

# Visão Geral de *Streams*



Prof. Jeferson Souza, MSc. (thejefecomp)

[jeferson.souza@udesc.br](mailto:jeferson.souza@udesc.br)



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

JOINVILLE  
CENTRO DE CIÊNCIAS  
TECNOLÓGICAS

# Introdução às Streams

## Afinal, o que são as famosas Streams [Boyarsky&Selikoff, 2016] [Oracle, 2021]?

*Streams* são sequências de elementos/dados nas quais operações podem ser aplicadas de forma sequencial ou paralela, a terminar na produção de um resultado esperado.

## Principais características das Streams...

- ◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡

# Introdução às Streams

## Definição de Stream Pipeline [Oracle, 2021]

Um **Stream Pipeline** consiste em uma fonte de dados (que teoricamente pode ser infinita), um conjunto finito de operações intermediárias, a permitir a transformação de uma *Stream* em outra, e uma operação terminal para produzir o resultado esperado.

## Pense em um Stream Pipeline Como...



Uma linha de produção em um chão de fábrica, onde cada operação representa um estágio da referida linha, da entrada até a produção do resultado final.

## 5 / 44



# Aplicação de Programação Funcional em uma Linguagem Orientada a Objetos

## Expressões Lambda em Java [Boyarsky&Selikoff, 2016]...

As expressões *Lambda* são definidas em Java como blocos de código-fonte que permitem a especificação de um conjunto de instruções de forma declarativa, a implicar em uma maior expressividade aquando combinadas com o poder da orientação a objetos presente na linguagem Java. *Interfaces* funcionais, as quais contém somente um único método abstrato, são a base para a definição e o uso das expressões *Lambda*. A interface `java.util.function.Predicate` é um exemplo de interface funcional.



## 8 / 44

# Aplicação de Programação Funcional em uma Linguagem Orientada a Objetos

## Exemplos de Expressões Lambda em Java...

1. `fruta`  $\rightarrow$  `System.out.println(fruta)`
2. `(Fruta fruta)`  $\rightarrow$  `{ System.out.println(fruta); }`
3. `fruta`  $\rightarrow$  `fruta.ehComestivel()`
4. `(Fruta fruta)`  $\rightarrow$  `{ return fruta.ehComestivel(); }`
5. `()`  $\rightarrow$  `new Fruta()`
6. `()`  $\rightarrow$  `{ return new Fruta(); }`
7. `(fruta, equipamento)`  $\rightarrow$  `equipamento.processar(fruta)`



## 11 / 44

## A Interface Funcional BiPredicate

## A interface funcional BiPredicate [Oracle, 2021]

A interface funcional *BiPredicate* representa um predicado de dois argumentos a ser especificado por um método de retorno booleano chamado *test()*, e a permitir verificar a validade do predicado quando aplicado a dois argumentos informados.

## @FunctionalInterface

```
public interface BiPredicate<T, U>{
```

... A omitir todos os métodos **default** e estáticos ...

```
boolean test(T t, U u);
```

}



## A Interface Funcional Function

## A interface funcional Function [Oracle, 2021]

A interface funcional *Function* representa um operação de conversão a ser especificada por um método chamado *apply()*, o qual recebe um argumento de um tipo parametrizado T, e retorna um valor convertido para o tipo parametrizado R.

## @FunctionalInterface

```
public interface Function<T, R>{
```

... A omitir todos os métodos **default** e estáticos ...

```
R apply(T t);
```

}







# A Interface Funcional BiFunction

## Exemplo de uso da interface funcional BiFunction [Boyarsky&Selikoff, 2016]

```
BiFunction<String, String, String> fConcat = (stringA, stringB)  
→ stringA.concat(stringB);
```

```
System.out.println(fConcat.apply("baby ", "chick")); // Imprime a  
string "baby chick" (sem as aspas duplas).
```

# A Interface Funcional UnaryOperator

## A interface funcional UnaryOperator [Oracle, 2021]

A interface funcional *UnaryOperator* é uma extensão da interface *Function*, a permitir somente um único tipo parametrizado *T* a ser utilizado como parâmetro e retorno do método *apply()*.

