



Academia Tehnică Militară "Ferdinand I"
Facultatea de Sisteme Informaticе și Securitate Cibernetică

PARADIGME DE PROGRAMARE (ÎN JAVA)

CURS 5 - PROGRAMARE FUNCȚIONALĂ

COL(R) TRAIAN NICULA

TEMATICĂ DE CURS

- Introducere în paradigme de programare (1)
- Programare orientată pe obiecte (OOP) (3)
- **Programare funcțională și asincronă (1/3)**
- Programare reactivă (1)
- Servicii web REST (2)
- Quarkus Framework (4)

CUPRINS CURS

- Java 8 - Noutăți
- Expresii Lambda
- Introducere în Stream-uri
- Stream-uri în detaliu
- Bibliografie



JAVA 8 - NOUTĂȚI

- Evoluția Java: 1.0 în 1996, 1.1 în 1997, 7 în 2011, 8 în 2014, 11 în 2018, 17 în 2021
- **Java 8 aduce cele mai importante schimbări din istoria Java cu scopul:**
 - **Reducerii verbozității**

înainte de Java 8

```
Collections.sort(inventory, new Comparator<Apple>() {  
    public int compare(Apple a1, Apple a2){  
        return a1.getWeight().compareTo(a2.getWeight());  
    }  
});
```

în Java 8 și după

```
inventory.sort(comparing(Apple::getWeight))
```

- **Utilizării eficiente a hardware-ului** (procesare paralelă pe CPU cu mai multe nuclee) fără a utiliza direct fire de execuție (threads)
- **Necesității de a procesa cantități mari de date** (big data) de la terabaiți în sus
- **Introduce Streams API** care **suportă operații paralele** pentru procesarea datelor și reasamblarea acestora **în mod declarativ** (similar SQL)
- **Permite trecerea de cod metodelor** (behavior parameterization) deschizând calea către **programarea funcțională** în Java



JAVA 8 - NOUTĂȚI

- Un **stream** este o secvență de elemente de date produsă la un moment dat, pe care un program le citește una câte una și le scrie într-un **stream** de ieșire
- Concept asemănător în Linux pentru **înlănțuirea comenziilor cu operatorul pipe (|)**

```
cat file1 file2 | tr "[A-Z]" "[a-z]" | sort | tail -3
```

- *cat* creează un stream de cuvinte, *tr* convertește caracterele, *sort* sortează cuvintele iar *tail -3* returnează ultimele 3 cuvinte din stream
- **Parametrizarea comportării sau capacitatea de a trece cod unei metode**, se traduce în exemplul din Linux în abilitatea de a trece comenzi *sort* parametrii linie comandă pentru diferite tipuri de sortări (acestea sunt predefinite)

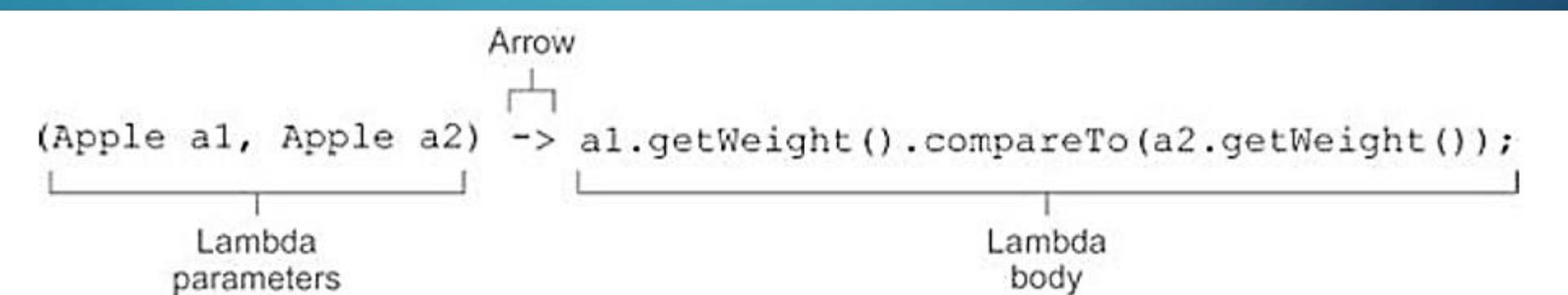
JAVA 8 - NOUTĂȚI

- **Paralelismul în Java 8** constă în abilitatea de a implementa **cod care se poate executa paralel și în siguranță pe bucăți** din stream-ul de intrare
- Acesta înseamnă că **programul nu are nevoie de acces concurrent la datele de intrare și nu le modifică** (*shared mutable data access*) ca în cazul firelor de execuție
- În **argoul programării funcționale** acestea poartă **numele de funcții pure**, sau funcții fără efecte secundare sau funcții fără stare
- **Funcțiile în Java 8 sunt tipuri noi de date** (și nu un sinonim pentru metode) pentru a facilita folosirea stream-urilor
- Java 8 introduce **tipul referințe de metode** (method reference) care trece codul metodei ca valoare
- Exemplu: *File::isHidden* poate fi trecut ca valoare



EXPRESII LAMBDA

- Reprezintă **funcții anonte** care au doar o listă de parametrii, corp și un tip returnat
- **Expresiile lambda nu au nume**
- Sunt funcții pentru că **nu sunt asociate cu nici o clasă**
- Pot fi **transmise ca argument** sau pot fi **stocate într-o variabilă**
- Expresie lambda:



EXPRESII LAMBDA

- Sintaxa:
(parameters) -> expression
- sau
(parameters) -> { statements; }
- Exemple de expresii lambda

```
// boolean expression
(List<String> list) -> list.isEmpty()
// creates an object
() -> new Apple(10)
// consumes an object
(Apple a) -> {
    System.out.println(a.getWeight());
}
// extracts from object
(String s) -> s.length()
(int a, int b) -> a * b
```



EXPRESII LAMBDA

INTERFEȚE FUNCȚIONALE

- *Interfața funcțională* este o interfață care specifică **numai o metodă abstractă**
- Exemple:

```
public interface Comparator<T> {           ← java.util.Comparator
    int compare(T o1, T o2);
}

public interface Runnable{                  ← java.lang.Runnable
    void run();
}
```

- **Expresia lambda reprezintă o implementare a metodei abstracte a interfeței funcționale *inline*** (se poate realiza același lucru cu clase anonte)
- **Semnătura metodei interfeței funcționale descrie semnătura expresiei lambda**
- Metoda abstractă poartă numele de **descriptorul funcției**



EXPRESII LAMBDA INTERFEȚE FUNCȚIONALE

```
package atm.paradigms.tests;

public class Lambdas {
    public static void main(String[] args) {
        Runnable r1 = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hello World Java 7");
            }
        };
        Runnable r2 = () -> System.out.println("Hello World Java 8");
        doIt(r1);
        doIt(r2);
        doIt(() -> System.out.println("Java 8 code passing"));
    }

    public static void doIt(Runnable r){
        r.run();
    }
}
```

Diverse implementări ale metodei `run()` aparținând interfeței funcționale `Runnable`, folosind o clasă anonimă sau prin expresii lambda.

