

# Metode avansate de programare

---

Informatică Româna, 2020-2021

Curs 4 - Java 8 features

Ref utile:

- <https://winterbe.com/posts/2014/03/16/java-8-tutorial/>
- <https://www.baeldung.com/java-8-new-features/>

# Pb Organizatorice

- Prima cerinta din proiectul extins va fi in Sapt 5 – citirea din baza de date (RepositoryDB)
- Grupa 221 – planificarea intalnire pt compunere de grille in Sapt 4

# Metode default sau statice în interfețe

```
interface AritmeticExpression{
    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);}

    default double patratBinom(double x, double y){ return Math.pow(x+y,2); }

    static double cubBinom(double x, double y){ return Math.pow(x+y,3); }

    static double suma(double x, double y) {return x+y;}
}
```

Java 8 permite interfețelor adaugarea **metodelor care nu sunt abstracte (default sau statice)** precum și a constantelor!!!

# Interfețe în contextul claselor interne anonime

```
AritmeticExpression patratBinom=new AritmeticExpression() {  
    @Override  
    public double calculate(double a, double b) {  
        return patratBinom(a,b);  
    }  
};
```

```
double a=2.1, b=2.2;  
double res=patratBinom.calculate(a,b);  
System.out.format("(%.2f+%.2f)^2=%.2f",a,b,res);
```

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

# Interfețe funcționale

- O interfață funcțională (functional interface) este orice interfață ce conține doar **o metodă abstractă**.
- Astfel **putem omite numele metodei** atunci când implementăm interfața și putem elimina folosirea claselor anonime. În locul lor vom avea **lambda expresii** sau **referințe la metode**
- O interfață funcțională este adnotată cu **@FunctionalInterface**

```
@FunctionalInterface
interface ArithmeticExpression {
    double calculate(double a, double b);
    // others default or static methods
}
```

# Utilizarea interfețelor funcționale

AritmeticExpression  $f$ =referintaLaMetoda sau oExpresie  
sau o functie lambda

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

# Referință la o metodă de clasă

```
@FunctionalInterface
interface AritmeticExpression {
    double calculate(double a, double b);
    // others default or static methods
}
```

```
class FormulaHelper{

    public static double patratBinom(double x, double y){
        return Math.pow(x+y,2);
    }
}
```

```
AritmeticExpression bin2=FormulaHelper::patratBinom; //method reference - static method
                                                         patratBinom

double a=2.1, b=2.2;
System.out.format("(%.2f+%.2f)^2=%.2f",a,b,bin2.calculate(a,b));
```

# Referință la o metodă de instanță

```
@FunctionalInterface
interface AritmeticExpression {
    double calculate(double a, double b);
    // others default or static methods
}
```

```
class FormulaHelper{
    private double a;
    private double b;

    public FormulaHelper(double a, double b) { this.a = a;this.b = b; }

    public double distanta(double x, double y){
        return Math.sqrt(Math.pow(x-a,2) +Math.pow(y-b,2));
    }
}
```

```
FormulaHelper helper=new FormulaHelper(a,b);
Formula dist=helper::distanta;
System.out.format("d(A(%.2f,%.2f),B(%.2f,%,2f))=%.2f",a,b,a,b,dist.calculate(a,b));
```



# Referință la o metodă de instanță .....

- "Reference to an instance method of an arbitrary object of a particular type"

```
class Boeing {  
    int height;  
  
    public Boeing(int height) {  
        this.height = height;  
    }  
  
    public int getHeight() {  
        return height;  
    }  
}
```

```
@FunctionalInterface  
interface Flyable<T> {  
    int canFly(T t); // the hight reached by T  
}
```

```
Flyable<Boeing> f=Boeing::getHeight;  
int n=f.canFly(new Boeing(23));
```

# Referință la constructor

```
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 funcție lambda (funcție anonimă) este o funcție definită și apelată fără a fi legată de un identificador.
- Funcțiile lambda sunt o formă de funcții „încuibate” (nested functions) în sensul că **permit accesul la variabilele din domeniul funcției în care sunt conținute.**

# Funcții Lambda. Exemplu

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

...

```
double a=2.1, b=2.2;
Formula f1=(x,y)->{ return FormulaHelper.patratBinom(a,b);};
System.out.format("(%.2f+%.2f)^2=%.2f",a,b,f1.calculate(a,b));
```

```
FormulaHelper helper=new FormulaHelper(a,b);
Formula f2=(x,y)->helper.distanta(a,b);
System.out.format("d(A(%.2f,%.2f),B(%.2f,%,2f))=%.2f",a,b,a,b,f1.calculate(a,b));
```

# Lambda. Domenii de accesibilitate

- Expresiile lambda pot avea acces la:
  - Variabilele statice
  - Variabile de instanță
  - Parametrii metodelor
  - Variabilele locale
- Amintiti-vă cum era în 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("(3 + 5)^2=%.0f %n",3,5,res1);  
}
```

Putem referi variabile locale în funcția lambda, dar acestea sunt implicit **final** (nu le putem modifica).