### @FunctionalInterface

```
public interface UnaryOperator<T> extends Function<T, T>{  
    ... A omitir todos os métodos default e estáticos ...  
}
```

Note que o método *apply()* não é declarado já que já está declarado na interface funcional *Function*. Entretanto, na ótica da interface *UnaryOperator*, o método é redefinido da seguinte forma:

```
T apply(T t);
```

## A Interface Funcional UnaryOperator

## Exemplo de uso da interface funcional UnaryOperator [Boyarsky&Selikoff, 2016]

```
UnaryOperator<String> fMaiuscula = string →
string.toUpperCase();
```

**System.out.println(fMaiuscula.apply(“chirp”));** // Imprime a string “CHIRP” (sem as aspas duplas).

# A Interface Funcional BinaryOperator

## A interface funcional BinaryOperator [Oracle, 2021]

A interface funcional *BinaryOperator* é uma extensão da interface *BiFunction*, a permitir somente um único tipo parametrizado *T* a ser utilizado como parâmetro e retorno do método *apply()*.

### @FunctionalInterface

```
public interface BinaryOperator<T> extends BiFunction<T, T,  
T>{  
    ... A omitir todos os métodos default e estáticos ...  
}
```

Note que o método *apply()* não é declarado já que já está declarado na interface funcional *BiFunction*. Entretanto, na ótica da interface *BinaryOperator*, o método é redefinido da seguinte forma:

```
T apply(T t1, T t2);
```

## A Interface Funcional BinaryOperator

## Exemplo de uso da interface funcional BinaryOperator [Boyarsky&Selikoff, 2016]

```
BinaryOperator<String> fConcat = (stringA, stringB) →
stringA.concat(stringB);
```

```
System.out.println(fConcat.apply("baby ", "chick")); // Imprime a
string "baby chick" (sem as aspas duplas).
```



# A Interface Funcional Supplier

## Exemplo de uso da interface funcional Supplier [Boyarsky&Selikoff, 2016]

```
Supplier<StringBuilder> sb = StringBuilder::new;  
Supplier<StringBuilder> sb2 = () → new StringBuilder();  
System.out.println(sb.get());  
System.out.println(sb2.get());
```

Ambos os *suppliers* são equivalentes, a criar um novo objeto do tipo `StringBuilder`. Entretanto, o primeiro *supplier* utiliza uma construção denominada de referência de método, onde essa referência é utilizada diretamente na definição da expressão de retorno. Nota-se o uso de dois “dois pontos” consecutivos (`::`) como forma de realizar tal referência. Pode ser utilizado também com outras interfaces funcionais já vistas anteriormente.



# A Interface Funcional Consumer

## A interface funcional Consumer [Oracle, 2021]

A interface funcional *Consumer* permite realizar uma operação sobre um parâmetro de tipo parametrizado *T*, por meio do método *accept()*, sem a necessidade de nenhum retorno (**void**).

@FunctionalInterface

```
public interface Consumer<T>{
```

... A omitir todos os métodos **default** e estáticos ...

```
void accept(T t);
```

```
}
```

# A Interface Funcional Consumer

## Exemplo de uso da interface funcional Consumer [Boyarsky&Selikoff, 2016]

```
Consumer<String> consumidor1 = System.out::println;
```

```
Consumer<String> consumidor2 = string →  
System.out.println(string);
```

```
consumidor1.accept("Annie");// Imprime a string "Annie" (sem  
as aspas duplas).
```

```
consumidor2.accept("Annie");// Imprime a string "Annie" (sem  
as aspas duplas).
```

# A Interface Funcional BiConsumer

## A interface funcional BiConsumer [Oracle, 2021]

A interface funcional *BiConsumer* permite realizar uma operação sobre dois parâmetros de tipos parametrizados *T* e *U*, por meio do método *accept()*, sem a necessidade de nenhum retorno (**void**).

