

Laboratorul 8: Colecții

Video introductiv: link [https://youtu.be/NrcJpB_XLp0]

Obiective

Pe parcursul laboratoarelor și temelor ați folosit structuri de date oferite de API-ul Java. În cadrul acestui laborator le vom aprofunda.

- lucrul cu cele trei tipuri principale de colecții din Java: List, Set, Map
- cunoașterea diferențelor dintre implementările colecțiilor (eficiență, sortare, ordonare etc)
- compararea elementelor unor colecții
- contractul equals-hashcode

Collections Framework

În pachetul **java.util** (pachet standard din JRE) există o serie de clase pe care le veți găsi folositoare. Collections Framework [<http://docs.oracle.com/javase/tutorial/collections/index.html>] 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 [<https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html>], pe care o implementează majoritatea claselor ce desemnează colecții din **java.util**. Explicații suplimentare găsiți pe Java Tutorials - Collection [<http://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html>]

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

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

Parcurea colecțiilor

Colecțiile pot fi parcurse (element cu element) folosind:

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

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 [<https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html>] 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()` [[http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html#remove\(\)](http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html#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`.

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[]`.

Genericitate

Fie următoarea porțiune de cod:

```
Collection c = new ArrayList();
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 de [Genericitate](#)

Interfața List

O listă este o colecție care poate fi **ordonată**. Listele **pot** conține elemente **duplicate**. Pe lângă operațiile moștenite de la `Collection`, interfața `List` [<https://docs.oracle.com/javase/8/docs/api/java/util/List.html>] 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`

[<http://docs.oracle.com/javase/7/docs/api/java/util/ListIterator.html>], 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 algoritmi implementați se numără:

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

În general, algoritmi pe colecții sunt implementați ca metode statice în clasa `Collections` [<http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html>].

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](http://docs.oracle.com/javase/tutorial/collections/algorithms/index.html) [<http://docs.oracle.com/javase/tutorial/collections/algorithms/index.html>]

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` [<https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>]
- `Comparator` [<https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html>]

	Comparable	Comparator
Logica de sortare	Logica de sortare trebuie să fie în clasa ale cărei obiecte sunt sortate. Din acest motiv, această metodă se numește <i>sortare naturală</i> .	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 <code>compareTo()</code> .	Clasa ale cărei instanțe se doresc a fi sortate nu trebuie să implementeze această interfață. Este nevoie de o altă clasă (poate fi și internă) care să implementeze interfața <code>Comparator</code> .
Metoda de comparare	<pre>int compareTo(Object o1)</pre> <p>Această metodă compară obiectul curent (<code>this</code>) cu obiectul <code>o1</code> și întoarce un întreg. Valoarea întoarsă este interpretată astfel:</p> <ol style="list-style-type: none"> 1. pozitiv – obiectul este mai mare decât <code>o1</code> 2. zero – obiectul este egal cu <code>o1</code> 3. negativ – obiectul este mai mic decât <code>o1</code> 	<pre>int compare(Object o1, Object o2)</pre> <p>Această metodă compară obiectele <code>o1</code> and <code>o2</code> și întoarce un întreg. Valoarea întoarsă este interpretată astfel:</p> <ol style="list-style-type: none"> 1. pozitiv – <code>o2</code> este mai mare decât <code>o1</code> 2. zero – <code>o2</code> este egal cu <code>o1</code> 3. negativ – <code>o2</code> este mai mic decât <code>o1</code>
Metoda de sortare	<pre>Collections.sort(List)</pre> <p>Aici obiectele sunt sortate pe baza metodei <code>compareTo()</code>.</p>	<pre>Collections.sort(List, Comparator)</pre> <p>Aici obiectele sunt sortate pe baza metodei <code>compare()</code> din <code>Comparator</code>.</p>
Pachet	<code>Java.lang.Comparable</code>	<code>Java.util.Comparator</code>

Exemplu `Comparable`:

- implementare:

```
public class Student implements Comparable<Student> {
    private String name;
    private String surname;

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    @Override
    public int compareTo(Student o) {
        if (surname.equals(o.surname)) {
            return name.compareTo(o.name);
        } else {
            return surname.compareTo(o.surname);
        }
    }
}
```

- folosire:

