**Streams**

* Au fost introduse in Java 8 si adaugat programarea functionala in Java
* Programarea functionala prevede utilizarea functiilor in code
* Streams sunt mult legate de lamba expression
* Stream API e foarte bun pentru operatii paralele
* **Stream** – secventa de elemente ce provin de la o sursa, care suporta operatii de procesare a datelor
* **Data Structures** sunt pentru a stoca datele, **Streams** sunt pentru a efectua calcule/operatii asupra lor
* Streams suporta database operations si diferite operatii din programarea functionala, ca **filter, map, reduce** etc.
* Operatiile de tip stream pot fi efectuate fie **secvential**, fie **paralel. Deci, asa nu trebuie sa ne batem capul cu multithreading, cu lock, cu fork-join framework etc., caci totul e administrat de Streams API**
* Putem defnini si **operatii intermediare** care returneaza stream, ceea ce permite mai multor operatii sa fie inlantuite sau chained. Asta se cheama **PIPELINING.** De ex metoda **filter()** tot returneaza un stream pe baza la un alt stream, adica cel care o apeleaza si aceasta metoda e un **pipeline**
* 
* Deci intai avem o sursa de date pentru stream, ca un ArrayList
* Apoi pot fi difertie operatii, ca filter, sort etc. care tot returneaza stream
* Terminal operation care returneaza valorile scalare, adica propiu zise
* Collections sunt iterate explicit cu iterators(external iteration) dar putem folosi si alta abordare cu streams, numita **internal iteration**
* **O operatie itermediara** returneaza un stream, deci putem apoi sa mai facem si alte operatii asupra celei returnate. Ex de operatie itermediara **filter()**
* **Atentie, operatiile terminale, ca min(), max(), etc., adica care returneaza valori scalare, nu streamuri, odata efectuate, nu ne mai permit efectuarea altor operatii asupra la stream caci el va fi inchis!**



**PipeLine exemple**

* Putem defnini si **operatii intermediare** care returneaza stream, ceea ce permite mai multor operatii sa fie inlantuite sau chained. Asta se cheama **PIPELINING.** De ex metoda **filter()** tot returneaza un stream pe baza la un alt stream, adica cel care o apeleaza si aceasta metoda e un **pipeline**
* int[] array = {1,2,3,4,5,6,7,8,9};  
  Arrays.*stream*(array).filter(x -> x>4).forEach(System.*out*::println);

.stream() a returnat un stream

tot acest stream a apelat metoda filter() ce a returnat un alt stream, si filter e un pipeline

**Exemplu**

* Sa zicem ca avem un array de valori integer. Acum, daca am vrea sa afisam toate elementele din array, ar trebui sa cream un for, si sa afisam fiecare element
* Totusi, am putea face asta mai usor cu streams:

public static void main(String[] args){  
 int[] array = {1,2,3,4,5,6,7,8,9};  
 Arrays.*stream*(array).forEach(System.*out*::println);

Deci, metoda stream din Arrays returneaza un stream pentru array introdus de noi, apoi folosim metoda forEach pentru obiectul de tip Stream primit si oferim o method reference pentru a arata ce sa faca cu fiecare element

* **Source** – array
* **Intermediate Operation** – stream() caci doar returneaza un stream
* **Terminal operation** - forEach
* **Arrays.stream(array)** – returneaza un stream