### @FunctionalInterface

```
public interface BiConsumer<T, U>{
```

... A omitir todos os métodos **default** e estáticos ...

```
void accept(T t, U u);
```

```
}
```

# A Interface Funcional BiConsumer

## Exemplo de uso da interface funcional BiConsumer [Boyarsky&Selikoff, 2016]

```
Map<String, Integer> mapa = new HashMap<>();
```

```
BiConsumer<String> insereMapa1 = mapa::put;
```

```
BiConsumer<String> insereMapa2 = (chave,valor) →  
mapa.put(chave,valor);
```

```
insereMapa1.accept("frango", 7); // Insere no mapa a chave  
"frango" associada ao valor 7.
```

```
insereMapa1.accept("pintinho", 1); // Insere no mapa a chave  
"pintinho" associada ao valor 1.
```

```
System.out.println(mapa);
```

# Streams - Gerar fontes de dados

## Gerar fontes de dados [Boyarsky&Selikoff, 2016]...

A interface `java.util.stream.Stream` possui alguns métodos para gerar fontes de dados finitas e infinitas. São eles:

- ▶ **empty()**: cria uma nova *Stream* sem nenhum elemento (vazia);
- ▶ **generate()**: cria uma nova *Stream* infinita, onde cada elemento é gerado por um dado fornecedor ("supplier");
- ▶ **iterate()**: cria uma nova *Stream* infinita, onde cada elemento é dado pelo aplicação consecutiva de um operador unário, a representar uma função de geração  $f$ . Possui também uma sobrecarga de método para criação de *Streams* finitas;
- ▶ **of()**: cria uma nova *Stream* finita com base em elementos fornecidos como parâmetro.

# Streams - Gerar fontes de dados finitas

## Exemplos de geração de fontes de dados finitas [Boyarsky&Selikoff, 2016]

```
Stream<String> streamVazia = Stream.empty();
```

```
Stream<String> streamElementoUnico = Stream.of(1);
```

```
Stream<String> streamMultiplosElementos = Stream.of(1, 2, 3);
```

# Streams - Gerar fontes de dados infinitas

## Exemplos de geração de fontes de dados infinitas [Boyarsky&Selikoff, 2016]

```
Stream<Double> streamRandom =  
Stream.generate(Math::random);
```

```
Stream<String> streamImpares = Stream.iterate(1, n → n + 2);
```

# Streams - Algumas Operações Intermediárias

## Algumas operações intermediárias utilizadas com Streams [Boyarsky&Selikoff, 2016]...

Algumas das operações intermediárias que podem ser utilizadas com *Streams* são:

- ▶ **filter()**: retorna uma nova *Stream* somente com os elementos que satisfazem uma expressão Lambda fornecida. Utiliza a interface funcional *Predicate*;
- ▶ **distinct()**: retorna uma nova *Stream* sem elementos duplicados;
- ▶ **limit()**: retorna uma nova *Stream* com um tamanho limitado  $n$ ;
- ▶ **skip()**: retorna uma nova *Stream* que deixa de fora os  $n$  primeiros elementos;
- ▶ **map()**: retorna uma nova *Stream* com elementos mapeados um a um de um tipo para outro. Utiliza a interface funcional *Function*.



# Streams - A Operação Intermediária filter()

## Exemplo de uso da operação intermediária filter() [Boyarsky&Selikoff, 2016]

```
Stream<String> stream = Stream.of("macaco", "gorila",  
    "chimpanzé");
```

```
stream.filter(animal →  
    animal.startsWith("m")).forEach(System.out::println); // Imprime  
a string "macaco" (sem as aspas duplas).
```

# Streams - A Operação Intermediária `distinct()`

## Exemplo de uso da operação intermediária `distinct()` [Boyarsky&Selikoff, 2016]

```
Stream<String> stream = Stream.of("pato", "pato", "pato",  
    "ganso");
```

```
stream.distinct().forEach(System.out::println); // Imprime as  
strings "pato" e "ganso" apenas uma vez cada uma (sem as aspas  
duplas).
```

