Laboratorul 11: Java features

Obiective

- înțelegerea conceptelor de expresii lambda și de streams
- familiarizarea cu metode default și cu metode statice în interfețe
- utilizarea de structuri sintactice introduse începând cu Java 8 (var)

Metode statice și metode default în interfețe

Începând cu Java 8, putem implementa metode în cadrul interfețelor, unde, în mod clasic, avem doar metode neimplementate (abstracte). Mai precis, noi putem să implementăm trei tipuri de metode: default (non-statice), statice, iar începând cu Java 9 și private.

Metode statice

Metodele statice, în cadrul unei interfețe, se comportă identic cu metodele statice din clasă, în sensul că ele se apelează sub forma `SomeInterface.metodaStatica()` și că acestea nu sunt moștenite nici de clasele care implementează interfața respectivă și nici de interfețele care extind respectiva interfață.

```
public interface SomeInterface {
    static void doStuff() {
        System.out.println("Doing something");
    }
}

// apelul metodei statice
SomeInterface.doStuff();
```

Metode default

O metodă default este o metodă obișnuită, non-statică, publică în mod default și marcată prin keyword-ul `default`, care se comportă ca o metodă moștenită (obișnuită și funcțională) în clasele ce implementează interfața ce conține metoda default respectivă.

Un motiv pentru care există metode default în Java este următorul: putem avea o interfață (o numim `Animal`), care are mai multe metode, inclusiv `shakeTail()`, și o clasă numită Human care implementează interfața respectivă. În acest caz, clasa `Human` trebuie să implementeze metoda `shakeTail()`, care nu este nicidecum relevantă, astfel am avea o implementare forțată a metodei `shakeTail()`, în care am avea un body gol sau să aruncăm o excepție, încălcând astfel unul dintre principiile SOLID, mai precis Interface Segregation Principle [https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#interface-segregation-principle]

Exemplu:

```
interface Animal {
    default void shakeTail() {
        System.out.println("Humans have no tail");
    }

    static void print() {
        System.out.println("This is an animal");
    }

    void makeSound();
}

class Human implements Animal {

    @Override
    public void makeSound() {
        System.out.println("Ahhhh");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal.print();
        Animal human = new Human();
        human.makeSound();
        human.shakeTail();
    }
}
```

Având în vedere că moștenirea multiplă nu este suportată în Java, mai precis extinderea a două clase în același timp, această problemă se propagă și la interfețe, în cazul metodelor default. Această problemă apare dacă o clasa implementează două interfețe, fiecare având o metodă default cu aceeași semnătură, în acest caz apărând eroare de compilare, care se rezolvă dacă respectiva clasă are propria sa implementare pentru metodele default din acele interfețe. Pentru o mai bună înțelegere, urmăriți exemplul de mai jos.

```
interface FirstInterface {
    default void doSomeAction() {
        System.out.println("FirstInterface action");
    }

    void doJob();
}

interface SecondInterface {
    default void doSomeAction() {
        System.out.println("SecondInterface action");
    }

    void doStuff();
}
```

```
class SomeClass implements FirstInterface, SecondInterface {
    // dacă nu am face override la doSomeAction, am avea eroare de compilare!
    @Override
    public void doSomeAction() {
        System.out.println("Action by SomeClass");
    }

    @Override
    public void doJob() {
        System.out.println("Do the job");
    }

    @Override
    public void doStuff() {
        System.out.println("Do the stuff");
    }
}
```

Metode private

Metodele private, în cadrul unei interfețe, se comportă identic cu metodele private din clasă, în sensul că ele nu pot fi apelate din exterior și că acestea nu sunt moștenite nici de clasele care implementează interfața respectivă și nici de interfețele care extind respectiva interfață.

```
public interface JavaFeaturesInterface {
        default int sum(int a, int b) {
                return a + b;
        }

        default int difference(int a, int b) {
                // Apelul metodei private
                return diff(a, b);
        }

        static int product(int a, int b) {
                return a * b;
        }

        private int diff(int a, int b) {
                return a - b;
        }
}

// apelul metodei private va genera eroare întrucât nu este accesibilă din exterior
SomeInterface.diff();
```

