

STREAM-URI

Un *stream*, așa cum îi spune numele, este un flux de date, respectiv o secvență de elemente preluate dintr-o sursă care suportă operații de procesare (parcurgere, modificare, ștergere etc.).

Noțiunea de *stream* a fost introdusă în versiunea Java 8 în scopul de a asigura prelucrarea datelor dintr-o sursă de date care suportă operații de procesare, într-o manieră intuitivă și transparentă. În versiunile anterioare, prelucrarea unei surse de date presupune utilizarea unor metode specifice sursei respective: selecția elementelor cu o anumită proprietate se poate realiza prin parcurgerea colecției cu ajutorul unei instrucțiuni iterative, operația de sortare se poate efectua folosind metoda `sort` din clasa utilitară `Collections` etc.

De exemplu, să presupunem faptul că se dorește extragerea dintr-o listă a informațiilor despre persoanele care au vârsta cel puțin egală cu 30 de ani și afișarea lor în ordine alfabetică. O soluție pentru o versiune anterioară versiunii 8 este prezentată mai jos:

```
ArrayList<Persoana> lp = new ArrayList<>();//lista de Persoane
lp.add(new Persoana("Matei", 23));
...
ArrayList<Persoana> ln = new ArrayList<>();//lista nouă

for(Persoana item: lp)
    if(item.getVarsta()>=30)
        ln.add(item);

Collections.sort(ln, new Comparator<Persoana>()
{
    public int compare(Persoana p1, Persoana p2) {
        return p1.getNume().compareTo(p2.getNume());
    }
});

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

O soluție pentru o versiune Java mai mare sau egală cu 8 este prezentată mai jos:

```
lp.stream().filter(p ->p.getVarsta()>=30).
sorted(Comparator.comparing(Persoana::getNume)).forEach(System.out::println);
```

Comparând cele două soluții, se poate observa faptul că utilizarea unui stream asociat unei colecții, alături de metode specifice, lambda expresii sau referințe către metode, conduce la o prelucrare mult mai facilă a datelor dintr-o colecție.

Caracteristicile unui stream

- Un stream nu este o colecție de date (un obiect de tip container), ci el poate fi asociat cu diferite colecții prin intermediul referințelor spre obiectele colecției respective. În consecință, un stream nu stochează elementele unei colecții, ci doar le prelucrează!
- Prelucrările efectuate asupra unui stream sunt asemănătoare interogărilor SQL și pot fi exprimate folosind lambda expresii și/sau referințe către metode.
- Un stream nu este reutilizabil, respectiv poate fi prelucrat o singură dată. Pentru o altă prelucrare a elementelor aceleiași colecții este necesară operația de asociere a unui nou stream pentru aceeași sursă de date.
- Majoritatea operațiilor efectuate de un stream furnizează un alt stream, care la rândul său poate fi prelucrat. În concluzie, se poate crea un lanț de prelucrări succesive.
- Stream-urile permit programatorului specificarea prelucrărilor necesare pentru o sursă de date, într-o manieră declarativă, fără a le implementa. Metodele utilizate pentru a prelucra un stream sunt implementate în clasa `java.util.stream.Stream`

Etapele necesare pentru utilizarea unui stream

- Crearea stream-ului
- Aplicarea unor operații de prelucrare succesive asupra stream-ului (operații intermediare)
- Aplicarea unei operații de închidere a stream-ului respectiv

În continuare, vom detalia fiecare dintre cele 3 etape necesare utilizării unui stream.

➤ Crearea unui stream

În sine, crearea unui stream presupune asocierea acestuia la o sursă de date. Astfel, în raport cu sursa de date cu care se asociază, un stream se poate crea prin mai multe modalități:

1. *deschiderea unui stream asociat unei colecții*: se utilizează metoda `Stream<T> stream()` existentă în orice colecție:

```
List<String> words = Arrays.asList(new String[]{"hello", "hola", "hallo"});
Stream<String> stream = words.stream();
```

2. *deschiderea unui stream asociat unei șir de constante*: se utilizează metoda statică cu număr variabil de parametri `Stream.of(T... values)` din clasa `Stream`:

```
Stream<String> stream = Stream.of("hello", "hola", "hallo", "ciao");
```

3. *deschiderea unui stream asociat unei tablou de obiecte*: se poate utiliza tot metoda `Stream.of(T... values)` menționată anterior:

```
String[] words = {"hello", "hola", "hallo", "ciao"};
Stream<String> stream = Stream.of(words);
```

4. *deschiderea unui stream asociat unei tablou de valori de tip primitiv*: se poate utiliza tot metoda `Stream.of(T... values)`, însă vom obține un stream format dintr-un singur obiect de tip tablou (array):

```
int[] nums = {1, 2, 3, 4, 5};
Stream<int[]> stream = Stream.of(nums)
System.out.println(Stream.of(nums).count()); // se va afișa valoarea 1
```

Se poate observa cum metoda `Stream.of` returnează un stream format dintr-un obiect de tip tablou cu valori de tip `int`, ci nu returnează un stream format din valorile de tip `int` memorate în tablou. Astfel, deschiderea unui stream asociat unui tablou cu elemente de tip primitiv se realizează prin apelul metodei `stream` din clasa `java.util.Arrays`:

```
int[] nums = {1, 2, 3, 4, 5};
System.out.println(Arrays.stream(nums).count()); // se va afișa valoarea 5
```

5. *deschiderea unui stream asociat cu un șir de valori generate folosind un algoritm specificat*: se poate utiliza metoda `Stream.generate(Supplier<T> s)` care returnează un stream asociat unui șir de elemente generate după regula specificată printr-un argument de tip `Supplier<T>`. Metoda este utilă pentru a genera un stream asociat unui șir aleatoriu sau unui șir de constante, cu o dimensiune teoretic infinită:

```
Stream.generate(()->Math.random()).forEach(System.out::println);
Stream.generate(new Random()::nextDouble).forEach(System.out::println);
```

Dimensiunea maximă a șirului generat poate fi stabilită folosind metoda `Stream<T> limit(long maxSize)`:

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

O altă posibilitate constă în utilizarea metodei `Stream<T> iterate(T seed, UnaryOperator<T> f)` care returnează un stream, teoretic infinit, asociat șirului de valori obținute prin apeluri succesive ale funcției `f`, iar primul element al șirului este indicat prin argumentul `seed`:

```
Stream.iterate(1, i->i * 2).limit(5).forEach(System.out::println);
```

Mai multe modalități de creare a unui stream sunt descrise în paginile <https://www.baeldung.com/java-8-streams> și <https://www.geeksforgeeks.org/10-ways-to-create-a-stream-in-java/>.

➤ Operații intermediare

După crearea unui stream, asupra acestuia se pot aplica operații succesive de prelucrare, cum ar fi: operația de sortare a elementelor după un anumit criteriu, filtrarea lor după o anumită condiție, asocierea lor cu o anumită valoare etc. O operație intermediară aplicată asupra unui stream furnizează un alt stream asupra căruia se poate aplica o altă operație intermediară, obținând-se astfel un șir succesiv de prelucrări de tip *pipeline* (vezi Figura 1 – sursa: <https://www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/stream-cheat-sheet.html>):

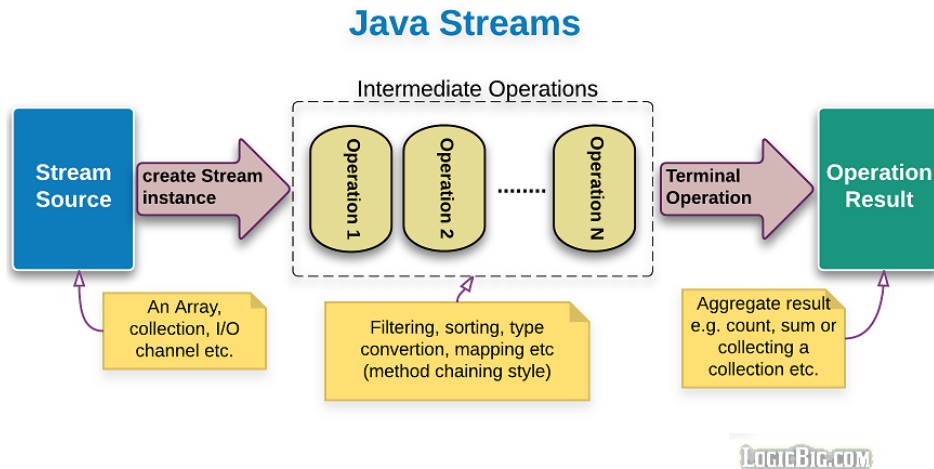


Figura 1. Etapele prelucrării unui stream

Observații:

- Operațiile intermediare nu sunt efectuate decât în momentul în care este invocată o operație de închidere!
- Operațiile intermediare pot fi de tip *stateful*, în care, intern, se rețin informații despre elementele prelucrate anterior (de exemplu: *sort*, *distinct*, *limit* etc.) sau pot fi de tip *stateless*, respectiv nu se rețin informații suplimentare despre elementele prelucrate anterior (de exemplu: *filter*).
- Operațiile intermediare de tip *stateless* pot fi efectuate simultan, prin paralelizare, în timp ce operațiile de tip *stateful* se pot executa doar secvențial.

Pentru prezentarea operațiilor intermediare, vom considera clasa *Persoana*, în care sunt definite câmpurile *String nume*, *int varsta*, *double salariu*, *String profesie* și *String[] competente*, metodele de tip *set/get* corespunzătoare, metoda *toString()* și constructori.

În continuare, sunt prezentate operații intermediare uzuale pentru o colecție cu obiecte de tip *Persoana*. Astfel, vom considera creată o listă de persoane *lp*:

```
List<Persoana> lp = new ArrayList<Persoana>();
lp.add(new Persoana(...));
lp.add(new Persoana(...));
.....
```

- `Stream<T> filter(Predicate<? super T> predicate)` – returnează un stream nou, format din elementele stream-ului inițial care îndeplinesc condiția specificată prin argumentul de tip `Predicate`.

Exemplu:

```
lp.stream().filter(p->p.getVarsta()>=40).forEach(System.out::println);
```

Într-o operație de filtrare se pot aplica mai multe criterii de selecție, prin utilizarea unor condiții specificate prin mai multe predicate:

```
Predicate<Persoana> c1 = p -> p.getVarsta()>=20;
Predicate<Persoana> c2 = p -> p.getSalariu()>=3000;
lp.stream().filter(c1.and(c2.negate())) .forEach(System.out::println);
```

- `Stream<T> sorted(Comparator<? super T> comparator)` – returnează un stream nou, obținut prin sortarea stream-ului inițial conform ordinii indicate prin comparatorul specificat prin argumentul de tip `Comparator`.

Exemplu:

```
lp.stream().sorted((p1,p2) -> p1.getVarsta() - p2.getVarsta()) .
    forEach(System.out::println);
```

Începând cu versiunea Java 8, în interfața `Comparator` a fost inclusă metoda statică `comparing` care returnează un obiect de tip `Comparator` creat pe baza unei funcții specificată printr-un argument de tip `Function<T>`:

```
lp.stream().sorted(Comparator.comparing(Persoana::getVarsta)) .
    forEach(System.out::println);
```

În plus, în interfața `Comparator` a fost introdusă și metoda `reversed()`, care permite inversarea ordinii de sortare din crescătoare (implicit!) în descrescătoare:

```
lp.stream().sorted(Comparator.comparing(Persoana::getVarsta).reversed()) .
    forEach(System.out::println);
```

- `Stream<T> sorted()` – returnează un stream nou, obținut prin sortarea stream-ului inițial conform ordinii naturale a elementelor sale (clasa corespunzătoare elementelor stream-ului, în acest caz clasa `Persoana`, trebuie să implementeze interfața `Comparable`).

Exemplu:

```
lp.stream().sorted().forEach(System.out::println);
```

- `Stream<T> limit(long maxSize)` – returnează un stream nou, format din cel mult primele `maxSize` elemente din stream-ul inițial.

Exemplu:

```
lp.stream().sorted(Comparator.comparing(Persoana::getVarsta).limit(3)) .
    forEach(System.out::println);
```

- `Stream<T> distinct()` – returnează un stream nou, format din elementele distincte ale stream-ului inițial. Implicit, elementele sunt comparate folosind hash-code-urile lor, ceea ce poate conduce la valori duplicate dacă în clasa respectivă nu sunt implementate corespunzător metodele `hashCode()` și `equals()`!

Exemplu:

```
lp.stream().distinct().forEach(System.out::println);
```

- `Stream<R> map(Function<T, R> mapper)` – returnează un stream nou, cu elemente de un tip `R`, obținut prin aplicarea asupra fiecărui obiect de tipul `T` din stream-ul inițial a regulii de asociere specificate prin funcția `mapper`.

Exemplu: afișarea profesiilor distincte ale persoanelor din lista `lp`

```
lp.stream().map(Persoana::getProfesie).distinct().
    forEach(System.out::println);
```

- `Stream<R> flatMap(Function<T, Stream<R>> mapper)` – returnează un stream nou, obținut prin concatenarea stream-urilor rezultate prin aplicarea funcției indicate prin argumentul de tip `Function` asupra fiecărui obiect de tip `T` din stream-ului inițial. Astfel, unui obiect din stream-ul inițial `i` se asociază un stream care poate să fie format dintr-unul sau mai multe obiecte de tip `R`!

Exemplu: afișarea competențelor distincte ale persoanelor din lista `lp`

```
lp.stream().flatMap(p->Stream.of(p.getCompetente())).distinct().
    forEach(System.out::println);
```

➤ **Operații de închidere**

Operațiile de închidere se aplică asupra unui obiect de tip `Stream` și pot returna fie un obiect de un anumit tip (primitiv sau nu), fie nu returnează nicio valoare (`void`).

- `void forEach(Consumer< T> action)` – operația nu returnează nicio valoare, ci execută o anumită prelucrare, specificată prin argumentul de tip `Consumer`, asupra fiecărui element dintr-un stream.

Exemplu:

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

- `T max(Comparator<T> comparator)` – operația returnează valoarea maximă dintre elementele unui stream, în raport cu criteriul de comparație precizat prin argumentul de tip `Comparator`.

Exemplu: afișarea unei persoane cu vârsta cea mai mare

```
System.out.println(lp.stream().max((p1,p2)->p1.getVarsta()-
    p2.getVarsta()));
```

- **T min**(Comparator<T> comparator) – operația returnează valoarea minimă dintre elementele unui stream, în raport cu criteriul de comparație precizat prin argumentul de tip Comparator.

Exemplu: afișarea unei persoane cu salariul minim

```
System.out.println(lp.stream().max(Comparator.  
    comparing(Persoana::getSalariu)));
```

- **T reduce**(T identity, BinaryOperator<T> accumulator) – efectuează o operație de reducere a stream-ului curent folosind o funcție de acumulare asociativă (care indică modul în care se reduc două obiecte într-unul singur) și returnează valoarea obținută prin aplicarea succesivă a funcției de acumulare.

Exemplu: afișarea sumei salariilor tuturor persoanelor din lista lp

```
Double ts = lp.stream().map(Persoana::getSalariu).  
    reduce(0.0, (x, y) -> x + y);  
System.out.println("Total salarii: " + ts);
```

- **R collect**(Collector<T,A,R> collector) – efectuează o operație de colectare, specificată prin argumentul de tip Collector, a elementelor asociate stream-ului inițial și poate returna fie o colecție, fie o valoare de tip primitiv sau un obiect.

Clasa `Collector` cuprinde o serie de metode statice care implementează operații specifice de colectare a datelor, precum definirea unei noi colecții din elementele asociate unui stream, efectuarea unor calcule statistice asupra elementelor asociate unui stream, gruparea elementele unui stream după o anumită valoare etc., astfel:

- colectorii **toList()**, **toSet()**, **toMap()** returnează o colecție de tipul specificat, formată din elementele asociate unui stream.

Exemplu: construcția unei liste cu obiecte de tip `Persoana` care au salariul mai mare sau egal decât 3000 RON

