Lucrare de laborator 6

COLECȚII

Obiective

După studierea acestui laborator studenții vor fi capabili:

- să lucreze cu cele trei tipuri principale de colecții din Java: ***List***, ***Set***, ***Map***;
- să explice diferențele dintre implementările colecțiilor (eficiență, sortare, ordonare etc.);
- să compare elementele unor colecții;
- să înțeleagă și să utilizeze metodele equals și hashcode.

1. Collections Framework

În pachetul **java.util** (pachet standard din JRE) există o serie de clase pe care le veți găsi folositoare. Collections Framework este o arhitectură unificată pentru reprezentarea și manipularea colecțiilor. Ea conține:

- **interfețe**: permit colecțiilor să fie folosite independent de implementările lor;
- **implementări**;
- **algoritmi**: metode de prelucrare (căutare, sortare) pe colecții de obiecte oarecare. Algoritmii sunt polimorfici: un astfel de algoritm poate fi folosit pe implementări diferite de colecții, deoarece le abordează la nivel de interfață.

Colecțiile oferă implementări pentru următoarele tipuri:

- **Set** (elemente neduplicate)
- **List** (o mulțime de elemente)
- **Map** (perechi cheie-valoare)

Există o interfață, numită Collection, pe care o implementează majoritatea claselor ce desemnează colecții din **java.util**. Explicații suplimentare găsiți pe Java Tutorials - Collection.

Exemplul de mai jos construiește o listă populată cu nume de studenți:

```
Collection names = new ArrayList();
names.add("Andrei");
names.add("Matei");
```

2. Parcurgerea colecțiilor

Colecțiile pot fi parcurse (element cu element) folosind:

- **iteratori**
- o construcție **for** specială (cunoscută sub numele de **for-each**)

2.1.Iteratori

Un iterator este un obiect care permite traversarea unei colecții și modificarea acesteia (ex: ștergere de elemente) în mod selectiv. Puteți obține un iterator pentru o colecție, apelând metoda sa **iterator()**. Interfața Iterator este următoarea:

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove(); // optional
}
```

Exemplu de folosire a unui iterator:

```
Collection<Double> col  = new ArrayList<Double>();
Iterator<Double> it = col.iterator();

while (it.hasNext()) {
   Double backup = it.next();
   // apelul it.next() trebuie realizat înainte de apelul it.remove()
   if (backup < 5.0) {
      it.remove();
   }
}
```

Apelul metodei remove() a unui iterator face posibilă eliminarea elementului din colecție care a fost întors la ultimul apel al metodei **next()** din același iterator. În exemplul anterior, toate elementele din colecție mai mici decât 5 vor fi șterse la ieșirea din bucla **while**.

2.2.For-each

Această construcție permite (într-o manieră expeditivă) traversarea unei colecții. **for-each** este foarte similar cu **for**. Următorul exemplu parcurge elementele unei colecții și le afișează.

```
Collection collection = new ArrayList();
for (Object o : collection)
   System.out.println(o);
```

Construcția **for-each** se bazează, în spate, pe un iterator, pe care îl ascunde. Prin urmare **nu** putem șterge elemente în timpul iterării. În această manieră pot fi parcurși și **vectori** oarecare. De exemplu, **collection** ar fi putut fi definit ca **Object[]**.

2.3.Genericitate

Fie următoarea porțiune de cod:

```
c.add("Test");
Iterator it  = c.iterator();

while (it.hasNext()) {
   String s = it.next();    //ERROR: next() returns an Object and it's needed an explicit cast to String
   String s = (String)it.next(); // OK
}
```

Am definit o colecție **c**, de tipul **ArrayList** (pe care îl vom examina într-o secțiune următoare). Apoi, am adăugat în colecție un element de tipul **String**. Am realizat o parcurgere folosind un iterator, și am încercat obținerea elementului nostru folosind apelul: *String s = it.next();*. Funcția **next** însă întoarce un obiect de tip **Object**. Prin urmare apelul va eșua. Varianta corectă este *String s = (String)it.next();*. Am fi putut preciza, din start, ce tipuri de date dorim într-o colecție:

```
Collection<String> c = new ArrayList<String>();
c.add("Test");
c.add(2);     // ERROR!
Iterator<String> it = c.iterator();

while (it.hasNext()) {
    String s = it.next();
```

```
}
```

Mai multe detalii despre acest subiect găsiți în laboratorul următor: **Genericitate**.

3. Interfața List

O listă este o colecție **ordonată**. Listele **pot** conține elemente **duplicate**. Pe lângă operațiile moștenite de la **Collection**, interfața List conține operații bazate pe poziție (**index**), de exemplu: *set*, *get*, *add* la un index, *remove* de la un index.

