Laborator 3: Interview Lab - PAO

1. Functional Interface, Lambda Function
3. Predicate & BiPredicate, Consumer & BiConsumer, Function & BiFunction & UnaryOperator, Supplier, Optional, Comparator & Comparable Interface
4. Collections, Stream API
5. Binary Search, Binary Tree
6. Threading (basic), Load Factor & Capacity
7. StringBuilder vs String Buffer
8. List Interface, ArrayList vs LinkedList + complexitati
9. Map Interface, HashMap vs Hashtable vs TreeMap vs LinkedHashMap + complexitati, HashMap Collisions (chaining, linear probing)
10. Set Interface, HashSet vs TreeSet vs LinkedHashSet + complexitati
11. Queue, Deque, Vector, Stack

==========

Functional Interface

O interfata functionala este o interfata care foloseste adnotarea `@FunctionalInterface` si are o singura metoda, care nu este implementata.

Exemplu:

```
@FunctionalInterface
public interface TestInteger {
	boolean isInteger(Double number);
}
```

Putem folosi interfata respectiva pentru a implementa o metoda sau implementand interfata in alta clasa, in felul urmator:

1. Implementand Interfata

```
public class TestIntegerImpl implements TestInteger {
```

```
	@Overriding
	public boolean isInteger(Double number) {
		return number == (int) number;
	}
}

public class Main {
	public static void main(String[] args) {
		TestIntegerImpl testInteger = new TestIntegerImpl();
		boolean isInt = testInteger.isInteger(5.0);
	}
}
```

2. Folosind o functie anonima (un lambda function)

```
public class Main {
	public static void main(String[] args) {
		TestInteger testInteger = (number) -> number == (int) number;

		boolean isInt = testInteger.isInteger(5.0);
	}
}
```

Lambda Function Best Practices: "Lambda Expressions and Functional Interfaces: Tips and Best Practices"

Predicate & BiPredicate

Predicate si BiPredicate sunt interfete in Java care ne ajuta sa stabilim niste predicate (evident dupa nume, nu?).

Intefata Predicate ne ofera acces in a pasa doar un argument, iar BiPredicate ne ofera acces in a pasa doua argumente, in final a testa daca "predicatul" nostru este adevarat sau nu. Putem aplica si operatorii logici OR si SAU.

De exemplu:

```
public class Main {

     String sampleText = “Hello Lab PAO 232”;
     Predicate<String> containsPredicate = (text) ->
sampleText.contains(text);

     BiPredicate<String, String> containsBiPredicate = (text,
pattern) -> text.contains(pattern);

     BiPredicate<String, String>
containsBiPredicateReferenceMethod = String::contains;

     public static void main(String[] args) {
          boolean result = containsPredicate.test(“Hello”);

          boolean resultNegate =
containsPredicate.negate().test(“Hello”);

          boolean andResult =
containsBiPredicate.and(containsBiPredicate.negate()).test(“He
llo”, “LAB”);
// Logical AND operator.

          booleanorResult =
containsBiPredicate.or(containsBiPredicate.negate()).test(“Hel
lo”, “LAB”);
     }
}
```

Consumer & BiConsumer

Consumer si BiConsumer sunt interfete in Java care ne ajuta sa stabilim niste actiuni prin baza unuia sau a doi parametri, transmisi prin lambda function. Consumer si BiConsumer nu returneaza niciun obiect.

Exemple:

```
public class Main {

     Consumer<String> upperCaseConsumer = (text) ->
System.out.println(text.toUpperCase());

     Consumer<String> lowerCaseConsumer = (text) ->
System.out.println(text.toLowerCase());

     Consumer<Double> logOfTenConsumer = (number) ->
System.out.println(Math.log10(number));

     BiConsumer<Integer, Integer> powConsumer = (base, power)
-> System.out.println(Math.pow(base, power));

     public static void main(String[] args) {
          upperCaseConsumer.accept("Hello World");
          lowerCaseConsumer.accept("Hello World");

          logOfTenConsumer.accept(1000.00);

          powConsumer.accept(2, 2);

          upperCaseConsumer
               .andThen(lowerCaseConsumer)
               .accept("Hello world");
     }
}
```

Function & BiFunction & UnaryOperator

Function este o intefata care cu ajutorul lambda function, primeste ca argument un anumit obiect si returneaza alt tip de obiect.