# Streams - As Operações Intermediárias `limit()` e `skip()`

**Exemplo de uso das operações intermediárias `limit()` e `skip()` [Boyarsky&Selikoff, 2016]**

```
Stream<Integer> stream = Stream.iterate(1, n → n + 1);  
  
stream.skip(5).limit(2).forEach(System.out::println); // Imprime os  
valores 6 e 7, a desconsiderar os cinco primeiros números, e limitar  
o número máximo de elementos a serem processados para 2.
```

# Streams - A Operação Intermediária `map()`

## Exemplo de uso da operação intermediária `map()` [Boyarsky&Selikoff, 2016]

```
Stream<String> stream = Stream.of("macaco", "gorila",  
    "chimpanzé");
```

```
stream.map(String::length).forEach(System.out::println); //
```

Imprime os valores 6, 5, e 9, os quais representam os tamanhos das strings presentes na stream.

# Streams - Algumas Operações Terminais

## Algumas operações terminais utilizadas com Streams [Boyarsky&Selikoff, 2016]...

Algumas das operações terminais que podem ser utilizadas com *Streams* são:

- ▶ **allMatch()**: indica se todos os elementos da *Stream* satisfazem uma expressão Lambda fornecida. Utiliza a interface funcional *Predicate*;
- ▶ **anyMatch()**: indica se, pelo menos, um dos elementos da *Stream* satisfaz uma expressão Lambda fornecida. Utiliza a interface funcional *Predicate*;
- ▶ **collect()**: permite a criação de diferentes estruturas de dados no momento da coleta dos elementos associados à *Stream* em questão;
- ▶ **count()**: retorna o número de elementos em uma *Stream* finita;

# Streams - Algumas Operações Terminais (Continuação)

## Algumas operações terminais utilizadas com Streams [Boyarsky&Selikoff, 2016] (Continuação)...

- ▶ **findAny():** retorna o primeiro elemento da *Stream* que chega até esta operação terminal. Muito útil quando está-se a trabalhar com *Streams* paralelas;
- ▶ **findFirst():** retorna o primeiro elemento da *Stream* na sequência de entrada no *Pipeline*. Caso os elementos não possuam qualquer tipo de ordem estabelecida, qualquer elemento pode ser retornado como resultado;
- ▶ **forEach():** processa cada elemento da *Stream* que já passou por todas as operações intermediárias. Utiliza a interface funcional *Consumer*;
- ▶ **reduce():** permite combinar todos os elementos de uma dada *Stream*, a ter um único objeto como retorno. Utiliza a interface funcional *BinaryOperator*.

# Streams - As Operações Terminais `allMatch()` e `anyMatch()`

Exemplo de uso das operações terminais `allMatch()` e `anyMatch()` [Boyarsky&Selikoff, 2016]

```
List<String> lista = Arrays.asList("macaco", "2", "chimpanzé");
```

```
Predicate<String> predLetra = caractere →  
Character.isLetter(caractere.charAt(0));
```

```
System.out.println(lista.stream().allMatch(predLetra)); // Imprime  
o valor false (falso).
```

```
System.out.println(lista.stream().anyMatch(predLetra)); // Imprime  
o valor true (verdadeiro).
```

# Streams - A Operação Terminal collect()

## Exemplos de uso da operação terminal collect() [Boyarsky&Selikoff, 2016]

```
List<String> lista = Arrays.asList("leões", "tigres", "ursos");
```

```
Set<Integer> conjunto = Set.of(1,2,3,4,5);
```

```
Set<String> conjuntoString =  
lista.stream().collect(Collectors.toSet()); //Coleta a lista como um  
conjunto.
```

```
List<Integer> listaNumeros =  
conjunto.stream().collect(Collectors.toList()); //Coleta o conjunto como  
uma lista.
```

```
Map<String, Integer> mapaString =  
lista.stream().collect(Collectors.toMap(string → string, String::length));  
//Coleta a lista como uma mapa, onde a chave é a string, e o valor o seu  
tamanho.
```



