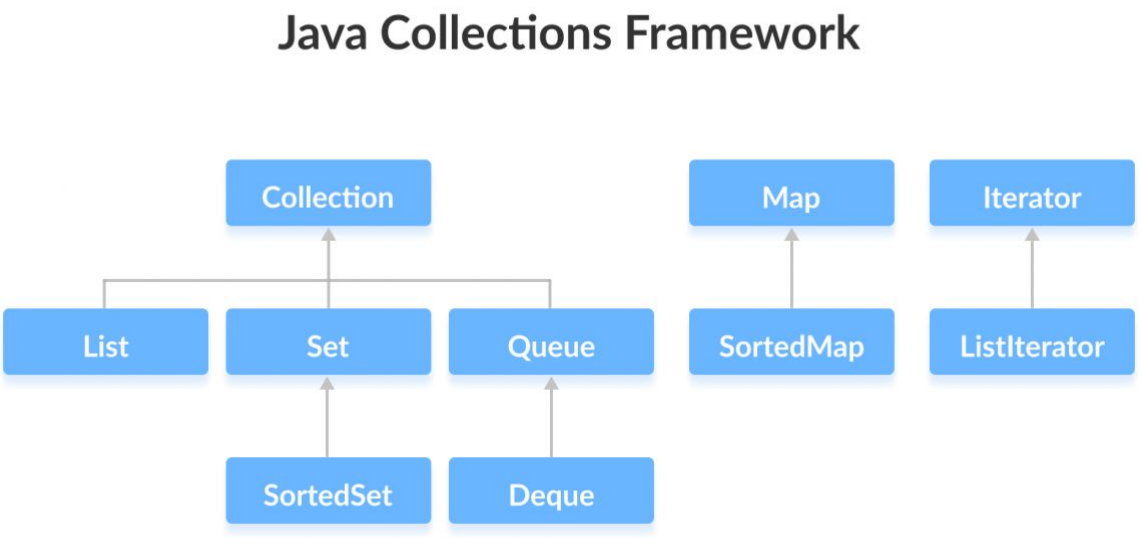
Laborator 7

Colecţii în Java

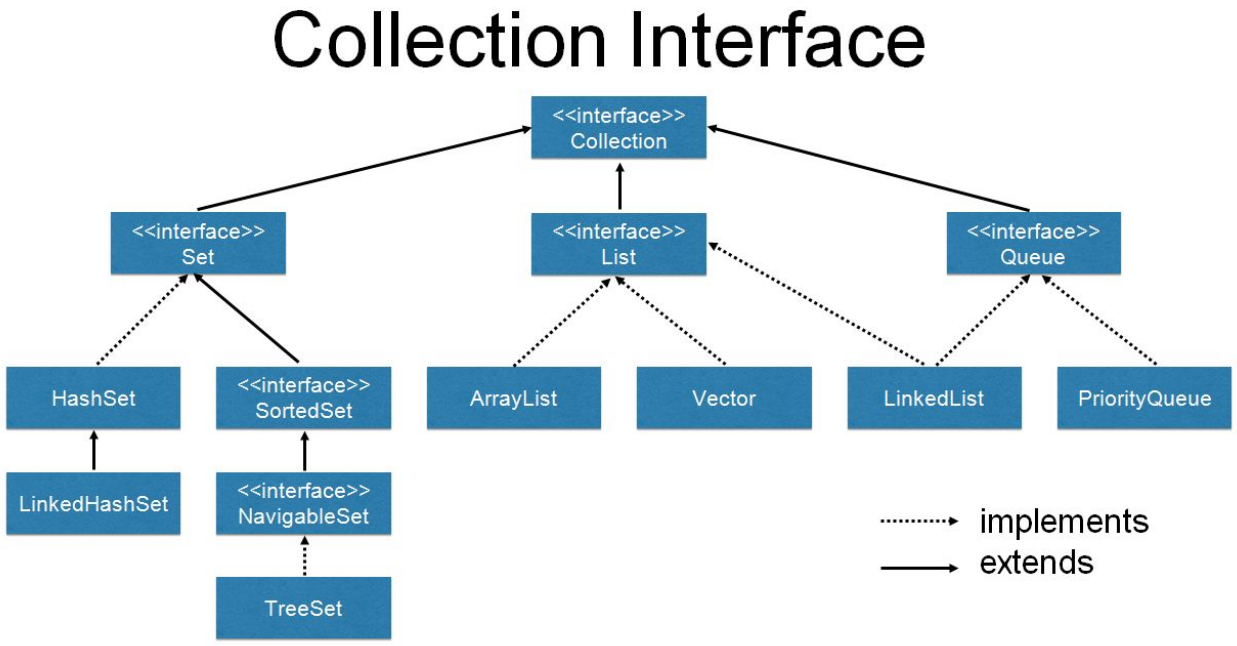
- O colectie reprezinta un grup de obiecte numite şi elemente**.**
- Colecţiile sunt folosite pentru stocarea, căutarea şi manipularea datelor.
- În Java, Collections Framework este o arhitectura unificată pentru reprezentarea şi manipularea colecţiilor. Ea este formată din:
    - **Interfeţe**: permit colecţiilor sa fie folosite independent de implementările lor.
    - **Implementări**: implementările concrete ale tipurilor de date abstracte din interfeţe
    - **Algoritmi**: o colecţie de metode care permit operaţii asupra colectiilor (cautare, sortare)

