Laborator 11: Colecții

1 Responsabil

Gabriel Guțu-Robu, gabriel.gutu@upb.ro
Publicat: 8 ianuarie 2023, 03:25

2 Introducere

În programarea orientată pe obiecte, o structură de date (numită și *container*) este o colecție de date, în sine chiar o clasă. Aceasta trebuie să conțină câmpuri și metode care să asigure manipularea datelor. **Java Collections Framework** are două tipuri de structuri de date: una pentru a stoca o colecție de elemente, numită **colecție** și una pentru a stoca perechi de tipul cheie-valoare, numită **map**.

Java Collections Framework are 3 tipuri de colecții: `Set`, `List` și `Queue`. `Set` stochează elemente neduplicate. `List` stochează o colecție ordonată de elemente. `Queue` stochează elemente ce respectă proprietatea primul intrat, primul ieșit. Toate interfețele și clasele definite în **Java Collections Framework** sunt grupate în pachetul `java.util`.

Figura 1 prezintă ierarhia colecțiilor în Java.

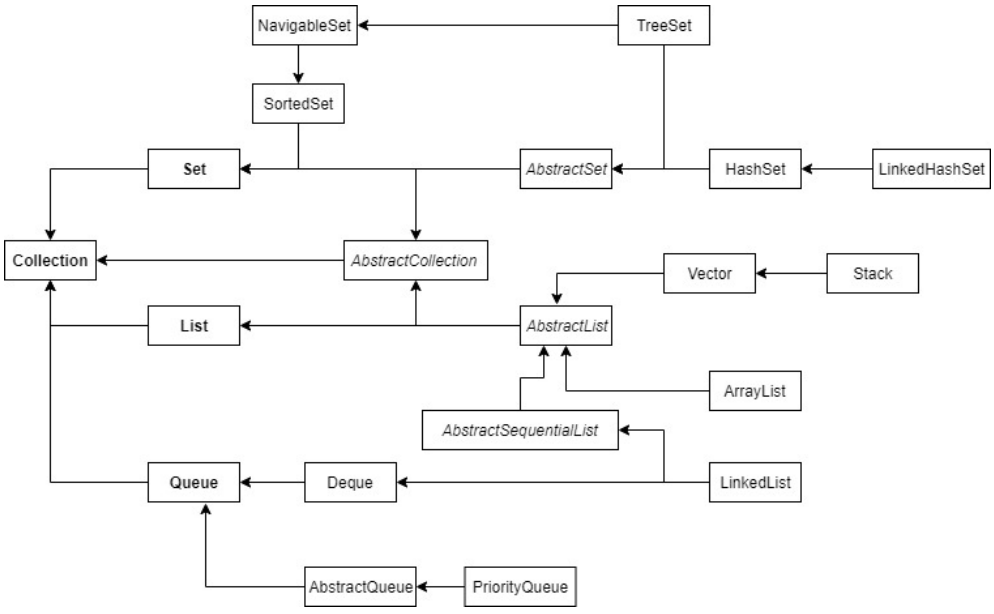

Figura 1. Ierarhia de colecții în Java.

Toate clasele implementează interfețele `Cloneable` și `Serializable`.

Interfața `Collection` este rădăcina pentru lucrul cu colecțiile. `AbstractCollection` este o clasă ce furnizează o implementare parțială a interfeței

`Collection`. Ea implementează toate metodele din `Collection`, mai puțin metodele `size` și `iterator`. Acestea sunt implementate în subclasele corespunzătoare. Metodele din interfața `Collection` sunt prezentate în Tabelul 1.

Tabelul 1. Metodele din interfața Collection.

