

Stream Api - представлява библиотека, която ни позволява да правим заявки към различни колекции и да обработваме информацията от тях.

Гледайки примерно една колекция от числа(1 1 2 3 4 5 5) , ние да и наредим няколко заявки и накрая тя да ни върне резултатът , който желаем.

Java Advanced - Built-in Query methods - Stream API - януари 2017 - Петър Пенев

Stream<T> Class

- Gives access to Stream API functions.
- Get an instance through:

Всяка една междинна функция в Stream Api ни връща резултат и той е Stream<T>, тоест стриймовете работят през този клас. Това <T> означава , че този клас е Generic и може да работи с всякакви други типове. Инстанция на този клас се взема през различни други неща. Примерно колекции. Един от начините е следния:

При масивите е малко по-различно, защото те са по-различни от всички останали колекции. Те са примитивен тип данни, докато колекциите са някакви обекти, които в някои случаи работят отдолу с масиви, а в други - не. Масивите са градивна част от езика и затова с тях правим по-различни неща от листовите например.

- An Array:

```
String[] array = new String[10];  
Stream<String> stream = Arrays.stream(array);
```

Инстанция върху HashMap:

- При хешмаповете също е различно, защото те не са едно измерими(с 1 измерение) а си имат ключове и стойности, тоест колекция от двойки ключове-стойности. Тук имаме свободата да извикаме stream върху която част от мапа искаме, тоест можем да извикаме stream само по ключовете , само по стойностите или върху двете взети заедно.(entryset).

- A Hash Map Collection:

```
HashMap<String, String> map = new HashMap<>();
```

Ако искаме да извикаме stream върху entrySet-а на мапа

```
HashMap<String, String> map = new HashMap<>();  
  
Stream<Map.Entry<String, String>> entries =  
    map.entrySet().stream();
```

Ако искаме стрийма да е върху ключовете , викаме `keySet()` както е показано на следваща картинка:

```
Stream<String> keys = map.keySet().stream();
```

По подобен начин е и със стойностите само че вместо `map.keySet()` , е `map.values()`.

Едното е `keySet`, защото ключовете винаги трябва да са уникални и сетовете осигуряват това.

Когато извикаме стрийм върху дадена колекция , примерно ключовете и след това повикаме някаква филтрираща функция дали тя ще филтрира и нещата в мапа ?

Отговорът е , че не , няма. Примерно `Stream`-а на горната снимка е напълно отделен от колекцията. `Stream<T>` не е колекция. Той е поток. Той знае за някаква информация и може да я чете. Тръгва от единия край и свършва в другия. Но той НЕ ПАЗИ ИНФОРМАЦИЯ.

Как се изпълняват функциите в `StreamAPI` ?

- Изпълняват се една след друга. Но има и различни видове функции - междинни и такива които затварят потока. Първите винаги връщат някакъв обект, който отново е стрийм от нещо.

- Each function call creates a new `Stream<T>` instance
- This allows method chaining

```
List<String> strings = new ArrayList<>();  
  
Stream<String> stringStream = strings.stream();  
  
Stream<Integer> intStream =  
    stringStream.map(s -> s.length());
```

Обобщение на това какво е `Stream` ?

`Stream` не е колекция. Той не държи в себе си информация за колекцията, която разглежда, по-скоро итерира(обхожда) всеки един елемент. Започва от единия край и итерира до другия, но в нито един момент не знае нищо за колекцията. Освен това стриймовете не модифицират(променят) информацията в колекцията , която обхождат.

`Streams` биват 2 вида:

- Generic streams - работят с всякакви типове данни.

- Can be of any type except primitives

```
List<String> strings = new ArrayList<>();  
Stream<String> strStream = strings.stream();
```

```
List<Integer> ints = new ArrayList<>();  
Stream<Integer> intStream = ints.stream();
```

```
List<Object> objects = new ArrayList<>();  
Stream<Object> objStream = objects.stream();
```

- Primitive streams - работят с примитивни типове данни. И понеже те не са класове затова трябва да имаме отделен стрийм за всеки примитивен тип отделно.

За всеки един примитивен тип си имаме и такъв стрийм.

```
int[] ints = { 1, 2, 3, 4 };  
IntStream intStream = IntStream.of(ints);
```

Друг начин по който можем да си изкараме примитивен поток е следния:

```
List<Integer> list = new ArrayList<>();  
IntStream mappedIntStream = list.stream()  
    .mapToInt(n -> Integer.valueOf(n));
```

Ако използваме .map вместо mapToInt , то потокът ще стане Generic , а не такъв какъвто в момента искаме - примитивен.

