Metode avansate de programare

Informatică Româna, 2017-2018

Curs 4 - Java 8 features

Interfețe cu o singură metodă abstractă

- **SAM Interfaces**
  - `java.lang.Runnable, void run();`
  - `java.awt.event.ActionListener, void actionPerformed(ActionEvent e);`
  - `java.util.Comparator, int compare(T o1, T o2);`
  - `java.util.concurrent.Callable, V call() throws Exception`
  - …. Si multe altele definite in JDCK sau de către programatori
- **Ce au in comun toate aceste interfete?**
  - Declara o singură metodă abstractă (de obicei, cu numele unor verbe precum: run, execute, perform, apply, compare,…..)

O singură ”metodă abstractă”?

Metode default în interfețe

```
@FunctionalInterface
interface Formula {
   double pi=3.14;
   double calculate(double a, double b);

   default double sqrt(double a) {
      return Math.sqrt(a);
   }
   default double power(double a, double  b) {
      return Math.pow(a, b);
   }
   default double numarLaPatrat(double nr)
   {
      return power(nr,2);
   }
   default double numarLaCub(double nr)
   {
      return power(nr,3);
   }
}
```

- Java 8 permite interfețelor adaugarea metodelor care nu sunt abstracte precum și a constantelor!!!

Interfețe în contextul claselor anonime

```
Formula patratBinomAnonim=new Formula(){
    @Override
    public double calculate(double a, double b) {
        return numarLaPatrat(a+b);
    }
};
c=patratBinomAnonim.calculate(a, b); //2+3 la patrat
System.out.format("(%.0f +%.0f)^2=%.0f",a,b,c);
```

Codul este lipsit de concizie, prea complicat!!!!

Interfețe funcționale

- O interfaţă funcţională (functional interface) este orice interfaţă ce conţine doar o metodă abstractă.
- Astfel putem omite numele metodei atunci când implementăm interfaţa şi putem elimina folosirea claselor anonime. În locul lor vom avea **lambda expresii** sau **referințe la metode**
- O interfaţă funcţională este anotată cu **`@FunctionalInterface`**

```
@FunctionalInterface
interface Formula {
   double pi=3.14;
   double calculate(double a, double b);

   default double sqrt(double a) {
      return Math.sqrt(a);
   }
}
```

Utilizarea interfețelor funcționale (SAM interface)

```
Formula f=referintaLaOMetoda sau oExpresie
```

Este posibil datorită faptului că avem o singură metodă abstractă in interfața Formula.

Referințe la metode

```
class FormuleMatematiceUzuale{
   public static double PatratBinom(double x, double y){ return Math.pow(x+y,2); }
   public static double CubBinom(double x, double y){ return Math.pow(x+y,3);}
   public double Suma(double x, double y) {return x+y;}
}
```

```
Formula f1=FormuleMatematiceUzuale::PatratBinom;
double patratBinom=f1.calculate(2,3); //(2+3)^2

f1=FormuleMatematiceUzuale::CubBinom;
double cubBinom=f1.calculate(2,3); //(2+3)^3

f1=new FormuleMatematiceUzuale()::Suma;
double suma=f1.calculate(2,3);

System.out.printf("(%d + %d)^2=%.0f %n",2,3,patratBinom);
System.out.printf("(%d + %d)^3=%.0f %n",2,3,cubBinom);
System.out.printf("(%d + %d)^3=%.0f %n",2,3,suma);
```

Referințe la constructori

```
interface StudentFactory<S extends Student> {
    S create(int id, String nume, float media);
}
```

```
//referinte la constructori
StudentFactory<Student> studentFactory=Student::new;
studentFactory.create(1, "POp", 8.9f);
```

Funcții lambda

- O functie lambda (funcţie anonimă) este o funcţie definită şi apelată fără a fi legată de un identificator.

- Funcţiile lambda sunt o formă de funcţii ,,incuibate” (nested functions) în sensul că permit accesul la variabilele din domeniul funcţiei în care sunt conţinute.

Funcții Lambda exemple

```
@FunctionalInterface
interface Formula {
    double calculate(int a, int b);
}
…
```

```
Formula patratulBinomuluiLambda1=(a, b)->{ return Math.pow(a+b,2);};
```

```
Formula patratulBinomuluiLambda1=(a, b)->Math.pow(a+b,2);
```

Lambda. Domenii de accesibilitate

- Expresiile lambda pot avea acces la:
  - Variabilele statice
  - Variabile de instanta
  - Parametrii metodelor
  - Variabilele locale

- Amintiti-va cum era in cazul caselor anonime.