**IntStream, LongStream, DoubleStream**

* **IntStream.of(number)** – returneaza un obiect de tip IntStream ce are reprezentat numarul trimis ca parametru ca stream
* IntStream a = IntStream.*of*(5);  
  a.forEach(System.*out*::println);
* **IntStream.of(number1,nr2,nr3,....,) –** returneaza un obiect de tip IntStream ce are reprezentant ca stream numerele trimise
* **IntStream.range(from,to)** – creaza un IntStream ce contine numere de la from pana la to
* forEach(lambda)
* sum() – returneaza suma elementelor in int
* min() – return OptionalInt
* max() - return OptionalInt
* count() – nr de elemente in stream
* map(lambda) – returneaza un stream nou creat prin efectuarea uneo operatii asupra fiecarui element
* **filter(lambda)** – e necesara o lamba expression care va avea un parametru, ce va fi fiecare element, si care sa returneze true daca acel element e pastrat si false daca nu. Noi facem un algoritm care sa decida in ce conditii un item e pastra sau nu. returneaza un obiect de tip IntStream
* limit(nr) – returnam doar atatea obiect cate scriem in nr pe baza la stream
* distinct() – returneaza stream cu elemente distincte. De ex, daca avem un array {1,2,3,4,1,5,6,1} si il transformam in stream si apoi folosim distinct(), vom avea avea {1,2,3,4,5,6}
* List<Integer> list = Arrays.*asList*(1,2,3,5,6,1,2,3,6);  
  list.stream().distinct().forEach(System.*out*::println);



**Stream Class**

* **of**(orice obiect) – returneaza un obiect de tip Stream a acelui obiect. Foloseste generics
* Clasa Stream e buna si pentru a lucra cu Stringuri
* String[] names = {"Eduard","Alex","Ion","Adelina","Pavel"};  
  Stream.*of*(names).sorted().forEach(System.*out*::println);

sau pentru a sorta invers:

String[] names = {"Eduard","Alex","Ion","Adelina","Pavel"};  
Stream.*of*(names).sorted(Comparator.*reverseOrder*()).forEach(System.*out*::println);

* toate metodele de la IntStream si restul tot sunt valabile
* filter de ex:
* String[] names = {"Eduard","Alex","Ion","Adelina","Pavel"};  
  Stream.*of*(names).filter(x -> x.startsWith("A")).forEach(System.*out*::println);

afisam doar numele ce incep cu A

**Collections and Streams**

* Collections sunt structuri de date din RAM
* Collections sunt pentru a stoca datele, streams pentru a le prelucra prin operatii complexe, deci collections sunt despre date, streams despre operatii asupra datelor
* In Collections putem adauga sau sterge date, in streams nu, ele transforma datele oferite in streams si mai mult nu pot fi modificate.
* Colectiile au nevoie de intreaga data din timp, adica daca de exemplu modificam elementele unei liste, chiar daca ele vor fi modficate rand pe rand, nu vom putea accesa primul element pana toate elementele nu vor fi modificate, si chiar de nu il accesam, ele oricum vor fi toate modificate(eager)
* Streams ofera interfate structurilor de date care reprezinta multimi secventiale de valori(ca cadrele video care sunt trimise ca stream in internet)
* Streams sunt structuri de date fixe ale caror elemente sunt calculate la cerere(deci streams sunt lazy collections), adica elementele sale sunt oferite doar la cerere, nu toate deodata.
* **Orice collection are metoda stream()**

**Lazy Streams and Eager Collections**

* Spre exemplu, fie codul:

list.stream()

.map(a -> a \* a)

.filter(a -> a > 0 && a < 100)

.map(a -> -a)

.forEach(a -> System.out.println(a));

