**Capítulo DEZESSEIS**

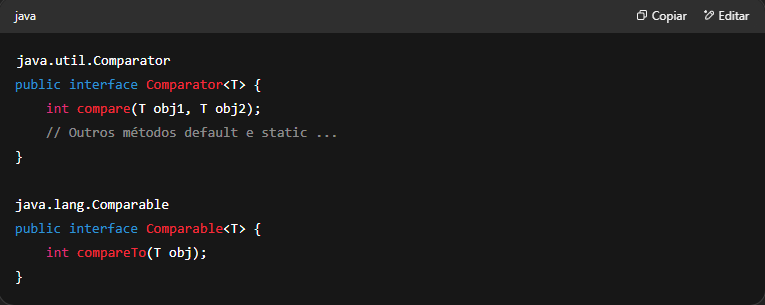
**Operações de Stream em Coleções**

**Objetivos do Exame**

* Desenvolver código que use métodos de dados de Stream e métodos de cálculo.
* Usar as interfaces java.util.Comparator e java.lang.Comparable.
* Ordenar uma coleção usando a API de Stream.

**Comparator e Comparable**

Para ordenar arrays ou coleções, o Java fornece duas interfaces muito semelhantes:



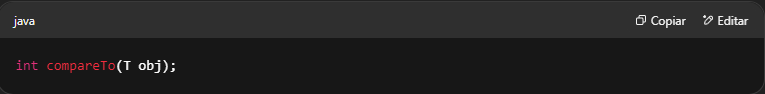
A diferença é que java.util.Comparator é implementada por uma classe que você usa para ordenar objetos de OUTRA classe, enquanto java.lang.Comparable é implementada pelo MESMO objeto que você deseja ordenar.

Com Comparator, você cria um objeto para comparar dois objetos de outro tipo a fim de ordená-los; é por isso que você recebe dois parâmetros, e o método se chama compare.

Com Comparable, você torna um objeto comparável a outro objeto do mesmo tipo para ordená-los; é por isso que você recebe apenas UM parâmetro e o método se chama compareTo. Como esta interface é mais fácil de compreender, vamos começar com ela.

**java.lang.Comparable**

O método a ser implementado é:



Como você pode ver, ele retorna um int. Aqui estão suas regras:

* Quando ZERO é retornado, significa que o objeto é IGUAL ao argumento.
* Quando um número MAIOR que zero é retornado, significa que o objeto é MAIOR que o argumento.
* Quando um número MENOR que zero é retornado, significa que o objeto é MENOR que o argumento.

Muitas classes do Java (como BigDecimal, BigInteger, wrappers como Integer, String, etc.) implementam essa interface com uma ordem natural (como 1, 2, 3, 4 ou A, B, C, a, b, c).

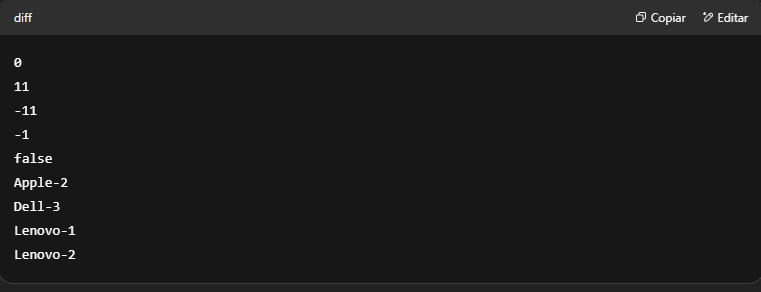
Como esse método pode ser usado para testar se um objeto é igual a outro, é recomendado que a implementação seja consistente com o método equals(Object) (se o método compareTo retorna 0, o método equals deve retornar true).

Uma vez que um objeto implementa essa interface, ele pode ser ordenado por Collections.sort() ou Arrays.sort(). Também pode ser usado como chave em um mapa ordenado (como TreeMap) ou em um conjunto ordenado (como TreeSet).

O seguinte é um exemplo de como um objeto pode implementar Comparable.