Metoda	**Scop**
`boolean add(E o)`	Adaugă un nou element `o` în colecție
`boolean addAll (Collection<? extends E> c)`	Adaugă elementele colecției `c` în colecție
`void clear()`	Șterge toate elementele colecției
`boolean contains(Object o)`	Returnează `true` dacă colecția conține elementul `o`
`boolean containsAll (Collection<?> c)`	Returnează `true` dacă această colecție conține toate elementele din `c`
`boolean equals(Object o)`	Returnează `true` dacă colecția este identică cu altă colecție `o`
`int hashCode()`	Returnează codul `hash` al acestei colecții
`boolean isEmpty()`	Returnează `true` dacă colecția nu conține elemente
`Iterator<E> iterator()`	Returnează un iterator pentru elementele din colecție
`boolean hasNext()`	Returnează `true` dacă iteratorul mai are elemente de traversat
`E next()`	Returnează următorul element al iteratorului
`void remove()`	Șterge ultimul element obținut utilizând metoda `next()`
`boolean remove(Object o)`	Elimină elementul `o` din colecție
`boolean removeAll(Collection<?> c)`	Elimină toate elementele din `c` din această colecție

`boolean retainAll(Collection<?> c)`	Păstrează elementele care sunt în `c` și în această colecție
`int size()`	Returnează numărul de elemente din colecție
`Object[] toArray()`	Returnează un tablou de elemente de tipul `Object` pentru elementele din această colecție

3 Set (mulțime)

Interfața `Set` extinde interfața `Collection`. Nu introduce noi metode, ci doar limitează elementele setului să fie unice. `AbstractSet` este o clasă abstractă care implementează `Set` și extinde `AbstractCollection`. Metodele `size` și `iterator` nu sunt implementate. Acestea sunt implementate în subclasele corespunzătoare.

`HashSet` este o clasă concretă care implementează `Set`. Se poate crea un `HashSet` cu ajutorul constructorului fără parametri sau dintr-o colecție existentă. Se poate da și o capacitate inițială. Dacă aceasta nu se precizează, se setează automat la `16`. Pentru eficiență, obiectele adăugate într-un `HashSet` trebuie să implementeze metoda `hashCode` într-o manieră care dispersează codul `hash`. Metoda `hashCode` este definită în clasa `Object`. Codul `hash` a două obiecte identice trebuie să fie egal. Metoda `hashCode` pentru clasa `Integer` returnează valoarea întreagă, pentru `Character`, codul Unicode al caracterului.

Elementele unui HashSet nu sunt stocate în ordinea în care sunt inserate!
Pentru a impune o ordine, se folosește LinkedHashSet.

```
import java.util.*;
public class TestHashSet {
     public static void main(String[] args) {
          Set<String> set = new HashSet<String>();
          set.add("Ana");
          set.add("Maria");
          set.add("Andrei");
          set.add("Mihai");
          set.add("Ana");
          System.out.println(set );
          Iterator<String> iterator = set.iterator();
          while(iterator.hasNext()) {
```

```
                System.out.print(iterator.next() + " ");
            }
        }
}
```

LinkedHashSet extinde HashSet cu o implementare dublu înlănțuită care asigură ordinea elementelor. Dacă nu este neapărat necesară păstrarea ordinii elementelor, este mai eficient să se folosească HashSet.

```
import java.util.*;
public class TestHashSet {
     public static void main(String[] args) {
          Set<String> set = new LinkedHashSet<String>();
          set.add("Ana");
          set.add("Maria");
          set.add("Andrei");
          set.add("Mihai");
          set.add("Ana");
          System.out.println(set );
          Iterator<String> iterator = set.iterator();
          for (Object o: set) {
               System.out.print(o.toString() + " ");
          }
     }
}
```

LinkedHashSet nu precizează ordinea în care sunt inserate elementele – crescător sau descrescător. Pentru aceasta, se folosește clasa TreeSet.

TreeSet

SortedSet este o subinterfață a lui Set, care garantează că elementele din set sunt sortate. Metodele first() și last() returnează primul și ultimul element din set, iar headSet(element) și tailSet(element) returnează o porțiune din set în care elementele sunt mai mici, respectiv mai mari sau egale cu element. NavigableSet extinde SortedSet și are metodele lower(e), floor(e), ceiling(e), higher(e), care returnează elementul mai mic, mai mic sau egal, mai mare sau egal sau mai mare decât e. Metodele pollFirst() și pollLast() elimină și returnează primul și ultimul element din treeset.