Accesarea variabilelor locale

```
public static void locVariable()
{
	int patrat=2;
	Formula patratulBinomuluiLambda1=(double a, double b)->{
		// patrat=5; eroare
		return Math.pow(a+b,patrat);
	};
	double res1=patratulBinomuluiLambda1.calculate(3.1,5);
	System.out.printf("(%d + %d)^2=%.0f %n",3,5,res1);
}
```

Putem referi variabile locale in functia lambda, dar acestea sunt implicit **final** (nu le putem modifica)

Accesarea membrilor de clasa și de instanță

```
class FormuleMatematice{
    private static int outerStaticPutere=1;
    private int outerPutere=1;
    public double PatratBinom(double x, double y){
        Formula f=(a,b)->{ outerPutere=2; return Math.pow(a+b,outerPutere);};
        return f.calculate(x,y);
    }
    public double CubBinom(double x, double y){
        return f.calculate(x,y);
    }
    Formula f=(a,b)->{ outerStaticPutere=3; return Math.pow(a+b,outerStaticPutere);};
}
```

In contrast cu variabilele locale, variabilele de clasa si cele de instanta pot fi accesate si modificate in functii lambda.

Accesare metodelor default în funcții lambda

```
interface Formula {
    double pi=3.14;
    double calculate(double a, double b);

    default double numarLaPatrat(double nr)
    {
        return power(nr,2);
    }
}
```

```
Formula patratBinomLambda2=(x,y)->numarLaPatrat(x+y); // eroare
```

In functii lambda nu putem accesa metode default din interfata

Built-in Functional Interfaces

- Predicates
- Functions
- Suppliers
- Consumers
- Comparators

Lista nu e exhaustivă! Vom mai folosi si altele ....

Predicate

https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html

- Predicatele sunt functii de un singur argument care intorc o valoare logica
- Verbul sugestiv pentru metoda abstractă: test

```
public static void printStudents(List<Student> studs, Predicate<Student> cond)
{
   for(Student s:studs)
      if (cond.test(s)) System.out.println(s);
   System.out.println();
}

public static void runExamplePredicate()
{
   List<Student> students=Arrays.asList(new Student(1,"Pop",3.6f), new Student(2,"Dan",8.6f),
   new Student(3,"Ana",5.6f));
   Predicate<Student> promovati;
   promovati=x->x.getMedia()>=5;   //functie lambda
   printStudents(students, promovati);

   promovati=PredicateStudents::isPromovat;   //referinta la metoda
   printStudents(students, promovati);
```

Predicate continuare

https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html

Modifier and Type	Method and Description
default Predicate<T>	and(Predicate<? super T> other) Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
static <T> Predicate<T>	isEqual(Object targetRef) Returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object).
default Predicate<T>	negate() Returns a predicate that represents the logical negation of this predicate.
default Predicate<T>	or(Predicate<? super T> other) Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
boolean	test(T t) Evaluates this predicate on the given argument.

```
Predicate<Student> promovatiSiIncepCuA=promovati.and(x->x.getNume().startsWith("A"));
printStudents(students, promovatiSiIncepCuA);
```

Predicate exemplu

```
Student s=new Student(3,"Ana",5.6f);
Predicate<Student> nonNull = Objects::nonNull;
System.out.println(nonNull.test(s));  //true


Predicate<Student> isNull = Objects::isNull;
System.out.println(isNull.test(s));  //false

Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNotEmpty = isEmpty.negate();
System.out.println(isEmpty.test("abracadrabra")); //false
```

Functions

https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html

- Funcțiile acceptă un argument și returnează o valoare
- Verbul sugestiv pentru metoda abstractă: apply

Modifier and Type	Method and Description
`default <V> Function<T,V>`	**`andThen`**`(Function<? super R,? extends V> after)` Returns a composed function that first applies this function to its input, and then applies the `after` function to the result.
`R`	**`apply`**`(T t)` Applies this function to the given argument.
`default <V> Function<V,R>`	**`compose`**`(Function<? super V,? extends T> before)` Returns a composed function that first applies the `before` function to its input, and then applies this function to the result.
`static <T> Function<T,T>`	**`identity`**`()` Returns a function that always returns its input argument.

