PROGRAMARE AVANSATĂ PE OBIECTE

Conf.univ.dr. Ana Cristina DĂSCĂLESCU

Temtică curs 12

- Programarea funcțională bază pe stream-uri

- Deschiderea, prelucrarea și închiderea unui stream

Stream

- Un ***stream***, este un flux de date, care se asociază la 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!**

- Exemplu: extragerea dintr-o listă(lp) a informațiilor despre **persoanele care au vârsta cel puțin egală cu 30 de ani și afișarea lor în ordine alfabetică**. O .

- **Soluția soluție pentru o versiune anterioară versiunii 8**

```
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);
```

Stream

- **Soluție pentru o versiune mai mare sau egală cu 8**

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

- Prelucrare facilă a unei surse de date

- Utilizare lambda expresii, inteftețe descriptor, respectiv referințe către metode

➢ Caracteristicile unui stream

- Stream-urile **nu sunt colecții de date** (obiecte container), ci ele pot fi asociate cu diferite colecții. În consecință, un stream nu stochează elementele unei colecției, 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 aceleași colecții este necesară operația de asociere a unui nou stream pentru aceeași sursă de date.

➢Caracteristicile unui stream

- **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 realizării 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

➢ Crearea unui stream

- Presupune asocierea unui stream la o sursă de date. 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()
```

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

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 statică **`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
```

- 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.
- 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
```

Stream

***5 deschiderea unui stream asociat cu un șir de valori generate folosind un algoritm specificat*:**

- **`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>`.

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

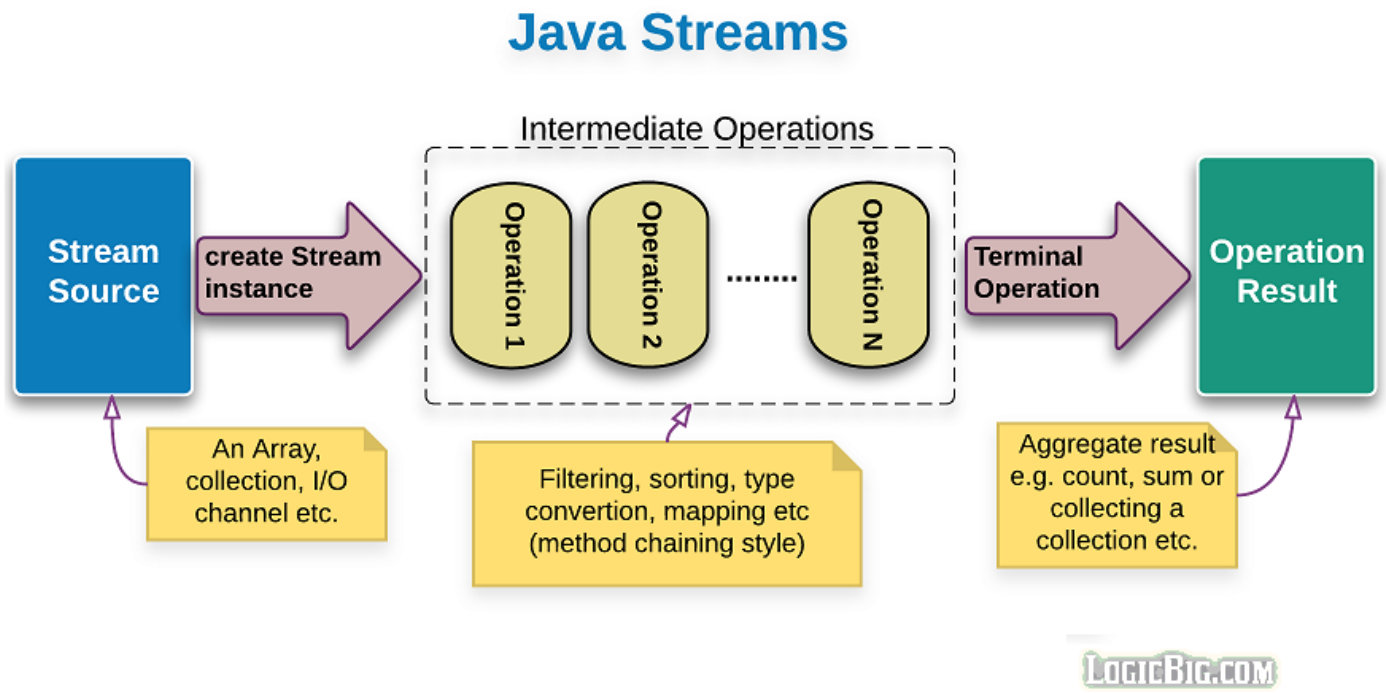
- Dimensiunea maximă a șirului generat poate fi stabilită folosind metoda `Stream<T> limit(long maxSize)`:

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

- 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

- 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*



➢ **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 reține informații despre elementele prelucrate anterior (de exemplu: `sort`, `distinct`, `limit` etc.) sau pot fi de ti

✓ ***stateless***, respectiv nu se reține 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.**

Stream

➢ `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`.

- ```
  lp.stream().filter(p-
  >p.getVarsta()>=40).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`.

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

Stream

- 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 (implicite!) în descrescătoare:

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

Stream

➢ **`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`**)

▪ `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.

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

Stream

➢ **`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()`!

- `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.

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

Stream

➢ **`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.

- ```
  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 referință), 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.

- **`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.

➢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 referință), 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.

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

Stream

- **`R(colectie_noua) 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 `Collectors` cuprinde o serie **de metode statice care implementează operații specifice de colectare a datelor**

✓ definirea unei noi colecții
✓ efectuarea unor calcule statistice
✓ gruparea elementele unui stream după o anumită valoare

Stream

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

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

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

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

Stream

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

```
Double sm = lp.stream().collect(averagingDouble(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>`.

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