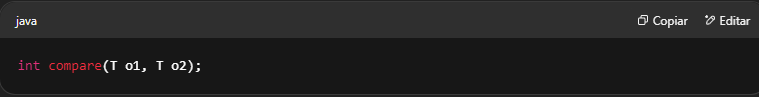


Ao executar este programa, esta é a saída:



**java.util.Comparator**

O método a ser implementado é:

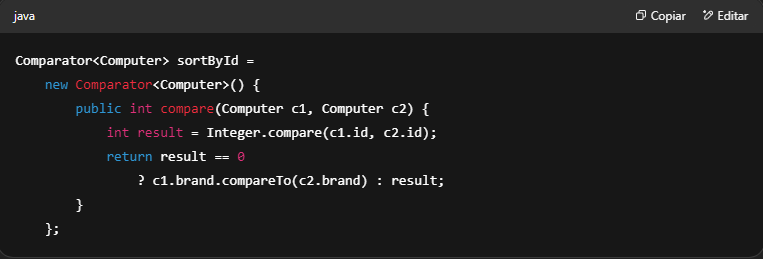


As regras do valor retornado são semelhantes às de Comparable:

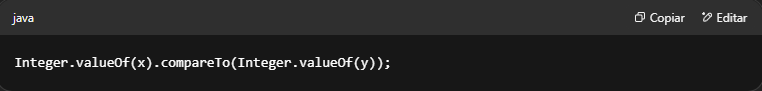
* Quando ZERO é retornado, significa que o PRIMEIRO argumento é IGUAL ao SEGUNDO argumento.
* Quando um número MAIOR que zero é retornado, significa que o PRIMEIRO argumento é MAIOR que o SEGUNDO argumento.
* Quando um número MENOR que zero é retornado, significa que o PRIMEIRO argumento é MENOR que o SEGUNDO argumento.

Uma vantagem de usar um Comparator em vez de Comparable é que você pode ter muitos Comparators para ordenar o mesmo objeto de diferentes formas.

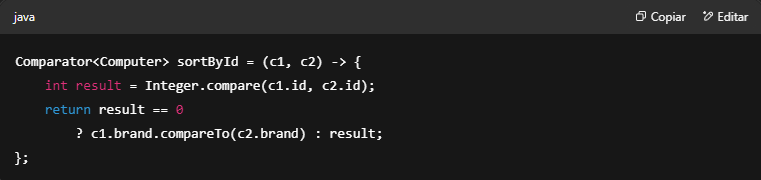
Por exemplo, podemos pegar a classe Computer do exemplo anterior para criar um Comparator que ordena primeiro pelo id e depois pela brand, e como as regras do valor retornado são praticamente as mesmas de Comparable, podemos usar o método compareTo:



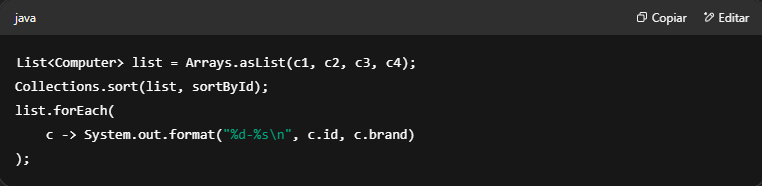
Além disso, Integer.compare(x, y) é equivalente a:



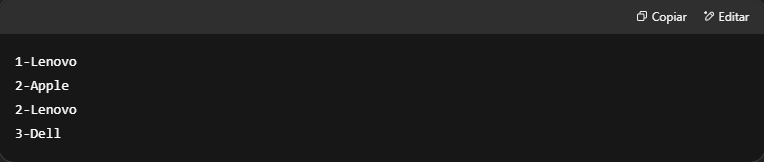
Felizmente, Comparator é uma interface funcional, então podemos usar uma expressão lambda em vez de uma classe interna:



Então, quando a usamos na lista do exemplo anterior:



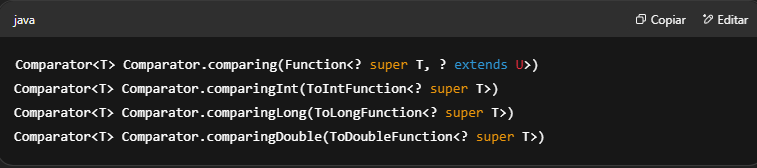
A saída é:



Caso esteja se perguntando, Comparable também é considerado uma interface funcional, mas como espera-se que Comparable seja implementado pelo objeto sendo comparado, você quase nunca a usará como uma expressão lambda.

