

**Pipeline** é o nome dado a junção desses métodos às listas de forma encadeada para obter um resultado. Um **Pipeline** é composto de:

1. **Souce**: Uma Collection
2. **Zero ou mais métodos intermediários**: filter, map, reduce...
3. **Um método finalizador**: forEach, sum, max, min...

- **Predicate**: aplicado a expressões lambda que retornam boolean
- **Consumer**: aplicado a expressões lambda que passa um objeto como parâmetro para métodos que retornam void
- **Function**: aplicado a expressões lambda que transforma um objeto T para U
- **Supplier**: aplicado a expressões lambda que instanciam um tipo T (como uma fábrica)

Generics < ? >

< ? super T >  
< ? extends T >

Dica: Wildcards (curingas)

Para entender melhor as declarações de **Generics** nas interfaces funcionais.

- **? super T**: A própria classe e todas as suas **superclasses**
- **? extends T**: A própria classe e todas as suas **subclasses**

< ? extends Funcionario > => Funcionário e suas subclasses

Predicate<T>

```
1: public class TestePredicate01 {
2:     public static void main(String[] args) {
3:         List<Produto> produtos = BDProduto.getProdutos();
4:         produtos.stream()
5:             .filter(p -> p instanceof Alimento)
6:             .forEach(p -> System.out.printf("%s \n\n", p));
7:     }
8: }
```

```
1: package java.util.function;
2:
3: public interface Predicate<T> {
4:     public boolean test(T t);
5: }
```

Consumer<T>

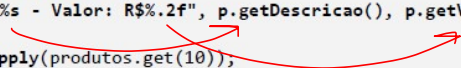
```
1: package java.util.function;
2:
3: public interface Consumer<T> {
4:     public void accept(T t);
5: }
```

```
1: public class TesteConsumer01 {
2:     public static void main(String[] args) {
3:         List<Produto> produtos = BDProduto.getProdutos();
4:         produtos.stream()
5:             .forEach(p -> System.out.printf("%s \n\n", p));
6:     }
7: }
```

## Function<T, R>

```
1: package java.util.function;
2:
3: public interface Function<T, R> {
4:     public R apply(T t);
5: }
```

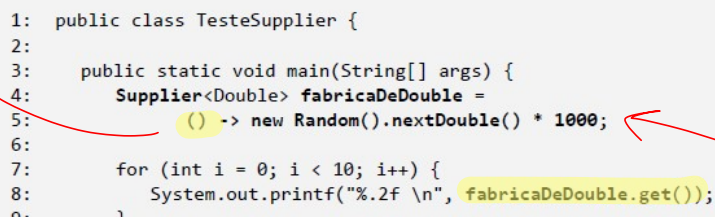
```
1: public class TesteFunction01 {
2:     public static void main(String[] args) {
3:         List<Produto> produtos = BDProduto.getProdutos();
4:         Function<Produto, String> obterPreco = (p ->
5:             String.format("Produto: %s - Valor: R$%.2f", p.getDescricao(), p.getValor()));
6:
7:         String preco = obterPreco.apply(produtos.get(10));
8:         System.out.println(preco);
9:     }
10: }
```



## Supplier<T>

```
1: package java.util.function;
2:
3: public interface Supplier<T> {
4:     public T get();
5: }
```

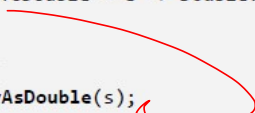
```
1: public class TesteSupplier {
2:
3:     public static void main(String[] args) {
4:         Supplier<Double> fabricaDeDouble =
5:             () -> new Random().nextDouble() * 1000;
6:
7:         for (int i = 0; i < 10; i++) {
8:             System.out.printf("%.2f \n", fabricaDeDouble.get());
9:         }
10:     }
11: }
```



## ToDoubleFunction<T>

```
1: package java.util.function;
2:
3: public interface ToDoubleFunction<T> {
4:     public double applyAsDouble(T t);
5: }
```

```
1: public class TesteToDoubleFunction01 {
2:     public static void main(String[] args) {
3:
4:         List<String> valoresString = List.of("123.44", "222.30", "456.00", "694.99");
5:         ToDoubleFunction<String> convertToDouble = e -> Double.valueOf(e);
6:
7:         double soma = 0;
8:         for (String s : valoresString) {
9:             soma += convertToDouble.applyAsDouble(s);
10:        }
11:         System.out.printf("Soma: %.2f \n", soma);
12:     }
13: }
```




