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*

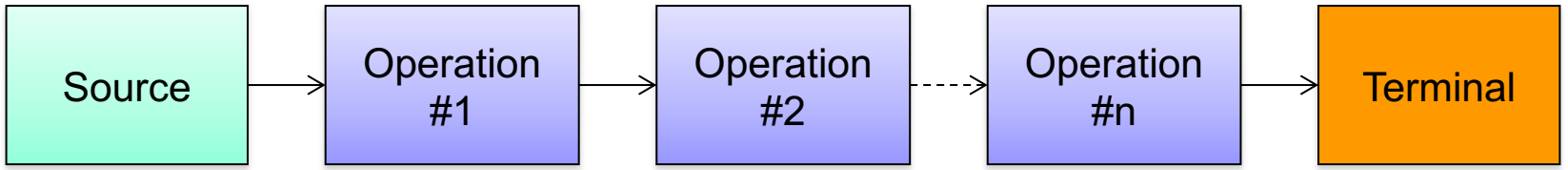
Aggregate Operations – Streams API

- 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
- (3) Perform one terminal operation



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

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

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, ..)*

java.util.stream – Intermediate operations

- **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
- **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);
```

java.util.stream – Terminating operations

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

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

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));


// other example with Map && Consumer

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

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);
```

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());

// 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) + " ");
    System.out.println();

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

    // 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);
    System.out.println();
}
```

```
Prego no Prato
Prego no Prato
Prego no Prato
PregonoPrato
Prego no Prato
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);
}
```

```
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`