* Daca streams ar fi eager, atunci dupa ce stream() creaza streamul, map() ar lua intregul stream si l-ar transforma pe tot in unul nou, apoi filter() ar lua intregul nou stream si ar transforma iar fiecare element din stream in unul nou si apoi map() iar in unul nou,si forEach l-ar folosi pe cel final in intregime, dar asta nu are loc.
* Deoarece streams sunt lazy, operatiile intermediare creaza cate o bucata de stream, necesara operatiei terminale. Deci, cand forEach va avea nevoie de primul element, abea atunci se va lua primul element din stream, si va fi trecut prin map() filter() map() si va fi generat asa o noua bucata de stream si apoi iar si iar, si bucata asta va ajunge la forEach pentru primul element, apoi cand va fi nevoie de al 2 element iar se va mai genera o bucata de stream de la fiecare metoda si tot asa
* Deci, asta inseamna ca stream e lazy, daca nu ar fi fost acel forEach, streamul returnat de stream() nici nu ar fi fost prelucrat de urmatoarele **operatii intermediare**
* La collections lucrurile nu stau asa. Ele modifica in intregime datele la apelarea unei metode, nu doar la cerere. Ele sunt eager. De ex, daca intr-o lista se modifica un element,dintr-un milion, nu vom putea accesa acel element pana nu vom avea intreaga lista cu elemente, deci daca de ex noi alocam memorie pentru acel element, nu vom putea sa il modificam, caci lista ofera deodata toate elementele si va vaea nevoie memorie cat pentru toate.
* Anume de asta **streams sunt mai rapide**, caci nu trebuie sa prelucreze toate datele deodata in operatiile intermediare, ci bucata cu bucata si le trimit bucata cu bucata la operatiile terminale, si asa de ex, am putea primi cate un element prelucrat rand pe rand, nu sa asteptam pentru toate odata.

[What does it mean intermediate operations are lazily executed whereas terminal operations are eagerly executed in java Stream API? - Stack Overflow](https://stackoverflow.com/questions/52926280/what-does-it-mean-intermediate-operations-are-lazily-executed-whereas-terminal-o)

Stream<String> stream = books.stream().filter(x->{  
 System.*out*.println("filtering");  
 return x.getPages() > 500;  
}).map(x->{  
 System.*out*.println("Mapping");  
 return x.getAuthor();  
});

Dar vedem ca nu avem niciun output, asa cum filter si merge nu au fost executate:



Ele vor fi executate doar cand va fi necesar de a accesa elememntele, de ex in forEach sau etc.

**Streams with custom Objects**

* Daca dorim sa transformam un stream intr-un obiect de tip collection, ca List de exemplu, putem folosi metoda **collect(Collectors.toTip),** care va transforma stream in acel tip de collection si o va returna

public class MyApp {  
 public static void main(String[] args){  
 List<Book> books = new ArrayList<>();  
 books.add(new Book("Being and Time", "Heidegger", 560, Type.*PHILOSOPHY*));  
 books.add(new Book("The Trial", "Franz Kafka", 240, Type.*NOVEL*));  
 books.add(new Book("Death on The Nile", "Agatha Christie", 370, Type.*THRILLER*));  
 books.add(new Book("Ancient Greece", "Robert F.", 435, Type.*HISTORY*));  
 books.add(new Book("Ancient Rome", "Robert F.", 860, Type.*HISTORY*));  
 books.add(new Book("Death of Virgil", "Hermann Broch", 590, Type.*NOVEL*));  
 books.add(new Book("The Stranger", "Albert Camus", 560, Type.*NOVEL*));  
  
 List<Book> result = books.stream().filter(x -> x.getType()==Type.*NOVEL*).collect(Collectors.*toList*());  
 result.stream().forEach(System.*out*::println);  
 }  
}  
class Book{  
 private String author;  
 private String title;  
 private int pages;  
 private Type type;  
  
 public Book(String title, String author, int pages, Type type) {  
 this.author = author;  
 this.title = title;  
 this.pages = pages;  
 this.type = type;  
 }  
  
 public String getAuthor() {  
 return author;  
 }  
  
 public void setAuthor(String author) {  
 this.author = author;  
 }  
  
 public String getTitle() {  
 return title;  
 }  
  
 public void setTitle(String title) {  
 this.title = title;  
 }  
  
 public int getPages() {  
 return pages;  
 }  
  
 public void setPages(int pages) {  
 this.pages = pages;  
 }  
  
 public Type getType() {  
 return type;  
 }  
  
 public void setType(Type type) {  
 this.type = type;  
 }  
  
 @Override  
 public String toString() {  
 return "Book{" +  
 "author='" + author + '\'' +  
 ", title='" + title + '\'' +  
 ", pages=" + pages +  
 ", type=" + type +  
 '}';  
 }  
}  
  
