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 veti găsi folositoare. Collections Framework [http://docs.oracle.com/javase/tutorial/collections/index.html] este o arhitectură unificată pentru reprezentarea şi manipularea colecţiilor. Ea conţine:

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

Colecţiile oferă implementări pentru următoarele tipuri:

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

Există o interfaţă, numită Collection [https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html], pe care o implementează majoritatea claselor ce desemnează colecţii din **java.util**. Explicaţii suplimentare găsiţi pe Java Tutorials - Collection [http://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html]

Exemplul de mai jos construieşte o listă populată cu nume de studenţi:

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

Parcurgerea colecţiilor

Colecţiile pot fi parcurse (element cu element) folosind:

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

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, apelând metoda sa **iterator()**. Interfaţa 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()] a unui iterator face posibilă eliminarea elementului din colecţie care a fost întors la ultimul apel al metodei **next()** din acelaşi iterator. În exemplul anterior, toate elementele din colecţie mai mici decât 5 for fi şterse la ieşirea din bucla `while`.

For-each

Această construcţie permite (într-o manieră expeditivă) traversarea unei colecţii. **for-each** este foarte similar cu `for`. Următorul exemplu parcurge elementele unei colecţii şi le afişează.

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

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

Genericitate

Fie următoarea porţiune 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 colecţie `c`, de tipul `ArrayList` (pe care îl vom examina într-o secţiune următoare). Apoi, am adăugat în colecţie un element de tipul `String`. Am realizat o parcurgere folosind un iterator, şi am încercat obţinerea elementului nostru folosind apelul: `String s = it.next();`. Funcţia `next` însă întoarce un obiect de tip `Object`. Prin urmare apelul va eşua. Varianta corectă este `String s = (String)it.next();`. Am fi putut preciza, din start, ce tipuri de date dorim într-o colecţie:

```
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 in laboratorul următor: Genericitate

Interfaţa List

O listă este o colecţie care poate fi **ordonată**. Listele **pot** conţine elemente **duplicate**. Pe lângă operaţiile moştenite de la `Collection`, interfaţa List [https://docs.oracle.com/javase/8/docs/api/java/util/List.html] conţine operaţii bazate pe poziţie (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 interfaţa ListIterator [http://docs.oracle.com/javase/7/docs/api/java/util/ListIterator.html], ce extinde interfaţa `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 înlănţuită. Prin urmare, accesul la un element nu se face în timp constant, fiind necesară o parcurgere a listei: `O(n)`.

Printre algoritmii implementaţi se numără:

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

În general, algoritmii pe colecţii sunt implementaţi ca metode statice în clasa Collections [http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html].

Atenţie: Nu confundaţi interfaţa `Collection` cu clasa `Collections`. Spre deosebire de prima, a doua este o clasă ce conţine exclusiv metode statice. Aici sunt implementate diverse operaţii asupra colecţiilor.

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 colecţii găsiţi pe Java Tutorials - Algoritmi pe liste [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 crescator. 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 conţine printre membrii săi o variabilă ce reţine media anilor. Acest lucru poate fi realizat folosind interfeţele:

- 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 numeşte *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 instanţe se doresc a fi sortate trebuie să **implementeze această interfaţă** şi, evident, să suprascrie metoda `compareTo()`.	Clasa ale cărei instanţe se doresc a fi sortate nu trebuie să implementeze această interfaţă. Este nevoie de o alta clasă (poate fi şi internă) care să implementeze interfaţa `Comparator`.
Metoda de comparare	`int compareTo(Object o1)` Această metodă compară obiectul curent (this) cu obiectul o1 şi 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 şi 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

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;
      }
});
```

Interfaţa Set

Un Set [http://docs.oracle.com/javase/7/docs/api/java/util/Set.html] (mulţime) este o colecţie ce nu poate conţine elemente duplicate. Interfaţa `Set` conţine doar metodele moştenite din `Collection`, la care adaugă restricţii 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 garanţii asupra **ordinii** de parcurgere. Doi iteratori **diferiţi** pot parcurge elementele mulţimii în ordine **diferită**.
- TreeSet [http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html]: memorează elementele sale sub formă de arbore roşu-negru [http://en.wikipedia.org/wiki/Red-black_tree]; elementele sunt ordonate pe baza valorilor sale. Implementarea este mai **lentă** decat `HashSet`.

- LinkedHashSet [http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashSet.html]: este implementat ca o **tabelă de dispersie**. Diferenţa faţă de `HashSet` este că `LinkedHashSet` menţine o listă dublu-înlănţuită peste toate elementele sale. Prin urmare (şi spre deosebire de `HashSet`), elementele rămân în **ordinea** în care au fost inserate. O parcurgere a `LinkedHashSet` va găsi elementele mereu în această ordine.

Atenţie: 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()], definită în clasa `Object`. De aceea, două obiecte **egale**, conform funcţiei `equals`, trebuie să întoarcă **acelaşi** rezultat din `hashCode`.

	HashSet	**LinkedHashSet**	**TreeSet**
Funcționarea internă	Elementele se memorează într-o tabelă de dispersie	Elementele sunt păstrate cu ajutorul unei liste înlănțuite	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. Daca 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 in 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 *equals()* și *hashCode()* pentru a compara obiectele	Folosește *equals()* și *hashCode()* pentru a compara obiectele	Folosește *compare()* și *compareTo()* pentru a compara obiectele

Explicaţii suplimentare găsiti pe Java Tutorials - Set [http://docs.oracle.com/javase/tutorial/collections/interfaces/set.html].