## DoubleFunction<R>

```
1: package java.util.function;
2:
3: public interface DoubleFunction<R> {
4:     public R apply(double value);
5: }
```


```
1: public class TesteDoubleFunction01 {
2:     public static void main(String[] args) {
3:         List<Produto> produtos = BDProduto.getProdutos();
4:
5:         DoubleFunction<String> toMoeda = e -> getCurrencyInstance().format(e);
6:         produtos.stream()
7:             .filter(p -> p instanceof Celular)
8:             .forEach(p -> System.out.printf("%s \n", toMoeda.apply(p.getValor())));
9:     }
10: }
```

## BiPredicate<T, U>

```
1: package java.util.function;
2:
3: public interface BiPredicate<T, U> {
4:     public boolean test(T t, U u);
5: }
```




```
1: public class TesteBiPredicate01 {
2:     public static void main(String[] args) {
3:         List<Produto> produtos = BDProduto.getProdutos();
4:
5:         BiPredicate<Produto, Double> valorMaior =
6:             (p, d) -> p.getValor() > d;
7:         produtos.stream()
8:             .filter(p -> valorMaior.test(p, 1000.00))
9:             .forEach(p -> System.out.printf("%s \n", p.getDescricao()));
10:     }
11: }
```



## UnaryOperator<T>

```
1: package java.util.function;
2:
3: public interface UnaryOperator<T> extends Function<T, T> {
4:     @Override
5:     public T apply(T t);
6: }
```

```
1: public class TesteUnaryOperator {
2:     public static void main(String[] args) {
3:         List<Produto> produtos = BDProduto.getProdutos();
4:
5:         UnaryOperator<String> aoContrario = e -> {
6:             String contra = "";
7:             for (int i = e.length() - 1; i >= 0; i--) {
8:                 contra += e.charAt(i);
9:             }
10:             return contra;
11:         };
12:
13:         produtos.stream()
14:             .filter(p -> p.getClass().getSimpleName().equals("Agenda"))
15:             .forEach(p -> System.out.println(aoContrario.apply(p.getDescricao())));
16:     }
17: }
```



map(Function<T, R>)

- Extrair dados do objeto
- Conversão

```
1: public class TesteMap01 {
2:
3:     public static void main(String[] args) {
4:         List<? extends Produto> produtos = new ArrayList<>();
5:         produtos = BDProduto.getProdutos();
6:
7:         produtos.stream()
8:             .map(p -> descricaoValorValor(p)) //entra Produto sai String
9:             .forEach(s -> System.out.println(s));
10:
11:     }
12:
13:     private static String descricaoValorValor(Produto p) {
14:         return String.format("Nome: %s - Valor: R$ %.2f", p.getDescricao(), p.getValor());
15:     }
16:
17: }
```

- O método **map** recebe como argumento uma Function:
  - Uma Function recebe um tipo genérico e retorna alguma outra coisa
- map para retorno de tipos primitivos:
  - **mapToInt()**, **mapToLong()** e **mapToDouble()**

Peek(Consumer<T>)

```
1: public class TestePeek01 {
2:
3:     public static void main(String[] args) {
4:         List<? extends Produto> produtos = new ArrayList<>();
5:         produtos = BDProduto.getProdutos();
6:
7:         produtos.stream()
8:             .peek(p -> System.out.printf("Submetido ao filtro: %s \n", p.getDescricao()))
9:             .filter(p -> p instanceof Bicicleta)
10:            .forEach(p -> System.out.printf("Passou pelo filtro: %s \n", p.getDescricao()));
11:     }
12: }
```

### Atenção: Boas práticas com peek

Usando o método **peek** você pode alterar os dados dos elementos dentro de uma **stream**, e essa alteração irá refletir na Collection subjacente, no entanto, essa não é uma boa prática pois os elementos não estão sendo acessados de forma **thread-safe**.