Functions exemplu

```
Function<String,Integer> toIntegerLambda= x->Integer.valueOf(x);
Function<String,Integer> toIntegerMethodReference=Integer::valueOf;

Integer fromString=toIntegerLambda.apply("12");
Integer fromString2=toIntegerMethodReference.apply("12");
System.out.println(fromString);
System.out.println(fromString2);

Function<String, String> backToString = toIntegerLambda.andThen(String::valueOf);
String s=backToString.apply("123");

Function<Double, Double> lg=(x)->Math.log10(x);
Function<Double,Double> compose=lg.compose(x->x*x);  //lg(x) compus cu x*x
System.out.println(compose.apply(10d));
```

Suppliers

- Produc un rezultat de un anumit tip generic. Spre deosebire de functii, nu admit nici un argument.
- Verbul sugestiv pentru metoda abstractă: get

Exemplu1: Referinta la constructor

```
Supplier<ArrayList> methodRef4 = ArrayList::new;
Supplier<ArrayList> lambda4 = () -> new ArrayList();

Supplier<ArrayList<String>> s1 =ArrayList<String>::new;
ArrayList<String> a1 = s1.get();
System.out.println(a1); //se va tipari lista vida
```

Exemplu 2: Generarea de valori fara data input

```
Supplier<LocalDate> s1 = LocalDate::now;
Supplier<LocalDate> s2 = () -> LocalDate.now();
LocalDate d1 = s1.get();
LocalDate d2 = s2.get();
System.out.println(d1);
System.out.println(d2);
```

Consumers

https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html

- Operatii effectuate pe un singur argument.
- Verbul sugestiv pentru metoda abstractă accept

```
Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);
greeter.accept(new Person(“Aprogramatoarei", “Dan"));
```

Comparatori

https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

- Verbul: compare

Modifier and Type	Method and Description
int	`compare(T o1, T o2)` Compares its two arguments for order.

```
List<Student> students= Arrays.asList(new Student(1,"Pop",3.6f), new Student(2,"Dan",8.6f),
      new Student(3,"Ana",5.6f));
Comparator<Student> comparatorMedie=(x,y)-> (int)(x.getMedia()-y.getMedia());
Comparator<Student> comparatorNume=(x,y)-> x.getNume().compareTo(y.getNume());

students.sort(comparatorMedie);
System.out.println(students);

students.sort(comparatorNume);
System.out.println(students);
```

Comparator clase anonime

```
List<Student> students2= Arrays.asList(new Student(1,"Pop",3.6f), new
Student(2,"Dan",8.6f),
      new Student(3,"Ana",5.6f));

students2.sort(new Comparator<Student>() {
   @Override
   public int compare(Student o1, Student o2) {
      return o1.getId()-o2.getId();
   }
});
```

Comparator – referință la metode

```
public class Student implements Comparable<Student>{
  . . .
    public static int comparaMedia(Student a, Student b)  { . . . }
}
```

```
Collections.sort(studs, Student::comparaMedia);
```

?