Interfaţa 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 funcţie: `primeşte` o entitate ca parametru (cheia), şi 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]

Particularităţile 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);
    }
}
```

Interfaţa 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** obişnuită 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 mulţimii de perechi, întoarse de `entrySet`. Explicaţii suplimentare găsiţi pe Java Tutorials - Map [http://docs.oracle.com/javase/tutorial/collections/interfaces/map.html].

Alte interfeţe

Queue [http://docs.oracle.com/javase/8/docs/api/java/util/Queue.html] este o interfaţă ce defineşte operaţii specifice pentru **cozi**:

- inserţia unui element
- ştergerea unui element
- operaţii de "inspecţie" a cozii

Implementări utilizate frecvente pentru `Queue`:

- `PriorityQueue`: coadă cu priorităţi / heap

Deque [http://docs.oracle.com/javase/8/docs/api/java/util/Queue.html] este o interfaţă, care extinde interfața `Queue`, ce defineşte operaţii 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:

- inserţia unui element
- ştergerea unui element
- operaţii de "inspecţie" a cozii / a stivei

Implementări utilizate frecvente pentru `Deque`:

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

Explicaţii suplimentare găsiţi 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 aceste operatiile 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).

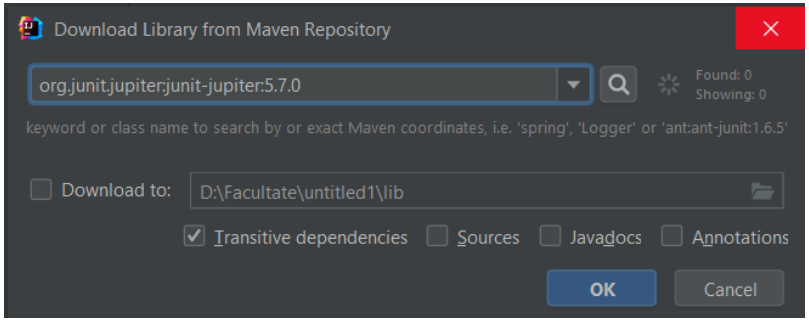
Elementul de bază în testare îl constituie "unit testing-ul". Acesta presupune scrierea de teste care verifică funcționalitatea unei singure componente: o clasă cu anumite metode, o structură de date, etc.

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 reactualizati 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 stucturi 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** colecţiilor se face în două moduri:
    - folosind iteratori (obiecte ce permit traversarea unei colecţii şi modificarea acesteia)

    - folosind construcţia specială **for each** (care nu permite modificarea colecţiei în timpul parcurgerii sale)
- Interfaţa **List** - colecţie ordonată ce **poate** conţine elemente **duplicate**.
- Interfaţa **Set** - colecţie ce **nu poate** conţine 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)
- Interfaţa **Map** - colecţie 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).

Exerciţii

1. (**2p**) În cadrul acestui exercițiu, veți implementa o clasă numită Student, care are patru membri:
    a. name (String)
    b. surname (String)
    c. id (long)
    d. averageGrade (double) - media unui student.
    - Clasa Student va implementa interfața Comparable [https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Comparable.html]<Student>, folosită la sortări, implementând metoda compareTo [https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Comparable.html#compareTo(T)]. În metoda compareTo, studenții vor fi comparați mai întâi după medie, apoi după numele de familie, apoi dupa 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 [https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/Collections.html#sort(java.util.List)]), apoi afisați conținutul din ArrayList.
3. (**1p**) Sortați ArrayList-ul de la punctul anterior cu metoda sort() din interfața List sau cu Collections.sort(), în care să folosiți o funcție lambda, în care se compară descrescător după medie.
4. (**2p**) Adăugați ArrayList-ul definit la subpunctul anterior într-un PriorityQueue (hint: Collection.addAll [https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/Collection.html#addAll(java.util.Collection)]), care folosește un Comparator [https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Comparator.html] (hint: constructor PriorityQueue) [https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/PriorityQueue.html#%3Cinit%3E(java.util.Comparator)] sau o funcție anonimă, unde elementele sunt sortate crescător după id (aici puteti folosi Long.compare ca să comparați două numere de tip long).

5. (**1p**) Suprascrieți metodele *equals* și *hashCode* în clasa Student (hint: puteți folosi generatorul de cod din IntelliJ).
6. (**1p**) Folosiți un *HashMap [https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/HashMap.html]<Student, LinkedList<String»*, în care se vor adăuga perechi de tipul (Student, lista de materii pe care le are studentul respectiv), iar apoi afisați conținutul colecției (hint: Map.Entry și entrySet()).
7. (**2p**) Extindeți clasa *LinkedHashSet<Integer>*, cu o clasă în care se vor putea adăuga doar numere pare. Va fi suprascrisă metoda *add*, 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 [https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/Iterator.html] (tipizat cu Integer) sau folosind forEach, afișând elementele din colecție. Înlocuiți LinkedHashSet cu HashSet - ce observați cu privire la ordinea de inserare a elementelor? Dar dacă ați înlocui cu TreeSet?
8. (**1p**) Rezolvați această problemă de pe LambdaChecker [https://lambdachecker.io/contest/24], care conține teste.
9. (**1p**) Pe baza exemplelor din laborator, implementați o clasă numită StudentTest în care să adăugați metodele aferente realizării unor UnitTeste (folosind JUnit), pentru testarea a cel puțin una dintre urmatoarele metodele din clasa Student:
    a. Equals
    b. CompareTo
    c. toString

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/]