**Métodos Default e Static em Comparator**

No Java 8, com a introdução de métodos default e static em interfaces, temos alguns métodos úteis em Comparator para simplificar nosso código como:



Esses métodos recebem uma Function (uma expressão lambda) que retorna o valor de uma propriedade do objeto que será usada para criar um Comparator usando o valor retornado por compareTo (observe também as versões para tipos primitivos).

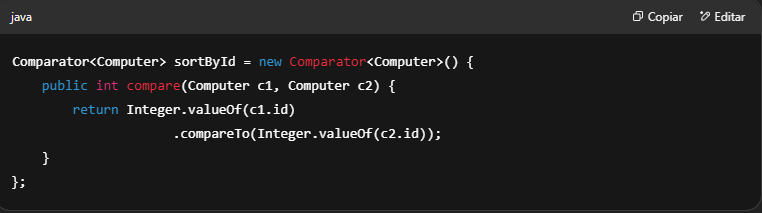
Por exemplo:



Ou:

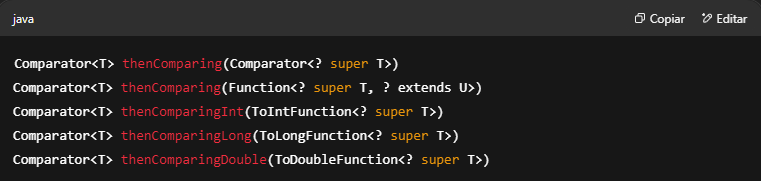


Eles são equivalentes a:

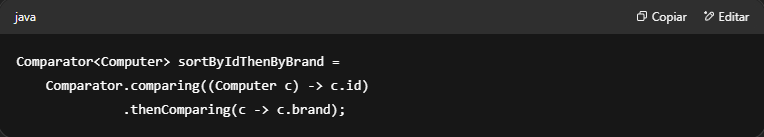


**thenComparing**

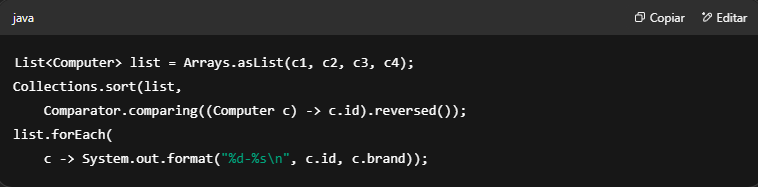
Outro método útil é thenComparing, que encadeia dois Comparators (observe que este **não é um método static**):



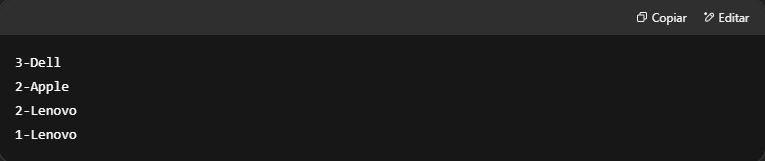
Dessa forma, podemos simplificar o código para criar um Comparator que ordena por id e depois por brand usando:



Por fim, o método default reversed() criará um Comparator que inverte a ordem do Comparator original:

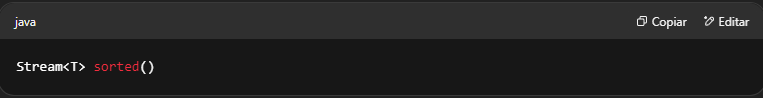


A saída:

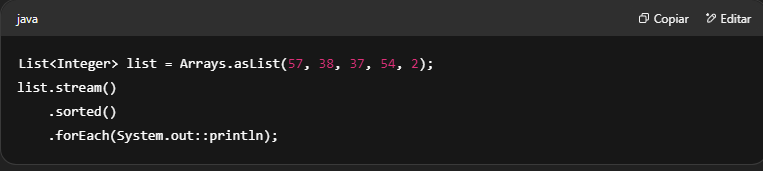


**Ordenando um Stream**

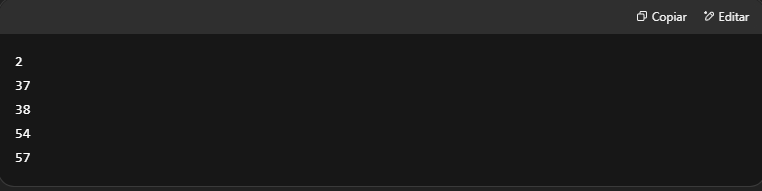
Ordenar um *stream* é simples. O método:



Retorna um *stream* com os elementos ordenados de acordo com sua **ordem natural**. Por exemplo:

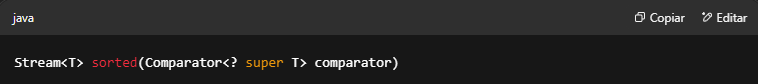


Irá imprimir:

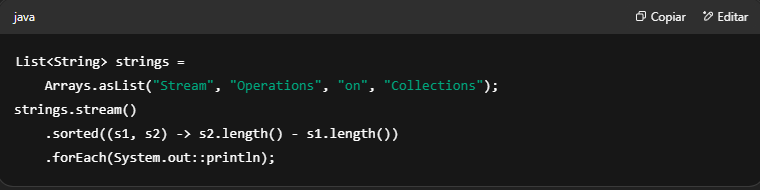


O único requisito é que os elementos do *stream* implementem java.lang.Comparable (dessa forma, eles são ordenados em ordem natural). Caso contrário, uma ClassCastException poderá ser lançada.

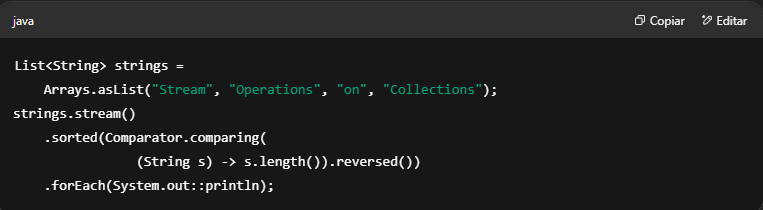
Se quisermos ordenar usando uma ordem diferente, há uma versão deste método que recebe um java.util.Comparator (essa versão **não está disponível** para *streams* primitivos como IntStream):



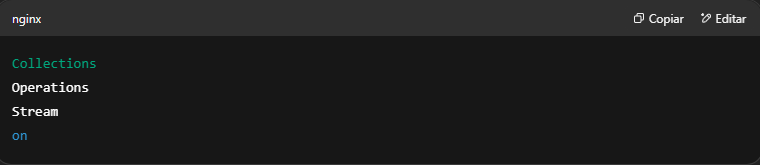
Por exemplo:



Ou:



Ambos irão imprimir:

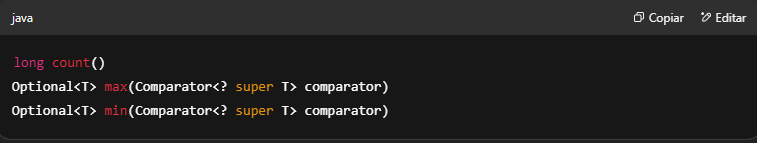


O primeiro trecho de código retornará um valor positivo se o comprimento da primeira String for menor que o da segunda, e um valor negativo caso contrário, para ordenar as Strings em **ordem decrescente**.

O segundo trecho de código criará um Comparator com o comprimento da String em ordem natural (**ordem crescente**) e depois inverterá essa ordem.