```
ArrayList<Student> students = new ArrayList<>();
// populate ArrayList with Student objects
Collections.sort(students);
```

Exemplu implementare și folosire Comparator:

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(5);
numbers.add(1);
numbers.add(3623);
numbers.add(13);
numbers.add(7);
Collections.sort(numbers, new Comparator<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2 - o1;
    }
});
System.out.println(numbers); // se afișează [3623, 13, 7, 5, 1]

// alternativ, putem sorta o colecție, folosind metoda sort() din interfața List, în acest mod:
numbers.sort(new Comparator<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2 - o1;
    }
});
```

Interfața Set

Un Set [<http://docs.oracle.com/javase/7/docs/api/java/util/Set.html>] (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 [<http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html>]: 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 [<http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html>]: memorează elementele sale sub formă de arbore roșu-negru [http://en.wikipedia.org/wiki/Red-black_tree]; elementele sunt ordonate pe baza valorilor sale. Implementarea este mai **lentă** decât HashSet.
- LinkedHashSet [<http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashSet.html>]: este implementat ca o **tabelă de dispersie**. Diferența față de HashSet este că LinkedHashSet menține o listă dublu-înălțuită peste toate elementele sale. Prin urmare (și spre deosebire de HashSet), elementele rămân în **ordine** 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 [[http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode\(\)](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode())], definită în clasa Object. De aceea, două obiecte **egale**, conform funcției equals, trebuie să întoarcă **același** rezultat din hashCode.

	HashSet	LinkedHashSet	TreeSet
Funcționarea internă	Elementele se memorează într-o tabelă de dispersie	Elementele sunt păstrate cu ajutorul unei liste înălțuite	Elementele se memorează într-un arbore de căutare
Utilizarea	Se folosește când dorești să stochezi o listă de perechi cheie-valoare fără a fi interesat de ordinea acestei memorări	Se folosește atunci când se dorește conservarea ordinii de la inserare	Se folosește când se dorește păstrarea elementelor într-o ordine stabilită cu ajutorul unui Comparator

Ordinea	Ordinea elementelor este total aleatoare	Se conservă ordinea în care au fost introduse elementele	Se folosește ordinea stabilită cu ajutorul unui Comparator. Dacă acesta nu este menționat, implicit elementele vor fi sortate crescător
Complexitatea operațiilor	$O(1)$ pentru toate operațiile de bază (inserare, ștergere, căutare)	$O(1)$ pentru toate operațiile de bază (inserare, ștergere, căutare)	Deoarece este folosit un arbore în spate, operațiile se execută în $O(\log(N))$
Performanța	Cel mai performant dintre cele 3 menționate	Performanța se află între cea a unui HashSet și a unui TreeSet deoarece în ciuda faptului că are complexitate $O(1)$ la operațiile principale, folosește intern și liste înlănțuite pentru păstrarea ordinii de la inserare	Din cauza faptului că după fiecare operație de adăugare și ștergere trebuie să conserve ordinea elementelor, are cea mai proastă performanță dintre cele 3 menționate
Compararea	Folosește <i>equals()</i> și <i>hashCode()</i> pentru a compara obiectele	Folosește <i>equals()</i> și <i>hashCode()</i> pentru a compara obiectele	Folosește <i>compare()</i> și <i>compareTo()</i> pentru a compara obiectele

Explicații suplimentare găsiți pe Java Tutorials - Set

[<http://docs.oracle.com/javase/tutorial/collections/interfaces/set.html>].

Interfața Map

Un Map [<http://docs.oracle.com/javase/7/docs/api/java/util/Map.html>] 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). Trei implementări pentru Map sunt:

- HashMap [<http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html>]
- TreeMap [<http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html>]
- LinkedHashMap [<http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html>]

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` [<http://docs.oracle.com/javase/7/docs/api/java/util/Map.Entry.html>] 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 [<http://docs.oracle.com/javase/tutorial/collections/interfaces/map.html>].

Alte interfețe

`Queue` [<http://docs.oracle.com/javase/8/docs/api/java/util/Queue.html>] este o interfață ce definește operații specifice pentru **cozi**:

- inserția unui element
- ștergerea unui element
- operații de "inspecție" a cozii

Implementări utilizate frecvente pentru `Queue`:

- `PriorityQueue`: coadă cu priorități / heap

`Deque` [<http://docs.oracle.com/javase/8/docs/api/java/util/Queue.html>] este o interfață, care extinde interfața `Queue`, ce definește operații specifice pentru **cozi cu două capete**, unul la început și celălalt la final. Având operații pentru ambele capete, rezultă faptul că o colecție de tip `Deque` poate fi folosită atât ca **stivă**, cât și drept **coadă**.