enum Type{  
 *NOVEL*, *FICTION*, *HISTORY*, *THRILLER*, *PHILOSOPHY*;  
}

**Atentie, toList returneaza List, nu ArrayList sau LinkedList sau altceva**

* **collect(Collectors.toList())** poate fi inlocuit cu .**toList()**
* Putem folosi si **sorted()** pentru a sorta obiectele noastre, dar avem nevoie sa-i trimited lui sorted() un parametru de tip Comparator:

List<Book> result = books.stream().filter(x -> x.getType()==Type.*NOVEL*)  
 .sorted(Comparator.*comparing*(Book::getAuthor))  
 .collect(Collectors.*toList*());

si aici spunem anume ca sortarea se va face dupa Author attribute

Nu putem in mod normal trimite o referinta la o metoda nonstatica prin Clasa::, dar aici putem, asa cum metoda e facuta special ca sa acceseze metoda trimisa anume de la fiecare element trimis catre ea in timpul sortarii

* Metoda **map(lambda)** mereu ia o lambda expression care trebuie neaparat sa returneze ceva. map() creaza un nou stream pe baza la cel vechi, aplicand o anumita operatie definita de noi asupra fiecarui element, deci fiecare element pur si simplu va fi dat acestei lambda expression, si ce va returna ea va fi pus in noul stream

Daca lambda expression returneaza String, pai atunci vom putea converti streamul doar intr-o lista List<String> si tot asa, deci depinde de ce returneaza ea.

De ex:

Fie ca vrem sa returnam o lista ce contine totii autorii cu uppercase name:

List<String> authors = books.stream().map(x->x.getAuthor()).map(String::toUpperCase).collect(Collectors.*toList*());  
authors.stream().forEach(System.*out*::println);

Deci primul map() a returnat autorii, al doilea a transformat fiecare nume in uppercase

* Putem pastra o referinta la un obiect de tip Stream cu **Stream <>**
* Stream<Book> stream = books.stream().sorted(Comparator.*comparing*(Book::getAuthor));
* Putem si grupa elementele dupa un atribut, de ex dupa tipul de roman:

Map<Type, List<Book>> map = books.stream().collect(Collectors.*groupingBy*(Book::getType));  
map.entrySet().stream().forEach(System.*out*::println);

Asa dar, cand am scris Type intre <>, Stream va sti ca sortarea se va face dupa Type, si apoi e logic ca fiecare type va fi un key, si value a sa va fi o lista cu elementele corespunzatoare

Folosim entrySet() deoarece Map nu are metoda stream(), dar metoda converteste Map in Set, si Set are metoda stream()

**Atentie! groupingBy returneaza un Map, nu HashMap!**

* limit()

De ex, sa afisam primele 2 carti din lista care au mai mult de 500 de pagini:

List<String> list = books.stream().filter(x -> x.getPages()>500).map(x->x.getTitle()).limit(2).collect(Collectors.*toList*());  
list.stream().forEach(System.*out*::println);



**External vs Internal iteration**

* **External Iteration e cea facuta cu for()**
* for(Book b : books)  
   System.*out*.println(b);

dar tot asta e echivalent cu asta:

Iterator<Book> iterator = books.iterator();  
while(iterator.hasNext())  
 System.*out*.println(iterator.next());

* Internal Iteration e cea pe care o putem face fara a folosi bucle for sau while, ca de ex:
* books.forEach(System.*out*::println);

sau

books.stream().forEach(System.*out*::println);

Internal Iteration e mai bun, deoarece External Iteration pur si simplu foloseste Iterator al structurii de date si ia itemele unul dupa altul, dar unele iteme ar putea fi poate plasate unul langa altul in memorie, si asa daca ar fi accesate pur si simplu unul dupa altul in memorie, s-ar optimiza viteza. Cu external iteration asa ceva nu e posibil, dar internatl iteration face anumite optimizari.

