



# Módulo 22





# BackEnd Java

---

Rodrigo Pires



A decorative pattern of hexagons in various shades of blue and cyan on the left side of the slide. Some hexagons contain icons: a lightbulb, a thumbs up, a smartphone, a magnifying glass, and a gear. A network diagram with a central node and five peripheral nodes is also visible.

1

# O que são Streams?



# O que são?

É um recurso que ajuda a manipular coleções de uma maneira simples e eficiente seguindo os princípios da programação funcional. Isso é interessante pois o controle de fluxo e loop ficam por conta da API onde temos que nos preocupar somente com a regra do negócio.






# Vantagens

A [proposta em torno da Streams API](#) é reduzir a preocupação do desenvolvedor com a forma de implementar controle de fluxo ao lidar com coleções, deixando isso a cargo da API. A ideia é iterar sobre essas coleções de objetos e, a cada elemento, realizar alguma ação, seja ela de filtragem, mapeamento, transformação, etc. Caberá ao desenvolvedor apenas definir qual ação será realizada sobre o objeto.

Todas as classes novas da API stream ficam no pacote:  
`java.util.stream`






# Como criar Streams

O primeiro passo para se trabalhar com streams é saber como criá-las. A forma mais comum é através de uma coleção de dados.

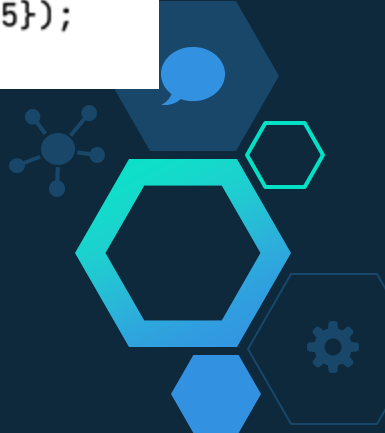
```
public static void main(String[] args) {  
    List<String> lista = List.of("Rodrigo", "Pires");  
    Stream<String> streamList = lista.stream();  
  
    Set<String> set = Set.of("Rodrigo", "Pires");  
    Stream<String> streamSet = set.stream();  
  
    Map<String, String> map = Map.of( k1: "Rodrigo", v1: "Pires");  
    Stream<String> streamMapValues = map.values().stream();  
    Stream<String> streamMapKeys = map.keySet().stream();  
}
```





# Como criar Streams

```
Stream numbersFromValues = Stream.of(1, 2, 3, 4, 5);  
IntStream numbersFromArray = Arrays.stream(new int[] {1, 2, 3, 4, 5});
```



A decorative graphic on the left side of the slide. It features a large central hexagon with a white number '2'. Surrounding this central hexagon are several smaller hexagons of varying shades of blue and cyan. Some of these smaller hexagons contain white icons: a lightbulb, a thumbs-up, a smartphone, a magnifying glass, and a gear. There is also a network-like icon with a central node and several connecting lines.

2

## Operações Intermediários

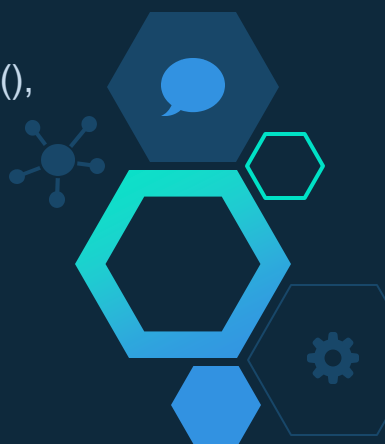




# Operações Intermediárias

Após conhecer alguns dos diferentes modos para criar e obter streams, o foco agora será em como processá-las. Para isso, será mostrado nos próximos tópicos a transformação e o processamento de streams fazendo uso de diferentes operações da interface Stream.


Algumas das operações intermediárias mais utilizadas são: `filter()`, `map()`, `sorted()`, `limit()` e `distinct()`.