BiFunction este o interfata care cu ajutorul lambda function, primeste ca argument doua obiecte si returneaza alt tip de obiect.

UnaryOperator este o interfata care cu ajutorul lambda function, primeste ca argument si returneaza acelasi tip de obiect.

Exemple:

```
public class Main {

     Function<String, String> toUpperCase = (text) ->
text.toUpperCase();

     Function<String, String> toLowerCase = (text) ->
text.toLowerCase();

     Function<Integer, Double> log10 = (number) ->
Math.log10(number);

     BiFunction<Integer, Integer, Integer> powerOf = (base,
power) -> (int) Math.pow(base, power);


     UnaryOperator<String> prefixText = (text) -> “Prefix” +
text;

     public static void main(String[] args) {

          int value = log10.apply(100);

          // Prima functie, dupa aceea a doua functie
          toUpperCase.andThen(toLowerCase).apply(“Hello
World”);

          // A doua functie, dupa aceea prima functie
          toUpperCase.compuse(toLowerCase).apply(“Hello
World”);


          int number = powerOf.apply(2, 2);

          prefixText.apply(“Base Text”);
     }
}
```

Supplier

Supplier este o interfata in Java care cu ajutorul careia returnam un obiect stabilit prin lambda function.

Exemple:

```
public class Main {

	Supplier<Double> PI = () -> (double) 3.14;
	Supplier<Double> randomSupplier = () -> Math.random();

	public static void main(String[] args) {
		System.out.println(PI.get());
		System.out.println(randomSupplier.get());
	}
}
```

Mai multe despre ele gasiti aici:
https://www.baeldung.com/java-8-functional-interfaces

Optional

Optional este o clasa care incapsuleaza un obiect si ne ajuta sa lucram foarte mult cu obiecte null. Avem metode prestabilite pentru a ne ajuta a trata/evita unele exceptii.

```
	Optional.of(); // daca e null, primim
NullPointerException
	Optional.empty();
	Optional.ofNullable();

	Optional.isPresent() → returns boolean value
	Optional.isEmpty() → returns boolean value

	Optional.ifPresent(objectPresent -> // code );
```

```
     Optional.ofNullable(null).orElse("john"); // default
value
     Optional.ofNullable(null).orElseGet(() -> objectPassed);
     Optional.ofNullable(null).orElseThrow(); // exception
based on object exception
     // Se mai pot pasa si exceptii custom, dar asta o sa va
arat la exceptii.


     Integer year = 2016;
     boolean booleanCheck = Optional.of(year).filter(number ->
number == 2016).isPresent();
```

Mai multe despre Optional aici:
https://www.baeldung.com/java-optional

Comparator & Comparable

Cele doua interfete au aceleasi scop, de a sorta compara obiectele. Insa, pe scurt, Comparable se foloseste si se implementeaza in clasa unde dorim sa facem compararea, iar Comparator ne ofera posibilitatea sa comparam doua obiecte atunci cand nu avem acces la codul sursa al clasei pe care vrem s-o comparam.

Exemple:

Comparable:

Avem acces la codul sursa al clasei:

```
public class Player implements Comparable<Player> {

     @Override
     public int compareTo(Player otherPlayer) {
```

```
          return Integer.compare(getRanking(),
otherPlayer.getRanking());

     }
}
```

Comparator:

NU avem acces la codul sursa al clasei (si a se observa ca se implementeaza interfata intr-o noua clasa):

```
public class PlayerComparatorImpl implements
Comparable<Player> {

     @Override
     public int compareTo(Player player1, Player player2) {
          return player1.getRanking() - player2.getRanking();
     }
}
```

Mai multe puteti citi aici:
https://www.baeldung.com/java-comparator-comparable

Stream API & Collections

Atunci cand avem de lucrat cu mai obiecte, de obicei cu liste, avem la dispozitie un tool care ne poate ajuta foarte bine in Java si anume colectiile si Stream API.

Stream API sunt un subiect foarte larg in general, dar o sa vorbim doar despre cateva chestii astazi, care sunt cele mai folosite.

Mai multe gasiti aici: https://www.baeldung.com/java-8-streams

Interfata List (despre care o sa vorbim mai jos, ne da la dispozitie o metoda care se cheama “stream()”). Aceasta metoda ne intoarce un “stream” cu toate obiectele noastre.

Putem forma un Stream folosind Stream.of(object).

