



# Stream API

- ✓ Manipulação de coleções com o paradigma funcional de forma paralela
- ✓ Imutável - Não altera a coleção origem, sempre cria uma nova coleção
- ✓ Principais funcionalidades
  - ✓ Mapping – Retorna uma coleção com mesmo tamanho da origem com os elementos alterados
  - ✓ Filtering – Retorna uma coleção igual ou menor que a coleção origem, com os elementos intactos
  - ✓ ForEach – Executa uma determinada lógica para cada elemento, retornando nada.
  - ✓ Peek - Executa uma determinada lógica para cada elemento, retornando a própria coleção.
  - ✓ Counting – Retorna um inteiro que representa a contagem de elementos.
  - ✓ Grouping – Retorna uma coleção agrupada de acordo com a regra definida.



# Exercício final

1 – Utilizando uma lista com um objeto complexo (Estudante, por exemplo) realize as seguintes operações:

- A) Transforme cada estudante em uma string com os atributos do objeto
- B) Conte a quantidade de estudantes tem na coleção
- C) Filtre estudantes com idade igual ou superior a 18 anos
- D) Exiba cada elemento no console.
- E) Retorne estudantes com nome que possui a letra B
- F) Retorne se existe ao menos um estudante com a letra D no nome
- G) Retorne o estudante mais velho (maior idade) da coleção. Retorne o mais novo também.



```
> Task :ExemploStreamAPI.main()
MÉTODO - estudantes.stream().count()
AÇÃO - Contagem: 7
```

```
> Task :ExemploStreamAPI.main()
MÉTODO - estudantes.stream().max(Comparator.comparingInt(String::length))
AÇÃO - Maior numero de letras: Optional[Marcelo]
```

```
> Task :ExemploStreamAPI.main()
MÉTODO - + estudantes.stream().min(Comparator.comparingInt(String::length))
AÇÃO - Menor numero de letras: Optional[Pedro]
```

```
> Task :ExemploStreamAPI.main()
MÉTODO - estudantes.stream().filter((estudante) -> estudante.toLowerCase().contains(r)).collect(Collectors.toList())
AÇÃO - Com a letra r no nome: [Pedro, Marcelo, Carla, Rafael]
```

```
> Task :ExemploStreamAPI.main()
MÉTODO - estudantes.stream().map(estudante -> estudante.concat(" - ").concat(String.valueOf(estudante.length()))).collect(Collectors.toList())
AÇÃO - Retorna uma nova coleção com a quantidade de letras: [Pedro - 5, Thayse - 6, Marcelo - 7, Carla - 5, Juliana - 7, Thiago - 6, Rafael - 6]
```

```
> Task :ExemploStreamAPI.main()
MÉTODO - estudantes.stream().limit(3).collect(Collectors.toList())
AÇÃO - Retorna os 3 primeiros elementos: [Pedro, Thayse, Marcelo]
```

```
> Task :ExemploStreamAPI.main()
MÉTODO - estudantes.stream().peek(System.out::println).collect(Collectors.toList())
Pedro
Thayse
Marcelo
Carla
Juliana
Thiago
Rafael
AÇÃO - Retorna os elementos: [Pedro, Thayse, Marcelo, Carla, Juliana, Thiago, Rafael]
```

```
> Task :ExemploStreamAPI.main()
MÉTODO - estudantes.stream().forEach(System.out::println)
AÇÃO - Retorna os elementos novamente:
Pedro
Thayse
Marcelo
Carla
Juliana
Thiago
Rafael
```

```
> Task :ExemploStreamAPI.main()
MÉTODO - estudantes.stream().allMatch((elemento) -> elemento.contains("W"))
AÇÃO - Tem algum elemento com W no nome? false
```

```
> Task :ExemploStreamAPI.main()
MÉTODO - estudantes.stream().allMatch((elemento) -> elemento.contains("W"))
AÇÃO - Tem algum elemento com W no nome? false
```

```
> Task :ExemploStreamAPI.main()
MÉTODO - estudantes.stream().anyMatch((elemento) -> elemento.contains("a"))
AÇÃO - Tem algum elemento com a minúscula no nome? true
```

```
> Task :ExemploStreamAPI.main()
MÉTODO - estudantes.stream().noneMatch((elemento) -> elemento.contains("a"))
AÇÃO - Não tem nenhum elemento com a minúscula no nome? false
```

```
> Task :ExemploStreamAPI.main()
MÉTODO - estudantes.stream().findFirst().ifPresent(System.out::println)
AÇÃO - Retorna o primeiro elemento da coleção: Pedro
```



## Exemplo de operação encadeada

```
// Exemplo de operação encadeada
System.out.print("Operação encadeada: ");
System.out.println(estudantes.stream()
    .peek(System.out::println)
    .map(estudante ->
        estudante.concat(" - ").concat(String.valueOf(estudante.length()))
    )
    .peek(System.out::println)
    .filter(estudante ->
        estudante.toLowerCase().contains("r"))
    .collect(Collectors.toList());
    .collect(Collectors.joining(", "));
    .collect(Collectors.toSet());
    .collect(Collectors.groupingBy(estudante -> estudante.substring(estudante.indexOf("-") + 1)))
);
```

```
> Task :ExemploStreamAPI.main()
Operação encadeada: Pedro
Pedro - 5
Thayse
Thayse - 6
Marcelo
Marcelo - 7
Carla
Carla - 5
Juliana
Juliana - 7
Thiago
Thiago - 6
Rafael
Rafael - 6
{ 5=[Pedro - 5, Carla - 5], 6=[Rafael - 6], 7=[Marcelo - 7]}
```

Considere a `java.util.List` de frutas conforme: `["Maçã", "Laranja", "Banana", "Limão"]`

Qual o estado resultante da chamada: `"frutas.stream().max(Comparator.comparingInt(String::length))"`?