- Colecţiile oferă implementării pentru următoarele tipuri: mulţimi (ordinea elementelor este importantă), liste (ordinea elementelor conteaza), tabel asociativ/map (perechi cheie-valoare)
- Avantajele utilizarii colecţiilor:
    - reducerea efortului de programare
    - îmbunătăţirea performanţei şi calităţii codului scris



Colecţii bazate pe Interfeţe

- Interfaţa Collection este interfaţa de baza din care sunt specializate celelalte interfete folosite pentru colecţii.
- Scopul ei este de a folosi colecţii la un nivel cat mai general.
- În general, denumirile claselor de colecţii sunt formate din doua parti, prima parte reprezinta structura de date folosită, iar cea de a doua parte reprezinta forma colectiei (interfata implementata). Exemplu: ArrayList, LinkedList



Metodele în interfaţa Collection se împart în trei categorii:

Operaţii de bază la nivel de element:

- int size();
- boolean isEmpty();
- boolean contains(Object element);
- boolean add(Object element);
- boolean remove(Object o)

Operaţii la nivel de colectie:

- boolean containsAll(Collection c);
- boolean addAll(Collection c);

- boolean removeAll(Collection c);
- boolean retainAll(Collection c);
- void clear();

Operaţii de conversie in vector:

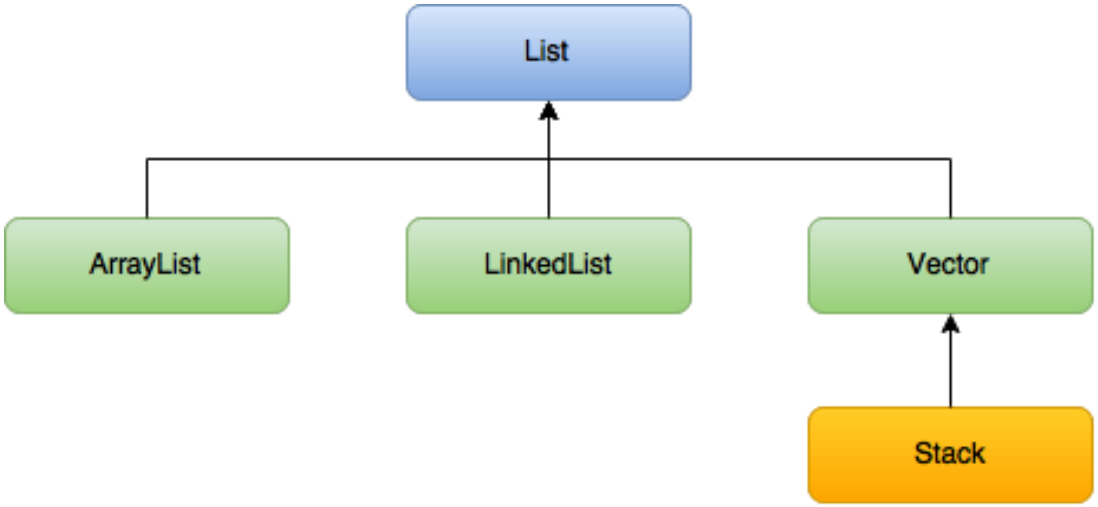
- Object[] toArray();
- Object[] toArray(Object a[]);

List

- Este o colecţie ordonată care permite şi duplicate
- Fiecare element este caracterizat prin poziţia sa în lista.
- Fata de interfaţa Collection, are şi câteva metode specifice pe baza indecsilor.