## Buscas:

findFirst(Predicate<T>)

```
1: public class TesteBusca01 {
2:
3:     public static void main(String[] args) {
4:         List<Produto> produtos = new ArrayList<>();
5:         produtos = BDProduto.getProdutos();
6:
7:         Optional<Produto> localizado = produtos.stream()
8:             .filter(p -> p.getValor() > 2_000.00)
9:             .findFirst();
10:
11:         boolean saoTodosCelulares = produtos.stream()
12:             .allMatch(p -> p instanceof Celular);
13:
14:         boolean nenhumEhCelular = produtos.stream()
15:             .noneMatch(p -> p instanceof Celular);
16:
17:         if (localizado.isPresent()) {
18:             System.out.printf("Produto encontrado: %s\n\n", localizado.get().getDescricao());
19:         }
20:         System.out.printf("Todos são celulares: %s \n", saoTodosCelulares ? "sim" : "não");
21:         System.out.printf("Nenhum é celulares: %s \n", nenhumEhCelular ? "sim" : "não");
22:
23:     }
24: }
```



- findAny()
- anyMatch()

## Métodos para análise de dados:

**count()**

- Retorna a quantidade de elementos de uma stream.

**max(Comparator<? super T> comparator)**

- Recebe um Comparator e retorna um **Optional<T>** com o maior elemento de acordo com o critério de comparação especificado.

**min(Comparator<? super T> comparator)**

- Recebe um Comparator e retorna um **Optional<T>** com o menor elemento de acordo com o critério de comparação especificado.

```
1: public class TesteAnalise01 {
2:     public static void main(String[] args) {
3:         List<Produto> produtos = new ArrayList<>();
4:         produtos = BDProduto.getProdutos();
5:
6:         Comparator<Produto> comparaValor = (p1, p2) -> p1.getValor().compareTo(p2.getValor());
7:
8:         Optional<Produto> maisCaro = produtos.stream().max(comparaValor);
9:         Optional<Produto> maisBarato = produtos.stream().min(comparaValor);
10:        long quantidade = produtos.stream().count();
11:
12:        System.out.printf("Produtos:\n\tmais caro: %s\n\tmais barato: %s\nQuantidade: %d",
13:            maisCaro.get().getDescricao(),
14:            maisBarato.get().getDescricao(),
15:            quantidade);
16:    }
17: }
```

## Cálculo sobre stream

**average()**

- Retorna um Optional com a média aritmética dos elementos da Stream
- Retorna um Optional vazio se a lista estiver vazia

**sum()**

- Retorna a soma dos elementos da Stream

Esses métodos são encontrados nas Stream primitivas: **DoubleStream, IntStream e LongStream.**

```

1: public class TesteCalculos01 {
2:     public static void main(String[] args) {
3:         List<Produto> produtos = new ArrayList<>();
4:         produtos = BDProduto.getProdutos();
5:
6:         OptionalDouble mediaValorBike = produtos.stream()
7:             .filter(p -> p instanceof Bicicleta)
8:             .mapToDouble(p -> p.getValor())
9:             .average();
10:
11:         double somaValorBike = produtos.stream()
12:             .filter(p -> p instanceof Bicicleta)
13:             .mapToDouble(p -> p.getValor())
14:             .sum();
15:
16:         System.out.printf("Bicicletas:\n\tMédia valores: %s\n\tSoma valores: %s",
17:             moeda(mediaValorBike.getAsDouble()),
18:             moeda(somaValorBike));
19:     }
20: }

```

Esses métodos são encontrados nas Stream primitivas: **DoubleStream**, **IntStream** e **LongStream**.

## Ordenação

**sorted()**

**sorted(Comparator<? super T> comparator)**

```

1: public class TesteOrdenacao01 {
2:     public static void main(String[] args) {
3:         List<Produto> produtos = new ArrayList<>();
4:         produtos = BDProduto.getProdutos();
5:
6:         System.out.println("Ordenação padrão: ");
7:
8:         produtos.stream()
9:             .sorted()
10:            .map(p -> descricaoValor(p))
11:            .forEach(System.out::println);
12:
13:         Comparator<Produto> ordemValor = (p1, p2) -> p1.getValor().compareTo(p2.getValor());
14:
15:         System.out.println("\n\nOrdenado pelo valor: ");
16:
17:         produtos.stream()
18:             .sorted(ordemValor)
19:             .map(p -> descricaoValor(p))
20:             .forEach(System.out::println);
21:     }
22:
23:     private static String descricaoValor(Produto p) {
24:         return String.format("Descrição: %s - Valor: %s",
25:             p.getDescricao(),
26:             moeda(p.getValor()));
27:     }
28: }

```