**Handle Files with Streams**

Stream<String> stream = Files.*lines*(Paths.*get*("C:\\Users\\user\\Desktop\\Sales.txt"));  
stream.forEach(System.*out*::println);

* Deci, cu Files.lines() putem transforma toate liniile din file in stream si lucra cum vrem cu ele
* Files. e din nio library

**Short-circuiting and loop fusion**

* Short circuiting si loop fusion sunt concepte care fac Streams foarte eficiente
* **Loop fusion** – proces in care diferite operatii sunt executate una dupa alta pentru fiecare element,exact ca intr-un loop
* List<String> list = books.stream().filter(x->{  
   System.*out*.println("filtering");  
   return x.getPages() > 500;  
   }).map(x->{  
   System.*out*.println("Mapping");  
   return x.getAuthor();  
   }).toList();



Vedem ca nu a fost executata metoda filter() pentru toate elementele deodata, ca sa faca un stream format din toate elementele, ce sa ajunga apoi la map(). In realitate, fiindca streams sunt lazy, intai s-a executat metoda filter() pentru primul element, apoi daca acesta a mers, s-a executat map() si apoi iar filter pentru urmatorul si tot asa.

* **Short-circuiting** – operatiile a caror executare este inutila, sunt eliminate si nu mai sunt executate. De ex:
* List<String> list = books.stream().filter(x->{  
   System.*out*.println("filtering");  
   return x.getPages() > 500;  
   }).map(x->{  
   System.*out*.println("Mapping");  
   return x.getAuthor();  
   }).limit(2).toList();  
  }



Daca nu era limit(2), s-ar fi facut 4 mapari, dar aici vedem ca s-au facut doar 2 mapari. Asta e din cauza ca Stream stie ca va trebui sa prelucreze doar 2 iteme, deci odata ce le-a prelucrat, pe restul nici nu le mai atinge, ca sa nu faca operatii degeaba. Iata inca un motiv de ce Stream e mai rapid ca Collections, caci in Collections asa ceva nu se va intampla.

**map() vs flatMap()**

* map() si flatMap() sunt ca **select** din SQL, caci putem selecta cate o valoare si sa facem operatii asupra ei
* flatMap() – metoda pentru a uni mai multe Stream in unul singur.

De ex, daca avem:

List<String> list = Arrays.*asList*("Eduard","Alex","Jack","Anthon","Evie");  
List<String[]> list2 = list.stream().map(x->x.split("")).distinct().collect(Collectors.*toList*());

ei bine, list2 e o lista de String[], nu de String, caci vom obtine asa:

{{“E”,”d”,”u”,”a”,”r”,”d”},{“A”,”l”,”e”,”x”},...}

Dar, noi vrem ca aceste arrays sa fie unite in unul singur, si anume aici si ne ajuta flatMap

List<String> list = Arrays.*asList*("Eduard","Alex","Jack","Anthon","Evie");  
List<String> list2 = list.stream().map(x->x.split("")).flatMap(Arrays::*stream*).distinct().collect(Collectors.*toList*());

Deci flatMap a transformat fiecare array intr-un stream si le-a combinat in unul.Deci, in loc de List<Stream<String[]>> avem List<String[]>. flatMap() a unit continuturile la toate streamurile in unul.

flatMap ne ajuta sa scapam de situatiile cand avem **Stream<Stream<T>>,** si le transformam in Stream<T>. sau List<Stream<String>> ca sa avem doar List<String>. Metoda ia continutul streamurilor si il transforma intr-un stream doar

**Atentie!Stream<Stream<T>> sau List/.../<Stream<...>> nu alt tip.**

List<String> list = Arrays.*asList*("Eduard","Alex","Jack","Anthon","Evie");  
List<String> list20 = list.stream().flatMap(x->Arrays.*stream*(x.split(""))).toList();