Metode ale Interfeţei List:

- **T get(int index)** - întoarce elementul de la poziţia index
- **T set(int index, T element)** - modifica elementul de la poziţia index void
- **add(int index, T element)** - adaugă un element la poziţia index
- **T remove(int index)** - şterge elementul de la poziţia index



ArrayList

- Oferă o implementare a unei liste utilizând un tablou unidimensional care poate fi redimensionat dinamic.
- Poate fi referit atât printr-o referinţă de tipul interfeţei implementate (List), cât şi printr-o referinţă de tipul colecţiei:

```
// prin referinta de lista
List<String> listaTablou = new ArrayList<>();
// prin referinta de arraylist
ArrayList<String> listaTablou2 = new ArrayList<>();
```

- O alta metoda de a defini o lista este folosind metoda asList() din clasa Arrays. Limitarea acestei metode, este ca lista o sa fie de dimensiune fixa (nu poţi face operaţii de adaugare sau ştergere).

```
List<String> stringList = Arrays.asList("1", "2", "3", "4", "5");

// operatii de adaugare sau stergere nu sunt permise
stringList.add("6");
stringList.remove( index: 1);
```

- O colecţie ArrayList poate conţine obiecte de tip: clase wrapper, clase definite într-un program, şi de tip colecţie. În schimb nu poţi avea un arraylist de primitive.

```
// allowed
ArrayList<ArrayList<Integer>> arrayListArrayList = new ArrayList<>();
ArrayList<Person> people = new ArrayList<>();
// not allowed
ArrayList<int> arrayListPrimitive = new ArrayList<int>();
```

```java
List<String> listOfStrings = new ArrayList<>();
listOfStrings.add("string1");
listOfStrings.add("string2");
listOfStrings.add("string3");
listOfStrings.add("string4");
listOfStrings.add("string5");

int index = 0;

System.out.println(listOfStrings.get(index));

System.out.println(listOfStrings.set(2, "newString"));

System.out.println(listOfStrings.remove( index: 3));

System.out.println(listOfStrings.lastIndexOf( o: "string5"));
```

LinkedList

- Oferă o implementare a unei liste utilizând o listă dublu înlănţuită.
- Fiecare nod al listei conţine o informaţie de tip generic, precum şi două referinţe: una către nodul anterior şi una către nodul următor.
- Methods:
  - **push()** - adaugă un element la începutul listei
  - **add()** - adaugă un element la finalul listei
  - **peek()** - întoarce elementul de la capul listei, fără să-l şteargă din lista
  - **pop(), poll()** - întoarce elementul de la capul listei şi îl şterge din lista

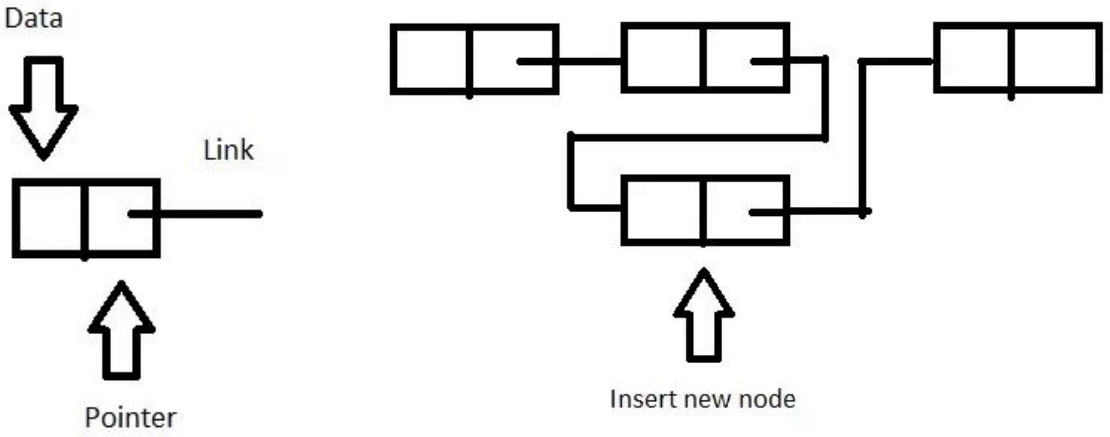
Diferenta intre **pop** şi **poll** sta în felul în care tratează cazul în care se fac operatiile pe o lista goala. Poll intoarce null pe o lista goala, in timp ce pop arunca o exceptie.