Какво е Optional<T> ?

Optional-ът е нещо, което стриймовете могат да ни върнат като резултат. Какво означава това ? .. Има функции , които ни връщат един единствен елемент и той може да го има или да го няма. Примерно на празен лист да му кажем да ни върне средно аритметичното на всички числа в него. Само че той е празен и би ни върнал null. От Oracle обаче са решили , че това не върши работа и са направили един wrapper class , който просто ни обгръща резултата. Optional-ът просто гледа дали един резултат го има или е null. Optional съхранява резултата на един единствен елемент. Да не се бърка с колекции като Лист например.

Optional<T>

- Some functions can return Optional<T>

```
Optional<String> first = elements.stream()
    .sorted()
    .findFirst();

if (first.isPresent()) {
    System.out.println(first.get());
} else {
    System.out.println("No matches.");
}
```

Различни видове операции.

Types of Operations

Intermediate, Terminal

- междинни - такива които връщат като резултат stream, който можем да chain-ваме (свързваме) колкото си искаме докато накрая не затворим потока с терминараща, крайна функция, която да върне конкретен резултат.

- крайни(финални, терминаращи и тн.) - такива са тези, които при извикването си затварят потока, тоест връщат конкретен резултат.

Междинни - те не затварят потока и връщат като резултат stream, който може да се свързва с още междинни функции докато се стигне до терминараща такава.

Intermediate Operations

- Does not terminate the Stream

```
List<String> elements = new ArrayList<>();
Collections.addAll(elements, "one", "two");
Stream<String> stream = elements.stream()
    .distinct()
    .sorted()
    .filter(s -> s.length() < 5)
    .skip(1)
    .limit(1);
```

Терминиращи операции - такива , които затварят потока и връщат конкретен резултат.

Java Advanced - Built-in Query methods - Stream API - януари 2017 - Петър Пенев



Terminal Operations

■ Terminates the stream

```
List<String> elements = new ArrayList<>();
Collections.addAll(elements, "one", "two");

elements.stream()
    .distinct()
    .forEach(s -> System.out.println(s))
```

Най-често използвани и полезни терминиращи функции са показани на следващата снимка:

■ Useful terminal operations:

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
forEach	side effect	to perform a side effect on elements

Друг начин, по който можем да разглеждаме тези операции е т.нар. pattern Map, Filter, Reduce. Какво представлява този pattern ?

- Много често използван начин за решаване на проблеми в програмирането, който ни казва да вземем елементите, да ги преобразуваме в нещо. След това да филтрираме тези, които ни трябва и най-накрая да се агрегират по някакъв начин.

Map, Filter, Reduce

Common pattern in data querying

List<String>

"10"

"2"

"5"

"6"

.mapToInt(Integer::valueOf())

.filter(x -> x > 4)

.reduce((x, y) -> x + y)

Map Operations(Мапващи, преобразуващи операции) - Просто трансформират обектите от stream-а от едно нещо в друго.

Transform the objects in the stream

```
Stream<String> strStream =  
    Stream.of("1", "2", "3");
```

```
Stream<Integer> numStream =  
    strStream.map(Integer::valueOf);
```

Filter Operations(Филтриращи операции) - Филтрирането работи с predicate(Подаваме някакъв елемент и връща булева стойност)

Filters objects by a given predicate

```
Stream<String> strStream =  
    Stream.of("one", "two", "three")  
        .filter(s -> s.length() > 3);
```


▪ Check for a given condition:

▪ Any element matches:

```
boolean any = stream1.anyMatch(x -> x % 2 == 0);
```

▪ All elements match:

```
boolean all = stream2.allMatch(x -> x % 2 == 0);
```

▪ None of the elements match:

```
boolean none = stream3.noneMatch(x -> x % 2 == 0);
```

Reduce Operations(Агрегиращи операции) - Те биват:

- Такива, които търсят някакъв елемент. (.anyMatch)
- Такива, които проверяват всички елементи дали са от дадено условие. (allMatch).
- Такива, при които нито един елемент не отговаря на дадено условие. (noneMatch).
- Има и такива за търсене. (findFirst()). Връща първият елемент, който е в stream-а до момента.
- findAny(). Връща, който и да е елемент от стрийма. Подобно на HashSet. Там елементите имат случайна подредба.