TreeSet implementează SortedSet. Se pot adăuga obiecte în treeset atâta timp cât pot fi comparate unul cu altul. Există două modalități de a compara obiectele:

- Cu ajutorul interfeței Comparable. Obiectele pot fi comparate cu ajutorul metodei compareTo;
- Cu ajutorul unui comparator din interfața Comparator.

```
import java.util.*;
public class TestHashSet {
     public static void main(String[] args) {
          Set<String> set = new HashSet<String>();
          set.add("Ana");
          set.add("Maria");
          set.add("Andrei");
          set.add("Mihai");
          set.add("Ana");
          System.out.println(set );
          TreeSet<String> treeSet = new TreeSet<String>(set);
          System.out.println("Multimea sortata: " + treeSet);
          System.out.println("Primul     element:     "     +
treeSet.first());
          System.out.println("Ultimul     element:     "     +
treeSet.last());
          System.out.println("Varful:           "           +
treeSet.headSet("Maria"));
          System.out.println("Coada:           "           +
treeSet.tailSet("Maria"));
          System.out.println("Eliminare primul element: " +
treeSet.pollFirst());
          System.out.println("Eliminare ultimul element: " +
treeSet.pollLast());
          System.out.println("Noul tree set: " + treeSet);
     }
}
```

Dacă nu trebuie să mențineți un set sortat atunci când introduceți elemente, este indicat să se folosească `HashSet` deoarece este mai ușor să se adauge și să se elimine elemente. Dacă se dorește ca set-ul să fie sortat, se poate crea un `SortedSet`.

Dacă se dorește inserarea de elemente într-un `TreeSet`, dar ele nu sunt instanțe ale lui `Comparable`, se poate defini un comparator pentru aceste obiecte. Pentru aceasta, se poate crea o clasă care implementează interfața `Comparator`, cu două metode, `compare` și `equals`.

- `public int compare(Object o1, Object o2)` – returnează o valoare negativă dacă `o1` este mai mic decât `o2`, o valoare pozitivă dacă `o1` este mai mare ca `o2` și `0` dacă sunt egale;
- `public boolean equals(Object o)` – returnează `true` dacă cele două obiecte sunt identice.

```
import java.util.Comparator;
public class ObiectGeometricComparator implements Comparator
<ObiectGeometric> {
     public int compare(ObiectGeometric o1, ObiectGeometric
o2) {
          double aria1 = o1.Aria();
          double aria2 = o2.Aria();
          if (aria1 < aria2)
               return -1;
          else if (aria1 == aria2)
               return 0;
          else
               return 1;
     }
}
import java.util.*;
public class Test {
     public static void main(String[] args) {
          Set<ObiectGeometric>    set    =    new    TreeSet
<ObiectGeometric>(new ObiectGeometricComparator());
          set.add(new Dreptunghi(4, 5));
          set.add(new Cerc(40));
          set.add(new Cerc(40));
          for (ObiectGeometric element: set)
               System.out.println("Aria      =      "      +
element.Aria());
     }
}
```

4 List (listă)

Interfața `List` extinde `Collection` și definește o colecție ordonată cu posibile duplicate. Metodele de care dispune interfața `List` sunt prezentate în Tabelul 2.

Tabelul 2. Metodele din interfața List.

Metodele din java.util.List<E>	
Metoda	**Scop**
`boolean add(int index, E element)`	Adaugă un nou element `E` la indexul specificat
`boolean addAll(int index, Collection<? extends E> c)`	Adaugă elementele din `c` în listă începând cu indexul specificat

`E get (int index)`	Returnează elementul de pe poziția `index`
`int indexOf(Object element)`	Returnează indexul primei apariții a lui `element`
`int lastIndexOf(Object element)`	Returnează indexul ultimei apariții a lui `element`
`ListIterator<E> listIterator()`	Returnează list iteratorul pentru elementele listei
`ListIterator<E> listIterator(int startIndex)`	Returnează list iteratorul pentru elementele listei începând cu `startIndex`
`E remove(int index)`	Elimină elementul de pe indexul specificat
`E set(int index, E element)`	Setează elementul la indexul specificat
`List<E> subList(int index1, int index2)`	Returnează o sublistă de la `index1` la `index2-1`