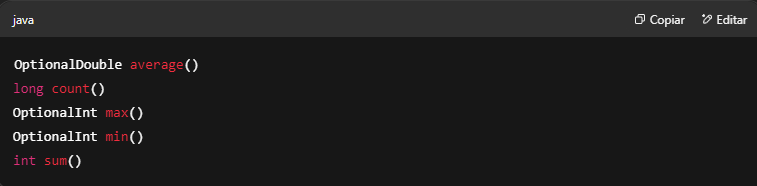
**Métodos de Dados e de Cálculo**

A interface Stream fornece os seguintes métodos de dados e cálculo:

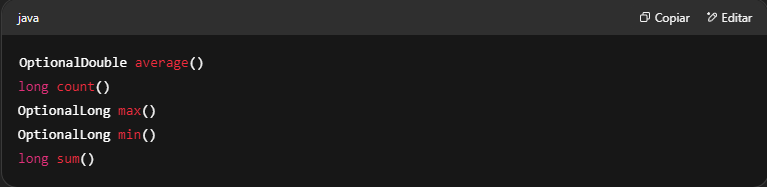


E, no caso das versões primitivas da interface Stream, temos os seguintes métodos:

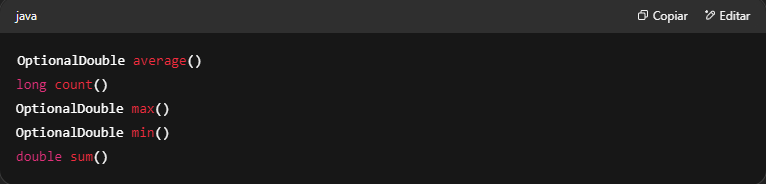
**IntStream**



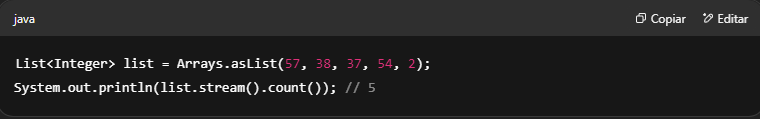
LongStream



DoubleStream



* count() retorna o número de elementos no *stream* ou zero se o *stream* estiver vazio:

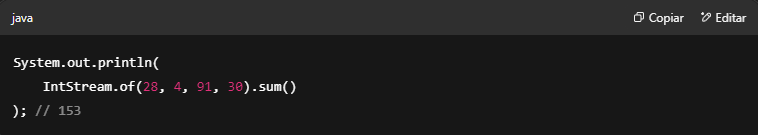


* min() retorna o valor mínimo no *stream* envolto em um Optional ou um Optional vazio se o *stream* estiver vazio.
* max() retorna o valor máximo no *stream* envolto em um Optional ou um Optional vazio se o *stream* estiver vazio.

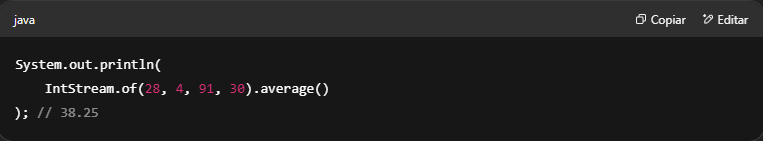
Quando falamos sobre primitivos, é fácil saber qual é o valor mínimo ou máximo. Mas quando estamos falando sobre objetos (de qualquer tipo), o Java precisa saber **como compará-los** para saber qual é o máximo e o mínimo. É por isso que a interface Stream precisa de um Comparator para max() e min():



sum() retorna a soma dos elementos no *stream* ou zero se o *stream* estiver vazio:



average() retorna a média dos elementos no *stream* envolta em um OptionalDouble ou um OptionalDouble vazio se o *stream* estiver vazio:



**Pontos-chave**

* java.util.Comparator é implementada por uma classe que você usa para ordenar objetos de OUTRA classe.
* java.lang.Comparable é implementada pelo MESMO objeto que você deseja ordenar.

Os principais métodos de ambas as interfaces retornam um int. Suas regras são muito semelhantes:

* Quando ZERO é retornado, significa que o objeto (ou primeiro argumento) é IGUAL ao (segundo) argumento.
* Quando um número MAIOR que zero é retornado, significa que o objeto (ou primeiro argumento) é MAIOR que o (segundo) argumento.
* Quando um número MENOR que zero é retornado, significa que o objeto (ou primeiro argumento) é MENOR que o (segundo) argumento.