Rezultat:
Hello World Java 7
Hello World Java 8
Java 8 code passing



EXPRESII LAMBDA BEHAVIOR PARAMETRIZATION

- Un exemplu de **reutilizare a codului** îl constituie lucrul cu resurse (fișiere, baze de date), deoarece este necesar să deschidem resursa, să procesăm datele apoi să închidem resursa
- **Partea inițială și finală sunt tot timpul la fel, partea de procesare fiind cea mai importantă și care diferă**
- Exemplu Java 7 care deschide un fișier și citește o linie de cod (notă: se folosește instrucțiunea **try-with-resources**):

```
public static String processFile() throws IOException{
    try(BufferedReader br = new BufferedReader(new FileReader("data.txt"))){
        // useful line
        return br.readLine();
    }
}
```

Exemplu de deschidere a unui fișier text folosind try cu parametru (**try-with-resources**). Resursa **br** este închisă automat la finalizarea operației de citire (fără finally). Operațiile Java I/O pot genera IOException. Metoda specifică excepția.

EXPRESII LAMBDA

BEHAVIOR PARAMETRIZATION

- Codul anterior este limitat deoarece poate citi doar o linie de cod
- Dar dacă dorim să citim 2 linii de cod? Alte modificări?
- **Se implementează o interfață funcțională** care să permită diferite comportamente prin expresii lambda

```
package atm.paradigms.tests;

import java.io.BufferedReader;
import java.io.IOException;

@FunctionalInterface
public interface BufferedReaderProcessor {
    String process(BufferedReader b) throws IOException;
}
```

Interfețele funcționale sunt decorate cu adnotarea **@FunctionalInterface**. Metoda **process()** va fi înlocuită de expresii lambda cu semnătura corespunzătoare. Pentru că primește ca parametru un stream de caractere (operație I/O) metoda generează **IOException**.



EXPRESII LAMBDA BEHAVIOR PARAMETRIZATION

```

package atm.paradigms.tests;

import java.io.*;

public class Lambdas {
    public static void main(String[] args) throws IOException {
        // read one line
        String oneL = processFile((BufferedReader br) -> br.readLine());
        System.out.println(oneL);
        // read two lines
        String twoL = processFile((BufferedReader br)
            -> br.readLine() + br.readLine());
        System.out.println(twoL);
    }

    public static String processFile(BufferedReaderProcessor p) throws IOException{
        InputStream in = Lambdas.class.getResourceAsStream("data.txt");
        try(BufferedReader br = new BufferedReader(new InputStreamReader(in))){
            return p.process(br);
        }
    }
}

```

Metoda **processFile()** primește ca parametru interfața funcțională anterioară. Creează un stream de caractere pe care îl trece metodei **process()**. Se cheamă metoda **processFile()** cu 2 expresii lambda, una care returnează o linie, iar cealaltă 2 linii concatenate.

Rezultat:
line1
line1 line2



EXPRESII LAMBDA

INTERFEȚE FUNCȚIONALE UTILE

```

package atm.paradigms.tests;
import java.util.*;
import java.util.function.Predicate;

public class PredicateTest {
    public static void main(String[] args) {
        List<String> myList = new ArrayList<>();
        myList.addAll(Arrays.asList("B", "A", "", "C", ""));
        System.out.println(myList);
        Predicate<String> p = (String s) -> !s.isEmpty();
        List<String> filtered = filter(myList, p);
        System.out.println(filtered);
    }
    public static <T> List<T> filter(List<T> list, Predicate<T> p){
        List<T> results = new ArrayList<>();
        for (T s : list){
            if (p.test(s))
                results.add(s);
        }
        return results;
    }
}

```

- **Predicate** – interfață `java.util.function.Predicate<T>` definește metoda abstractă `test` care acceptă un obiect de tip `T` și returnează boolean

Rezultat:

[B, A, , C,]
[B, A, C]

Metoda generică `filter()` folosește `Predicate<T>` pentru a elimina din lista primită ca parametru anumite valori. Returnează la final lista rezultată. Metoda poate face filtrări diferite în funcție de lambda primit.

EXPRESII LAMBDA

INTERFEȚE FUNCȚIONALE UTILE

- **Consumer** – Interfață `java.util.function.Consumer<T>` definește metoda `accept` care primește obiectul de tip **T** și nu returnează nimic. Interfața se folosește când se efectuează operații pe obiecte

```
package atm.paradigms.tests;

import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class ConsumerTest {
    public static void main(String[] args) {
        forEach(Arrays.asList(1, 2, 3, 4, 5),
                (Integer i) -> System.out.println(i));
    }

    public static <T> void forEach(List<T> list, Consumer<T> c){
        for (T i : list){
            c.accept(i);
        }
    }
}
```

Rezultat:

1
2
3
4
5

Metoda generică `forEach ()` folosește `Consumer<T>` pentru procesa lista primită ca parametru. Nu returnează nimic. Metoda poate face diferite manipulări ale datelor din listă în funcție de lambda primit.



EXPRESII LAMBDA

INTERFEȚE FUNCȚIONALE UTILE

- **Function** – Interfață `java.util.function.Function<T, R>` definește metoda `apply` care primește un obiect de tip **T** și returnează unul de tip **R**
- **Se folosește pentru a mapa informații** dintr-un un obiect de intrare la altul de ieșire

```
package atm.paradigms.tests;
import java.util.*;
import java.util.function.Function;

public class FunctionTest {
    public static void main(String[] args) {
        List<Integer> l = map(Arrays.asList("lambdas", "in", "action"),
                               (String s) -> s.length());
        System.out.println(l);
    }
    public static <T, R> List<R> map(List<T> list, Function<T, R> f){
        List<R> result = new ArrayList<>();
        for (T s : list){
            result.add(f.apply(s));
        }
        return result;
    }
}
```

Rezultat:
[7, 2, 6]

Metoda generică `map()` folosește `Function<T, R>` pentru a transforma elementele listei primite ca parametru. Returnează lista rezultată. Metoda poate face diferite transformări ale datelor din listă în funcție de lambda primit.