# Criar classe Pessoa

```
public class Pessoa {  
  
    String id;  
    String nome;  
    String nacionalidade;  
    int idade;  
  
    public Pessoa(){}  
  
    public Pessoa (String id, String nome, String nacionalidade, int idade){  
        this.id = id;  
        this.nome = nome;  
        this.nacionalidade = nacionalidade;  
        this.idade = idade;  
    }  
  
    public List populaPessoas(){  
        Pessoa pessoa1 = new Pessoa( id: "p1" , nome: "Matheus Henrique", nacionalidade: "Brasil", idade: 18);  
        Pessoa pessoa2 = new Pessoa( id: "p2" , nome: "Hernandez Roja", nacionalidade: "Mexico", idade: 21);  
        Pessoa pessoa3 = new Pessoa( id: "p3" , nome: "Mario Fernandes", nacionalidade: "Canada", idade: 22);  
        Pessoa pessoa4 = new Pessoa( id: "p4" , nome: "Neymar Junior", nacionalidade: "Brasil", idade: 22);  
        return List.of(pessoa1,pessoa2,pessoa3,pessoa4);  
    }  
}
```






# Filter

O método `filter()` é usado para filtrar elementos de uma stream de acordo com uma condição (predicado). Para isso, ele recebe como parâmetro um objeto que implementa a interface `Predicate<T>` e retorna uma nova stream contendo apenas os elementos que satisfazem à condição.


Primeiramente é criada uma lista com alguns objetos do tipo `Pessoa`. Em seguida, com a chamada ao método `stream()` é criada a stream. Logo após, o método `filter()` recebe como parâmetro uma condição, representada por uma expressão lambda, que tem por objetivo buscar todas as pessoas que nasceram no Brasil.





# Filter


```
private static void filter() {  
    List<Pessoa> pessoas = new Pessoa().populaPessoas();  
  
    Stream stream = pessoas.stream().filter(pessoa -> pessoa.getNacionalidade().equals("Brasil"));  
  
    Predicate<Pessoa> predi = pessoa -> pessoa.getNacionalidade().equals("Brasil");  
    Stream stream1 = pessoas.stream().filter(predi);  
  
    Predicate<Pessoa> predi1 = new Predicate<Pessoa>() {  
        @Override  
        public boolean test(Pessoa pessoa) {  
            return pessoa.getNacionalidade().equals("Brasil");  
        }  
    };  
    Stream stream2 = pessoas.stream().filter(predi1);  
}
```





# Map


Algumas situações se faz necessário realizar transformações em uma lista de dados. O método `map()` permite realizar essas mudanças sem a necessidade de variáveis intermediárias, apenas utilizando como argumento uma função do tipo `java.util.function.Function`, que, assim como `Predicate<T>`, também é uma interface funcional. Essa função toma cada elemento de uma stream como parâmetro e retorna o elemento processado como resposta. O resultado será uma nova stream contendo os elementos mapeados a partir da stream original.





# Map

```
private static void map() {  
    List<Pessoa> pessoas = new Pessoa().populaPessoas();  
  
    Stream<Integer> stream = pessoas.stream()  
        .filter(pessoa -> pessoa.getNacionalidade().equals("Brasil"))  
        .map(Pessoa::getIdade);  
  
    IntStream streamInt = pessoas.stream()  
        .filter(pessoa -> pessoa.getNacionalidade().equals("Brasil"))  
        .mapToInt(Pessoa::getIdade);  
}
```






# Map

Nesse trecho de código pode-se ter um problema com a utilização do método `map()`, haja vista que seu retorno é do tipo `Stream<Integer>`. Esse fato gera o boxing dos valores inteiros, isto é, a necessidade de converter o tipo primitivo retornado pelo método `getIddade()` em seu correspondente objeto wrapper.

Pensando nisso, a Streams API oferece implementações para os principais tipos primitivos:

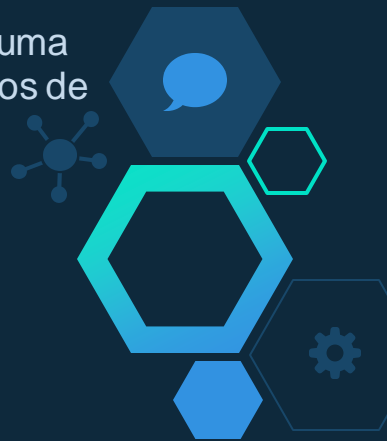
`IntStream`, `DoubleStream` e `LongStream`. Neste exemplo, portanto, pode-se usar o `IntStream` para evitar o autoboxing e chamar o método `mapToInt()` ao invés do `map()`.





# Sorted

A ordenação de elementos em coleções é uma tarefa recorrente no dia a dia de todo desenvolvedor. No Java 8, felizmente, isso foi bastante facilitado, eliminando a necessidade de implementar o verboso `Comparator`, assim como as classes internas anônimas, proporcionando ao código clareza e simplicidade. Para isso, a Streams API oferece a operação `sorted()`. Esse método retorna uma nova stream contendo os elementos da stream original ordenados de acordo com algum critério.