```
List<Persoana> lnoua=lp.stream().filter(p->p.getSalariu()>=3000).  
    collect(Collectors.toList());
```

- colectorul **joining**(String delimiter) returnează un șir de caractere obținut prin concatenarea elementelor unui stream format din șiruri de caractere, folosind șirul delimiter indicat prin parametrul său.

Exemplu:

```
String s = lpers.stream().filter(p -> p.getSalariu()>=3000).  
    map(Persoana::getNume).collect(Collectors.joining(", "));
```

- colectorul **counting()** returnează numărul de elemente dintr-un stream.

Exemplu:

```
Long result = givenList.stream().collect(counting());
```

- colectorii **averagingDouble()**, **averagingLong()** și **averagingInt()** returnează media aritmetică a elementelor de tip double, long sau int dintr-un stream.

Exemplu: calcularea salariului mediu al persoanelor din lista lp

```
Double sm = lp.stream().collect(averagingDouble(Persoana::getSalariu));
```

- colectorii **summingDouble()**, **summingLong()** și **summingInt()** returnează suma elementelor de tip double, long sau int dintr-un stream.

Exemplu: calcularea sumei tuturor salariilor persoanelor din lista lp

```
Double st = lp.stream().collect(summingDouble(Persoana::getSalariu));
```

- colectorul **groupingBy()** realizează o grupare a elementelor după a anumită valoare, returnând astfel o colecție de tip Map, ale cărei elemente vor fi perechi de forma <valoare de grupare, lista obiectelor corespunzătoare>.

Exemple:

1. *gruparea persoanelor pe categorii de vârstă*

```
Map<Integer, List<Persoana>> lgv = lp.stream().
    collect(groupingBy(Persoana::getVarsta));
```

2. *suma salariilor pe categorii de vârstă*

```
Map<Integer, Double> lgs = lp.stream().collect(groupingBy(
    Persoana::getVarsta, summingDouble(Persoana::getSalariu)));
```

În continuare, vom prezenta o modalitate de afișare a cuvintelor distincte din fișierul text *exemplu.txt* în ordinea descrescătoare a frecvențelor lor:

```
//creăm un stream format din toate liniile
//din fișierul text "exemplu.txt"
Files.lines(Paths.get("exemplu.txt"))
    //creăm un stream în care înlocuim fiecare linie
    //cu un tablou format din cuvintele de pe linia respectivă
    .map(linie -> linie.split("[^\\w]+"))

    //creăm un stream în care înlocuim fiecare tablou de cuvinte
    //cu un stream format din cuvintele din tabloul respectiv
    .flatMap(tablou -> Arrays.stream(tablou))

    //păstrăm doar cuvintele cu lungime nenulă, adică
    //eliminăm liniile vide din fișierul text dat
    .filter(cuvant -> cuvant.length() > 0)

    //grupăm cuvintele și calculăm frecvența fiecărui cuvânt,
    //rezultatul fiind obținut într-un Map cu perechi cheie-valoare de
    //forma (cuvânt, frecvență_cuvânt)
    .collect(Collectors.groupingBy(cuvant -> cuvant,
                                    Collectors.counting()))
```



```
//creăm un nou stream din perechile din Map-ul obținut anterior
.entrySet().stream()

//sortăm perechile în ordinea descrescătoare a frecvențelor
.sorted(Map.Entry.<String, Long>comparingByValue().reversed())

//afișăm rezultatul
.forEach(System.out::println);
```

În afara operațiilor de prelucrare (intermediare) și a celor de închidere prezentate anterior, mai există și alte operații de acest tip, pe care le puteți studia în paginile următoare: <https://javaconceptsoftheday.com/java-8-stream-intermediate-and-terminal-operations/> și <https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>.