

# Java Avançado

## Manipulando Coleções com a Stream API



---

---

---

---


---

---

---

---

Tópicos Abordados



- O que é a Stream API
- Exemplo de uso na prática
- O tipo *Stream<T>*
- Operações
  - **sorted()**
  - **limit()**
  - **filter()**
  - **distinct()**
  - **map()**
  - **collect()**
- Referenciando construtores
- Streams paralelas

---

---

---

---


---

---

---

---

Stream API



- API que permite combinar operações
  - Usada principalmente em coleções do Java
- Introduzida no Java 8
- Aproveita o uso de expressões lambda
- Funciona bem para coleções pequenas e também para coleções muito grandes
  - Usa a abordagem de *lazy evaluation*

---

---

---

---

---

---

---

---

## Exemplo de uso na prática

**ArrayList**

[1, 10, 8, 9, 6, 2]

➡

**Problema**

Lista em ordem crescente dos números entre 2 e 8 com cada elemento elevado ao quadrado

↓

[1, 10, 8, 9, 6, 2] → sorted() → [1, 2, 6, 8, 9, 10] → filter() → [2, 6, 8] → map() → [4, 36, 64]

- A Stream API permite encadear as operações (pipeline)

```
List<Integer> newList = list.stream()
    .sorted()
    .filter(e -> e >= 2 && e <= 8)
    .map(e -> e * e)
    .collect(Collectors.toList());
```

---

---

---

---

---

---

---

---

## Obtendo um objeto *Stream<T>*

- As coleções *List* e *Set* possuem o método *stream()*

```
Stream<Integer> stream = list.stream();
```

- O objeto *Stream<T>* é a porta de entrada para a Stream API
- O tipo parametrizado *T* depende do tipo da coleção
- Para arrays, o código muda

```
int[] array = new int[10];
Stream<int> stream = Stream.of(array);
```

---

---

---

---

---

---

---

---

## Operações da Stream API

- A Stream API possui uma série de operações para manipulação de dados
- 2 tipos
  - Intermediárias
    - Retornam um novo objeto *Stream<T>*
    - Possibilitam o pipeline
    - Ex: *sorted()*, *limit()*, *filter()*, *map()*
  - Terminais
    - Geram um resultado final (redução)
    - Finalizam o uso da stream
    - Ex: *collect()*, *reduce()*, *count()*, *max()*

---

---

---

---


---

---

---

---

Operação: *sorted()*



- Ordena os elementos da coleção
- Permite fornecer ou não um *Comparator<T>*
- Operação intermediária

```
stream.sorted((e1, e2) -> e1.getNome().compareTo(e2.getNome()))
```

Expressão lambda que substitui uma instância de *Comparator<T>*

---

---

---


---

---

---

---

Operação: *limit()*



- Define um tamanho máximo para a coleção
- Os elementos que excedem o limite fornecido são removidos
- Operação intermediária

```
stream.limit(10);
```

Só os 10 primeiros elementos são mantidos

---

---

---


---

---

---

---

Operação: *filter()*



- Filtra os resultados de acordo com um critério
- Recebe um parâmetro *Predicate<T>*
- Operação intermediária

```
stream.filter(e -> e > 10);
```

Expressão lambda que determina a filtragem de elementos maiores que 10

---

---

---

---


---

---

---

3

Operação: *distinct()*



- Remove elementos duplicados
- Operação intermediária

```
stream.distinct();
```

---

---

---

---


---

---

---

---

Operação: *map()*



- Mapeia um elemento em outro elemento (transformação)
- Recebe um parâmetro *Function<T, R>*
- Operação intermediária

```
stream.map(e -> e + 2);
```

Expressão lambda que incrementa 2 unidades a cada elemento

- Existem também mapeamentos especializados
  - mapToInt()* → *IntStream*
  - mapToDouble()* → *DoubleStream*
  - mapToLong()* → *LongStream*

---

---

---

---


---

---

---

---

Operação: *collect()*



- Finaliza o pipeline, gerando um resultado
- Operação terminal
- A classe *Collectors* tem métodos estáticos que auxiliam nesta operação
- Exemplos:

<code>stream.collect(Collectors.toList())</code>	➡	<code>List&lt;T&gt;</code>
<code>stream.collect(Collectors.toSet())</code>	➡	<code>Set&lt;T&gt;</code>
<code>stream.collect(Collectors.counting())</code>	➡	<code>Long</code>
<code>stream.collect(Collectors.summingInt(e -&gt; e.getIdade()))</code>	➡	<code>Integer</code>

---

---

---

---


---

---

---

---

Operação: `count()`



- Faz a contagem de elementos
- Operação terminal

```
long c = stream.count();
```

---

---

---


---

---

---

---

Referenciando construtores



- Além de referenciar métodos, é possível também referenciar construtores

```

public class Pessoa {
    private String nome;

    public Pessoa(String nome) {
        this.nome = nome;
    }
}

List<String> nomes = Arrays.asList(
    "José", "Maria", "Pedro");

List<Pessoa> pessoas = nomes.stream()
    .map(Pessoa::new)
    .collect(Collectors.toList());

```

"José"

"Maria"

"Pedro"

→

{ Pessoa: nome = "José" }

{ Pessoa: nome = "Maria" }

{ Pessoa: nome = "Pedro" }

Equivalente a:

`.map(e -> new Pessoa(e))`

---

---

---


---

---

---

---

Streams Paralelas



- Uma stream paralela pode ser processada simultaneamente por vários núcleos de processamento

```
List<Integer> list = list.parallelStream()
    .sorted()
    .map(e -> e * e)
    .collect(Collectors.toList());
```

Stream paralela

- Para que o resultado seja consistente, as operações intermediárias precisam ser independentes

---

---

---

---

---

---

---



---

---

---

---

---

---

---