EXPRESII LAMBDA

INTERFEȚE FUNCȚIONALE UTILE

- Pentru **evitarea operațiilor costisitoare de boxing /unboxing** Java 8 furnizează **interfețe specializede pentru tipurile primitive**

```
package atm.paradigms.tests;

import java.util.function.IntPredicate;
import java.util.function.Predicate;

public class IntPedicateTest {
    public static void main(String[] args) {
        // reference Predicate
        Predicate<Integer> even = (Integer i) -> i % 2 == 0;
        System.out.println("Result with boxing: "
                + even.test(1000));
        // primitive specialized Predicate
        IntPredicate odd = (int i) -> i % 2 == 1;
        System.out.println("Result without boxing: "
                + odd.test(1000));
    }
}
```

Rezultat:

Result with boxing: true

Result without boxing: false



EXPRESII LAMBDA

INTERFEȚE FUNCȚIONALE UTILE

| Interfață funcțională | Descriptor funcție | Specializare primitive |
|----------------------------------|-----------------------------|--|
| Predicate<T> | T -> boolean | IntPredicate, LongPredicate, DoublePredicate |
| Consumer<T> | T -> void | IntConsumer, LongConsumer, DoubleConsumer |
| Function<T, R> | T -> R | IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T> |
| Supplier<T> | () -> T | BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier |
| UnaryOperator<T> | T -> T | IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator |
| BinaryOperator<T> | (T, T) -> T | IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator |
| BiPredicate<L, R> | (L, R) -> boolean | |
| BiConsumer<T, U> | (T, U) -> void | ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T> |
| BiFunction<T, U, R> | (T, U) -> R | ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, .ToDoubleBiFunction<T, U> |

EXPRESII LAMBDA

VERIFICAREA TIPULUI, DEDUCEREA TIPULUI, RESTRIȚII

- Tipul expresiei lambda este dedus din contextul în care se folosește
- Tipul așteptat poartă numele de tip țintă
- Exemplu de verificare a tipului

`List<Apple> filtered = filter(inventory, (Apple a) -> a.getWeight() > 150);`

1. Se analizează declarația metodei **filter**
2. Al 2-lea parametru al metodei este de tip **Predicate<Apple>** (tip țintă)
3. **Predicate<Apple>** este o interfață funcțională cu o singură metodă **test**
4. Metoda **test** este un descriptor de funcție care **acceptă un obiect Apple și returnează boolean**
5. Argumentul primit de metoda **filter** trebuie să satisfacă cerințele de mai sus
 - Dacă expresia lambda generează o excepție și metoda abstractă a interfeței trebuie să conțină clauza **throws** corespunzătoare

EXPRESII LAMBDA

VERIFICAREA TIPULUI, DEDUCEREA TIPULUI, RESTRIȚII

- Compilerul deduce interfața funcțională asociată cu expresia lambda (tipul țintă) din context, deducând semnătura corespunzătoare expresiei lambda
- **Când lambda are doar un parametru parantezele pot fi omise**
- De asemenea **tipul parametrilor din expresia lambda pot fi omisi**

```
List<Apple> greenApples =  
    filter(inventory, a -> "green".equals(a.getColor()));
```

No explicit type on
the parameter a

- **Expresiile lambda pot folosi variabile definite în afara scopului lor doar dacă sunt definite ca *finale* (final) sau *efectiv finale***

```
// effectively final  
int port = 8080;  
Runnable r = () -> System.out.println(port);  
port = 24; // ERROR
```

Prin folosire în expresia lambda, variabila **port** devine efectiv finală și nu mai poate fi modificată.

EXPRESII LAMBDA REFERINȚE DE METODE

- Permite reutilizarea definițiilor metodelor existente și trecerea codului asemănător expresiilor lambda
- Referințele de metode permit crearea de expresii lambda din metode implementate

```
inventory.sort(comparing(Apple::getWeight));
```

- **Apple::getWeight** este o referință de metodă pentru metoda `getWeight` din clasa `Apple`
- Referința de metodă este echivalentul expresiei lambda `(Apple a) -> a.getWeight()`
- Alt exemplu **System.out::println** este echivalent pentru `(String s) -> System.out.println(s)`



EXPRESII LAMBDA REFERINȚE DE METODE

- Există 3 tipuri de referințe la metode
- La o metodă statică – exemplu: metoda *parselnt* a clasei *Integer* se scrie *Integer::parselnt*
- La metoda instanței unui tip arbitrar – exemplu: metoda *length* a unui obiect de tip *String* se scrie *String::length*
- La metoda instanței unui obiect existent stocat într-o variabilă locală – exemplu: variabila *locTransaction* este o referință la tipul *Transaction* care are metoda *getValue*, se scrie *locTransaction ::getValue*

```
List<String> list = Arrays.asList( "b", "B", "A", "a");
list.sort(comparing(String::compareToIgnoreCase));
System.out.println(list);
```

Rezultat:
[A, a, b, B]

EXPRESII LAMBDA REFERINȚE DE METODE

- Se pot crea **referințe la constructori** existenți folosind numele acestuia și cuvântul cheie **new**, **ClassName::new**
- Funcționează în mod similar metodelor statice
- Exemplu:

```
package atm.paradigms.tests;

public class Apple {
    private int weight;
    private String color;

    public Apple(int weight) {
        this.weight = weight;
    }
    public Apple(int weight, String color) {
        this.weight = weight;
        this.color = color;
    }

    @Override
    public String toString() {
        return "Apple [color=" + color
               + ", weight=" + weight + "]";
    }
}
```



EXPRESII LAMBDA

REFERINȚE DE METODE

```

package atm.paradigms.tests;

import java.util.*;
import java.util.function.Function;

public class LambdaTest {
    public static void main(String[] args) {
        List<Integer> weights = Arrays.asList(7, 3, 4, 8, 10);
        List<Apple> apples = map(weights, Apple::new);
        System.out.println(apples);
    }

    public static List<Apple> map(List<Integer> list,
                                    Function<Integer, Apple> f){
        List<Apple> result = new ArrayList<>();
        for (Integer e : list){
            result.add(f.apply(e));
        }
        return result;
    }
}
  
```

Rezultat:

[Apple [color=null, weight=7],
 Apple [color=null, weight=3],
 Apple [color=null, weight=4],
 Apple [color=null, weight=8],
 Apple [color=null, weight=10]]

Metoda **map()** primește ca parametrii o listă și o interfață funcțională de tip **Function**. Aplică funcția pe fiecare element din listă, generează o instanță **Apple**, care este apoi adăugată unei alte liste de rezultate. Metoda **map()** este utilizată cu constructorul **new Apple(int weight)**, transmis ca referință de metodă.