Cel mai utilizat caz in care am folosit Stream.of() a fost cand am avut de concatenat doua liste si anume in felul urmator putem face:
(Mai multe exemple aici:
https://www.baeldung.com/java-merge-streams)

```
public class Main {

     public static void main(String[] args) {
          List<Integer> integers1 = List.of(1, 2, 3);
          List<Integer> integers2 = List.of(4, 5, 6);

          Stream<Integer> streamConcat =
Stream.concat(integers1.stream(), integers2.stream());

          // creez iar o lista:
          List<Integer> listConcat = streamConcat.toList();
          // toList() este o metoda din clasa Stream aparuta
in Java 9+.
```

Daca am de lucrat cu o lista de obiecte si vreau sa transform fiecare element din lista, pot face in felul urmator:

```
public class Main {

     public static void main(String[] args) {
          List<Integer> integers1 = List.of(1, 2, 3);
          List<Integer> integers2 = List.of(4, 5, 6);

          List<String> stringIntegers = integers1.stream()
                         .map(element ->
String.valueOf(element)) // metoda de a transforma fiecare
element in String → practic metoda primeste un lambda function
care reprezinta elementul din lista si returneaza valoarea
functiei.
                         .toList(); // refac lista
```

Pot sa filtrez dupa o anumita conditie obiectele mele dintr-o lista in felul urmator:

```
public class Main {

     public static void main(String[] args) {
          List<Integer> integers1 = List.of(1, 2, 3);
          List<Integer> integers2 = List.of(4, 5, 6);

          List<Integer> integersFiltered = integers1.stream()
                         .filter(element -> element % 2 ==
0)// iau doar numerele pare, boolean expression
                         .toList(); // returnez totul intr-o
lista noua
```

Daca vreau sa gasesc daca exista vreun element cu o anumita conditie, de exemplu daca am numere pare in lista pot folosi *stream().anyMatch(element -> boolean expression)* -> returneaza un boolean:

```
public class Main {

     public static void main(String[] args) {
          List<Integer> integers1 = List.of(1, 2, 3);
          List<Integer> integers2 = List.of(4, 5, 6);

          if( integers1.stream()
                         .anyMatch(element -> element % 2 ==
0)// verific daca exista numere pare, boolean expression
)     {
                // COD DACA E NUMAR PAR IN LISTA
          } else {
                // ELSE
          }
}
```

La fel se poate folosi si negarea, daca nu exista niciun element cu o conditie specifica *stream().noneMatch(element -> boolean expression)* -> returneaza un boolean:

```
public class Main {
```

```
     public static void main(String[] args) {
          List<Integer> integers1 = List.of(1, 2, 3);
          List<Integer> integers2 = List.of(4, 5, 6);

          if( integers1.stream()
                         .noneMatch(element -> element % 2 ==
0)// daca nici unul nu este par
)      {
               // COD DACA NU E NICIUNUL PAR IN LISTA
          } else {
               // ELSE
          }
}
```

Mai exista *stream().count(), stream().distinct(), stream().sorted()* si *stream().forEach(element -> // code)*.

Prima returneaza cate elemente sunt in stream, a doua ia doar elementele distincte, a treia sorteaza elementele (se poate adauga si o noua interfata comparator, primeste doua variabile prin lambda function si returneaza functia de comparare, iar ultima ia fiecare element in parte si face ceva conform codului.

Exemple:

```
public class Main {

     public static void main(String[] args) {
          List<Integer> integers1 = List.of(1, 2, 3);
          List<Integer> integers2 = List.of(4, 5, 6);

          int count = integers1.stream().count(); //
returneaza 3

          List<Player> players = List.of(new Player(id=1,
name='A', name='B'), new Player(id=2), new Player(id=3,
name='C'));
```

```
          var distinctPlayers =
players.stream().distinct().toList(); // distinct() returneaza
un Stream.

          var sortedPlayers =
players.stream().sorted().toList(); // sorted() returneaza un
Stream.

          var sortedByComparator = players.stream().sorted(
(p1, p2) -> p1.getName().compareTo(p2.getName())).toList();


          players.stream()
               .forEach(player -> System.out.println(player));
     }
```

De fapt, despre ceea ce am vorbit acum este vorba de niste operatii. Operatiile sunt INTERMEDIARE SI FINALE.

Operatiile intermediare nu returneaza un obiect, iar cele finale returneaza un obiect.