# Sorted

```
private static void sorted() {  
    System.out.println("sorted");  
    List<Pessoa> pessoas = new Pessoa().populaPessoas();  
    Stream stream = pessoas.stream().  
        filter(pessoa -> pessoa.getNacionalidade().equals("Brasil"))  
        .sorted(Comparator.comparing(Pessoa::getNome));  
  
    Stream stream1 = pessoas.stream().  
        filter(pessoa -> pessoa.getNacionalidade().equals("Brasil"))  
        .sorted(Comparator.comparing(Pessoa::getIdade));  
  
    stream1.forEach(i->System.out.println(i));  
  
    Stream stream2 = pessoas.stream().  
        filter(pessoa -> pessoa.getNacionalidade().equals("Brasil"))  
        .sorted((p1, p2) -> Integer.compare(p2.getIdade(), p1.getIdade()));  
  
    stream2.forEach(i->System.out.println(i));  
}
```





# Distinct

A operação `distinct()` retorna uma stream contendo apenas elementos que são exclusivos, isto é, que não se repetem, de acordo com a implementação do método `equals()`.





# Distinct

```
private static void distinct() {  
    System.out.println("distinct");  
    List<Pessoa> pessoas = new Pessoa().populaPessoas();  
    Stream<Pessoa> stream = pessoas.stream().distinct();  
}
```





# Limit


O método `limit()` é utilizado para limitar o número de elementos em uma stream. É uma operação conhecida como curto-circuito devido ao fato de não precisar processar todos os elementos. Como exemplo, o código a seguir demonstra como retornar uma stream com apenas os dois primeiros elementos:





# Limit

```
private static void limit() {  
    System.out.println("limit");  
    List<Pessoa> pessoas = new Pessoa().populaPessoas();  
    Stream<Pessoa> stream = pessoas.stream().limit(2);  
}
```



A decorative graphic on the left side of the slide. It features a large cyan hexagon with a white number '3' inside. Surrounding this central hexagon are several smaller hexagons of varying shades of blue and cyan. Some of these smaller hexagons contain white icons: a lightbulb, a thumbs-up, a smartphone, a magnifying glass, and a gear. There is also a network-like icon with a central node and several smaller nodes connected by lines.

3

# Operações Terminais



# Operações Terminais

Esse tipo de operação pode ser identificada pelo tipo de retorno do método, uma vez que uma operação terminal nunca retorna uma interface Stream, mas sim um resultado (List, String, Long, Integer, etc.) ou void.

Algumas das operações terminais mais utilizadas são: ForEach, Collect, Count, anyMatc e allMatch.





# ForEach

Através do método `forEach()` é possível realizar um loop sobre todos os elementos de uma stream e executar algum tipo de processamento. No exemplo a seguir, o parâmetro que o método `forEach()` recebe uma expressão lambda que invoca o método `getNome()` do objeto `pessoa` e imprime o seu retorno no console. Assim, serão exibidos os nomes de todas as pessoas presentes na coleção.








# ForEach

```
private static void forEach() {  
    System.out.println("*** forEach");  
    List<Pessoa> pessoas = new Pessoa().populaPessoas();  
    pessoas.stream().forEach(pessoa -> System.out.println(pessoa.getNome()));  
    pessoas.forEach(pessoa -> System.out.println(pessoa.getNome()));  
}
```





# Count

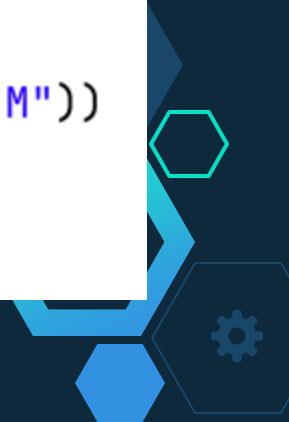
O método `count()` retorna a quantidade de elementos presentes em uma stream. Ele é classificada como uma operação de redução (reduction). Como exemplo, o trecho de código a seguir mostra como obter o número de pessoas em uma lista cujo nome começa com a letra "M":





# Count

```
private static void count() {  
    System.out.println("*** count");  
    List<Pessoa> pessoas = new Pessoa().populaPessoas();  
    long count = pessoas.stream()  
        .filter(pessoa -> pessoa.getNome().startsWith("M"))  
        .count();  
}
```



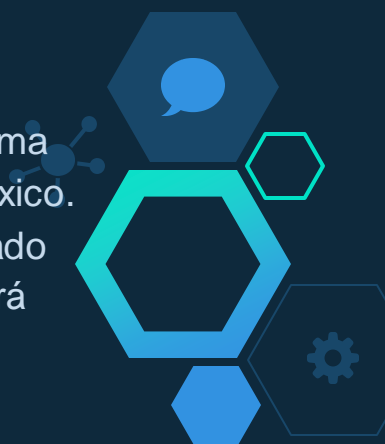


# AllMatch

Um padrão de processamento comum em aplicações consiste em verificar se os elementos de uma coleção correspondem a um determinado predicado, isto é, a uma característica ou propriedade do objeto.