EXPRESII LAMBDA REFERINȚE DE METODE

```
package atm.paradigms.tests;
import java.util.*;
import java.util.function.BiFunction;

public class LambdaTest {
    public static void main(String[] args) {
        Map<Integer, String> hm = new HashMap<Integer, String>();
        hm.put(12, "red");
        hm.put(10, "green");
        hm.put(13, "yellow");
        List<Apple> apples = map(hm, Apple::new);
        System.out.println(apples);
    }
    public static List<Apple> map(Map<Integer, String> map,
        BiFunction<Integer, String, Apple> f){
        Set<Map.Entry<Integer, String>> set = map.entrySet();
        List<Apple> result = new ArrayList<>();
        for (Map.Entry<Integer, String> a : set){
            result.add(f.apply(a.getKey(), a.getValue()));
        }
        return result;
    }
}
```

Rezultat:

[Apple [color=green, weight=10],
Apple [color=red, weight=12],
Apple [color=yellow, weight=13]]

Metoda **map()** primește ca parametrii un **Map** și o interfață funcțională de tip **BiFunction**. Aplică funcția pe fiecare pereche cheie/valoare dat de **entrySet()**, generează o instanță **Apple**, care este apoi adăugată unei alte liste de rezultate.
Metoda **map()** este utilizată cu constructorul **new Apple(int weight, String color)**, transmis ca referință de metodă.



EXPRESII LAMBDA

ALTE DETALII

- Semnătura metodei **abstracte** (*descriptorul funcției*) **descrie semnătura expresiei lambda**
- **Interfața Comparator** are un **descriptor de funcție de forma** $(T, T) \rightarrow \text{int}$
- Soluția de sortare a unei liste este

```
inventory.sort((Apple a1, Apple a2)-> a1.getWeight().compareTo(a2.getWeight()));
```

- Care poate fi rescrisă folosind deducerea tipului la

```
inventory.sort((a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

- **Interfața Comparator** are o **funcție statică ajutătoare comparing** care știe să extragă cheia de comparație dintr-un singur argument

```
import static java.util.Comparator.comparing;
```

```
inventory.sort(comparing((a) -> a.getWeight()));
```

EXPRESII LAMBDA

ALTE DETALII

- Instrucțiunea anterioară poate fi scrisă mai concis
`inventory.sort(comparing(Apple::getWeight));`
- În Java 8 Comparator-i pot și înlănuiri
- Metoda statică `Comparator.comparing` returnează tot un Comparator
- Pentru a sorta descrescător nu este necesar să facem o nouă sortare, doar să chemăm metoda `reversed`
`inventory.sort(comparing(Apple::getWeight).reversed());`

- Pentru a introduce un alt criteriu de sortare atunci când rezultatul este egal folosește metoda `thenComparing`

```
inventory.sort(comparing(Apple::getWeight)
    .reversed()
    .thenComparing(Apple::getCountry));
```



INTRODUCERE ÎN STREAM-URI

- Colecțiile sunt omniprezente în aplicațiile Java fiind folosite pentru a grupa și procesa datele
 - Cu toate acestea manipularea colecțiilor nu este simplă fiind necesară utilizarea iteratorilor și reimplementarea acelorași operații
 - Ce se întâmplă dacă avem colecții de dimensiuni mari care necesită paralelizare și utilizarea procesoarelor cu mai multe nuclee?
 - Scrierea de cod paralelizat cu iteratori este o sarcină complexă
 - Soluția o reprezintă stream-urile care ne permit manipularea datelor în mod declarativ
 - Se declară ce se dorește fără a ne interesa detaliile de implementare
 - Interogarea bazelor de date se face declarativ utilizând interogări SQL
- SELECT name FROM population WHERE age < 40**
- Exemplu: returnează numele persoanelor cu vîrstă sub 40 de ani sortate după vîrstă

INTRODUCERE ÎN STREAM-URI

EXEMPLU ÎN JAVA 7

```
public static List<String> filterJ7(List<Person> population){  
    // filter elements with an accumulator  
    List<Person> people = new ArrayList<>();  
    for (Person p : population){  
        if (p.getAge() < 40){  
            people.add(p);  
        }  
    }  
    // sort people under 40  
    Collections.sort(people, new Comparator<Person>() {  
        @Override  
        public int compare(Person p1, Person p2) {  
            return Integer.compare(p1.getAge(), p2.getAge());  
        }  
    });  
    // process and extract names  
    List<String> result = new ArrayList<>();  
    for (Person p : people){  
        result.add(p.getName());  
    }  
    return result;  
}
```

Exemplu de filtrare în stil Java 7 și anterior (vezi proiectul course7)

Metoda **filterJ7()** primește ca parametru o listă cu elemente de tip **People**.

Folosind iteratori extrage persoanele cu vârstă sub 40 de ani într-o listă nouă.

Sortează lista folosind un Comparator implementat ca o clasă anonimă.

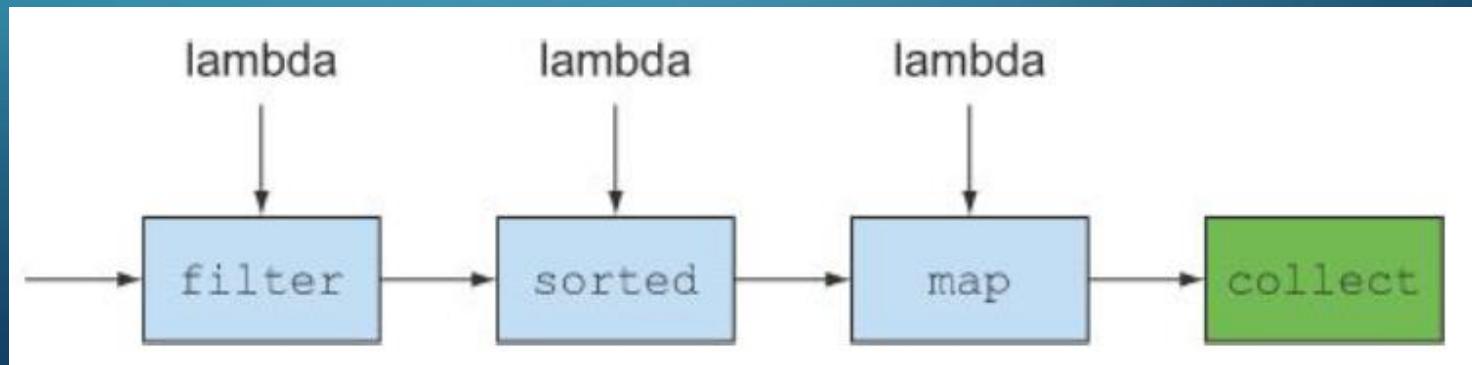
Extrage numele persoanelor într-o altă listă de **String** și o returnează.

INTRODUCERE ÎN STREAM-URI

EXEMPLU ÎN JAVA 8