# Accesarea membrilor de clasă 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);};
}
```

În contrast cu variabilele locale, **variabilele de clasă și cele de instanță** pot fi accesate și modificate în funcții lambda.

# Built-in Functional Interfaces

- Predicates
- Functions
- Suppliers
- Consumers
- Comparators
- ...



# Predicate

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

- Predicatele sunt funcții de un singur argument care întorc o valoare logică
- Verbul sugestiv pentru metoda abstractă: **test**

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

```
Predicate<Student> estePromovat=x->x.getMedia()>=5; //Lambda function
```

```
Predicate<Student> estePromovat2=StudentHelper::promovat; //method reference  
System.out.println(estePromovat.test(new Student(24,"Birlanescu",4.6f)));
```

# Predicate – default, static methods

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

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

```
Predicate<Student> promovatSiIncepeCuA=estePromovat.and(x->x.getNume().startsWith("A"));
System.out.println(estePromovat.test(new Student(24,"Birlanescu",4.6f))); //false
```

# Predicate. Alte Example

```
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
```

# Consumer

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

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

## Method Summary

All Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type		Method and Description	
void		<b>accept</b> (T t)	Performs this operation on the given argument.
default	<b>Consumer</b> <T>	<b>andThen</b> ( <b>Consumer</b> <? super T> after)	Returns a composed Consumer that performs, in sequence, this operation followed by the after operation.

```
Consumer<Student> consumer=System.out::println; //method reference
consumer.accept(new Student(123,"Dan",4.5f));
```

```
Consumer<Student> consumer2=x-> System.out.println(x); //Lambda
consumer.accept(new Student(123,"Dan",4.5f));
```

```
Consumer<Student> consumer3=Student::toString; //method reference
consumer.accept(new Student(123,"Dan",4.5f));
```

# Metode adiționale colecțiilor - forEach, removeIf

```
List<Student> list= new ArrayList(Arrays.asList(  
    new Student(22, "Aprogramatoarei", 5.6f),  
    new Student(23, "Popescu", 9.6f),  
    new Student(24, "Birlanescu", 8.6f)));
```

```
list.forEach(x-> System.out.println(x));  
list.forEach(System.out::println);
```

```
Predicate<Student> estePromovat=x->x.getMedia()>=5;  //lambda function
```

```
list.forEach(x-> {if (estePromovat.test(x)) System.out.println(x);} );
```

```
list.removeIf(estePromovat.negate());
```

```
list.forEach(System.out::println);
```

# Exerciții

- Să se șteargă dintr-o listă de șiruri de caractere, toate elementele care încep cu “a”.
- Să se șteargă dintr-o listă de șiruri de caractere, toate elementele care sunt prefixele unui cuvânt. De exemplu “Anamaria”. Folositi funcție lambda si referință la metodă.
- Să se șteargă dintr-o listă de șiruri de caractere, toate elementele care conțin șirul vid. Folositi funcție lambda si referință la metodă.
- Să se șteargă toți studenții a căror nume începe cu “B”, dintr-o listă de studenți.

# Function

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

# Function. Exemplan

```
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));
```



# UnaryOperator

- Caz particular de functie - Function<T,T>

```
UnaryOperator<String> uo1 = String::toUpperCase;  
UnaryOperator<String> uo2 = x -> x.toUpperCase();
```

```
System.out.println(uo1.apply("ana"));  
System.out.println(uo2.apply("ana"));
```

# BinaryOperator

- Caz particular de BiFunction - BiFunction<T,T,T>

```
BinaryOperator<String> bo1 = String::concat;  
BinaryOperator<String> bo2 = (x,y)->x.concat(y);
```

```
System.out.println(bo1.apply("ana", " blandiana"));  
System.out.println(bo2.apply("ana", " blandiana"));
```

# Supplier

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

## Exemplu 1: 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);
```

# Comparatori

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

- Verbul: **compare**

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

```
public class StudentHelper{
    public static int compareByAverage(Student a, Student b)
    {
        return (int) (a.getMedia()-b.getMedia());
    }

    public static int compareByName(Student a, Student b)
    {
        return (int) a.getNume().compareTo(b.getNume());
    }

    public static int compareById(Student a, Student b)
    {
        return (int) a.getNume().compareTo(b.getNume());
    }
}
```

## Exerciții:

- 1) Folosiți clasa *StudentHelper* și definiți comparatori pt sortarea unei liste de studenți. Folosiți referință la metodă.
- 2) Sortați o listă de studenți, definind comparatori ca funcții lambda.