Operații specifice:

- inserția unui element
- ștergerea unui element
- operații de "inspecție" a cozii / a stivei

Implementări utilizate frecvente pentru `Deque`:

- `LinkedList`: pe lângă `List`, `LinkedList` implementează și `Deque` (deci și `Queue`)
- `ArrayDeque`: este mai rapidă decât `LinkedList`, în caz ca este folosită drept coadă

Explicații suplimentare găsiți pe Java Tutorials - Queue

[<http://docs.oracle.com/javase/tutorial/collections/interfaces/queue.html>], `Deque`

[<https://docs.oracle.com/javase/tutorial/collections/interfaces/deque.html>]

În Java, există colecții care sunt marcate ca fiind *obsolete*, adică nu mai sunt recomandate să fie folosite. Exemple de astfel de colecții:

- **Vector** - operațiile prin care colecția este modificată (adăugare, ștergere) sunt sincronizate (detalii legate de sincronizări veți studia la APD [<https://ocw.cs.pub.ro/courses/apd/laboratoare/06>], în anul 3), în timp ce operațiile la **ArrayList** (care este recomandat în locul lui **Vector**) nu sunt sincronizate, permițând astfel programatorului să aibă mai mult control asupra operațiilor în cod
- **Hashtable** - operațiile prin care colecția este modificată (adăugare, ștergere) sunt sincronizate, în timp ce aceste operații la **HashMap** (care este recomandat în locul lui **Hashtable**) nu sunt sincronizate
- **Stack** - acesta reprezintă implementarea de operații specifice pentru stivă și extinde clasa **Vector**, despre care am vorbit anterior. Colecția recomandată în locul acesteia este **ArrayDeque**.

Funcții lambda

În cadrul laboratorului de clase interne, am vorbit despre funcții anonime (funcții lambda) și despre cum le putem folosi în Java.

Putem folosi funcții anonime pentru a executa diverse operații pe liste (de exemplu **removeIf**, care filtrează elementele unei colecții pe baza unui predicat, și **replaceAll**, care aplică o operație pe toate elementele unei colecții).

Exemple:

```
List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));

// incrementează toate numerele din colecție cu 1
list.replaceAll((x) -> x + 1);

// șterge din colecție numerele impare
list.removeIf((x) -> x % 2 == 1);
```

O altă utilitate a funcțiilor anonime reprezintă în implementarea comparatorilor folosiți la sortare sau la crearea de colecții sortate (**TreeSet**, **TreeMap**).

Exemple:

```
// o variantă
Collections.sort(list, (o1, o2) -> o2 - o1);

// alta variantă, prin care se folosim de metoda sort() din interfața List
list.sort((o1, o2) -> o2 - o1);

// colecții sortate
TreeSet<Integer> sortedSet = new TreeSet<>((o1, o2) -> o1 - o2);
TreeMap<Integer, Integer> sortedMap = new TreeMap<>((o1, o2) -> o1 - o2);
```

Unit Testing

În procesul de dezvoltare software, o parte foarte importantă a acestuia este și verificarea dacă codul scris se comportă în modul așteptat sau nu. Tot ce presupune verificarea funcționalității codului se poate încadra sub umbrela termenului de "Testing", dintre care există mai multe tipuri (unit testing, functional testing, integration testing, printre altele).

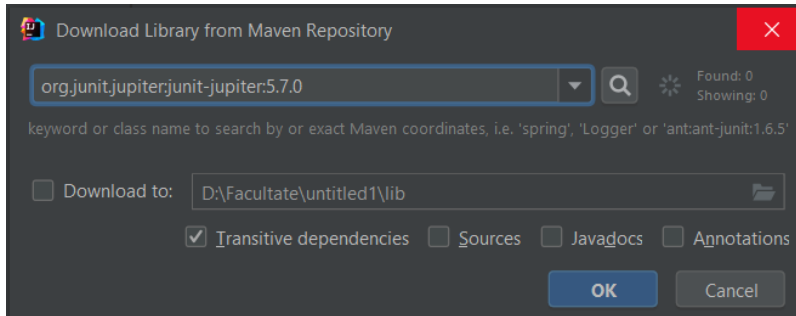
Elementul de bază în testare îl constituie "unit testing-ul". Acesta presupune scrierea de teste care verifică funcționalitatea unei singure componente: o clasă cu anumite metode, o structură de date, etc.

Framework-ul de testare despre care vom vorbi este JUnit5. Pentru o analiză mai amănunțită a acestuia, consultați următoarea pagină: [JUnit5 Basics](#). Scopul prezentării în acest laborator este de a vă prezenta un punct de start în învățarea acestui framework.