```
public static List<String> filterJ8(List<Person> population){  
    return population.stream()  
        .filter(p -> p.getAge() < 40) // filter people under 40  
        .sorted(comparing(Person::getAge)) // sort by age  
        .map(Person::getName) // extract names  
        .collect(toList()); // collect list of names  
}
```

- **Operațiile precum `filter`, `sorted`, `map` și `collect` sunt disponibile ca blocuri de nivel înalt și nu trebuie să ne preocupăm de firele de execuție sau cum să paralelizăm anumite operații**
- **Codul este declarativ:** se specifică ceea ce se dorește să se realizeze și nu cum se implementează



INTRODUCERE ÎN STREAM-URI

- **Un stream este o secvență de elemente** provenind de la o sursă care suportă operații de procesare a datelor
 - **Secvență de elemente** – similar colecțiilor, cu deosebirea că **stream-urile se ocupă cu operații pe date**, pe când **colecțiile sunt structuri de date** care stochează datele
 - **Sursă** – Stream-urile consumă datele dintr-o sursă precum colecții, vectori sau resurse I/O
 - **Operații de procesare a datelor** – Stream-urile suportă operații comune de manipulare a datelor precum **filter, map, reduce, find, match, sort** etc. în mod similar SQL. Operațiile pe stream-uri pot fi secvențiale sau paralele
- **Stream-urile** au și caracteristicile:
 - **Pipelining** – majoritatea operațiilor cu stream-uri **returnează tot un stream** permitând înlănuirea acestora
 - **Internal iteration** – Spre deosebire de colecții care folosesc iteratori expliciti, **stream-urile ascund iteratorii**

INTRODUCERE ÎN STREAM-URI

COD PENTRU EXEMPLE

```
package atm.paradigms.tests;

public class Person {
    private final String name;
    private final boolean employed;
    private final int age;
    private final Type gender;

    public Person(String name, boolean employed,
                  int age, Type gender) {
        this.name = name;
        this.employed = employed;
        this.age = age;
        this.gender = gender;
    }

    public String getName() {
        return name;
    }
    public boolean isEmployed() {
        return employed;
    }
}
```

```
public int getAge() {
    return age;
}
public Type getGender() {
    return gender;
}

@Override
public String toString() {
    return name;
}

public enum Type {MALE, FEMALE};
}
```

(vezi proiectul course7, pachetul atm.paradigms.others)



INTRODUCERE ÎN STREAM-URI

COD PENTRU EXEMPLE

- Pentru **generarea unui liste conținând populația** folosim metoda statică de mai jos:

```
public static List<Person> populate(){
    return Arrays.asList(
        new Person("Ion", true, 46, Person.Gender.MALE),
        new Person("Maria", false, 15, Person.Gender.FEMALE),
        new Person("Vasile", false, 25, Person.Gender.MALE),
        new Person("Violeta", true, 50, Person.Gender.FEMALE),
        new Person("Marius", false, 5, Person.Gender.MALE),
        new Person("Alexia", true, 35, Person.Gender.FEMALE),
        new Person("George", true, 28, Person.Gender.MALE),
        new Person("Cristina", false, 19, Person.Gender.FEMALE),
        new Person("Mihai", true, 55, Person.Gender.MALE)
    );
}
```



INTRODUCERE ÎN STREAM-URI

- Sursa este lista de persoane **returnată de metoda `populate()`**
- Procesăm datele din stream prin **`filter`, `map`, `limit` și `collect`**. În afară de `collect` toate operațiile returnează tot un stream
- **`collect` pornește operația pipeline** ce poate fi văzută ca o interogare pe sursă și returnează o listă. **Fără `collect` nu se produce nici un rezultat**

```
List<Person> population = populate();
List<String> threeOver40 = population.stream()
    .filter(p -> p.getAge() > 40)
    .map(Person::getName)
    .limit(3)
    .collect(toList());
System.out.println(threeOver40);
```

Rezultat:
[Ion, Violeta, Mihai]

INTRODUCERE ÎN STREAM-URI

- Stream-urile, la fel ca iteratorii, **pot fi traversate doar o dată**
- Codul de mai jos generează o eroare care spune că stream-ul este consumat deja

```
List<String> words = Arrays.asList("title", "line", "letter");
Stream<String> a = words.stream();
a.forEach(System.out::println);
a.forEach(System.out::println);
```

Rezultat:

title

line

letter

Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed

INTRODUCERE ÎN STREAM-URI

- Colecțiile necesită ca iterarea să fie făcută explicit de utilizator (iterație externă)
- Stream-urile folosesc iterare internă

