Stream API

UA.DETI.POO

Iterar sobre coleções

- Iterator

```
List<String> names = Arrays.asList("Ana", "Ze", "Rui");
Iterator<String> it = names.iterator();
while (it.hasNext())
  System.out.println(it.next());
```

- ciclo "for each"

```
List<String> names = Arrays.asList("Ana", "Ze", "Rui");
for (String name : names)
    System.out.println(name);
```

- Método forEach

```
List<String> names = Arrays.asList("Ana", "Ze", "Rui");
names.forEach(s -> System.out.println(s)); // forEach com lambda
names.forEach(System.out::println); // forEach com referência de método
```

- Stream operations
  - *Aggregate operations*

Aggregate Operations – Streams API

- The preferred method of iterating over a collection is to obtain a stream and perform aggregate operations on it.
- Aggregate operations are often used in conjunction with lambda expressions
  - to make programming more expressive, using less lines of code.

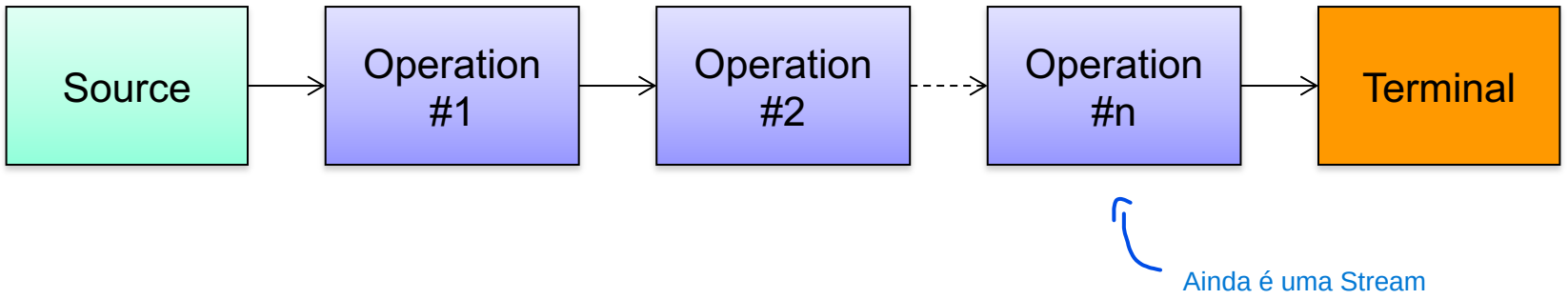
- Package java.util.stream
  - The key abstraction introduced in this package is stream.

Stream Pipeline

- (1) Obtain a stream from a source
- (2) Perform one or more intermediate operations

não são executadas até eu necessitar de executar

- (3) Perform one terminal operation



- Usage: Source.Op1.Op2 .. .Terminal

Sem operação terminal a pipeline existe mas não opera nada na "fonte"

java.util.stream

- Streams differ from collections in several ways:
- No storage
  - A stream is not a data structure that stores elements; instead, it conveys elements through a pipeline of computational operations.
- Functional in nature
  - An operation on a stream produces a result but does not modify its source.
- Laziness-seeking ('process-only, on-demand' strategy)
  - Many stream operations, such as filtering or mapping, can be implemented lazily, exposing opportunities for optimization. Intermediate operations are always lazy.

não são executadas até eu necessitar de executar, faz com que o compilador otimize as operações

- Possibly unbounded
  - While collections have a finite size, streams need not.
- Consumable

Imaginem que a Stream são valores lançados aleatoriamente, e nunca param. (ex: API Twitter)

  - The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

Para ver outra vez a mesma fonte temos de criar outra Stream! (ex: ler e operar um ficheiro em loop)

java.util.stream – Sources

- Streams sources include:
  - From a Collection via the stream() and parallelStream() methods;
  - From an Array via Arrays.stream(Object[]);
  - *and many more (files, random, ..)*

tem o processo de paralelismo, para que os métodos aplicados a seguir sejam aplicados em paralelo

java.util.stream – Intermediate operations

(Não devolvem nada! Permitem depois correr a Stream)

Para filtrar preciso de um "teste", implementar a class Predicate

- **filter** - excludes all elements that don't match a Predicate
- **map** - perform transformation of elements using a Function
- **flatMap** - transform each element into zero or more elements by way of another Stream

se o resultado da transformação do map, for uma lista (teria uma lista de listas) e o flatMap expandia/concatenava essas listas.

- **peek** - performs some action on each element
- **distinct** - excludes all duplicate elements (equals())
- **sorted** - ordered elements (Comparator)
- **limit** - maximum number of elements
- **substream** - range (by index) of elements
- *(and many more -> see java.util.stream.Stream<T>)*

```
List<Person> people = ...;
Stream<Person> tenPersonsOver18 = people.stream()
                .filter(p -> p.getAge() > 18)
                .limit(10);
```

São 2 operações intermédias, devolvem uma Stream. Só quando fizer uma operação terminal devolverá uma collection

Predicate, porque devolve um boolean

10 ou menos (se a lista não tiver 10 elementos fica menor)

java.util.stream – Terminating operations

- Reducers
  - reduce(), count(), findAny(), findFirst()
- Collectors
  - collect()
- forEach
- iterators

int

recolher os elementos de uma Stream para uma estrutura de dados

documentação java

Exemplos de Collectors:

-> .collect(Collectors.toCollection(TreeSet::new))
-> .collect(Collectors.joining(", "))
-> .collect(Collectors.summingInt(Employee::getSalary))
-> .collect(Collectors.groupingBy(Employee::getDepartment))
-> .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD))