Instalare JUnit5

Pentru a folosi biblioteca JUnit într-un anumit proiect, urmează următorii pași:

1. Într-un proiect, intrați pe File → Project Structure → Libraries
2. din meniul de Libraries, apăsați pe "+", după care pe "From Maven"
3. se va deschide un pop-up, în care va trebui să scrieți în fereastra de căutare "org.junit.jupiter:junit-jupiter:5.7.0", după care apăsați "Ok"
4. IntelliJ vă va întreba dacă vreți să adăugați biblioteca la proiectul curent, apăsați "Ok"
5. la final, în fereastra inițială apăsați "Apply" → "Ok"



Pentru a putea testa funcționalitatea bibliotecii "JUnit" puteți folosi codul exemplu de aici (<https://ocw.cs.pub.ro/courses/poo-ca-cd/alte-resurse/junit-java> [https://ocw.cs.pub.ro/courses/poo-ca-cd/alte-resurse/junit-java]). În cazul în care aveți erori, puteți să încercați să reactualizați cache-ul aplicației (File → Invalidate Caches), timp în care JAR-urile și SDK-ul Java vi se vor reindexa (progresul operației se poate observa în colțul din dreapta jos sub formă de "loading bar").

Utilizarea framework-ului

Pentru a folosi JUnit5, este important de învățat două concepte: cel de adnotări și cel de assert-uri. Cel mai mare avantaj al JUnit este viteza rapidă de scriere a testelor, iar lucrul acesta este posibil datorită adnotărilor. Adnotările sunt termeni standardizați, prefațați de semnul "@", plasați fix înaintea semnăturii unei funcții. Scopul acestora este că, în momentul compilării, compilatorul să știe să adauge funcționalitate în plus metodei căreia a fost adăugat. Printre adnotările de bază din JUnit5 se numără:

- **@Test** - metoda va funcționa ca test, aceasta va trebui să întoarcă o valoare de True dacă funcționalitatea testată funcționează în modul dorit, False în mod contrar (lucru posibil prin assert-uri, despre care vom vorbi mai jos)
- **@BeforeEach** - metoda va fi executată înaintea fiecărui test
- **@AfterEach** - metoda va fi executată după fiecare test
- **@DisplayName("Some_String")** - se folosește împreună cu @Test; când testul se rulează, la consolă va apărea la output cu numele "Some_String"

Pentru a vedea toate adnotările disponibile în JUnit5, consultați următoarea pagină de documentație <https://www.swtestacademy.com/junit-5-annotations/> [https://www.swtestacademy.com/junit-5-annotations/].

Așa cum am menționat mai sus, cel de-al doilea concept necesar scrierii testelor este cel de assert-uri. Acestea metode statice, care sunt găsite în clasa org.junit.jupiter.api.Assertions, afirmă valoarea de adevăr a diferite expresii. În continuare, vă vom prezenta câteva exemple de assert-uri des folosite:

- **assertEquals(value_1, value_2)** - verifică dacă cele două valori sunt egale
- **assertTrue(boolean value)**, respectiv **assertFalse(boolean value)** - verifică dacă value este True, respectiv False
- **assertNull(obj)**, respectiv **assertNotNull(obj)** - verifică dacă obj este Null, respectiv dacă nu este

Pentru mai multe detalii despre toate funcțiile de assert existente, consultați următoarea pagină de documentație: <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

[<http://junit.sourceforge.net/javadoc/org/junit/Assert.html>]

Pentru a scrie teste corect și cu bună vizibilitate, aveți în minte următoarele lucruri:

- puneți nume sugestive pentru fiecare metodă de test pe care o implementați
- scrieți metode de test care să testeze doar o anumită funcționalitate, nu mai multe deodată
- țineți minte că testele sunt făcute să verifice că implementările făcute de voi funcționează în modul dorit de voi

Exemplu de folosire JUnit5

FloatCalculator class

```
package main;

public class FloatCalculator {

    public float add(float first, float second) {
        return first + second;
    }

    public float multiply(float first, float second) {
        return first * second;
    }

    public float divide(float first, float second) {
        return first / second;
    }

    public boolean isNegative(float num) {
        return num < 0;
    }
}
```