Banana.

Maçã.

Limão.

Erro de compilação, método "max" não existe.

Laranja.



PRÓXIMA PERGUNTA



Considere a `java.util.List` de frutas conforme: `["Maçã", "Laranja", "Banana", "Limão"]`

Qual o estado resultante da chamada: `"frutas.stream().count()"`?

Um número de tipo `long` com valor 5.

Um número de tipo `int` com valor 4.



Um número de tipo `long` com valor 4.



Um número de tipo `double` com valor 4.

Um número de tipo `float` com valor 4.

Considere a `java.util.List` de frutas conforme: ["Maçã", "Laranja", "Banana", "Limão"]

Qual o estado resultante da chamada: `"frutas.stream().min(Comparator.comparingInt(String::length))"`

Limão.

Erro de compilação, método "min" não existe.

Banana.

Laranja.

Maçã.



Considere a `java.util.List` de frutas conforme: `["Maçã", "Laranja", "Banana", "Limão"]`

Qual o estado resultante da chamada: `"frutas.stream().filter(fruta -> fruta.length() > 5 ).collect(Collectors.toList())"`?

Erro de compilação, método "filter" não existe.

`["Laranja", "Banana", "Limão"]`

`["Laranja", "Banana"]`



`["Maçã", "Laranja", "Banana", "Limão"]`

Erro de execução.

PRÓXIMA PERGUNTA

Considere a `java.util.List` de frutas conforme: `["Maçã", "Laranja", "Banana", "Limão"]`

Qual o estado resultante da chamada: `"frutas.stream().min(Comparator.comparingInt(String::length))"`

Erro de compilação, método "min" não existe.

Maçã.



Banana.

Laranja.

Limão.

PRÓXIMA PERGUNTA

Considere a `java.util.List` de frutas conforme: `["Maçã", "Laranja", "Banana", "Limão"]`

Qual o estado resultante da chamada: `"frutas.stream().map(fruta -> fruta + " - " + (fruta.length() * 2)).collect(Collectors.toList())"`?

`["Maçã - 8", "Laranja - 14", "Banana - 6", "Limão - 10"]`



Erro de execução.

`["Maçã - 0", "Laranja - 0", "Banana - 0", "Limão - 0"]`

Erro de compilação, método "map" não existe.

`["Maçã - 4", "Laranja - 7", "Banana - 6", "Limão - 5"]`

PRÓXIMA PERGUNTA

Considere a `java.util.List` de frutas conforme: `["Maçã", "Laranja", "Banana", "Limão"]`

Qual o estado resultante da chamada: `frutas.stream().collect(Collectors.groupingBy(fruta -> fruta.substring(0, 1)))`

Erro de compilação.

`{M=["Maçã"], L=["Laranja", "Limão"], B=["Banana"]}`



`["Maçã", "Laranja", "Banana", "Limão"]`

`[]`

Erro de execução.

PRÓXIMA PERGUNTA

**Assinale a alternativa correta:**

O método "peek" retorna void.

O método "forEach" retorna um único elemento.

O método "peek" retorna o mesmo Stream de entrada.



O método "forEach" retorna um objeto de mesmo tipo do objeto de entrada.

O método "peek" lança NoSuchElementException se a entrada for nula.

PRÓXIMA PERGUNTA



Considere a `java.util.List` de frutas conforme: ["Maçã", "Laranja", "Banana", "Limão"]

Qual o estado resultante da chamada: `frutas.stream().noneMatch(fruta -> fruta.length() >= 10)`?

false

Erro de execução.

Erro de compilação.

Um novo Stream com os elementos que atendem a condição parametrizada.

true



PRÓXIMA PERGUNTA

Considere a `java.util.List` de frutas conforme: ["Maçã", "Laranja", "Banana", "Limão"]

Qual o estado resultante da chamada: `frutas.stream().anyMatch(fruta -> fruta.contains("X"))`?

false



Erro de execução, nenhuma fruta tem a letra "X".

Erro de compilação, nenhuma fruta tem a letra "X".

true

Erro de compilação, método "anyMatch" não existe.

PRÓXIMA PERGUNTA

**Assinale a alternativa correta:**

A Stream API é performática pois usa algoritmos de inteligência artificial

A Stream API não é performática

A Stream API não deve ser utilizada para grandes coleções.

A Stream API é performática pois executa as operações de forma paralelizada



A Stream API é performática pois executa as operações de forma síncrona

FINALIZAR