Deci, map ar trebui sa returneze un Stream<Stream<T>>, dar flatMap are grija sa combine toate in unul, deci Stream<T>

Deci, map ia pur si simplu cate un element si la final returneaza un stream cu toate elementele modificate, dar flatStream are grija sa elimine Stream<Stream<?>>

Exemplu:

Integer[] list1 = {1, 2, 3};  
Integer[] list2 = {4, 5};

List<List<List<Integer>>> test = Arrays.*stream*(list1).map(x->  
 Arrays.*stream*(list2).map(y->Arrays.*asList*(x,y)).toList()).toList();

Primim un List<List<List<Integer>>> sau un Stream<Stream<List<Integer>>>

Fiecare pereche va fi o lista de tip Integer, si toate combinarile facute de un numar cu altele, adica toate perechele ce un nr le creaza cu altele iar va fi o List<List<>> si apoi toate perechile la toate numerele iar vor fi o List<List<List<Integer>>>

Solutie:

Integer[] list1 = {1, 2, 3};  
Integer[] list2 = {4, 5};  
  
List<List<Integer>> test = Arrays.*stream*(list1).flatMap(x->  
 Arrays.*stream*(list2).map(y->Arrays.*asList*(x,y))).toList();

**min() si max() method**

* Ambele metode au nevoie de o lambda expression pentru a compara 2 elemente, si sa returneze 1,0 sau -1. Iau un parametru de tip **Comparator**
* Putem folosi pentru asta **Integer::compare, daca sunt doar int values**
* Aceste metode returneaza un obiect de tip **Optional<>**
* **Optional<>** este pur si simplu un container pentru a stoca o valoare, sau nici una, in loc de a primi un null
* Metode pentru Optional:
* get()
* isPresent()
* isEmpty()
* map()
* stream()
* ifPresent(lambda expression) – daca e prezenta o valoare, executa lambda expression
* orElse(0) – returnam valoarea 0 daca nu exista nicio valaore stocata in Optional object, dara daca exista, va returna pe cea existenta.
* List<Integer> lista = Arrays.*asList*(10,20,25,50,14,58,7);  
  Optional<?> max = lista.stream().max(Integer::*compare*);  
  Optional<?> min = lista.stream().min(Integer::*compare*);  
  System.*out*.println(max.get());  
  System.*out*.println(min.get());

List<Integer> lista = Arrays.*asList*(10,20,25,50,14,58,7);  
lista.stream().min(Integer::*compare*).ifPresent(System.*out*::println);

* **Putem crea si noi propria expresie lambda, de ex daca vrem sa returnam cartea cu cele mai multe pagini, dar evident vom face o metoda identica cu compare cu cea din Comparator, asa cum are nevoie de un parmetru de tip Comparator:**

books.stream().max((b1,b2)->b1.getPages() > b2.getPages() ? 1 : (b1.getPages() < b2.getPages() ? -1 : 0)).ifPresent(System.*out*::println);

* **Putem folosi Comparator.comparing(Clasa::metodaGetter) pentru a arata dupa care atribut sa se caute min**

books.stream().min(Comparator.*comparing*(Book::getPages))

**reduce(prev,lambda)**

* Combina elementele dintr-un stream pentru a returna o singura valoare ca rezultat
* Sunt si alte metode care fac asta, ca min(),max() etc.
* **prev** – elementul cu care se va incepe reducerea elementelor
* **lambda –** metoda care face ceva cu 2 elemente si returneaza un rezultat. Lambda expression trebuie sa aiba neaparat 2 parametrii. Primul parametru va fi rezultatul aflat pana la un anumit element din stream, si al doilea parametru va fi acel element.
* **De ex:**

List<Integer> list = Arrays.*asList*(1,5,6,8,9,10);  
int sum = list.stream().reduce(0,(x,y)->x+y);  
System.*out*.println(sum);