FloatCalculatorTest class

```
package main;

import org.junit.jupiter.api.*;

public class FloatCalculatorTest {
    private FloatCalculator calculator;

    @BeforeEach
    public void setUp() {
        this.calculator = new FloatCalculator();
    }

    @AfterEach
    public void clean() {
        this.calculator = null;
    }

    @Test
    @DisplayName("Add test")
    public void testAdd() {
        Assertions.assertEquals(5, calculator.add(2, 3));
        Assertions.assertNotEquals(5, calculator.add(2, 2));
    }

    @Test
    @DisplayName("Multiply test")
    public void testMultiply() {
        Assertions.assertEquals(6, calculator.multiply(2, 3));
        Assertions.assertNotEquals(7.5f, calculator.multiply(2.5f, 4));
    }

    @Test
    @DisplayName("Divide test")
    public void testDivide() {

```

```

        Assertions.assertEquals(10, calculator.divide(100, 10));
        Assertions.assertNotEquals(5.5f, calculator.divide(55, 12));
    }

    @Test
    @DisplayName("IsNegative test")
    public void testIsNegative() {
        Assertions.assertTrue(calculator.isNegative(-5));
        Assertions.assertFalse(calculator.isNegative(10));
    }
}

```

TL;DR

- Pachetul `java.util` [<https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html>] oferă implementări ale unor structuri de date și algoritmi pentru manipularea lor: ierarhiile `Collection` [<https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html>] și `Map` [<https://docs.oracle.com/javase/8/docs/api/java/util/Map.html>] și clasa cu metode statice `Collections` [<https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>].
- **Parcurgerea** colecțiilor se face în două moduri:
 - folosind iteratori (obiecte ce permit traversarea unei colecții și modificarea acestora)
 - folosind construcția specială **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` [<http://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html>] (neordonat, nesortat), `TreeSet` [<http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html>] (set sortat) și `LinkedHashSet` [<http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html>] (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` [<http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>] (neordonat, nesortat), `TreeMap` [<http://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html>] (map sortat) și `LinkedHashMap` [<http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html>] (map ordonat)
- **Contractul equals - hashCode**: dacă *obj1 equals obj2 atunci hashCode obj1 == hashCode obj2*. Dacă implementați `equals`, implementați și `hashCode` dacă doriți să folosiți acele obiecte în colecții bazate pe hash-uri (e.g. `HashMap`, `HashSet`).

Exerciții

1. În cadrul acestui exercițiu, veți implementa o clasă numită `Student`, care are patru membri:

1. `name` (`String`)
2. `surname` (`String`)
3. `id` (`long`)
4. `averageGrade` (`double`) - media unui student.

Clasa `Student` va implementa interfața `Comparable<Student>`, folosită la sortări, prin implementarea metodei `compareTo`. În metoda `compareTo`, studenții vor fi comparați mai întâi după medie, apoi după numele de familie, apoi după prenume. După implementarea clasei, sortați elementele listei "students" din metoda `main` folosind metoda `Collections.sort()`.

2. Sortați lista "copyStudents" din metoda `main` folosind, de această dată, o expresie lambda.

3. Adăugați lista "anotherCopyStudents" într-un `PriorityQueue` (cu ajutorul metodei `Collection.addAll`), care folosește un `Comparator` (utilizați constructorul `PriorityQueue`) sau o funcție anonimă. Elementele vor fi

sortate crescător după id.

4. Suprascrieți metodele equals și hashCode în clasa Student (puteți folosi generatorul de cod din IntelliJ). După aceasta, adăugați în lista asociată studenților din "studentMap" patru materii aleatorii. Pentru a obține materiile aleatorii, urmăriți indicațiile din codul din funcția main.

5. Extindeți clasa LinkedHashSet<Integer> cu o clasă în care se vor putea adăuga doar numere pare. Metoda add va fi suprascrisă astfel încât să nu permită adăugarea de numere impare în colecție. Efectuați aceeași operațiune și pentru clasele TreeSet și HashSet. Observați diferențele privind ordinea de inserare a elementelor între cele trei clase menționate.

Scheletele il puteți găsi pe github [<https://github.com/oop-pub/oop-labs/blob/master/src/lab8/Main.java>]. Soluția trebuie încărcată pe LambdaChecker [<https://beta.lambdachecker.io>].

Resurse

- [Exerciții din alți ani](#)

Linkuri utile

- Streams [<https://www.baeldung.com/java-8-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 [<https://www.geeksforgeeks.org/stream-in-java/>]
 - Filter streams examples [<https://www.geeksforgeeks.org/stream-filter-java-examples/>]

poo-ca-cd/laboratoare/colectii.txt · Last modified: 2023/11/30 19:33 by albert.daraban