Interfețe funcționale, funcții și expresii lambda

În Java 8, au fost introduse interfețele funcționale, care reprezintă interfețe ce conțin fix o metodă (neimplementată / abstractă). Aceste interfețe pot fi implementate sub forma de expresii lambda (despre care am vorbit în cadrul laboratorului de clase interne [https://ocw.cs.pub.ro/courses/poo-ca-cd/laboratoare/clase-interne#expresii_lambda], introduse, de asemenea, în Java 8, pentru a reduce numărul de linii de cod, pentru a putea pasa funcții ca parametri la metode și pentru folosirea evaluării leneșe, despre care veți discuta la Paradigme de Programare, în semestrul următor), ele fiind folosite pentru a implementa funcții anonime în Java. Pentru a marca o interfață funcțională, este de recomandat să adăugam adnotarea `@FunctionalInterface`, prin care se permite existența unei singure metode abstracte în cadrul unei interfețe funcționale.

```
@FunctionalInterface
interface FunctionalInterface {
    int sum(int a, int b);
}

class Main {
    public static void main(String[] args) {
        FunctionalInterface sumOp = (a, b) -> a + b;
        System.out.println(sumOp.sum(1, 2));
    }
}
```

În cadrul pachetului java.util.function [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/package-summary.html], există interfețe funcționale predefinite, care pot fi folosite mai ales în cadrul colecțiilor (streams).

Consumer

Consumer<T> [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Consumer.html] - reprezintă o funcție care primește un parametru de tip T și nu întoarce nimic (o funcție void).

Exemplu:

```
public void whenNamesPresentConsumeAll(){
    Consumer<String> printConsumer = t -> System.out.println(t);
    Stream<String> cities = Stream.of("Sydney", "Dhaka", "New York", "London");
    cities.forEach(printConsumer);
}
```

Interfața Consumer<T> [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Consumer.html] expune și o metodă andThen(Consume<T>) [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Consumer.html#andThen(java.util.function.Consumer)] care poate înlănțui alte operații de tipul Consumer.

Exemplu:

```
public void whenNamesPresentUseBothConsumer(){
    List<String> cities = Arrays.asList("Sydney", "Dhaka", "New York", "London");

    Consumer<List<String>> upperCaseConsumer = list -> {
        for(int i=0; i< list.size(); i++){
            list.set(i, list.get(i).toUpperCase());
        }
    };
```

```
    Consumer<List<String>> printConsumer = list -> list.stream().forEach(System.out::println);

    upperCaseConsumer.andThen(printConsumer).accept(cities);
}
```

Există și o interfață BiConsumer<T, U> [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/BiConsumer.html] care primește două argumente în loc de unul singur, dar are aceeași funcționalitate.

Predicate

Predicate<T> [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Predicate.html] - reprezintă o funcție booleană care primește un singur parametru de tip T. Ea este folosită în general împreună cu funcțiile de filter pe stream-uri.

```
public void testPredicate(){
    List<String> names = Arrays.asList("John", "Smith", "Samueal", "Catley", "Sie");
    Predicate<String> nameStartsWithS = str -> str.startsWith("S");
    names.stream().filter(nameStartsWithS).forEach(System.out::println);
}
```

Interfața Predicate<T> [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Predicate.html] expune și metodele and(Predicate<T>) [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Predicate.html#and(java.util.function.Predicate)], not(Predicate<T>) [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Predicate.html#not(java.util.function.Predicate)], or(Predicate<T>) [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Predicate.html#or(java.util.function.Predicate)] care pot înlănțui alte operații de tipul Predicate.

Exemplu:

```
public void testPredicateAndComposition(){
    List<String> names = Arrays.asList("John", "Smith", "Samueal", "Catley", "Sie");
    Predicate<String> startPredicate = str -> str.startsWith("S");
    Predicate<String> lengthPredicate = str -> str.length() >= 5;
    names.stream().filter(startPredicate.and(lengthPredicate)).forEach(System.out::println);
}
```

Există și o interfață BiPredicate<T, U> [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/BiPredicate.html] care primește două argumente în loc de unul singur, dar are aceeași funcționalitate.