* Deci, reduce, ca si max si min, ia fiecare element din stream rand pe rand
* Acum, x va fi rezultatul obtinut pana la elementul selectat, si y va fi elementul selectat
* cand abea luam primul element, adica 1, metoda lambda nici nu a fost inca executata, si nu e niciun rezultat precedent, de aia metoda reduce are un parametru care va fi valoarea initiala pentru rezultat, adica 0
* deci, intai x=0 si y = 1, deci rezultatul e 1
* apoi, se ia 5, deci x = 1, y = 5, rezultatul de 6
* apoi se ia 6, dexi x = 6, y = 6 deci rezultatul e 12
* apoi se ia 8, deci x = 12, y = 8 rezultat = 20 si tot asa
* Putem avea si **reduce(Lambda**) – fara un parametru initial. Atunci se vor lua deodata primul si al 2 element, nu doar primul. Apoi se va lua rezultaltul si al 3, apoi noul rezultat si al 4 si tot asa. Adica, se pune 0 automat ca si cum.
* int[] array = {1,2,3,4,5,6,7,8,9,10};  
    
  Optional<Integer> optional = Arrays.*stream*(array).boxed().reduce((x,y)->x+y);  
  System.*out*.println(optional.get());

x = 1 y = 2 , deci 3

x = 3, y = 3, deci 6

x = 6, y = 4, deci 10

....

**Sum() si mapToInt(),boxed() method**

* Metoda sum() este disponibila doar pentru IntStream, DoubleStream si LongStream
* map() returneaza un stream, dar metoda mapToInt() returneaza un IntStream

De ex, sa sumam toate paginile la toate cartile:

IntStream intStream = books.stream().mapToInt(Book::getPages);  
System.*out*.println(intStream.sum());

* boxed() – transforma IntStream in Stream
* IntStream intStream = books.stream().mapToInt(Book::getPages);  
  Stream<?> stream = intStream.boxed();

**OptionalInt/Long/Double**

* Metodele min(), max() de la IntMap/DoubleMap si LongMap returneaza nu Optional, dar OptionalInt/Long/Double. Atentie! Ele nu mostenesc Optional!

Optional<?> o = books.stream().mapToInt(Book::getPages).max();

**Vom primi eroare! Ne trebuie OptionalInt<>!!!**

**allMatch(),noneMatch(),findFirst() findAny()**

* **allMatch(Predicate)** – returneaza true daca toate elementele corespund conditiei predicat, si false daca nu toate
* **noneMatch(Predicate) –** returneaza true daca nici-un element nu corespunde conditiei, sau false in caz contrar
* **findAny() –** returneaza un element din stream random. Returneaza un obiect de tip Optional. E pentru parallelization
* **findFirst() –** returneaza primul element din Stream**.** Returneaza un obiect de tip Optional. E pentru sequantial approach

**Parallelization**

* Parallelization e foarte usoara cu Stream API
* Tot ce trebuie sa facem e sa apelam metoda **parallel()**
* Totusi, nu e nicio garantie ca va fi executat mereu in mod paralel, deoarece trebuie sa fim siguri ca streamul poate fi **parallelized**
* De exemplu, findFirst() nu e pentru abordarea paralela, asa cum mereu returneaza doar primul element, dar findAny() returneaza orice element random din streamul dat.
* **Stream API foloseste Fork-Join Framework**
* public class Parallel {  
   public static void main(String[] args){  
   long nr = 1000000000;  
   long time = System.*currentTimeMillis*();  
   *sequential*(nr);  
   System.*out*.println("Sequential: "+(System.*currentTimeMillis*() - time));  
   time = System.*currentTimeMillis*();  
   *parallel*(nr);  
   System.*out*.println("Parallel: "+(System.*currentTimeMillis*()-time));  
   }  
   public static void sequential(long nr){  
   System.*out*.println(LongStream.*range*(0,nr).reduce(0,Long::*sum*));  
   }  
   public static void parallel(long nr){  
   System.*out*.println(LongStream.*range*(0,nr).parallel().reduce(0,Long::*sum*));  
   }