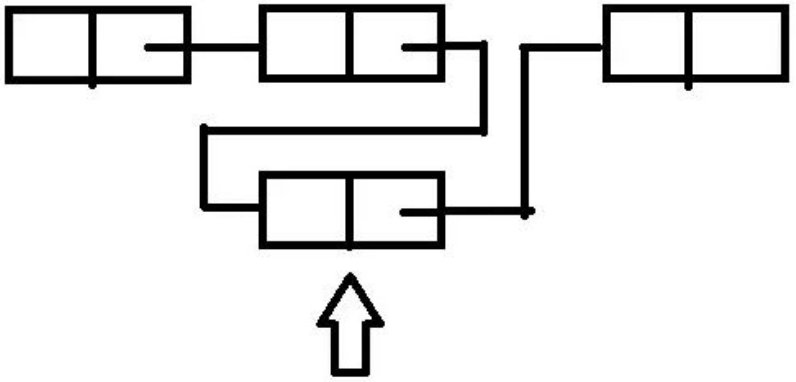
```java
LinkedList<String> lista = new LinkedList<>();
lista.add("A");
lista.add("B");
lista.addLast( e: "C");
lista.addFirst( e: "D");
lista.add( index: 2,  element: "E");
lista.add("F");
lista.add("G");

System.out.println(lista); // [D, A, E, B, C, F, G]

lista.pop();
lista.peek();
lista.poll();

System.out.println(lista); // [E, B, C, F, G]
```





LinkedList vs ArrayList

- Accesul la elemente din lista este mai rapid în ArrayList fără de LinkedList.
- Manipularea elementelor din lista este mai rapid în LinkedList.
- Structura internă pentru stocarea elementelor în LinkedList este lista dublu inlantuita, iar în ArrayList este un tablou dinamic.

Iteratori

- Un iterator este un obiect care permite traversarea unei colecţii şi modificarea acesteia (ex: ştergere de elemente) în mod selectiv.
- Puteţi obţine un iterator pentru o colecţie, apeland metoda sa iterator().
- Interfata Iterator este următoarea:

```
public interface Iterator {
    boolean hasNext();
```

E **next()**;
void **remove()**;
}

Metode:

- hasNext întoarce true dacă mai exista elemente parcurse încă de iteratorul respectiv
- next întoarce următorul element
- remove elimina din colecţie ultimul element întors de next.

Este util sa folosim iteratori cand dorim: ştergerea elementului curent, în timpul iterarii sau cand dorim sa iteram mai multe colecţii în paralel.

```
public class Ex7 {
    public static void main(String [] args){
        Collection c = new ArrayList();
        Iterator it = c.iterator();
        while (it.hasNext()) {
            //verificari asupra elementului curent: it.next();
            it.remove();
        }
    }
}
```