Operatii intermediare: stream()

.map(element -> {
// code
return !!obligatoriu!!;
})

.peek(element -> // code) → este map fara return in blocuri

.filter()

.distinct()

.sorted()

Operatii finale: stream()

.count()

.toList()

.collect(Collectors.toList())
.collect(Collectors.toMap())
.collect(Collectors.toUnmodifiableSet()) // etc. → se gasesc cand se lucreaza

!!!! Mai exista si .parallelStream(), dar nu o sa intru acum in el, nefiind folosit foarte des si dat pe la interviuri. Dar, **este bine sa cititi despre el SI SA STITI CAND SE FOLOSESTE**:
https://www.baeldung.com/java-when-to-use-parallel-stream

How to: -> practic, la interviuri s-a mai dat

Luam din baza de date o lista cu obiecte si vrem sa accesam in O(1) un obiect.

Solutia: Se returneaza List<Object> din baza de date (care evident obiectul are id-ul incapsulat) si se folosesc stream-uri pentru a face o mapare de la List<Object> la Map<String, Object> unde key-ul este id-ul, iar value este obiectul in sine.

Astfel, putem obtine un obiect dintr-o lista in O(1) fara a cunoaste index-ul, ci doar id-ul.

Mai multe se pot gasi aici:
https://www.baeldung.com/java-collectors-tomap

Clasa Collections o sa fie studiata la curs mai pe larg, dar se poate folosi cautare binara folosind *Collections.binarySearch()*, se poate găsi minimul folosind *Collections.min()*, maximul folosind *Collections.max().*

Binary Tree: Un arbore este binar daca are doi copii, cel din stanga si cel din dreapta. Un arbore are o radacina. Cautarea intr-un arbore se face in log(n).

Threading (basic)

De la sisteme de operare, stiti ca atunci cand avem nevoie sa lucram cu multe date, lucram cu mai multe thread-uri. Atunci cand lucram cu mai multe thread-uri si actionam pe mai multe obiecte trebuie sa ne asiguram ca resursa nu este accesata de mai multe thread-uri in acelasi timp. Clasificam un obiect ca fiind ‘thread-safe’ atunci cand poate fi accesat de mai multe thread-uri fara probleme.

Load Factor & Capacity

Aproape toate structurile de date din Java (daca nu chiar toate), au doua proprietati: load factor si capacity.

Astea sunt folosite pentru a stii cand se face resize a structurilor de date. Load factor-ul decide cand sa creasca capacitatea, iar capacity este capacitatea pe care o are structura de date.

StringBuilder vs StringBuffer

StringBuilder si StringBuffer sunt doua clase care ne ajuta pentru a lucra cu String-uri. Ce este de retinut, este ca cu ele putem face 'String' sa fie mutabil (stim ca String este imutabil).

Diferenta principala dintre cele doua este ca StringBuilder NU este thread-safe, iar StringBuffer este. De aici, deducem ca StringBuilder este mai rapid, iar StringBuffer este mai lent.

Se pot citi mai multe despre performanta aici: https://www.digitalocean.com/community/tutorials/string-vs-stringbuffer-vs-stringbuilder

List interface, ArrayList vs LinkedList

List este o interfata cu anumite metode prestabilite. ArrayList si LinkedList implementeaza interfata List.

Cea mai cunoscuta metoda (si cea mai folosita de mine) este: *List.of()* (Java 9+) prin care creem o lista imutabila. O metoda asemanatoare este *Arrays.asList()* prin care creem o lista cu elemente fixe (nu putem adauga elemente) si care este mutabila. (Putem seta alte elemente in lista pe un index existent).

Diferenta dintre List.of() si Arrays.asList() se poate citi aici: https://www.baeldung.com/java-arrays-aslist-vs-list-of

ArrayList si LinkedList nu sunt thread-safe.

Complexitati:

Cum functioneaza ArrayList? Este un tip de date abstract in care se face resize automat. Atunci cand capacitatea este maxima, add() este de complexitate O(n).