O método `allMatch()` verifica se todos os elementos de uma stream atendem a um critério passado como parâmetro, através de um `Predicate`, e retorna um valor booleano.


No exemplo a seguir, cada elemento da stream é submetido a uma condição, que nesse caso é verificar se a pessoa nasceu no México. Se todos os elementos obedecem a essa condição, será retornado `true`. Caso algum dos elementos não satisfaça ao predicado, será retornado `false`.





# AllMatch

```
private static void allMatch() {  
    System.out.println("*** allMatch|");  
    List<Pessoa> pessoas = new Pessoa().populaPessoas();  
    boolean todosMexicanos = pessoas.stream()  
        .allMatch(pessoa -> pessoa.getNacionalidade().equals("Mexico"));  
}
```





# Collect


O método `collect()` possibilita coletar os elementos de uma stream na forma de coleções, convertendo uma stream para os tipos List, Set ou Map.





# Collect

```
private static void collect() {  
    System.out.println("*** collect");  
    List<Pessoa> pessoas = new Pessoa().populaPessoas();  
    List<Pessoa> pessoasComM = pessoas.stream()  
        .filter(pessoa -> pessoa.getNome().startsWith("M"))  
        .collect(Collectors.toList());  
    pessoasComM.forEach(pessoa -> System.out.println(pessoa));  
}
```






# Collect

```
Set<Pessoa> treeset = pessoas.stream()  
    .filter(pessoa -> pessoa.getNome().startsWith("M"))  
    .collect(Collectors.toSet());
```

```
ArrayList<Pessoa> list = pessoas.stream()  
    .filter(pessoa -> pessoa.getNome().startsWith("M"))  
    .collect(Collectors.toCollection(ArrayList::new));
```

```
TreeSet<Pessoa> treeset1 = pessoas.stream()  
    .filter(pessoa -> pessoa.getNome().startsWith("M"))  
    .collect(Collectors.toCollection(TreeSet::new));
```








# Collect

```
Map<Integer, Pessoa> map = pessoas.stream()  
    .collect(Collectors.toMap(Pessoa::getIdade, Pessoa::new));
```

```
Map<Integer, List<Pessoa>> grupoPorIdade = pessoas.stream()  
    .collect(Collectors.groupingBy(Pessoa::getIdade));
```

```
Map<String, List<Pessoa>> grupoPorNacionalidade = pessoas.stream()  
    .collect(Collectors.groupingBy(Pessoa::getNacionalidade));
```

```
Map<String, Integer> grupoPorNacionalidadeSomadosIdades = pessoas.stream()  
    .collect(Collectors.groupingBy(Pessoa::getNacionalidade,  
        Collectors.summingInt(Pessoa::getIdade)));
```



A decorative graphic on the left side of the slide. It features a large central hexagon with a blue-to-teal gradient containing the number '4'. Surrounding this central hexagon are several smaller hexagons of varying shades of blue and teal. Some of these smaller hexagons contain white icons: a lightbulb, a thumbs-up, a smartphone, a magnifying glass, and a gear. There is also a network-like icon with a central node and radiating lines, and a speech bubble icon.

4

# Valores Opcionais



# Opcional / Optional


Optionals surgiram para evitar `nullPointerExceptions` e antes de tentar obter algo, podemos validar se realmente existe.





# Opcional / Optional

```
private static void optioanl() {  
    System.out.println("*** optioanl");  
    List<Pessoa> pessoas = new Pessoa().populaPessoas();  
  
    Optional<Pessoa> max = pessoas.stream()  
        .max(Comparator.comparing(Pessoa::getIdade));  
    if (max.isPresent()) {  
        System.out.println(max.get());  
    }  
  
    max.ifPresent(System.out::println);  
}
```






# Opcional / Optional

```
Optional<Pessoa> min = pessoas.stream()
    .min(Comparator.comparing(Pessoa::getIdade));

min.ifPresentOrElse(System.out::println, new Runnable() {
    @Override
    public void run() {
        //Buscar outra vez em algum lugar
    }
});

Pessoa value = min.orElseThrow();
```





# Referências

[Exemplos disponíveis no meu github:](#)

<https://github.com/digaomilleniun/backend-java-ebac>