```
List<String> fruits = new ArrayList<>(Arrays.asList("Apple", "Orange", "Grape"));
fruits.add("Apple");            // metodă moștenită din Collection
fruits.add(2, "Pear");          // [Apple, Orange, Pear, Grape, Apple]
System.out.println(fruits.get(3)); // Grape
fruits.set(1, "Cherry");        // [Apple, Cherry, Pear, Grape, Apple]
fruits.remove(2);
System.out.println(fruits);     // [Apple, Cherry, Grape, Apple]
```

Alături de **List**, este definită interfața ListIterator, ce extinde interfața **Iterator** cu metode de parcurgere în ordine inversă. **List** posedă două implementări standard:

- **ArrayList** - implementare sub formă de vector. Accesul la elemente se face în timp constant: O(1)
- **LinkedList** - implementare sub formă de listă dublu înlănțuită. Prin urmare, accesul la un element nu se face în timp constant, fiind necesară o parcurgere a listei: O(n).

Printre algoritmii implementați se numără:

- **sort** - realizează sortarea unei liste
- **binarySearch** - realizează o căutare binară a unei valori într-o listă sortată

În general, algoritmii pe colecții sunt implementați ca metode statice în clasa Collections.

Atenție: Nu confundați interfața Collection cu clasa Collections. Spre deosebire de prima, a doua este o clasă ce conține exclusiv metode statice. Aici sunt implementate diverse operații asupra colecțiilor.

Iată un **exemplu** de folosire a sortării:

```
List<Integer> l = new ArrayList<Integer>();
l.add(5);
l.add(7);
l.add(9);
l.add(2);
l.add(4);

Collections.sort(l);
System.out.println(l);
```

Mai multe detalii despre algoritmi pe colecții găsiți pe Java Tutorials - Algoritmi pe liste.

4. Compararea elementelor

Rularea exemplului de sortare ilustrat mai sus arată că elementele din **ArrayList** se sortează crescător. Ce se întâmplă când dorim să realizăm o sortare particulară pentru un tip de date complex? Spre exemplu, dorim să sortăm o listă *ArrayList<Student>* după media anilor. Să

presupunem că Student este o clasă ce conține printre membrii săi o variabilă ce reține media anilor. Acest lucru poate fi realizat folosind interfețele:

- Comparable
- Comparator

	Comparable	Comparator
Logica de sortare	Logica de sortare trebuie sa fie în clasa ale cărei obiecte sunt sortate. Din acest motiv, această metodă se numește *sortare naturală*.	Logica de sortare se află într-o **clasă separată**. Astfel, putem defini mai multe metode de sortare, bazate pe diverse câmpuri ale obiectelor de sortat.
Implementare	Clasa ale cărei instanțe se doresc a fi sortate trebuie să **implementeze această interfață** și, evident, să suprascrie metoda *compareTo()*.	Clasa ale cărei instanțe se doresc a fi sortate nu trebuie să implementeze această interfață. Este nevoie de o alta clasă (poate fi și internă) care să implementeze interfața Comparator.
Metoda de comparare	*int compareTo(Object o1)* Această metodă compară obiectul curent (**this**) cu obiectul **o1** și întoarce un întreg. Valoarea întoarsă este interpretată astfel: 1. pozitiv – obiectul este mai mare decât o1 2. zero – obiectul este egal cu o1 3. negativ – obiectul este mai mic decât o1	*int compare(Object o1, Object o2)* Această metodă compară obiectele **o1 and o2** și întoarce un întreg. Valoarea întoarsă este interpretată astfel: 1. pozitiv – o2 este mai mare decât o1 2. zero – o2 este egal cu o1 3. negativ – o2 este mai mic decât o1
Metoda de sortare	*Collections.sort(List)* Aici obiectele sunt sortate pe baza metodei *compareTo()*.	*Collections.sort(List, Comparator)* Aici obiectele sunt sortate pe baza metodei *compare()* din Comparator.
Pachet	Java.lang.Comparable	Java.util.Comparator

5. Interfața Set

Un Set (mulțime) este o colecție ce nu poate conține elemente duplicate. Interfața Set conține doar metodele moștenite din *Collection*, la care adaugă restricții astfel încât elementele duplicate să nu poată fi adăugate. Avem trei implementări utile pentru **Set**:

- HashSet: memorează elementele sale într-o **tabelă de dispersie** (hash table); este implementarea cea mai performantă, însă nu avem garanții asupra **ordinii** de parcurgere. Doi iteratori **diferiți** pot parcurge elementele mulțimii în ordine **diferită**.
- TreeSet: memorează elementele sale sub formă de arbore roșu-negru; elementele sunt ordonate pe baza valorilor sale. Implementarea este mai **lentă** decât HashSet.
- LinkedHashSet: este implementat ca o **tabelă de dispersie**. Diferența față de **HashSet** este că **LinkedHashSet** menține o listă dublu-înlănțuită peste toate elementele sale. Prin urmare (și spre deosebire de **HashSet**), elementele rămân în **ordinea** în care au fost inserate. O parcurgere a **LinkedHashSet** va găsi elementele mereu în această ordine.

Atenție: Implementarea HashSet, care se bazează pe o **tabelă de dispersie**, calculează codul de dispersie al elementelor pe baza metodei hashCode, definită în clasa Object. De aceea, două obiecte **egale**, conform funcției equals, trebuie să întoarcă **același** rezultat din hashCode.
Explicații suplimentare găsiți pe Java Tutorials - Set.

6. Interfața Map

Un Map este un obiect care mapează **chei** pe **valori**. Într-o astfel de structură **nu** pot exista chei duplicate. Fiecare cheie este mapată la exact o valoare. **Map** reprezintă o modelare a conceptului de funcție: primește o entitate ca parametru (cheia), și întoarce o altă entitate (valoarea). Cele trei implementări pentru **Map** sunt:

- HashMap
- TreeMap
- LinkedHashMap

Particularitățile de implementare corespund celor de la **Set**. Exemplu de folosire:

```
class Student {
   String name;
   float avg;

   public Student(String name, float avg) {
      this.name = name;
      this.avg  = avg;
   }

   public String toString() {
      return "[" + name + ", " + avg + "]";
   }
}

public class Test {
   public static void main(String[] args) {

      Map<String,Student> students = new HashMap<String, Student>();

      students.put("Matei",  new Student("Matei",  4.90F));
      students.put("Andrei", new Student("Andrei", 6.80F));
      students.put("Mihai",  new Student("Mihai",  9.90F));

      System.out.println(students.get("Mihai"));

      // adaugăm un element cu aceeași cheie
      System.out.println(students.put("Andrei", new Student("", 0.0F)));
      // put(...) întoarce elementul vechi

      // si îl suprascrie
      System.out.println(students.get("Andrei"));

      // remove(...) returnează elementul șters
      System.out.println(students.remove("Matei"));

      // afișăm structura de date
      System.out.println(students);
   }
}
```

Interfața Map.Entry desemnează o pereche (cheie, valoare) din map. Metodele caracteristice sunt:

- **getKey**: întoarce cheia
- **getValue**: întoarce valoarea
- **setValue**: permite stabilirea valorii asociată cu această cheie

O **iterare** obișnuită pe un map se va face în felul următor:

```
for (Map.Entry<String, Student> entry : students.entrySet())
   System.out.println(entry.getKey() + " has the following average grade: " + entry.getValue().getAverage());
```

În bucla for-each de mai sus se ascunde, de fapt, iteratorul mulțimii de perechi, întoarse de *entrySet*. Explicații suplimentare găsiți pe Java Tutorials - Map.

7. Alte interfețe

Queue definește operații specifice pentru **cozi**:
- inserția unui element
- ștergerea unui element
- operații de „inspecție" a cozii

Implementări utilizate frecvente pentru **Queue**:
- **LinkedList**: pe lângă **List**, **LinkedList** implementează și **Queue**
- **PriorityQueue**;

Explicații suplimentare găsiți pe Java Tutorials - Queue

Concluzii

- Pachetul java.util oferă implementări ale unor structuri de date și algoritmi pentru manipularea lor: ierarhiile Collection și Map și clasa cu metode statice Collections.
- **Parcurgerea** colecțiilor se face în două moduri:
  - folosind iteratori (obiecte ce permit traversarea unei colecții și modificarea acesteia)
  - folosind construcția speciala **for each** (care nu permite modificarea colecției în timpul parcurgerii sale)
- Interfața **List** - colecție ordonată ce **poate** conține elemente **duplicate**.
- Interfața **Set** - colecție ce **nu poate** conține elemente **duplicate**. Există trei implementări utile pentru Set: HashSet (neordonat, nesortat), TreeSet (set sortat) și LinkedHashSet (set ordonat)
- Interfața **Map** - colecție care mapează **chei** pe **valori**. Într-o astfel de structură nu pot exista chei duplicate. Cele trei implementări pentru Map sunt HashMap (neordonat, nesortat), TreeMap (map sortat) și LinkedHashMap (map ordonat)
- **Contractul equals - hashcode**: dacă *obj1 equals obj2 atunci hashcode obj1 == hascode obj2*. Dacă implementați *equals* implementați și *hashcode* daca doriți să folosiți acele obiecte în colecții bazate pe hashuri (e.g. HashMap, HashSet).

Linkuri utile

- Streams, introduse din Java 8, pot fi folosite și pentru a aplica operații pe colecții. Nu le folosim momentan la laborator însă le puteți utiliza la teme:
  - Java streams