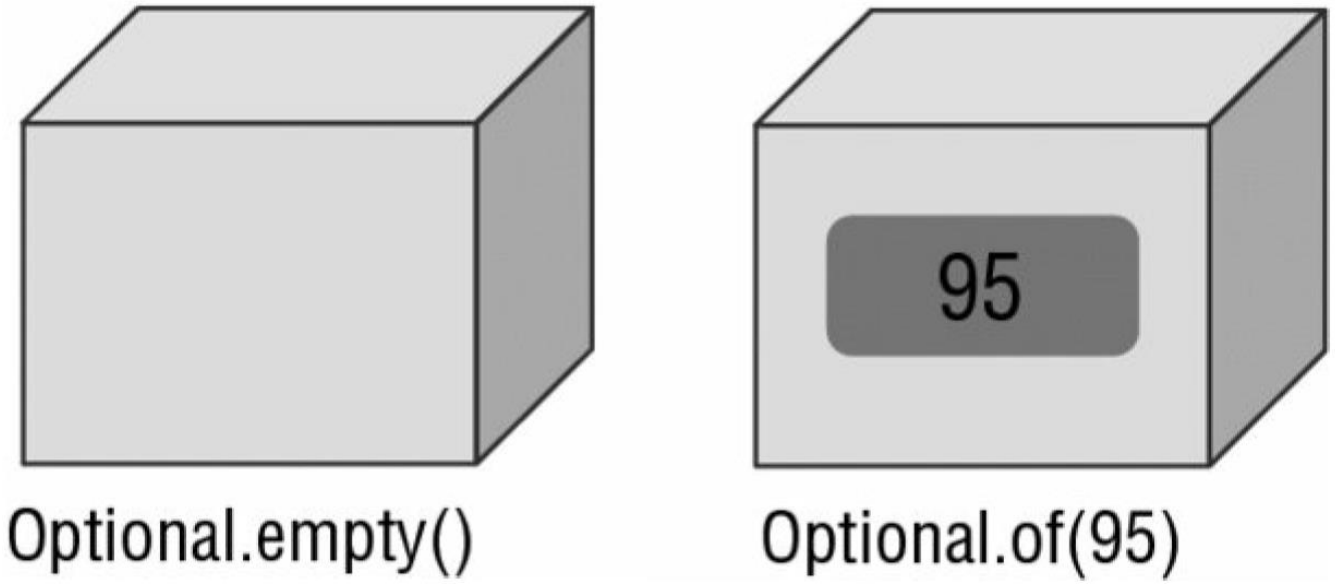
```
1: _____<ArrayList<String> ex1 = x -> "".equals(x.get(0));
2: _____<Long> ex2 = (Long l) -> System.out.println(l);
3: _____ <String, Boolean> ex3 = (s1) -> s1.contains("a");
```

R

```
Predicate<ArrayList<String>> ex1 = x -> "".equals(x.get(0));
Consumer<Long> ex2 = (Long l) -> System.out.println(l);
Function<String, Boolean> ex3 = (s1) -> s1.contains("a");
```

Optionals

- Un container care retine o valoare sau nu retine nimic
- Previne NullPointerException
- Nu este interfata functională!!!



Optional exemplu

```
public static Optional<Double> average(int... scores) {
    if (scores.length == 0) return Optional.empty();
    int sum = 0;
    for (int score: scores) sum += score;
    return Optional.of((double) sum / scores.length);
}
```

```
Optional<Double> avg=average(1,2,3);
if(avg.isPresent())
   System.out.println(avg.get()); //2
// sau
avg.ifPresent(System.out::println);
//sau
avg.ifPresent((x)->System.out.println(x));
```

Optional exemplu AbstractRepository

```
@Override
public Optional<E> delete(ID id) {
        return Optional.ofNullable(entities.remove(id));
}

@Override
public Optional<T> update(T entity) throws ValidatorException {
        validator.validate(entity);
        if (entities.containsKey(entity.getId())) {
             entities.put(entity.getId(), entity);
             return Optional.empty();
        }
        return Optional.of(entity);
}
```

Metode adiționale pentru colecții

Iterarea colecțiilor

```
//using an iterator
List<String> names = Arrays.asList("Ana", "Ioana");
for(String n: names)
    System.out.println(n);

//forEach lambda
names.forEach(c -> System.out.println(c));

//forEach method reference
names.forEach(System.out::println);
```

Metode adiționale pentru colecții

```
boolean removeIf(Predicate<? super E> filter); // remove conditionally
```

```
List<String> list1=new ArrayList<String>
        (Arrays.asList("ana", "alearga", "la ", "sala"));
list1.removeIf(x->x.startsWith("a"));
list1.forEach(x -> System.out.println(x));

//Method reference on a specific instance:
String pref="sa";
list1.removeIf(pref::startsWith);
```

streams

- Un java.util.Stream reprezintă o secvență de elemente pe care se pot efectua una sau mai multe operații.
- Operațiile pe stream(flux) sunt fie intermediare fie terminale.
- În timp ce operațiile terminale returnează un rezultat al unui anumit tip, operațiile intermediare returnează fluxul în sine, astfel încât se pot inlănțui mai multe operații pe flux.
- Fluxurile sunt create pe o sursă, de exemplu, un java.util.Collection cum ar fi List sau Set (dictionarele nu sunt acceptate).
- Operațiunile pe flux pot fi executate fie secvențial sau parallel (streamuri seriale sau paralele.