## Streams - A Operação Terminal count()

## Exemplo de uso da operação terminal

```
Stream<String> streamAnimal = Stream.of("macaco", "gorila",  
"chimpanzé");
```

**System.out.println(streamAnimal.count());**//Imprime o valor 3 (três).

# Streams - A Operação Terminal findAny()

## Exemplo de uso da operação terminal findAny() [Boyarsky&Selikoff, 2016]

```
Stream<String> streamInfinita = Stream.generate(() →  
    "chimpanzé");  
  
streamInfinita.findAny().ifPresent(System.out::println); // Imprime  
a string "chimpanzé".
```

## Streams - A Operação Terminal findFirst()

## Exemplo de uso da operação terminal

```
Stream<String> streamAnimal = Stream.of("macaco", "gorila",  
"chimpanzé");
```

```
streamAnimal.findFirst().ifPresent(System.out::println);//Imprime
a string "macaco".
```

## Streams - A Operação Terminal forEach()

## Exemplo de uso da operação terminal forEach() [Boyarsky&Selikoff, 2016]

```
Stream<String> streamAnimal = Stream.of("macaco", "gorila",  
"chimpanzé");
```

```
streamAnimal.forEach(System.out::println); // Imprime as strings  
presentes na Stream, cada uma em uma linha distinta.
```

## Streams - A Operação Terminal reduce()

## Exemplos de uso da operação terminal reduce() [Boyarsky&Selikoff, 2016]

```
Stream<String> streamLetra = Stream.of("w", "o", "l", "f");
String palavra = streamLetra.reduce("", (c1,c2) → c1 + c2);
System.out.println(palavra); //Imprime a string "wolf".

palavra = streamLetra.reduce("super", (c1,c2) → c1 + c2);
System.out.println(palavra); //Imprime a string "superwolf".

Stream<Integer> streamNumero = Stream.of(1,3,5);
streamNumero.reduce((n1,n2) → n1 +
n2).ifPresent(System.out::println); //Imprime o valor 9 (nove).
```

# Bibliografia



GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G.; BUCKLEY, A.; SMITH, D.; BIERMAN, G. **“The Java Language Specification: Java SE 16 Edition”**. Oracle. 2021. Disponível em: <https://docs.oracle.com/javase/specs/jls/se16/jls16.pdf>. Acesso em: 12 Jul. 2021.



ORACLE, INC. **“Java Platform, Standard Edition & Java Development Kit Version 16 API Specification”**. Oracle. 2021. Disponível em: <https://docs.oracle.com/en/java/javase/16/docs/api/index.html>. Acesso em: 26 Jul. 2021.

# Bibliografia (Continuação)

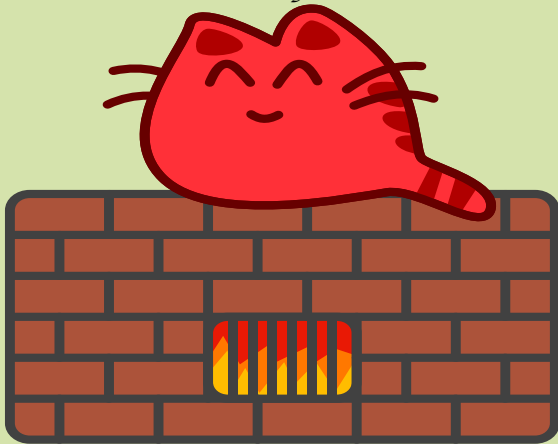


BOYARSKY, J. and SELIKOFF, S. **“Oracle Certified Associate Java SE 8 Programmer I: Study Guide”**. Sybex. Indianapolis, Indiana, USA. 2015.



BOYARSKY, J. and SELIKOFF, S. **“Oracle Certified Associate Java SE 8 Programmer II: Study Guide”**. Sybex. Indianapolis, Indiana, USA. 2016.

*That's it folks!*



*Thank you for your attention!*