Function

Function<T, R> [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Function.html] - reprezintă o funcție care primește un parametru de tip T și întoarce un rezultat de tip R. Ea este folosită în general împreună cu funcțiile de map pe stream-uri.

```
public void testFunctions(){
    List<String> names = Arrays.asList("Smith", "Gourav", "Heather", "John", "Catania");
    Function<String, Integer> nameMappingFunction = String::length;
    List<Integer> nameLength = names.stream().map(nameMappingFunction).collect(Collectors.toList());
    System.out.println(nameLength);
}
```

Interfața Function<T, R> [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Function.html] expune și metode care pot înlănșui alte operașii de tipul Function.

- andThen(Function<T, R>) [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Function.html#andThen(java.util.function.Function)] - functia data ca parametru este executata dupa cea la care este apelata metoda
- compose(Function<T, R>) [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Function.html#compose(java.util.function.Function)] - functia data ca parametru este executata inainte de cea la care este apelata metoda

Există și o interfață BiFunction<T, U, R> [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/BiFunction.html] care primește două argumente în loc de unul singur, dar are aceeași funcționalitate.

Supplier

Supplier<T> [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/Supplier.html] - reprezintă o funcție care nu primește niciun parametru și întoarce un rezultat de tip T, prin funcția `get()` și este folosită ca target pentru o funcție lambda.

```
public void supplier(){
    Supplier<Double> doubleSupplier1 = () -> Math.random();
    DoubleSupplier doubleSupplier2 = Math::random;

    System.out.println(doubleSupplier1.get());
    System.out.println(doubleSupplier2.getAsDouble());
}
```

De asemenea, interfața Comparator [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Comparator.html] este o interfață funcțională, având o singură metodă abstractă - compare [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Comparator.html#compare(T,T)].

Streams

Stream-urile [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/stream/package-summary.html] au fost introduse în Java 8, pentru a permite programatorului să efectueze operații de tipul filter, map și reduce pe colecții într-un mod elegant și eficient.

Stream-urile pot fi obținute prin mai multe modalități, în special de la liste și set-uri, care expun metodele `stream()` și `parallelStream()`. Diferența constă în faptul că cea dintâi face operațiile secvențial în timp ce a doua le face în paralel pe un mediu de procesare cu mai multe core-uri.

Exemple:

```
Arrays.asList("a1", "a2", "a3")
    .stream()
    .findFirst()
    .ifPresent(System.out::println);  // a1
```

```
Stream.of("a1", "a2", "a3")
    .findFirst()
    .ifPresent(System.out::println);  // a1
```

```
IntStream.range(1, 4)
    .forEach(System.out::println);
```

```
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average()
    .ifPresent(System.out::println);  // 5.0
```

Map, Filter, Peek, Reduce

Provenite din Programarea functionala [https://www.geeksforgeeks.org/functional-programming-paradigm/], functiile de `map()`, `filter()`, `peek()` și `reduce()` au fost importate în Java ca principal mijloc de lucru cu Stream-uri. Ele ajută la reducerea considerabilă a codului scris, dar pot, în același timp, să îngreuneze înțelegerea acestuia, cu atât mai mult pentru programatorii aflați la început de carieră.

Map

Permite conversia datelor dintr-un Stream prin schimbare a tipului sau modificare a valorii.

```
String[] myArray = new String[]{"bob", "alice", "paul", "ellie"};
Stream<String> myStream = Arrays.stream(myArray);
Stream<String> resultStream1 = myStream.map(s -> s.toUpperCase()); // ["BOB", "ALICE", "PAUL", "ELLIE"]
resultStream1.forEach(System.out::println);
```

```
String[] myArray = new String[]{"bob", "alice", "paul", "ellie"};
Stream<Integer> resultStream2 = myStream.map(s -> s.length()); // [3, 5, 4, 5]
resultStream2.forEach(System.out::println);
```

```
class Student{
    String name;
    Integer age;

    Student(String name, Integer age) {
        this.name = name;
        this.age = age;
    }
}
Stream<Student> resultStream3 = myStream.map(s -> new Student(s, s.length())); // [Student: {"Bob", 3}, Student: {"Alice", 5}, Student: {"Paul", 4}, Student: {"Ellie", 5}]
resultStream3.forEach(System.out::println);
```