.map(Employee::getSalary)
.sum()

```
// Accumulate names into a List
List<Person> people = ...;
List<String> names = people.stream()
                           .map(Person::getName)
                           .collect(Collectors.toList());
```

operação intermédia

operação terminal, que devolve uma coleção (devolve uma lista neste caso de String)

Stream.Filter

- Filtering a stream of data is the first natural operation that we would need.
- Stream interface exposes a filter method that takes in a Predicate that allows us to use lambda expression to define the filtering criteria:

```
List<String> l = Arrays.asList(”Ana Maria", ”Mariana", ”Rui”);

l.stream().filter(n -> n.length()>3)
          .forEach(System.out::println);
```

Stream.Map

- The map operations allows us to apply a function that takes in a parameter of one type and returns something else.

```
Stream<Student> map = persons.stream()
        .filter(p -> p.getAge() > 18)
        .map(person -> new Student(person));
```

(Outra utilidade: criar objetos)

```
// other example with Map && Consumer

List<String> l = Arrays.asList("Ana", "Ze", "Rui");
l.stream().map(n -> "Nome = " + n)
          .forEach(System.out::println);
```

Output:

```
Nome = Ana
Nome = Ze
Nome = Rui
```

Stream.Reduce

- A reduction operation takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation
- For instance, finding the sum or maximum of a set of numbers, or accumulating elements into a list.

```
// example with Map & Reduce
List<Integer> costBeforeTax = Arrays.asList(100, 200, 300, 400, 500);

double bill = costBeforeTax.stream()
           .map(cost -> (cost*1.23))
           .reduce(0.0,(sum, cost) -> sum + cost));
System.out.println("Total : " + bill);
```

acumulador

Vantagem de eu poder implementar vários métodos através da função lambda

[Poderia apenas usar o .sum()]

Stream.Collect

- The Stream API provides several "terminal" operations.
- The collect() method is one of those, which allows us to collect the results of the operations:

```
List<Student> students = persons.stream()
        .filter(p -> p.getAge() > 18)
        .map(Student::new)
        .collect(Collectors.toList());
```

Existem mais métodos do Collectors. Para diferentes estruturas (ex: Collectors.toMap())

```
// other example with Map && Collect
List<String> l = Arrays.asList("Ana", "Ze", "Rui");
List<String> res = l.stream()
        .map(n -> "Nome: " + n)
        .collect(Collectors.toList());
res.forEach(System.out::println);
```

Some examples using a list of strings

```
public static void listExample() {
    List<String> words = new ArrayList<String>();
    words.add("Prego");
    words.add("no");
    words.add("Prato");
    // old fashioned way to print the words
    for (int i = 0; i < words.size(); i++)
        System.out.print(words.get(i) + " ");      [0]
    System.out.println();

    // Java 5 introduced the foreach loop and Iterable<T> interface
    for (String s : words)
        System.out.print(s + " ");                 [1]
    System.out.println();

    // Java 8 has a forEach method as part of the Iterable<T> interface
    // The expression is known as a "lambda" (an anonymous function)
    words.stream().forEach(n -> System.out.print(n + " "));   [2]
    System.out.println();

    // but in Java 8, why use a lambda when you can refer directly to the
    // appropriate function?
    words.stream().forEach(System.out::print);     [3]
    System.out.println();

    // Let's introduce a call on map to transform the data before it is printed
    words.stream().map(n -> n + " ").forEach(System.out::print);
    System.out.println();                          [4]

    // obviously these chains of calls can get long, so the convention is
    // to split them across lines after the call on "stream":
    words.stream()
        .map(n -> n + " ")
        .forEach(System.out::print);               [5]
    System.out.println();
}
```

```
[0] Prego no Prato
[1] Prego no Prato
[2] Prego no Prato
[3] PregonoPrato
[4] Prego no Prato
[5] Prego no Prato
```

Some examples with an array of int

```
public static void arraysExample() {
    int[] numbers = {3, -4, 8, 73, 507, 8, 14, 9, 3, 15, -7, 9, 3, -7, 15};

    // want to know the sum of the numbers?  It's now built in
    int sum = Arrays.stream(numbers)
        .sum();
    System.out.println("sum = " + sum);

    // how about the sum of the evens?
    int sum2 = Arrays.stream(numbers)
        .filter(i -> i % 2 == 0)
        .sum();
    System.out.println("sum of evens = " + sum2);

    // how about the sum of the absolute value of the evens?
    int sum3 = Arrays.stream(numbers)
        .map(Math::abs)
        .filter(i -> i % 2 == 0)
        .sum();
    System.out.println("sum of absolute value of evens = " + sum3);

    // how about the same thing with no duplicates?
    int sum4 = Arrays.stream(numbers)
        .distinct()
        .map(Math::abs)
        .filter(i -> i % 2 == 0)
        .sum();
    System.out.println("sum of absolute value of distinct evens = " + sum4);
}
```

é um reducer específico!

remove os duplicados

A ordem importa !!!

```
sum = 649
sum of evens = 26
sum of absolute value of evens = 34
sum of absolute value of distinct evens = 26
```

Sumário

- JAVA Stream API
- java.util.stream
  - Interfaces

    `BaseStream`
    `Collector`
    `DoubleStream`
    `DoubleStream.Builder`
    `IntStream`
    `IntStream.Builder`
    `LongStream`
    `LongStream.Builder`
    `Stream`
    `Stream.Builder`
  - Classes

    `Collectors`
    `StreamSupport`