Laborator 8

Expresii Lambda

- Reprezinta o bucata de cod care implementează o interfata funcţională, fără crearea unei clase anonime sau concrete.
- Practic este o metoda anonimă.

O lambda expresie constă:

- Dintr-o listă de parametri formali, separați prin virgulă şi cuprinşi eventual între paranteze rotunde.
- Săgeata direcțională ->.
- Un body ce constă dintr-o expresie sau un bloc de instrucțiuni.

```
// ( int arg1, String args2 )      ->        System.out.println( "Two arguments " + arg1 + " and " + arg2);
//   argument list            arrow token    Body of lambda expression

// no argument - un singur statement, fara acolade
LambdaInterface lambdaInterface = () -> System.out.println("Hello World");

lambdaInterface.show();

// one argument - single statement with curly braces

//      LambdaInterface anotherLambdaInterface = (message) -> { System.out.print(message);};
LambdaSecondInterface secondInterface = (message) -> System.out.println(message);

secondInterface.show( a: 5);

// with two argument and types
LambdaThirdInterface lambdaThirdInterface = (int a, int b) -> {
    return a * b;
};
System.out.println(lambdaThirdInterface.compute( a: 5,  b: 4));

// two arguments - multiple statements
LambdaFourthInterface lambdaFourthInterface = (int a, int c) -> {
    if (a > c) {
        System.out.println("Primul element este mai mare");
    } else {
        System.out.println("Al doilea element este mai mare");
    }
};

lambdaFourthInterface.doSomething( a: 4,  c: 20);
```

- O interfață funcțională (*functional interface*) este orice interfață ce conține doar o metodă abstractă. Din această cauză putem omite numele metodei atunci când implementăm interfața şi putem elimina folosirea claselor anonime. În locul lor vom avea *lambda expresii*.

```java
@FunctionalInterface
public interface FunctionalInterfaceEx {

    void singleMethod();
}
```

Interfeţe Funcţionale în Java:

```java
/*
    java.util.function.Function
    - reprezinta o functie/metoda care primeste un singur parametru si intoarce o singura valoare

    public interface Function<T,R> {

        public <R> apply(T parameter);
    }
 */

Function<Long, Long> adder = new AddThree();
Long result = adder.apply((long) 4);


Function<Long, Long> adder2 = (value) -> value + 3;
Long resultLambda = adder2.apply((long) 8);
System.out.println("resultLambda = " + resultLambda);
```

```java
/*
    java.util.function.Predicate
    - reprezinta o functie/metoda care primeste un singur parametru si intoarce true sau false

    public interface Predicate {
        boolean test(T t);
    }
 */

Predicate<Long> predicate = new CheckForNull();
Boolean status = predicate.test( t: null);
System.out.println(status);

Predicate<Long> lambdaPredicate = s -> s !=null;
Boolean statusLambda = lambdaPredicate.test( t: null);
System.out.println(statusLambda);
```

```java
/*
    java.util.function.Consumer
    - reprezinta o functie/metoda care consuma un  parametru fara sa intoarca vreun rezultat

 */

Consumer<Integer> consumer = (value) -> System.out.println(value);
consumer.accept( t: 20);
```

```java
/*
    java.util.function.Supplier
    - reprezinta o functie/metoda care nu primeste niciun parametru dar intoarce o valoare
 */

Supplier<Double> randomValue = () -> Math.random();
System.out.println(randomValue.get());
```

Avantaje:

- Expresiile lambda reduc numărul de linii de cod scrise
- Ca parte a API-ului Collections, java.util.stream oferă suport pentru operații funcționale pe *stream*-uri de elemente

Tipuri de operaţii pe Stream-uri:

- **intermediare**, care întorc un stream nou pe care se pot face procesări.
- **terminale**, care marchează stream-ul ca fiind consumat, punct în care nu mai poate fi folosit.