Operații pe stream

- Filter – operatie intermediară (returnează fluxul în sine)

```
List<String> stringCollection =
      Arrays.asList("ddd2", "aaa2","bbb1","aaa1","bbb3",
            "ccc1", "bbb2","ddd1","aaa3","aaa4");
List<String> res=stringCollection
      .stream()
      .filter(x->x.startsWith("a"))
      .collect(Collectors.toList());
```

collect este o operație finală utilizată pentru a transforma elementele fluxului într-un alt tip de rezultat.

Operații pe stream

- Map – operatie intermediară (Convertește fiecare elem din stream într-un alt obiect (conform unei funcții)

```
List<String> stringCollection =
      Arrays.asList("ddd2", "aaa2","bbb1","aaa1","bbb3",
            "ccc1", "bbb2","ddd1","aaa3","aaa4");
//List<String> res=stringCollection
stringCollection
      .stream()
      .filter(x->x.startsWith("a"))
      .map(x->x.toUpperCase())
      .forEach(System.out::println);
      //.collect(Collectors.toList());
```

Operații pe stream

- Sorted – operatie intermediară, returnează o vedere ordonată a stream-lui.

```
List<String> stringCollection =
      Arrays.asList("ddd2", "aaa2","bbb1","aaa1","bbb3",
            "ccc1", "bbb2","ddd1","aaa3","aaa4");
//List<String> res=stringCollection
stringCollection
      .stream()
      .filter(x->x.startsWith("a"))
      .map(x->x.toUpperCase())
      .sorted((x,y)->x.compareTo(y))
      .forEach(System.out::println);
```

Operații pe stream

- Reduce – operație finală, determină o reducere a elem. stream-lui. Poate avea două argumente (un elem neutru și o expresie lambda). Returnează un Optional dacă specificarea elem neutru lipsește).

```
List<String> list =
      Arrays.asList("ddd2", "aaa2","bbb1","aaa1","bbb3",
            "ccc1", "bbb2","ddd1","aaa3","aaa4");
Optional<String> op=list
      .stream()
      .filter(x->x.startsWith("a"))
      .reduce( (x,y) -> x.concat(y));
op.ifPresent(System.out::println);

  //    if (op.isPresent())
  //       System.out.println(op.get());
```

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

Reduce

- Exemplu: Determină valoarea maximă

```
Integer[] l = {1, 9, 5, 2 };
Optional<Integer> res;
res = Arrays.stream(l)
          .reduce((x, y)-> x>y ? x :y);
res.ifPresent(System.out::println);
```

Operații pe stream

- Match – operație finală (returneaza true/false) anyMatch, allMatch,

```
boolean anyStartsWithA =
        Arrays.asList("ddd2", "aaa2","bbb1","aaa1","bbb3",
        "ccc1", "bbb2","ddd1","aaa3","aaa4")
        .stream()
        .anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA);      // true

boolean allStartsWithA =
        Arrays.asList("ddd2", "aaa2","bbb1","aaa1","bbb3",
                "ccc1", "bbb2","ddd1","aaa3","aaa4")
        .stream()
        .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA);      // false
```

Operații pe stream

- Match – operație finală (returneaza true/false) - noneMatch

```
boolean noneStartsWithZ =
        Arrays.asList("ddd2", "aaa2","bbb1","aaa1","bbb3",
                "ccc1", "bbb2","ddd1","aaa3","aaa4")
        .stream()
        .noneMatch((s) -> s.startsWith("z"));
System.out.println(noneStartsWithZ);      // true
```

Operații pe stream

- **Count** – operatie terminală– returnează numărul de elemente din stream (long).

```
long startsWithB =
        Arrays.asList("ddd2", "aaa2","bbb1","aaa1","bbb3",
                "ccc1", "bbb2","ddd1","aaa3","aaa4")
                .stream()
                .filter((s) -> s.startsWith("b"))
                .count();

System.out.println(startsWithB);    // 3
```

Citirea/Scrierea din/în fișier – Java NIO and Stream

```
public static void readWriteStuds()
{
   Path path = Paths.get("./src/data/Studs.txt");
   Stream<String> lines;
   try {
      lines = Files.lines(path);   //Files – helper class
      lines.forEach(s -> System.out.println(s));
   } catch (IOException e) { . . . }

   try (BufferedWriter bufferedWriter = Files.newBufferedWriter(path, StandardOpenOption.APPEND)) {
      bufferedWriter.write(student.toString());
      bufferedWriter.newLine();
   } catch (IOException e) { e.printStackTrace() }
}
```

Cursul următor

- Reflecție în Java