Metodele din java.util.ListIterator<E>

Metoda	Scop
`void add(E o)`	Adaugă obiectul `o` în listă
`boolean hasPrevious()`	Returnează `true` dacă list iteratorul mai are elemente la o traversare inversă
`int nextIndex()`	Returnează indexul următorului element
`E previous()`	Returnează elementul anterior în list iterator
`int previousIndex()`	Returnează indexul elementului anterior
`void set(E o)`	Înlocuiește ultimul element returnat de `next()` sau de `last()` cu elementul `o`

Clasele `ArrayList` și `LinkedList` sunt două implementări concrete ale interfeței `List`. `ArrayList` stochează elementele într-un tablou creat dinamic. Dacă capacitatea tabloului este depășită, un tablou mai mare este creat și elementele din tabloul curent sunt copiate în noul tablou. Constructorii clasei `ArrayList` sunt:

- `ArrayList()` – creează o listă vidă cu capacitate inițială implicită;
- `ArrayList(Collection<? extends E> c)` – creează un `ArrayList` dintr-o colecție existentă;
- `ArrayList(int capacitateInitiala)` – creează un `ArrayList` cu o capacitate inițială specificată.

`LinkedList` stochează elementele într-o listă înlănțuită, și are metode pentru căutarea, inserare și ștergerea elementelor de la ambele capete ale listei.

- `void addFirst(E o)` – adaugă un element la începutul listei;
- `void addLast(E o)` – adaugă un element la finalul listei;
- `E getFirst()` – returnează primul element al listei;
- `E getLast()` – returnează ultimul element al listei;
- `E removeFirst()` – returnează și șterge primul element al listei;
- `E removeLast()` – returnează și șterge ultimul element al listei.

Fie următorul program:

```
import java.util.*;
public class Test {
     public static void main(String[] args) {
          List<Integer> lista = new ArrayList<Integer>();
          lista.add(1);
          lista.add(2);
          lista.add(3);
          lista.add(4);
          lista.add(5);
          System.out.println(lista);

          LinkedList<Integer>    listaInlantuita    =    new
LinkedList<Integer>(lista);
          listaInlantuita.add(1, 10);
          listaInlantuita.removeFirst();
          listaInlantuita.removeLast();

          ListIterator<Integer>          iterator          =
listaInlantuita.listIterator();
          while (iterator.hasNext()) {
               System.out.print(iterator.next() + " ");
          }
```

```
            System.out.println();
            iterator                                                =
listaInlantuita.listIterator(listaInlantuita.size());
            while (iterator.hasPrevious()) {
                  System.out.print(iterator.previous()+" ");
            }
      }
}
```

Se va afișa:

```
[1, 2, 3, 4, 5]
10 2 3 4
4 3 2 10
```

`ArrayList` este eficient pentru regăsirea elementelor și pentru inserări și ștergeri la capete. `LinkedList` este eficient pentru inserarea și ștergerea elementelor de oriunde în listă.

4.1 Metode statice din clasa Collections

Metodele statice din clasa `Collections` includ `binarySearch`, `reverse`, `shuffle`, `copy`, etc.

```
List<String> lista = Arrays.asList("Mihai", "Ana", "George");
Collections.sort(lista);
System.out.println(lista);
List<String> lista = Arrays.asList("Mihai", "Ana", "George");
Collections.reverse(lista);
System.out.println(list);
List<String>    lista1    =    Arrays.asList("Mihai",    "Ana",
"George");
List<String> lista2 = Arrays.asList("Andrei", "Sebastian");
Collections.copy(lista1, lista2);
System.out.println(lista1);
```

Pentru a vedea de câte ori apare un element în colecție, se folosește metoda `frequency`.

```
Collection<String> colectie = Arrays.asList("Ana", "George",
"Ana");
System.out.println(Collections.frequency(colectie, "Ana"));
```

4.2 Vector. Stack (stivă).

Clasa `Vector` este similară cu `ArrayList`, cu diferența că are metode sincronizate pentru accesul și modificarea vectorului. Metodele sincronizate previn coruperea datelor când un vector este accesat și modificat de două fire de execuție simultan. Pentru aplicațiile care nu necesită sincronizare, utilizarea clasei `ArrayList` este mai eficientă și rapidă decât al clasei `Vector`.