Interfata poate fi folosită pentru orice colecţie.
Set

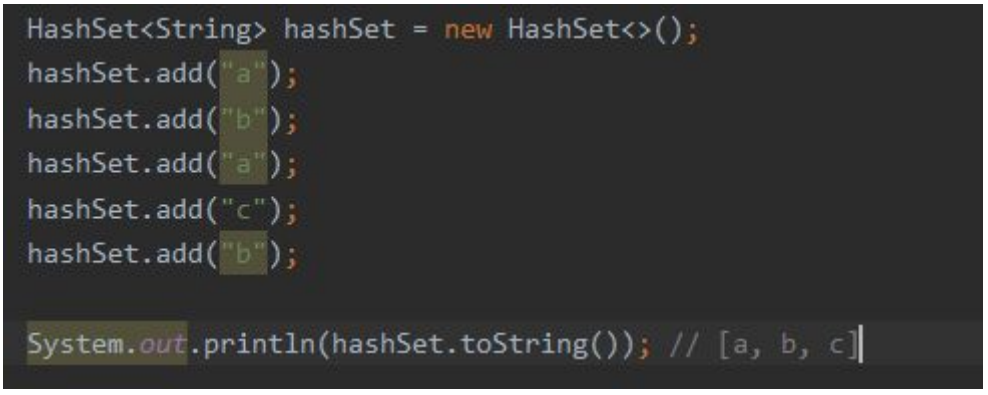
- Reprezinta o colectie care nu permite duplicate.

- Conţine doar metodele moştenite din Collection, la care adaugă restricţii astfel incat elementele duplicate să nu fie poată fi adaugate.

- Implementării are interfeţei Set: HashSet, TreeSet, LinkedHashSet

HashSet

- Stocheaza elementele colectiei intr-un tablou hash.
- Este implementarea cea mai performanta de Set.
- Obiectele inserate într-un HashSet nu sunt garantate sa fie inserate în ordinea în care au fost adaugate. Obiectele sunt inserate pe baza hash-ului



```
HashSet<String> hashSet = new HashSet<>();
hashSet.add("a");
hashSet.add("b");
hashSet.add("a");
hashSet.add("c");
hashSet.add("b");

System.out.println(hashSet.toString()); // [a, b, c]
```

- Pentru un HashSet de obiecte, când se testeaza daca un element mai e în mulţime conteaza ca atat metoda equals sa returneze egalitate cât şi ca obiectele sa aibă acelaşi hash.

```java
public class Valoare {
    private int v;

    public Valoare(int v) { this.v = v; }

    public int getV() { return v; }

    public void setV(int v) { this.v = v; }

    @Override
    public boolean equals(Object o) {...}

    @Override
    public int hashCode() { return Objects.hash(v); }

    @Override
    public String toString() {...}
}
```

```java
HashSet<Valoare> valoareHashSet = new HashSet<>();
Valoare v1 = new Valoare( v: 1);
Valoare v2 = v1;
Valoare v3 = new Valoare( v: 1);
valoareHashSet.add(v1);
valoareHashSet.add(v2);
valoareHashSet.add(v3);

System.out.println(valoareHashSet.toString()); // [Valoare{v=1}]
```

TreeSet

- Nu poate sa contina valori duplicate.
- În implementarea internă foloseşte ca structură de date un TreeMap.
- Nu poate sa contina valori **null**.
- Nu este o colecţie thread safe.
- Pentru sortarea elementelor se poate da ca parametru o clasa Comparator, la crearea unui obiect TreeSet (sau ca clasa care

defineste tipul obiectelor din TreeSet sa implementeze interfața Comparable)

```
public class FruitNameComparator implements Comparator<Fruit> {

    @Override
    public int compare(Fruit o1, Fruit o2) {
        return o1.getName().compareToIgnoreCase(o2.getName());
    }
}
```

```
TreeSet<Fruit> fruitTreeMap = new TreeSet<>(new FruitNameComparator());
fruitTreeMap.add(new Fruit( name: "bapple"));
fruitTreeMap.add(new Fruit( name: "fapple"));
fruitTreeMap.add(new Fruit( name: "capple"));
fruitTreeMap.add(new Fruit( name: "oapple"));
fruitTreeMap.add(new Fruit( name: "iapple"));

System.out.println(fruitTreeMap);
//[Fruit{name='bapple'}, Fruit{name='capple'}, Fruit{name='fapple'}, Fruit{name='iapple'}, Fruit{name='oapple'}]
```

Map

- Este un obiect care mapează chei pe valori.
- Nu pot exista chei duplicate.
- Tipuri de Map-uri: **HashMap**, **TreeMap**

```
HashMap map = new HashMap();
map.put("key1", "value1");
map.put("key2", "value2");
map.put("key3", "value3");
map.put("key4", "value4");
map.put("key5", "value5");

System.out.println(map); // {key1=value1, key2=value2, key5=value5, key3=value3, key4=value4}
```