Exemple de conversie inversă (de la Stream la un tip de date):

```
String[] myNewArray = resultStream1.toArray(String[]::new);
List<String> myNewArray2 = (ArrayList<String>)resultStream1.collect(Collectors.toList());
Set<Integer> myNewSet = (Set<Integer>) resultStream2.collect(Collectors.toSet());
```

Filter

Așa cum spune și numele, ajuta la filtrarea elementelor unui Stream, mai precis la eliminarea elementelor nedorite din acesta. Precum funcția de `map()`, așteaptă ca argument o expresie `lambda`, doar că, de data aceasta, expresia trebuie să întoarcă un boolean ce va determina dacă o valoare **rămâne** în Stream.

```
ArrayList<String> myArray = (ArrayList<String>) Arrays.asList("dog", "cat", "monkey", "elephant", "rat", "lion", "zebra");
String[] myNewArray =  Arrays.stream((String[])myArray.toArray())
      .filter(x -> x.length() > 4)
      .toArray(String[]::new);
```

Peek

Returnează un Stream în urma aplicării unui Consumer pe elementele unui stream deja existent. Conform documentației [https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#peek(java.util.function.Consumer)] Java, scopul principal al acestei metode este debugging-ul. Pe deasupra, un scenariu în care se poate dovedi foarte utilă e reprezentat de posibilitatea de a modifica informațiile de stare internă ale unui obiect.

```
class Student{
    String name;
    Integer age;

    Student(String name) {
        this.name = name;
        this.age = age;
    }

    // getters, setters and toString
}

Stream<Student> studentStream = Stream.of(new Student("Alice", 21), new Student("Bob", 22), new Student("Gigel", 20));
studentStream.peek(st -> st.setName(st.getName().toLowerCase()))
  .forEach(System.out::println); // Se vor afișa studenții având numele scris cu litere mici
```

Toate operațiile prezentate anterior fac parte din categoria operațiilor `intermediare`, ele nu ajung sa fie executate până când nu avem nevoie de un rezultat explicit(se apelează o metodă `terminală`).

Reduce

Metodele de tip `reduce` sunt metode ce obțin un **rezultat** în urma unei operații pentru întregul set de date. De aceea, le vom numi și metode `terminale`, deoarece setul de operații

funcționale pe care îl vom aplica asupra Stream-ului se va termina aproape întotdeauna cu o operație de tip `reduce`.

Deja ați întâlnit operația `toArray()` anterior, care este o operație de tip `reduce`. Alte operații similare ar fi `sum`, `average` și `count`.

Funcția în sine de `reduce()`, spre deosebire de `map` și `filter`, acceptă două argumente: un element identitate (sau acumulator) și o expresie lambda.

```
String[] myArray = { "this", "is", "a", "sentence" };
String result = Arrays.stream(myArray)
                .reduce("", (a,b) -> a + b);
```

flatMap și groupingBy

flatMap

Similar cu map, această funcțională permite aplicarea unei metode pe elementele unui stream. Ce aduce în plus această metodă este că aplică și o operație de flat pe stream-ul rezultat în urma aplicării acelei operații pe toate elementele din cadrul stream-ului. Cel mai bun exemplu ar fi dacă avem de-a face cu o listă de liste, aplicăm funcționala flatMap(), iar rezultatul va fi o listă cu elementele combinate din acele liste:

```
List<List<Integer>> listOfLists =
        List.of(List.of(1, 2, 3, 4), List.of(10, 9, 8, 7), List.of(5, 6));
    List<Integer> singleList = listOfLists.stream().flatMap(List::stream).toList();
    System.out.println(singleList);
```

Iar output-ul va fi:

```
[1, 2, 3, 4, 10, 9, 8, 7, 5, 6]
```

groupingBy

Această funcțională se folosește pentru a grupa elementele dintr-un stream după o anumita condiție (de exemplu grupatul unei liste de numere în par și impar). Aceasta întoarce o instanță de tip Map:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    Map<String, List<Integer>> parity =
        numbers.stream().collect(Collectors.groupingBy(i -> i % 2 != 0 ? "Odd" : "Even"));
    for (String paritate : parity.keySet()) {
      System.out.println(paritate + " " + parity.get(paritate));
    }
```

Iar output-ul va fi:

```
Even [2, 4, 6, 8, 10]
Odd [1, 3, 5, 7, 9]
```

var

În Java 10, `var` a fost întrodus pentru a face munca unui programator mai lejeră și acesta poate fi folosit doar în interiorul blocurilor de cod (nu poate fi folosit în declararea câmpurilor unei clase sau la semnătura unei metode).

`var` poate fi folosit doar când o variabilă este declarată și i se atribuie în același timp o valoare, prin care se deduce tipul variabilei (se aplică doar dacă tipul variabilei se identifică în timpul compilării).

Exemplu:

```
var n = 10; // se deduce ca tipul este int
var list = new ArrayList(); // se deduce ca tipul este ArrayList

var p; // aceasta declaratie este eronata, genereaza eroare de compilare,
// deoarece nu se poate deduce tipul variabilei (trebuie neaparata atribuita o valoare la declaratie)
```

Nu este recomandată folosirea în exces al lui `var`, deoarece duce la un cod mai greu de înțeles și care poate provoca erori cu ușurință.

Limbajul Kotlin

Kotlin reprezintă un limbaj de programare orientat pe obiecte, creat încât să fie interoperabil cu Java, mai precis Kotlin rulează în același mediu cu Java (în JVM), astfel noi putem să avem un proiect în care să avem clase scrise în Java și clase scrise în Kotlin, o practică care este întâlnită în proiecte din industrie (backend, Android).

Spre deosebire de Java, Kotlin are o sintaxă simplificată și mai concisă decât Java și are câteva features care face acest limbaj atractiv, de exemplu null checks, safe calls, valori default pentru parametrii unei metode, named parameters, extension functions, data classes etc.

Dacă sunteți interesați să învățați acest limbaj, puteți începe cu acest curs practic [https://developer.android.com/courses/kotlin-bootcamp/overview]. De asemenea, o serie de exerciții interactive este disponibilă aici [https://kotlinlang.org/docs/koans.html].

Un exemplu ce ilustrează interoperabilitatea dintre Java și Kotlin poate fi găsit aici [https://github.com/oop-pub/oop-labs/tree/master/src/lab12/kotlin].

Exerciții

Laboratorul trebuie rezolvat pe platforma LambdaChecker, fiind găsit aici [https://beta.lambdachecker.io/contest/23]

Vi se dă scheletul pentru o aplicație ce simulează o situație mai mult sau mai puțin fictivă. Mai multe firme cooperează cu o bancă pentru a oferi salariul și beneficiile salariaților lor. Firmele au mai mulți salariați și mai multe proiecte în istoric. Este posibil ca unii salariați să fi schimbat firma la care lucrează, dar li se pastrează în istoricul personal proiectele la care au lucrat. Un salariat poate avea mai multe conturi bancare făcute prin firma la care lucrează.

Aveți de implementat:

- TODO-urile din clasele `Business`, `Bank` si `Employee` - întoarceți tipurile corespunzatoare de date, dar asigurați-vă că sunt imutabile (hint: `Collections.unmodifiable*`)
- metodele statice din clasele `BankReport` și `BusinessReport`

Pentru testare sunt folosite 10 teste, fiecare dintre acesta reprezentand **10 puncte.**

Dezambiguizare:

- Customer = Employee of the Business
- Business = a client of the Bank
- Customers of the Bank = all the Employees that work for the Businesses that are clients of the Bank

Resurse

- Features din Java 8 [https://www.journaldev.com/2389/java-8-features-with-examples]
- Features din Java 10 [https://www.journaldev.com/20395/java-10-features]
- Features din Java 12 [https://www.journaldev.com/28666/java-12-features]
- Features din Java 17 [https://www.baeldung.com/java-17-new-features]
- When to use var in Java 10 [https://www.lewuathe.com/when-to-use-var-in-java-10.html]
- Streams [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/stream/package-summary.html]

poo-ca-cd/laboratoare/java-features.txt · Last modified: 2024/01/10 09:13 by andrei.vasiliu2211