```
// iteration with for-each
List<String> names = new ArrayList<>();
for (Person p : population){
    names.add(p.getName());
}
```

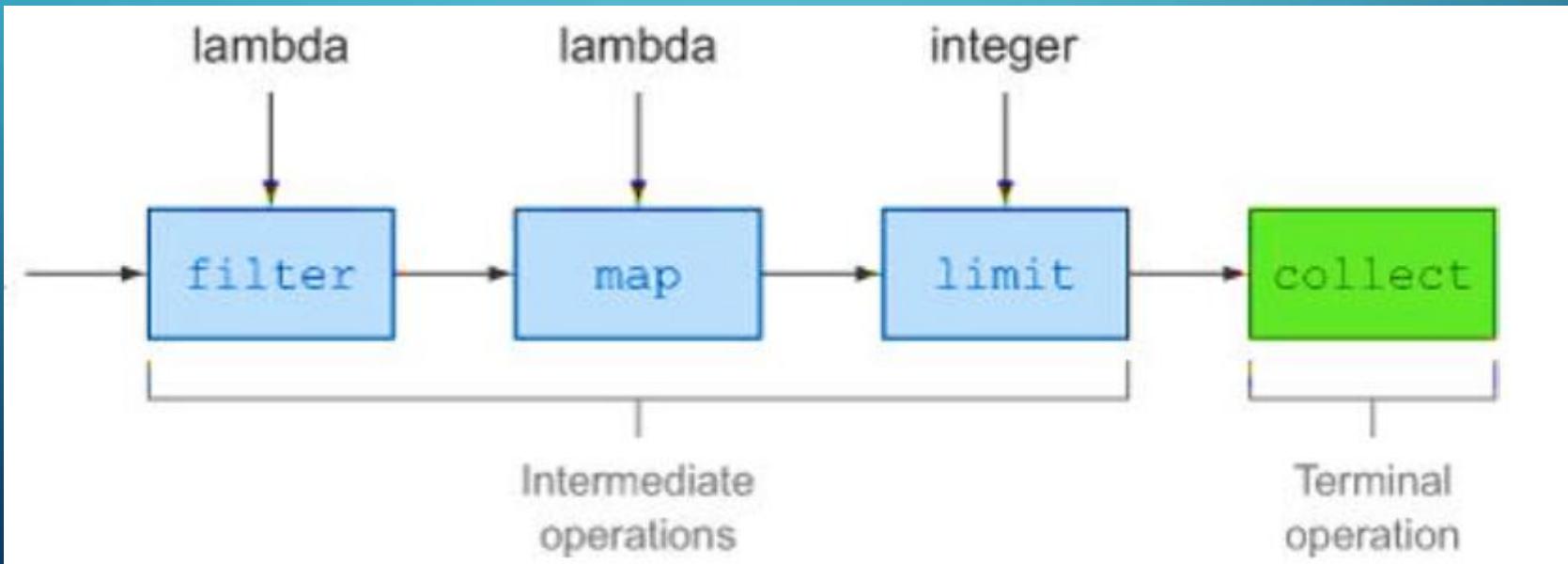
```
// internal iteration
List<String> names = population.stream()
    .map(Person::getName)
    .collect(toList());
```

```
// iteration with iterator
List<String> names = new ArrayList<>();
Iterator<Person> iterator =
    population.iterator();
while (iterator.hasNext()){
    Person p = iterator.next();
    names.add(p.getName());
}
```

INTRODUCERE ÎN STREAM-URI

OPERAȚII CU STREAM-URI

- Interfața Stream din pachetul `java.util.stream` definește multe operații ce pot fi încadrate în 2 categorii:
- **Operații intermediare** – precum `filter`, `sort`, `map`, `limit` etc. ce pot fi conectate pentru a forma **pipeline-uri**
- **Operații terminale** – care duc la execuția pipeline-ului și la închiderea lui



INTRODUCERE ÎN STREAM-URI

OPERAȚII CU STREAM-URI

- **Operațiile intermediare**, precum *filter* sau *sort*, **returnează un alt stream**
- **Operațiile intermediare** nu execută procesarea până la **invocarea operației terminale**
- **Operațiile intermediare sunt leneșe (lazy)** putând fi unite și executate într-o singură trecere

```
List<Person> population = populate();
List<String> threeOver40 = population.stream()
    .filter(p -> {
        System.out.println("filtering " + p.getName());
        return p.getAge() > 40;
    })
    .map(p -> {
        System.out.println("mapping " + p.getName());
        return p.getName();
    })
    .limit(3)
    .collect(toList());
System.out.println(threeOver40);
```

Operațiile **filter** și **map** sunt executate într-o singură trecere (**lazy**). Se utilizează expresii lambda cu blocuri pentru logging.

Rezultat:

filtering Ion
mapping Ion
filtering Maria
filtering Vasile
filtering Violeta
mapping Violeta
filtering Marius
filtering Alexia
filtering George
filtering Cristina
filtering Mihai
mapping Mihai
[**Ion, Violeta, Mihai**]

INTRODUCERE ÎN STREAM-URI

OPERAȚII CU STREAM-URI

- **Operațiile terminale produc un rezultat dintr-un pipeline**, precum o listă, un întreg sau chiar void

```
population.stream().forEach(System.out::println);
```

- **forEach** este o operație terminală deoarece returnează **void**

```
long count = population.stream()
    .filter(p -> p.getAge() > 40)
    .distinct()
    .count();
```

- În exemplu, **count** este operație terminală celelalte sunt intermediare

Lucrul cu stream-uri presupune 3 elemente:

- **O sursă de date** pe care executăm o interrogare
- **Un pipeline** alcătuit din operații intermedie înlántuite
- **O operație terminală** care execută pipeline-ul și produce un rezultat

INTRODUCERE ÎN STREAM-URI

OPERAȚII CU STREAM-URI

Operații intermedie utilizate

| Operație | Returnează | Argument | Definiție funcție |
|-----------------|------------|----------------|-------------------|
| filter | Stream<T> | Predicate<T> | T -> boolean |
| map | Stream<T> | Function<T, R> | T -> R |
| limit | Stream<T> | Integer | |
| sorted | Stream<T> | Comparator<T> | (T, T) -> int |
| distinct | Stream<T> | | |

Operații terminale utilizate

| Operație scop | descriere |
|----------------|---|
| forEach | Consumă fiecare element din stream și aplică o expresie lambda pe el. Operația returnează void |
| count | Returnează numărul de elemente din stream. Tipul returnat este long |
| collect | Reduce stream-ul la o colecție (List, Map) sau chiar un întreg |



STREAM-URI ÎN DETALIU

FILTERING

- Am folosit deja metoda ***filter***. Aceasta primește ca argument un **Predicate** și returnează un **stream conținând toate elementele care satisfac condiția**

```
List<Person> employedPeople = population.stream()
    .filter(Person::isEmployed)
    .collect(toList());
```

- Pentru **filtrarea elementelor unice** se folosește metoda ***distinct***

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 3, 1, 2, 4, 5, 4, 6, 5);
numbers.stream()
    .filter(i -> i % 2 == 0)
    .distinct()
    .forEach(System.out::println);
```

- Se pot sări elemente din stream cu metoda ***skip***

```
List<Person> peopleOver40 = population.stream()
    .filter(p -> p.getAge() > 40)
    .skip(2)
    .collect(toList());
```



STREAM-URI ÎN DETALIU

MAPPING

- Metoda **map** ia ca argument o **funcție** care este **aplicată fiecărui element rezultând un element nou**

```
List<String> names = population.stream()
    .map(Person::getName)
    .collect(toList());
```

Operațiile **map** primește ca argument **Function<Person, String>** cu semnătura **(Person) -> String**. Returnează **Stream<String>**, iar **collect** va crea o listă de **String**.