O stivă (`Stack`) este o structură de tipul last-in, first-out (ultimul venit, primul servit). Metodele din clasa `Stack` sunt:

- `Stack()` – creează o stivă nouă;
- `boolean empty()` – returnează `true` dacă stiva este vidă;
- `E peek()` – returnează elementul de la vârful stivei;
- `E pop()` – returnează și elimină elementul de la vârful stivei;
- `E push(E o)` – adaugă elementul `o` la vârful stivei;
- `int search(Object o)` – returnează poziția elementului specificat în stivă.

5 Queue (coadă). PriorityQueue (coadă cu priorități)

O coadă (`Queue`) este o structură de tipul first-in, first-out (primul venit, primul servit). Metodele din interfața `java.util.Queue` sunt:

- `boolean offer(E element)` – inserează un element în coadă;
- `E poll()` – returnează și elimină vârful cozii, sau `null` dacă coada este vidă;
- `E remove()` - returnează și elimină vârful cozii, sau aruncă o excepție dacă coada este vidă;
- `E peek()` – returnează, dar nu elimină vârful cozii, sau returnează `null` dacă coada este vidă;
- `E element()` – returnează, dar nu elimină vârful cozii, sau aruncă o excepție dacă coada este vidă.

```
public class TestQueue {
     public static void main(String[] args) {
          java.util.Queue<String>      coada      =      new
java.util.LinkedList<String>();
          coada.offer("rosu");
          coada.offer("verde");
          coada.offer("albastru");
          while (coada.size() > 0)
               System.out.print(coada.remove() + " ");
     }
}
```

Într-o coadă cu priorități, elementelor le sunt atribuite priorități. Când sunt accesate, elementul cu prioritatea cea mai mare este eliminat primul. Clasa `PriorityQueue` implementează o coadă cu priorități. Coada cu priorități ordonează elementele în ordinea lor naturală, utilizând interfața `Comparable`. Elementului cu cea mai mică valoare i se asignează cea mai mare prioritate și astfel este eliminat primul din coadă. Se poate

specifica o ordine, folosind un comparator în constructor: `PriorityQueue (initialCapacity, comparator)`.

6 Map

Un `map` este un container care stochează chei și valorile asociate lor. Cheile pot fi orice obiecte. Un map nu poate conține chei duplicate. O cheie, împreună cu valoarea sa, formează o intrare (en., *entry*). Există 3 tipuri de map-uri: `HashMap`, `LinkedHashMap` și `TreeMap`. Interfața `java.util.Map.Entry` are următoarele metode:

- `K getKey()` – returnează cheia corespunzătoare intrării;
- `V getValue()` – returnează valoarea corespunzătoare intrării;
- `void setValue (V value)` – înlocuiește valoarea în intrare.

Interfața `SortedMap` extinde interfața `Map` și menține maparea în ordine ascendentă a cheilor, cu metodele `firstKey()` și `lastKey()` care returnează cea mai mică și cea mai mare cheie, `headMap(key)` ce returnează porțiunea din map ale cărei chei sunt mai mici decât `key`, și `tailMap(key)` ce returnează porțiunea din map ale cărei chei sunt mai mari sau egale decât `key`.

Clasa `HashMap` este eficientă pentru găsirea unei valori, inserarea sau ștergerea unei intrări. `LinkedHashMap` extinde `HashMap`, cu o implementare înlănțuită, ce asigură ordonarea intrărilor în map. Intrările din `HashMap` nu sunt ordonate, însă intrările din `LinkedHashMap` pot fi găsite fie în ordinea în care au fost inserate în map, fie în ordinea în care au fost ultima dată accesate. Clasa `TreeMap` este eficientă pentru traversarea cheilor în ordine sortată. Cheile pot fi sortate utilizând interfețele `Comparable` sau `Comparator`. Pentru cazul când se implementează interfața `Comparator`, trebuie utilizat constructorul `TreeMap(Comparator comparator)`.