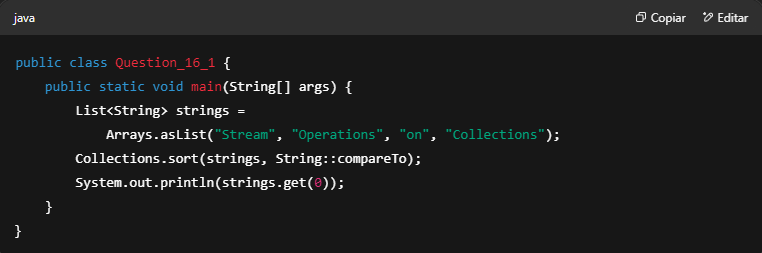
Os métodos comparing(), thenComparing() e reversed() são métodos auxiliares da interface Comparator adicionados no Java 8.

O método sorted() da interface Stream retorna um *stream* com os elementos ordenados de acordo com sua ordem natural. Você também pode passar um Comparator como argumento.

* count() retorna o número de elementos no *stream* ou zero se o *stream* estiver vazio.
* min() retorna o menor valor no *stream* envolto em um Optional ou um vazio se o *stream* estiver vazio.
* max() retorna o maior valor no *stream* envolto em um Optional ou um vazio se o *stream* estiver vazio.
* sum() retorna a soma dos elementos no *stream* ou zero se o *stream* estiver vazio.
* average() retorna a média dos elementos no *stream* envolta em um OptionalDouble ou vazio se o *stream* estiver vazio.

**Autoavaliação**

**1. Dado:**



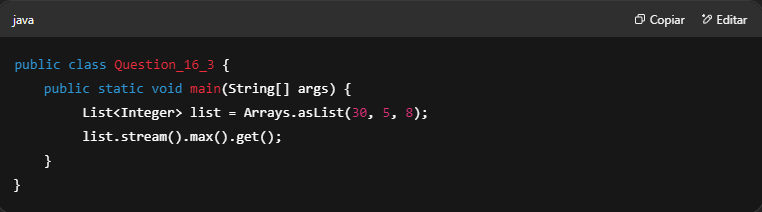
**Qual é o resultado?**

A. Collections  
B. on  
C. Falha de compilação  
D. Uma exceção ocorre em tempo de execução

**2. Qual das seguintes instruções retorna um Comparator válido?**

A. (String s) -> s.length();  
B. Comparator.reversed();  
C. Comparator.thenComparing((String s) -> s.length());  
D. Comparator.comparing((String s) -> s.length() \* -1);

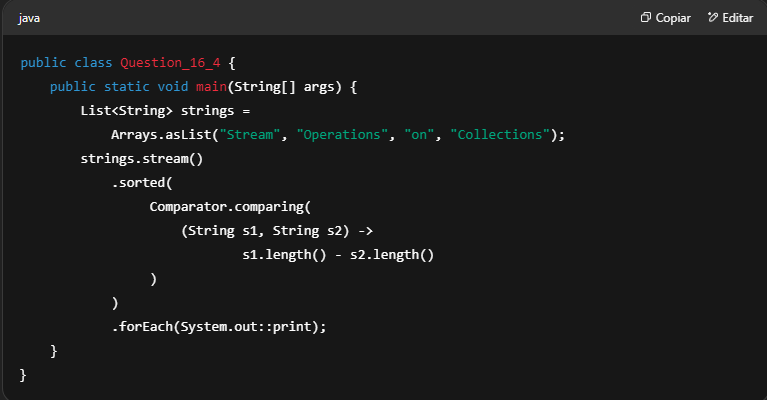
**3. Dado:**



**Qual é o resultado?**

A. 5  
B. 30  
C. Falha de compilação  
D. Uma exceção ocorre em tempo de execução

**4. Dado:**



**Qual é o resultado?**

A. CollectionsOperationsStreamOn  
B. onStreamOperationsCollections  
C. Falha de compilação  
D. Uma exceção ocorre em tempo de execução