Exemplu: dintr-o listă de cuvinte rezultă o listă cu lungimea fiecărui cuvânt în caractere

```
List<String> words = Arrays.asList("Java8", "new", "features", "are", "cool");
List<Integer> wordLengths = words.stream()
    .map(String::length)
    .collect(toList());
System.out.println(wordLengths);
```

- Rezultat: [5, 3, 10, 3, 4]

Operațiile **map** primește ca argument **Function<String, Integer>** cu semnătura **(String) -> Integer**. Returnează **Stream<Integer>**, iar **collect** va crea o listă de **Integer**.

STREAM-URI ÎN DETALIU

FLATTENING

- Problemă: Cum extragem caractere unice dintr-o listă de cuvinte?
- O soluție imediată dar greșită

```
wordLengths = words.stream()
    .map(w -> w.split(""))
    .distinct()
    .collect(toList());
```

- Metoda `split` returnează `String[]` pentru fiecare cuvânt, iar stram-ul rezultat va fi `Stream<String[]>` și nu `Stream<String>` cum aveam nevoie
- Soluția o constituie metoda ***flatMap*** care transformă fiecare vector în elemente individuale pe care le adaugă la stream
- Toate stream-urile generate prin aplicarea ***map*** sunt amestecate (flattened) într-un singur stream



STREAM-URI ÎN DETALIU

FLATTENING

- Există o metodă statică **Arrays.stream()** care primește un vector și produce un stream

```
List<String> words = Arrays.asList("Java8", "new", "features", "are", "cool");
List<String> uniqueChars= words.stream()
// converts each word in letter array
        .map(w -> w.split(""))
// flattens each stream into one single stream
        .flatMap(Arrays::stream)
        .distinct()
        .collect(toList());
System.out.println(uniqueChars);
```

Operația **map** primește un cuvânt și îl transformă stream de vector de litere (**Stream<String[]>**). Folosind **flatMap** vectorul de litere este convertit în stream de litere (**Stream<String>**) pe care se aplică apoi celelalte operații

Rezultat:

[J, a, v, 8, n, e, w, f, t, u, r, s, c, o, l]



STREAM-URI ÎN DETALIU

FINDING & MATCHING

- Procesarea datelor pentru găsirea elementelor care corespund unei proprietăți date
- Streams API pune la dispoziție următoarele metode: *allMatch*, *anyMatch*, *noneMatch*, *findFirst* și *findAny*
- **Metoda *anyMatch*:**
 - oferă răspuns la întrebarea Există vreun element în stream care corespunde unui condiții date (Predicate)?
 - returnează boolean și este o operatie terminală

```
if (population.stream().anyMatch(Person::isEmployed)){
    System.out.println("Some people insist to work! Why?");
}
```

STREAM-URI ÎN DETALIU

FINDING & MATCHING

- **Metoda allMatch:** verifică dacă **toate elementele** unui stream **verifică o condiție dată**

```
boolean isEmployed = population.stream().allMatch(Person::isEmployed);  
System.out.println(isEmployed);
```

- Rezultat: *false*
- **Metoda noneMatch:** metoda opusă **allMatch**, verifică dacă **nici un element nu verifică o condiție dată**
- **Metoda findAny:** returnează **un element arbitrar** din stream-ul curent **ca Optional**

```
Optional<Person> person = population.stream()  
        .filter(p -> p.getAge() > 40)  
        .findAny();
```

- Pipeline-ul va fi optimizat să execute totul într-o singură trecere și să termine atunci când a găsit rezultatul (*short-circuiting*)

STREAM-URI ÎN DETALIU

FINDING & MATCHING

- Clasa **Optional<T>** este o **clasă container** pentru a **reprezenta prezența sau absența unei valori**
- Metoda **findAny** poate să **nu returneze nici un element**, iar pentru a evita returnarea de null Java 8 introduce **Optional<T>**

| Metodă | Descriere |
|---|--|
| isPresent() | Returnează true dacă Optional conține o valoare, false dacă nu conține |
| ifPresent(Consumer<T> block) | Execută blocul dacă este prezentă o valoare |
| T get() | Returnează dacă există, generează excepție dacă nu există |
| T orElse(T) | Returnează valoarea dacă există, dacă nu returnează valoarea implicită |

```
population.stream()
    .filter(p -> p.getAge() > 40)
    .findAny()
    .ifPresent(p -> System.out.println(p.getName()));
```



STREAM-URI ÎN DETALIU

FINDING & MATCHING

- Metoda ***findFirst*** – găsește **primul element** dintr-un stream folosind **ordinea în care elementele apar**
- Exemplu: dintr-o listă de numere să se găsească primul pătrat perfect care se divide cu 3

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);
list.stream()
    .map(x -> x * x)
    .filter(x -> x % 3 == 0)
    .findFirst()
    .ifPresent(System.out::println);
```

Rezultat: 9

STREAM-URI ÎN DETALIU REDUCING

- **Operațiile de reducere** constau în **combinarea elementelor** dintr-un stream **într-o sigură valoare** (exemplu: suma valorilor din stream)

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);
int sum = list.stream().reduce(0, (a, b) -> a + b);
System.out.println(sum);
```

- **Metoda reduce** – are 2 parametrii:
- **Valoarea inițială** (0 în exemplu)
- Interfața **BinaryOperator<T>** care combină 2 elemente de același tip pentru a produce o nouă valoare

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);
int product = list.stream().reduce(1, (a, b) -> a * b);
System.out.println(product);
```

STREAM-URI ÎN DETALIU REDUCING

- Folosind metoda **reduce** putem calcula valoarea maximă și minimă dintr-un stream:
- Este nevoie de o funcție lambda care să returneze maximul sau minimul a 2 elemente date
- Clasa **Integer** pune la dispoziție metodele statice **max**, **min** (dar și **sum**)
- Există o variantă a funcției **reduce** care nu ia o valoare inițială dar returnează un obiect de tip **Optional**

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6);
list.stream()
    .reduce(Integer::max)
    .ifPresent(System.out::println);
```

```
list.stream()
    .reduce(Integer::min)
    .ifPresent(System.out::println);
```



| Operație | Tip | Tip returnat | Interfață funcțională | Descrierea funcției |
|------------------|-------------|--------------|------------------------|---------------------|
| filter | Intermediar | Stream<T> | Predicate<T> | T -> boolean |
| distinct | Intermediar | Stream<T> | | |
| skip | Intermediar | Stream<T> | long | |
| limit | Intermediar | Stream<T> | long | |
| map | Intermediar | Stream<T> | Function<T, R> | T -> R |
| flatMap | Intermediar | Stream<R> | Function<T, Stream<R>> | T -> Stream<R> |
| sorted | Intermediar | Stream<T> | Comparator<T> | (T, T) -> int |
| anyMatch | Terminal | boolean | Predicate<T> | T -> Boolean |
| noneMatch | Terminal | boolean | Predicate<T> | T -> Boolean |
| allMatch | Terminal | boolean | Predicate<T> | T -> Boolean |
| findAny | Terminal | Optional<T> | | |
| findFirst | Terminal | Optional<T> | | |
| forEach | Terminal | void | Consumer<T> | T -> void |
| collect | Terminal | R | Collector<T, A, R> | |
| reduce | Terminal | Optional<T> | BinaryOperator<T> | (T, T) -> T |
| count | Terminal | long | | |