▪ Find an element:

▪ Gets the first element of the stream:

```
Optional<Integer> first = list.stream()  
    .findFirst();
```

▪ Gets any element of the stream:

```
Optional<Integer> first = list.stream()  
    .findAny();
```

General Reduction - Това е Reduce метод, който можем да си ползваме сами. ТАЗИ REDUCE ФУНКЦИЯ НЕ НИ ПРЕДПАЗВА ДА ПРАВИМ ГРЕШКИ, ТОЕСТ АКО ПОДАДЕМ ГРЕШНА ЛАМБДА ТЯ ЩЕ НИ ВЪРНЕ ГРЕШЕН РЕЗУЛТАТ.

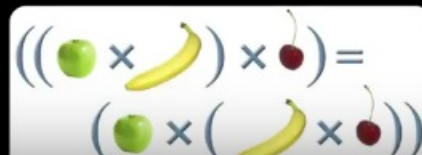
General Reduction

- Applies a given lambda:

```
Optional<Integer> first = list.stream()
    .reduce((x, y) -> x + y);
```

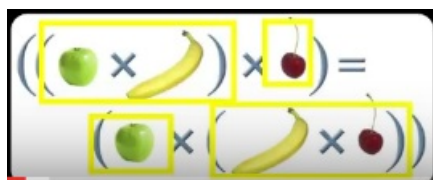
- Consider associativity:

$r(a, r(b, c))$ should be equal to $r(r(a, b), c)$



За да можем да прилагаме този Reduce метод, има едно условие което трябва да изпълним, за да имаме верен резултат накрая. Нариса се associativity(асоциативност). Какво означава това ?

Функцията reduce от a и функцията reduce от b и c , трябва да ни върнат същия резултат като функцията reduce. от функцията reduce на a и b и c.



Примерно имаме числата 1, 2, 3 и искаме да намерим тяхната сума -> $(1 + 2) + 3 = 6$

след това в обратен ред -> $1 + (2 + 3) = 6$

Втори пример с делене:

Имаме числата 12, 6, 3 -> Започваме отначалото $(12 / 6) / 3 = 2 / 3 = 2,3$

А след това наобратно -> $12 / (6 / 3) = 6$

Както се вижда от втория пример тук няма асоциативност и правилото не се спазва.

СТРИЙМ ОТ ПРИМИТИВЕН ТИП Е ЕДИН ЕДИНСТВЕН КЛАС И В НЕГО МОЖЕ ДА ИМА ТОЧНО ДЕФИНИРАНИ МЕТОДИ. ПРИМЕРНО IntStream МОЖЕ ДА ИМА ГОТОВИ ЗА ПОЛЗВАНЕ МЕТОДИ КАТО SUM, AVERAGE И ДРУГИ ПОЛЕЗНИ, КОИТО ДА ПОЛЗВАМЕ НАГОТОВО. ЗА РАЗЛИКА ОТ ПРИМИТИВНИТЕ ТИПОВЕ, GENERICS КАТО STREAM<T>, ПОТОКЪТ НЯМА КАК ДА ЗНАЕ КАКВО ИМА В <T> ПО ВРЕМЕ НА ИЗПЪЛНЕНИЕ НА ПРОГРАМАТА.

Сортиращи операции

Sorting

- Sort by passing a comparator as lambda:

```
List<Integer> numbers = new ArrayList<>();  
Collections.addAll(numbers, 7, 6, 3, 4, 5);  
  
numbers.stream()  
    .sorted((x1, x2) -> Integer.compare(x1, x2))  
    .forEach(System.out::println);
```

Потоци над HashMaps

Creating the Stream

- Use any dimension of the Hash Map:

- Stream over the Entry set:

```
Stream<Map.Entry<String, String>> entries =  
    map.entrySet().stream();
```

- Stream over the Key set:

```
Stream<String> keys = map.keySet().stream();
```

- Stream over the Value set:

```
Stream<String> keys = map.values().stream();
```

Collectors - За да съберем цял stream елементи в една колекция , трябва да използваме функцията .collect

Collectors

- Collecting a Stream into a list:

```
String[] strings = { "22", "11", "13" };  
List<Integer> numbers = Arrays.stream(strings)  
    .map(Integer::valueOf)  
    .collect(Collectors.toList());
```



Можем да collect-ваме към различни колекции - Arrays, Lists , Sets, HashMaps etc..

- You can collect streams into different collections:
 - Arrays, Sets, HashMaps, etc.

