

Visão Geral de *Streams*



Prof. Jeferson Souza, MSc. (thejefecomp)

jeferson.souza@udesc.br



UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

JOINVILLE
CENTRO DE CIÊNCIAS
TECNOLÓGICAS

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.

◀ ◻ ▶ ◀ ◼ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ ↺ 🔍 ↻

Introdução às Streams

Exemplo de uso das Streams [Oracle, 2021]

```
int soma = widgets.stream()  
    .filter(w → w.getCor() == VERMELHO)  
    .mapToInt(w → w.getPeso())  
    .sum();
```

No exemplo acima temos especificado o processamento de uma coleção de *widgets*, onde está-se interessado em somar o peso de cada um dos *widgets* a possuir a cor vermelha. Portanto, o *pipeline* para a dada *Stream* é estabelecido pela coleção de entrada *widgets*, as operações intermediárias *filter()* e *mapToInt()*, e a operação terminal *sum()*.

Aplicação de Programação Funcional em uma Linguagem Orientada a Objetos

Exemplos de Expressões Lambda em Java...

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

A interface funcional Predicate [Oracle, 2021]

A interface funcional *Predicate* representa um predicado de um único argumento a ser especificado por um método de retorno booleano chamado *test()*, e a permitir verificar a validade do predicado quando aplicado a um dado argumento.

@FunctionalInterface

```
public interface Predicate<T>{
    ... A omitir todos os métodos default e estáticos ...
    boolean test(T t);
}
```

Exemplo de uso da interface funcional Predicate

◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡

A interface funcional BiPredicate [Oracle, 2021]

@FunctionalInterface

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

[illegible]

Exemplo de uso da interface funcional BiPredicate

```
System.out.println(comecaCom.test("Corsa", "C")); // imprime
true (verdadeiro).
```

A Interface Funcional Function

A interface funcional Function [Oracle, 2021]

A interface funcional *Function* representa uma 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 Function

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

```
Function<String, Integer> fTamanho = string → string.length();  
System.out.println(fTamanho.apply("cluck")); // Imprime o valor  
5.
```


A Interface Funcional BiFunction

A interface funcional BiFunction [Oracle, 2021]

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

@FunctionalInterface

```
public interface BiFunction<T, U, R>{
```

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

```
R apply(T t, U u);
```

```
}
```

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

A interface funcional Supplier [Oracle, 2021]

A interface funcional *Supplier* permite fornecer um retorno de um tipo parametrizado T , por meio da invocação do método `get()`, sem a necessidade de receber nenhum parâmetro de entrada.

@FunctionalInterface

```
public interface Supplier<T>{  
    T get();  
}
```

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);
```

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

- ▶ **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 pela 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.

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)

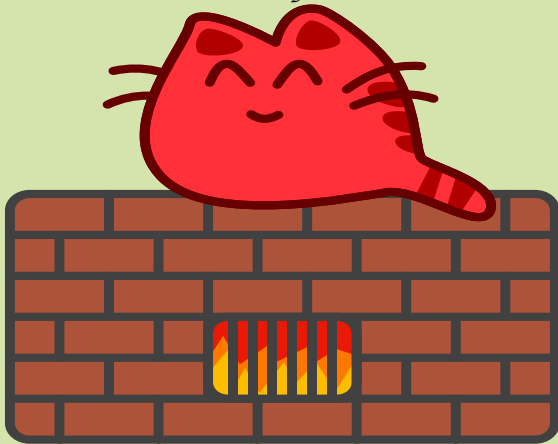


BOYARSKY, J. and SELIKOFF, S. **“Oracle Certified Associate Java SE 8 Programmer I: Study Guide”**. Sybex. Indianapolis, Indiana, USA. 2015.



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!