STREAM-URI ÎN DETALIU

STREAM-URI NUMERICE

- Pentru a evita problemele de performanță pentru tipuri primitive Java 8 introduce **stream-uri specializare** precum *IntStream*, *DoubleStream*, *LongStream*
- **Convertirea** unui stream normal la un stream specializat se face cu *mapToInt*, *mapToDouble*, *mapToLong*

```
int calories = menu.stream()
    .mapToInt(Dish::getCalories)
    .sum();
```

Returns a Stream<Dish>

Returns an IntStream

- *IntStream* are metoda **sum** care returnează suma elementelor sau 0 pentru stream gol
- **Conversia** unui stream de primitive la un stream general se face prin chemarea metodei **boxed**

STREAM-URI ÎN DETALIU

INTERVALE NUMERICE

- Java 8 a introdus 2 metode statice **range** și **rangeClosed** în interfețele **IntStream** și **LongStream** care pot genera **intervale numerice**
- **range** este exclusiv (nu conține capetele) iar **rangeClosed** este inclusiv (conține capetele)

```
IntStream even = IntStream.rangeClosed(1, 100)
    .filter(n -> n % 2 ==0);
System.out.println(even.count());
```

Rezultat: 50

STREAM-URI ÎN DETALIU

CONSTRUIREA STREAM-URILOR

- **Construirea stream-urilor din valori** – metoda statică `Stream.of` poate lua un număr oricât de mare de parametri și returnează un stream

```
Stream<String> stream = Stream.of("Java8", "in", "action");
stream.map(String::toUpperCase).forEach(System.out::println);
```

Rezultat:

JAVA8

IN

ACTION

- Pentru a **crea un stream gol** se folosește

```
Stream<String> emptyStream = Stream.empty();
```

STREAM-URI ÎN DETALIU

CONSTRUIREA STREAM-URILOR

- Se pot crea stream-uri din vectori
- Metoda statică **Arrays.stream** primește un **vector ca parametru și returnează un Stream**

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7};  
int sum = Arrays.stream(numbers).sum();  
System.out.println(sum);
```

Rezultat: 28

- Java NIO API (non-blocking IO) oferă suport pentru stream-uri
- Multe metode statice din `java.nio.file.Files` returnează un stream
- De exemplu `Files.lines` returnează un stream de linii de text (String) dintr-un fișier

STREAM-URI ÎN DETALIU

CONSTRUIREA STREAM-URILOR

```
long uniqueWords = 0;
try (Stream<String> lines =
    Files.lines(Paths.get("data.txt"))){
    uniqueWords = lines.flatMap(l -> Arrays.stream(l.split(" ")))
        .distinct()
        .count();
} catch (IOException e) {
    e.printStackTrace();
}
```

Folosind *try-with-resources* se obține un stream de linii de text (lines). Folosind operația **flatMap** în combinație cu `Arrays.stream` și metoda `split`, liniile de text sunt sparte în cuvinte individuale. Se numără apoi cuvintele unice cu operațiile **distinct** și **count**.

STREAM-URI ÎN DETALIU

CONSTRUIREA STREAM-URILOR

- Streams API pune la dispoziție 2 metode statice ce pot fi folosite pentru a genera **stream-uri dintr-o funcție**: **Stream.iterate** și **Stream.generate**
- Metodele **iterate** și **generate** permit **crearea de stream-uri infinite** (care nu au o dimensiune fixă)
- Metoda iterate:**

```
Stream.iterate(0, n -> n + 2)
    .limit(20)
    .forEach(System.out::println);
```

Rezultat:

0

2

4

6

8

...

- ia o **valoare inițială** (0 în exemplu) și o **expresie lambda** (de tip **UnaryOperator<T>**) care se aplică succesiv pe fiecare nouă valoare produsă
- Operația produce un **stream infinit** (nemărginit)



STREAM-URI ÎN DETALIU

CONSTRUIREA STREAM-URILOR

- Putem construi **seria Fibonacci** (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55....) cu **iterate**
- Primul element al seriei este perechea (0, 1) – tuple

```
Stream.iterate(new int[]{0, 1},  
               t -> new int[]{t[1], t[0] + t[1]})  
    .limit(10)  
    .map(t -> t[0])  
    .forEach(System.out::println);
```

- Pentru că metoda primește ca **argument UnaryOperator<T>** trebuie să folosim un **stream de tuple** reprezentate ca vectori cu 2 elemente
- Primul este new int[]{0, 1}**
- Pentru a tipări seria Fibonacci utilizăm operația **map** pentru a extrage doar primul element din tuple

Rezultat:

0

1

1

2

3

5

8

13

21

34



STREAM-URI ÎN DETALIU

CONSTRUIREA STREAM-URILOR

- Metoda **generate** produce un **stream infinit de valori calculate**
- Primește ca argument o **expresie lambda de tip Supplier<T>**, cu definiția
 $() \rightarrow T$, care returnează noua valoare
- Nu aplică succesiv a funcție pentru a produce noua valoare ca *iterate*

```
Stream.generate(Math::random)
    .limit(5)
    .forEach(System.out::println);
```

Rezultat:

0.29267096637211754
0.9551296280285726
0.7701335941535129
0.522304048718223
0.460392872702028

BIBLIOGRAFIE

- **Java 8 in Action**, Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft, Manning Publications Co, 2015
- **Java EE 8 Design Patterns and Best Practices**, Rhuan Rocha, João Purificação, Packt Publishing, 2018
- **Java™: The Complete Reference, Tenth Edition**, Herbert Schildt, McGraw-Hill, 2005