Fie următorul program:

```
import java.util.*;
public class Test {
public static void main(String[] args) {
     Map<String, Integer> hashMap = new HashMap<String,
Integer>();
     hashMap.put("Radu", 30);
     hashMap.put("George", 31);
     hashMap.put("Bogdan", 29);
     hashMap.put("Valentin", 29);
     System.out.println(hashMap);
     Map<String, Integer> treeMap = new TreeMap<String,
Integer>(hashMap);
     System.out.println(treeMap);
```

```
     Map<String,     Integer>     linkedHashMap     =     new
LinkedHashMap<String, Integer>();
     linkedHashMap.put("Radu", 30);
     linkedHashMap.put("George", 31);
     linkedHashMap.put("Bogdan", 29);
     linkedHashMap.put("Valentin", 29);
     System.out.println(linkedHashMap);
     }
}
Se va afișa:
{Radu=30, Bogdan=29, Valentin=29, George=31}
{Bogdan=29, George=31, Radu=30, Valentin=29}
{Radu=30, George=31, Bogdan=29, Valentin=29}
```

7 Aplicații

7.1 Studenți

Să se creeze clasa `Student`, cu următorii membri:

- câmpurile `nume` (de tipul `String`) și `media` (de tipul `float`);
- un constructor cu parametri care inițializează numele și media;
- metoda `getMedia()` ce returnează media studentului;
- metoda `String toString()`, ce returnează numele și media studentului concatenate, sub forma unui `String`, separate printr-un spațiu.

Să se definească o clasă ce va reprezenta un Map în care se vor stoca informații despre studenți, astfel:

- Map-ul va conține chei de la `0` la `10` (corespunzătoare mediilor posibile);
- Fiecărei chei îi va fi asociată o listă (de tipul `ArrayList`) care va reține toți studenții cu media rotunjită egală cu cheia. De exemplu, considerăm că un student are media rotunjită `8` dacă media sa este în intervalul `[7.50, 8.49];`
- Map-ul va menține cheile (mediile) ordonate descrescător. Se va realiza o implementare potrivită a interfeței `Map`, care să permită acest lucru, și se va folosi un `Comparator` pentru stabilirea ordinii cheilor;

```
Map<Integer,ArrayList<Student>>  m  =  new  TreeMap<Integer,
ArrayList<Student>>(new Comparator<Integer>(){
            public int compare(Integer o1, Integer o2) {
                return o2.compareTo(o1);
            }
        });
```

- Să va defini în clasă metoda `adaugaStudent(Student s)`, ce va adăuga un student în lista corespunzătoare mediei lui.

- Se vor citi de la tastatură datele (numele și media) pentru un număr de `n` studenți;
- Să se itereze pe map, utilizându-se varianta specifică de `for-each` și să se afișeze fiecare pereche de tipul cheie-valoare, respectiv media și lista de studenți corespunzătoare.

```
for(Map.Entry<Integer, ArrayList<Student> > intrare : m.entrySet()) {
     System.out.print(intrare.getKey() + "  ");
     System.out.println(intrare.getValue());
}
```

7.2 Pacienți

Implementați clasa `Pacient` care implementează interfața `Comparable<Pacient>`. Clasa `Pacient` are variabilele membru `String nume` și `int prioritate`, ce desemnează prioritatea cu care va intra un pacient în vizită la medic.

Să se suprascrie metoda `int compareTo(Pacient pacient)`, care întoarce `-1` dacă prioritatea elementului curent este mai mare decât a lui pacient, `1` dacă este mai mică și `0` dacă sunt egale. Să se creeze o coadă de priorități cu câteva elemente de tipul `Pacient` și să se elimine pe rând elementele. Dacă metoda `compareTo()` a fost redefinită în mod corect, elementele vor fi eliminate în ordinea descrescătoare a priorităților.