1. ArrayList
    - add() - O(1) cand nu facem resize, O(n) cand se face resize.
    - add(index, element) - O(n)
    - get() - O(1) - e pe baza de index
    - indexOf() - O(n) - se itereaza peste toata lista
    - contains() - O(n) - se itereaza peste toata lista
    - remove() - O(n) cand nu stim index-ul

2. LinkedList
    - add() - O(1) - se adauga mereu in coada
    - add(index, element) - O(n) - se itereaza peste lista
    - get() - O(n) - se itereaza peste lista
    - remove() - O(n) - se itereaza peste lista
    - remove(index) - O(n) - se itereaza peste lista
    - contains() - O(n) - se itereaza peste lista

Map interface, HashMap vs Hashtable vs TreeMap vs LinkedHashMap

Map este o interfata cu anumite metode prestabilite. HashMap, Hashtable, TreeMap si LinkedHashMap implementeaza interfata Map.

De recapitulat este ca un hashtable este o structura de date care stocheaza datele sub forma <key, value>, unde in Java key si value pot fi obiecte, nu se pot folosi primitive.

Ah, apropo, v-am zis ca <> este diamond operator (java 7+)? (ref: https://www.baeldung.com/java-diamond-operator)

De retinut este faptul ca HashMap NU este thread-safe, iar Hashtable, iar inserarea in HashMap nu garanteaza ca elementele se vor insera in ordine. HashMap accepta sa se insereze elemente null si o singura cheie nula, iar Hashtable nu.

TreeMap este un tip de date care are implementata automat interfata Comparator, iar in mod implicit sortarea se face bazata pe cheile map-ului. Ca baza, are implementat un arbore rosu-negru (Red-Black tree). Nu este thread-safe si nu accepta null keys, dar accepta null values.

LinkedHashMap, fata de HashMap asigura ordinea inserarii elementelor. Mai multe se pot citi aici: https://www.baeldung.com/java-linked-hashmap

Complexitati:

Cum functioneaza HashMap? Este un tip de date abstract in care se face resize automat. Cheia este encodata intr-un hashcode. Este posibil ca la un moment dat sa avem acelasi hashcode pentru doua valori diferite. In Java, asta se trateaza cu liste inlantuite (mai multe aici:). In general, tehnica asta se cheama ‘chaining’, dar mai exista si o alta varianta care se cheama ‘linear probing’ (tehnica asta cauta prin tot hashmap-ul pana

cand gaseste un ‘loc liber’ si insereaza elementul. → creste complexitatea? - Da, evident, O(n) la .put()). Elementele prin care se encodeaza se cheama buckets. *Daca sunt mai mult de 8 bucket-uri, o sa fie convertit intr-un arbore balansat, modul in care sunt stocate bucket-urile. (Ref: https://www.javacodemonk.com/difference-between-hashmap-linkedhashmap-and-treemap-d33da4e2)*
Mai multe despre cum functioneaza HashMap-ul aveti aici: https://www.baeldung.com/java-hashmap-load-factor .

Va rog sa cititi si pe paginile de java documentation, nu dureaza mult:
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Hashtable.html
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/HashMap.html
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/TreeMap.html

Complexitati:

1. HashMap/Hashtable/LinkedHashMap
   - contains() - O(1)
   - get() - O(1)
   - put() - O(1)
   - remove() - O(1)

2. TreeMap
   - put(), get(), remove(), containsKey() - O(log n) - red-black-tree complexity

Puteti citi mai multe si in *cursul 8 de anul trecut*.

Set interface, HashSet vs TreeSet vs LinkedHashSet

Set este o interfata cu anumite metode prestabilite. HashSet, TreeSet si LinkedHashSet implementeaza interfata Set. Nici una dintre ele nu este thread-safe.

Este o lista fara duplicate, avand complexitatile asemanatoare unui hashtable.

Despre cum functioneaza totul gasiti aici:
https://www.baeldung.com/java-hashset

Complexitati:

1. HashSet/LinkedHashSet
    - add(), remove() - O(1) - O(n) cand se face resize
    - contains() - O(1)

2. TreeSet
    - add(), remove(), contains() - O(log n) - implementare de red-black tree. (TreeMap implementation).

Mai gasiti si aici, despre ele:
https://www.baeldung.com/java-collections-complexity

Gasiti cei primii k termeni dintr-o lista folosind Java Collections:
https://www.baeldung.com/java-array-top-elements

In Java mai exista si clasele Queue, Deque, Vector si Stack, dar nu le-am folosit niciodata.

Se pot citi despre ele aici:
https://www.baeldung.com/java-deque-vs-stack
https://www.baeldung.com/java-stack
https://www.baeldung.com/java-arraylist-vs-vector