Operation	Return Type	Type Of Operation	What It Does?
filter()	Stream<T>	Intermediate	Returns a stream of elements which satisfy the given predicate.
map()	Stream<R>	Intermediate	Returns a stream consisting of results after applying given function to elements of the stream.
distinct()	Stream<T>	Intermediate	Returns a stream of unique elements.
sorted()	Stream<T>	Intermediate	Returns a stream consisting of elements sorted according to natural order.
limit()	Stream<T>	Intermediate	Returns a stream containing first *n* elements.
skip()	Stream<T>	Intermediate	Returns a stream after skipping first *n* elements.
forEach()	void	Terminal	Performs an action on all elements of a stream.
toArray()	Object[]	Terminal	Returns an array containing elements of a stream.
reduce()	type T	Terminal	Performs reduction operation on elements of a stream using initial value and binary operation.
collect()	Container of type R	Terminal	Returns mutable result container such as List or Set.
min()	Optional<T>	Terminal	Returns minimum element in a stream wrapped in an Optional object.
max()	Optional<T>	Terminal	Returns maximum element in a stream wrapped in an Optional object.
count()	long	Terminal	Returns the number of elements in a stream.
anyMatch()	boolean	Terminal	Returns true if any one element of a stream matches with given predicate.
allMatch()	boolean	Terminal	Returns true if all the elements of a stream matches with given predicate.
noneMatch()	boolean	Terminal	Returns true only if all the elements of a stream doesn't match with given predicate.
findFirst()	Optional<T>	Terminal	Returns first element of a stream wrapped in an Optional object.
findAny()	Optional<T>	Terminal	Randomly returns any one element in a stream.

```java
/*
forEach()
 - este cea mai simpla si comuna operatie, parcurge elementele stream-ului
 apeland functia data ca parametru pe fiecare element
 - exista in Iterable si Map
 - este o operatie terminala, dupa ce este apelata, nu mai putem sa facem operatii
 pe stream-ul respectiv este considerat ca fiind consumat
*/
empList.stream().forEach( e -> System.out.println(e.getName()));
```

```java
/*
map()
 - creaza un stream nou dupa aplicarea unei functii la fiecare element al stream-ului original
 - stream-ul nou poate sa fie de tip diferit
 - operatie intermediara
 */
List<Integer> ids = empList.stream()
        .map(e -> e.getId())
        .collect(Collectors.toList());
System.out.println(ids);
```

```java
/*
    collect()
     - cea mai comuna metoda de a prelua elemente dintr-un stream dupa prelucrare lor, si a le
       impacheta intr-o structura
     - operatie terminala
 */
List<Employee> employees = empList.stream().collect(Collectors.toList());
Set<Employee> employeesSet = empList.stream().collect(Collectors.toSet());
Map<Integer, String> employeesMap = empList.stream().collect(Collectors.toMap(Employee::getId, Employee::getName));
```

```java
/*
    filter()
    - creaza un stream nou pe baza elementelor din stream-ul vechi care respecta conditia din filter
    - operatie intermediara
 */

List<Employee> filteredEmployees = employees.stream()
        .filter(e -> e.getSalary() > 150000.0)
        .collect(Collectors.toList());
filteredEmployees.stream().forEach(e -> System.out.println("Name: " + e.getName() + "  salary: " + e.getSalary()));
```

```java
// unele operatii sunt considerate operatii de scurt circuitare
// operatiile de scurt-circuitare permit operatii pe stream-uri de date infinite in timp finit
Stream<Integer> infiniteStream = Stream.iterate( seed: 2, i -> i * 2);

List<Integer> collect = infiniteStream
        .skip(3)
        .limit(5)
        .collect(Collectors.toList());
System.out.println(collect);
```

```java
/*
    findFirst()
    - intoarce un obiect de tip Optional pentru prima intrare din stream
    - operatie terminala
 */
Employee employee = employees.stream()
        .filter(e -> e.getSalary() > 150000.0)
        .findFirst()
        .orElse( other: null);
System.out.println(employee.getName() + " " + employee.getSalary());
```

```java
/*
    flatMap()
    - un stream poate sa contina structuri de date complexe
        Stream<List<String>>    -> flatMap ->   Stream<String>

        { {1,2}, {3,4}, {5,6} } -> flatMap -> {1,2,3,4,5,6}

        { {'a','b'}, {'c','d'}, {'e','f'} } -> flatMap -> {'a','b','c','d','e','f'}
 */

List<List<String>> namesNested = Arrays.asList(
        Arrays.asList("Jeff", "Bezos"),
        Arrays.asList("Bill", "Gates"),
        Arrays.asList("Mark", "Zuckerberg"));

List<String> namesFlatStream = namesNested.stream()
        .flatMap(Collection::stream)
        .collect(Collectors.toList());

System.out.println(namesFlatStream);
```

```
/*
    One of the most important characteristics of streams is that they allow for significant optimizations through lazy evaluations.

    A possible reason against this is if longer methods are put into the if statement, maybe with a few parameters,
     it will start to get messy and you might struggle to read it.

*/
public static void main(String[] args) {
    final int number = 4;
    //      final boolean computeResult = compute(number);
    //      final boolean processResult = process(number);

    final Supplier<Boolean> computeResult = () -> compute(number);
    final Supplier<Boolean> processResult = () -> process(number);
    if (computeResult.get() && processResult.get()) {
        System.out.println("TRUE");
    } else {
        System.out.println("FALSE");
    }
}

public static boolean compute(final int number) {
    System.out.println("computing number : " + number);
    return number > 5 ? true : false;
}

public static boolean process(final int number) {
    System.out.println("processing number : " + number);
    return number % 3 == 0 ? true : false;
}


*
Through the use of some lazy lambdas we can keep our code easy to read without sacrificing performance by executing unneeded operations.
```