}



* **parallel Stream** cu metoda parallel() se asigura sa divida streamul in mai multe streamuri, si sa execute taskul pentru fiecare pe mai multe cores, si apoi sa ia rezultatele de la fiecare si sa il combine in unul

**More about Collectors**

* Pe langa groupingBy(),toList(),toSet() mai sunt si alte metode la Collectors
* joining(separator) - ne ajuta sa unim mai multe stringuri in unul, si toate sa fie separate prin separator

public static void main(String[] args) {  
 List<String> list = List.*of*("user1","user2","user3");  
  
 String together = list.stream().collect(Collectors.*joining*(","));  
 System.*out*.println(together);  
}

Vom avea: user1,user2,user3

* counting() - doar calculeaza numarul de elemente si returneaza un long

List<String> list = List.*of*("user1","user2","user3");  
  
Long together = list.stream().collect(Collectors.*counting*());  
System.*out*.println(together);

**Rezultat: 3**

* maxBy(Comparator), minBy(Comparator) - gaseste elementul maxim sau minim dupa comparator

List<String> list = List.*of*("abc","abcderg","gfgfg");  
  
Optional<String> longest = list.stream().collect(Collectors.*maxBy*(  
 (s1,s2) -> {  
 if(s1.length() == s2.length())  
 return 1;  
 else if(s1.length() > s2.length())  
 return 1;  
 else  
 return -1;  
 })  
);  
System.*out*.println(longest.get());

* **groupingBy(method1, MetodaDeTransformat) -** method1 e criteriul dupa care se grupeaza, iar urmatorul argument efectueaza o metoda asupra la value din hashMap returnat, fiind anume value

method1 - key va fi, MetodaDeTransformat - value

ist<String> list = List.*of*("abc","abcderg","gfgfg","fre");  
  
Map<Integer, Set<String>> map = list.stream().collect(Collectors.*groupingBy*(String::length,Collectors.*toSet*()));

Deci, vom avea Map<Integer, Set<String>>

* **partitioningBy(Predicate)** - returneaza un Map ce va avea true si false ca key, si elementele ce corespund predicatului vor fi la true, celelalte la false

List<String> list = List.*of*("abc","abcderg","gfgfg","fre");  
  
Map<Boolean, List<String>> map = list.stream().collect(Collectors.*partitioningBy*(s->s.length() > 3));  
map.entrySet().forEach(System.*out*::println);



* **teeing(MinFunc,MaxFunc,BiFunc) -** aflam minimul, maximul si facem ceva cu ele si returnam ceva
* **mapping(Function, Collector.metoda) -** fiecare element, inainte de a trece la collector method, trece prin function, se returneaza unul nou si apoi se face collectarea

List<String> list = List.*of*("abc","abcderg","gfgfg","fre");  
  
List<String> list2 = list.stream().collect(Collectors.*mapping*(String::toUpperCase,Collectors.*toList*()));

**Function.identity**

Simpla metoda ce returneaza ce-i dam:

(t) -> t, nu face nimic.

Asa anume daca avem o lista de stringuri, putem returna un hashmap cu stringurile si numarul lor de aparitii

List<String> list = new ArrayList<>();

list.add("Hello");

list.add("Hello");

list.add("World");

Map<String, Long> counted = list.stream()

.collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));

**toMap(forValueMethod, ForValueKey)**

Valoarea pastrata din stream va fi trimisa la prima metoda, ca sa fie ca key

Si asa am transforma un List in HashMap

Fie List<User>

Map<Privilege, List<User>> map = users.stream().flatMap(x -> x.getPrivileges().  
 stream()).distinct()  
 .collect(Collectors.*toMap*(Function.*identity*(), v2 -> new ArrayList<>()));