# Exerciții

- 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"`);

# Rezolvări

```
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");
```

# Metode adiționale pe Map

## putIfAbsent

```
@Override
public E save(E entity) throws ValidationException {
    if (entity==null)
        throw new IllegalArgumentException("entity must be not null");
    validator.validate(entity);
    return entities.putIfAbsent(entity.getID(),entity);
}
```

# Metode adiționale pe Map

## computeIfAbsent

```
Function<String, Integer> mapper3 =  
    (v)-> 1;
```

```
Map<String, Integer> countsIfAbs = new HashMap<>();  
counts.put("Jenny", 15);  
Integer jenny3=counts.computeIfAbsent("John",mapper3);  
System.out.println(counts); //{Jenny=15}  
System.out.println(jenny3);
```



# Metode adiționale pe Map

## computeIfPresent

```
BiFunction<String, Integer, Integer> mapper2 =  
    (v1, v2)-> v2+1;  
  
Map<String, Integer> counts = new HashMap<>();  
counts.put("Jenny", 1);  
Integer jenny2=counts.computeIfPresent("Jenny",mapper2);  
System.out.println(counts); //{Jenny=2}
```

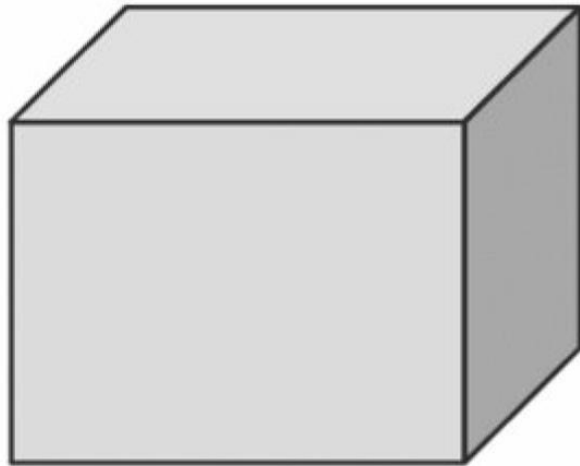
# Metode adiționale pe Map

## merge

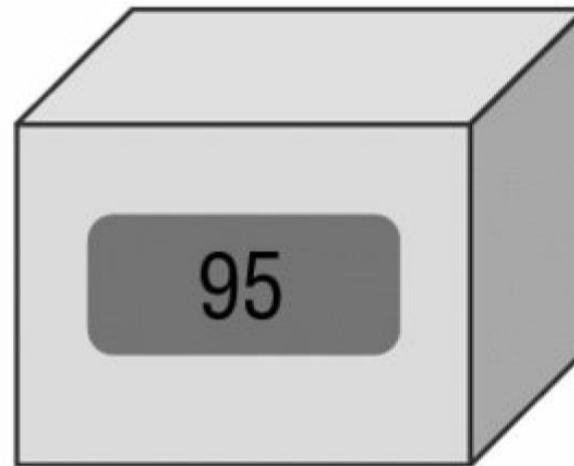
```
BiFunction<String, String, String> mapper =  
    (v1, v2)-> v1.length() > v2.length() ? v1: v2;  
  
Map<String, String> favorites = new HashMap<>();  
favorites.put("Jenny", "Bus Tour");  
favorites.put("Tom", "Tram");  
  
String jenny = favorites.merge("Jenny", "Skyride",  
    mapper );  
String tom = favorites.merge("Tom", "Skyride",  
    mapper );  
  
System.out.println(favorites); // {Tom=Skyride, Jenny=Bus Tour}  
System.out.println(jenny); // Bus Tour  
System.out.println(tom); // Skyride
```

# Optional

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



`Optional.empty()`



`Optional.of(95)`

# 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);  
}
```

# Stream

- Un *java.util.Stream* reprezintă o secvență de elemente pe care se pot efectua una sau mai multe operații.
- Operațiile pe Stream sunt fie intermediare fie terminale.
- În timp ce operațiile terminale returnează un rezultat de un anumit tip, operațiile intermediare returnează fluxul în sine, astfel încât se pot înlanțui 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 *secvențial* sau *parallel* (stream-uri seriale sau paralele).
- <https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>

# Operații pe stream

- **Filter** – operație **intermediară** (returnează fluxul în sine)

```
Stream<String> myStream=Stream.of("ddd2",  
    "aaa2", "bbb1", "aaa1", "bbb3",  
        "ccc1", "bbb2", "ddd1", "aaa3", "aaa4");  
myStream  
    .filter(s -> s.startsWith("c"))  
    .foreach(System.out::println);
```

*foreach* - este o operație finală

# Operații pe stream

- **Map** – operație **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());
```

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



# Operații pe stream

- **Sorted** – operație **intermediară**, returnează o vedere ordonată a stream-ului.

```
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-ului. 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** (returnează 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** (returnează 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** – operație **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.
- JDBC
- Interfețe grafice.