  - Filter streams examples

ÎNTREBĂRI DE CONTROL

1. Ce este Collections Framework?
2. Descrieți structura Collections Framework.

3. Care implementări oferă collecțiile?
4. Ce reprezintă interfața Collection?
5. Enumerați metode de parcurgere a colecțiilor.
6. Ce reprezintă un iterator?
7. Descrieți construcția for-each.
8. Ce presupune noțiunea de genericitate?
9. Descrieți interfața List.
10. Descrieți implementările standard ale interfeței List.
11. Descrieți interfețele Comparable și Comparator.
12. Numiți și descrieți metodele de comparare implementate în interfețele Comparable și Comparator.
13. Numiți și descrieți metodele de sortare implementate în interfețele Comparable și Comparator.
14. Ce reprezintă interfața Set?
15. Enumerați și descrieți implementările utile pentru interfața Set.
16. Ce reprezintă interfața Map?
17. Enumerați și descrieți implementările interfeței Map.
18. Enumerați și descrieți metodele caracteristice interfeței Map.

EXERCIȚII

1. (**2p**) În scheletul de laborator, aveți un fișier cu o clasă (Student), care are trei membri: name (String), surname (String), id (long) și averageGrade (double) - media unui student.
   - Clasa Student va implementa interfața Comparable<Student>, folosită la sortări, implementând metoda compareTo. În metoda compareTo, studenții vor fi comparați mai întâi după medie, apoi după numele de familie, apoi după prenume (adică dacă doi studenți au aceeași medie, ei vor fi comparați după numele de familie și dacă au același nume de familie, atunci vor fi comparați după prenume). Recomandăm să suprascrieți metoda *toString*, pentru a putea afișa datele despre un student.
2. (**1p**) Creați 5 obiecte de tip Student și adăugați-le într-un ArrayList, pe care să îl sortați (hint: Collections.sort), apoi afișați conținutul din ArrayList.
3. (**2p**) Adăugați ArrayList-ul definit la subpunctul anterior într-un PriorityQueue (hint: Collection.addAll), care folosește un Comparator, unde elementele sunt sortate crescător după id (aici puteți folosi Long.compare ca să comparați două numere de tip long).
4. (**1p**) Suprascrieți metodele *equals* și *hashCode* în clasa Student (hint: puteți folosi generatorul de cod din IntelliJ).
5. (**2p**) Folosiți un *HashMap<Student, LinkedList<String»*, în care se vor adăuga perechi de tipul (Student, lista de materii pe care le are studentul respectiv), iar apoi afișați conținutul colecției (hint: Map.Entry și entrySet()).
6. (**2p**) Extindeți clasa *LinkedHashSet<Integer>*, cu o clasă în care se vor putea adăuga doar numere pare. Vor fi suprascrise metodele *add* și *addAll*, în așa fel încât să nu fie permise adăugarea de numere impare în colecție. Pentru testare, adăugați numere pare și impare, iar după aceea iterați prin colecție, folosind Iterator (tipizat cu Integer) sau folosind forEach, afișând elementele din colecție. Inlocuiți LinkedHashSet cu HashSet - ce observați cu privire la ordinea de inserare a elementelor? Dar dacă ați înlocui cu TreeSet?

EXERCIȚII SUPLIMENTARE

1. Instanţiati o colecţie care să **nu** permită introducerea elementelor duplicate, folosind o implementare corespunzătoare din bibliotecă. La introducerea unui element existent, semnalaţi eroare. Colecţia va reţine String-uri şi va fi parametrizată.
2. Creaţi o clasă Student.
   a. Adăugaţi următorii membri:

- **câmpurile** nume (de tip String) şi medie (de tip float)
- un **constructor** care îi iniţializează
- metoda toString.

b. Folosiţi codul de la **exerciţiul anterior** şi modificaţi-l astfel încât colecţia aleasă de voi să reţină obiecte de tip Student. Testaţi prin adăugare de elemente duplicate, având aceleaşi valori pentru toate câmpurile, instanţiindu-le, de fiecare dată, cu new. Ce observaţi?

c. Prelucraţi implementarea de mai sus astfel încât colecţia să reprezinte o tabelă de dispersie, care calculează codul de dispersie al elementelor după un criteriu ales de voi (puteţi suprascrie funcţia **hashCode**).
   - În Student suprascrieți metoda equals astfel încât să se ţină cont de câmpurile clasei, şi încercaţi din nou. Ce observaţi?

- *Hint:* Set.add, Object.equals, Object.hashCode

3. Plecând de la implementarea exerciţiului anterior, realizaţi următoarele modificări:
   - Supraîncărcaţi, în clasa Student, metoda equals, cu o variantă care primeşte un parametru Student, şi care întoarce întotdeauna false.
   - Testaţi comportamentul prin crearea unei colecţii ce conţine instanţe de Student şi iteraţi prin această colecţie, afişând la fiecare pas element.equals(element) şi ((Object)element).equals(element) (unde element este numele de variabilă ales pentru fiecare element al colecţiei). Cum explicaţi comportamentul observat? Dacă folosiţi un iterator, acesta va fi şi el **parametrizat**.
4. Creați clasa Gradebook, de tip Map, pentru reţinerea studenţilor după medie: cheile sunt mediile și valorile sunt liste de studenți. Gradebook va menţine cheile **ordonate descrescător**. Extindeţi o implementare potrivită a interfeţei Map, care să permită acest lucru.
   . Caracteristicile clasei definite sunt:
   I. Cheile pot avea valori de la 0 la 10 (corespunzătoare mediilor posibile). Verificați acest lucru la adăugare.
   II. Valoarea asociată fiecărei chei va fi o listă (List) care va reţine toţi studenţii cu media rotunjită egală cu cheia. Considerăm că un student are media rotunjită 8 dacă media sa este în intervalul [7.50, 8.49].

   a. Implementați un Comparator pentru stabilirea ordinii cheilor. Gradebook va primi un parametru de tip Comparator în constructor și îl va da mai departe constructorului clasei moștenite.

   b. Definiţi în clasă metoda add(Student), ce va adăuga un student în lista corespunzătoare mediei lui. Dacă, în prealabil, nu mai există niciun student cu media respectivă (rotunjită), atunci lista va fi creată la cerere.

   c. Testați clasa:
      . instanțiați un obiect Gradebook și adăugați in el câţiva studenţi.
      I. iteraţi pe Gradebook şi sortaţi alfabetic fiecare listă de studenţi pentru fiecare notă. Pentru a sorta, se va folosi metoda Collections.sort, iar clasa Student va implementa o interfață care specifică modul în care sunt comparate elementele.
         - clasa Student va implementa interfaţa Comparable, suprascriind metoda compareTo.
5. Creaţi o clasă care moşteneşte HashSet<Integer>.
   - Definiţi în această clasă o variabilă membru care reţine numărul total de elemente adăugate. Pentru a contoriza acest lucru, suprascrieți metodele add și addAll. Pentru adăugarea efectivă a elementelor, folosiţi implementările din clasa părinte (HashSet).
   - Testaţi, folosind atât add cât şi addAll. Ce observaţi? Corectaţi dacă este cazul.
   - Modificaţi implementarea astfel încât clasa voastră să moştenească LinkedList<Integer>. Ce observaţi? Ce **concluzii** trageţi?
   - Hint: Collection.add, Collection.addAll.
   - Hint: implementarea addAll din sursele pentru HashSet şi LinkedList.