**Capítulo TREZE**

**Iterando e Filtrando Coleções**

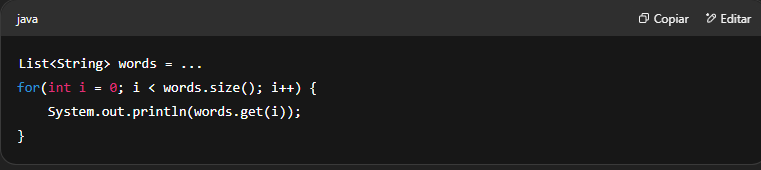
**Objetivos do Exame**

* Streams e Filtros de Coleções.
* Iterar usando métodos forEach de Streams e List.
* Filtrar uma coleção usando expressões lambda.

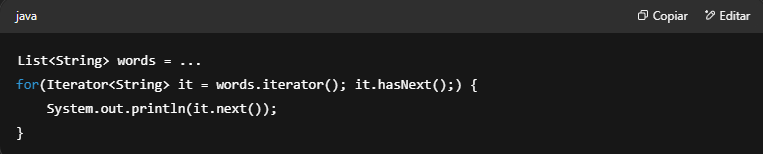
**Iteração**

Geralmente, quando você tem uma lista, você quer iterar sobre seus elementos. Uma maneira comum é usar um bloco for.

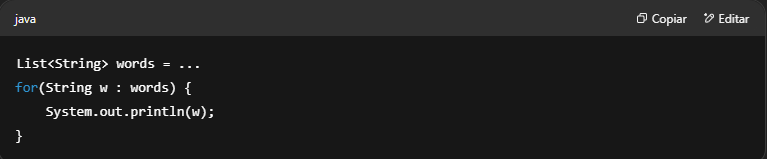
Ou com um índice:



Ou com um iterador:

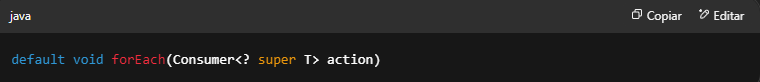


Ou com o chamado loop *for-each*:



Além de parecer feio, os dois primeiros adicionam pontos onde um erro pode acontecer (as variáveis de índice e iterador). A forma recomendada é usar o *for-each* sempre que possível.

Portanto, aproveitando as interfaces funcionais, o Java 8 adiciona outra opção para iterar listas baseada no *for-each*, o método foreach:



Como este método é definido na interface Iterable, ele não está disponível apenas para listas, mas para todas as implementações desta interface, como Queues, Sets, Deques e até algumas exceções relacionadas a SQL, como SQLException.

Observe também que este é um método default, o que significa que há uma implementação padrão que as classes que implementam podem sobrescrever (e muitas fazem, principalmente para lidar com modificações concorrentes).

Esta é a implementação padrão:



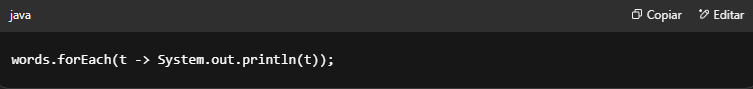
Basicamente, é um loop *for-each* usando os novos recursos funcionais do Java 8.

Para usá-lo, podemos começar com uma classe anônima:

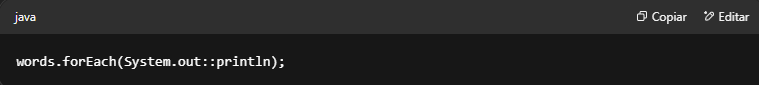


Lembre-se de que a interface Consumer representa uma operação que recebe um parâmetro, mas não retorna nenhum resultado.

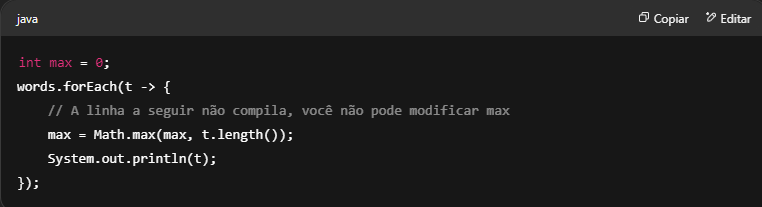
Essa classe anônima pode ser transformada em uma expressão lambda:



Ou, neste exemplo específico, uma referência de método:

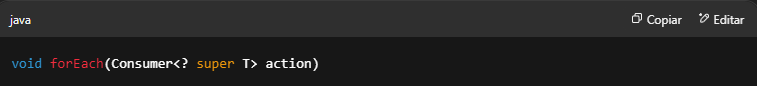


Lembre-se das regras sobre o uso de variáveis final ou efetivamente final dentro de classes anônimas ou expressões lambda. Código como o seguinte não é válido:



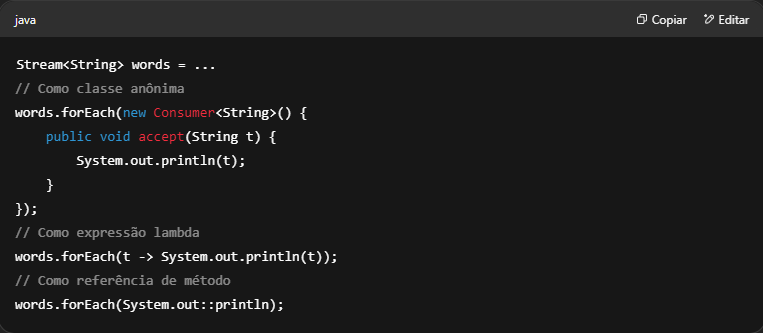
Se você quiser fazer coisas assim (obter o comprimento máximo de todas as strings em uma lista), é melhor usar *streams* para iterar sobre a coleção e aplicar outras operações (para esse exemplo específico, veremos como calcular o comprimento máximo com o método reduce em um capítulo posterior).

A interface Stream fornece um método forEach correspondente:

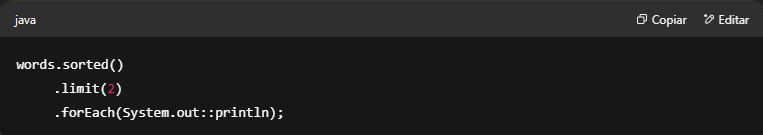


Como este método não retorna um stream, ele representa uma operação terminal.

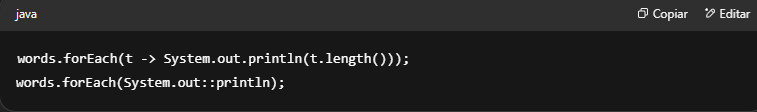
Usá-lo não é diferente da versão Iterable:



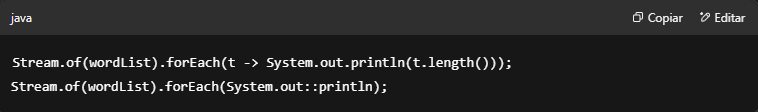
Claro, a vantagem de usar streams é que você pode encadear operações, por exemplo:



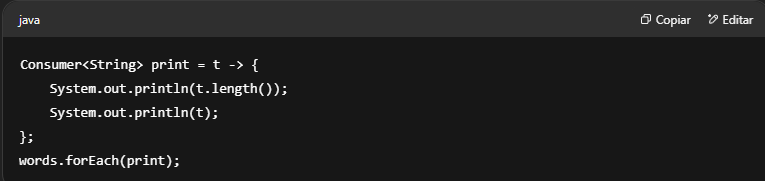
Como é uma operação terminal, você não pode fazer coisas como:



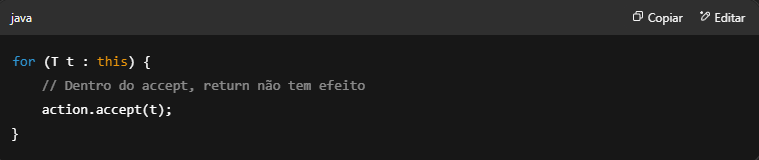
Se quiser fazer algo assim, crie um novo stream a cada vez:



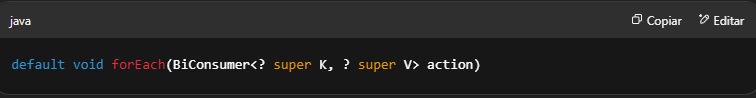
Ou encapsule o código dentro de uma única lambda:



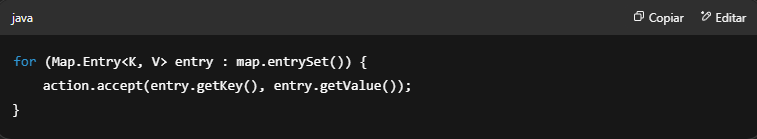
Você não pode usar return, break ou continue para terminar uma iteração. break e continue gerarão um erro de compilação, pois não podem ser usados fora de um loop, e return não faz sentido quando vemos que o método foreach é basicamente implementado assim:



Como nota (já que não é coberto no exame), o Java 8 também adicionou um método forEach à interface Map. No entanto, como um mapa tem uma chave e um valor, este novo método usa um BiConsumer:



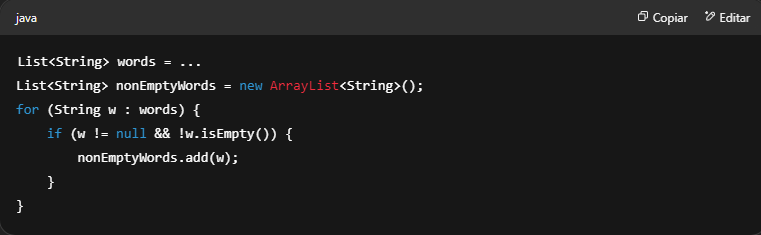
Com uma implementação padrão equivalente a:



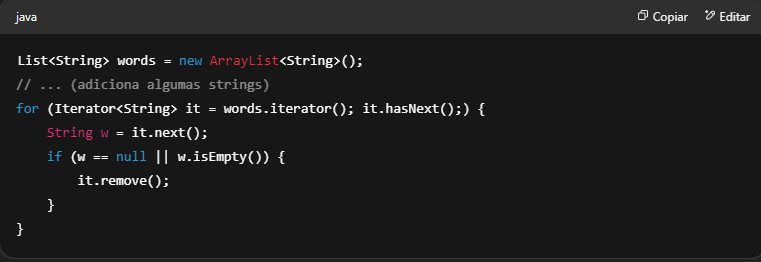
**Filtragem**

Outro requisito comum é filtrar (ou remover) elementos de uma coleção que não correspondam a uma condição específica.

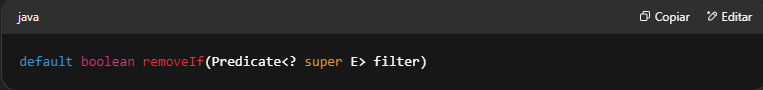
Normalmente você faz isso copiando os elementos correspondentes para outra coleção:



Ou removendo os elementos que não correspondem diretamente da coleção usando um iterador (somente se a coleção suportar remoção):

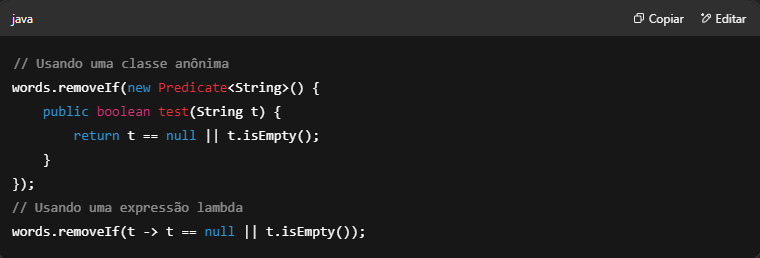


Agora, no Java 8, há um novo método na interface Collection:

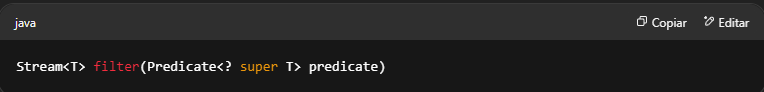


Que remove todos os elementos da coleção que satisfaçam o predicado fornecido (a implementação padrão usa a versão com iterador).

Isso torna o código mais simples, usando expressões lambda ou referências de método:

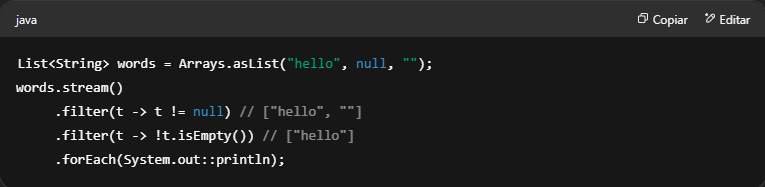


Para o caso em que você copia os elementos correspondentes para outra coleção, você tem o método filter da interface Stream:

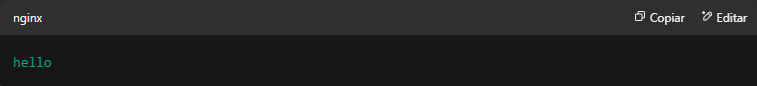


Que retorna um novo stream contendo os elementos que satisfazem o predicado fornecido.

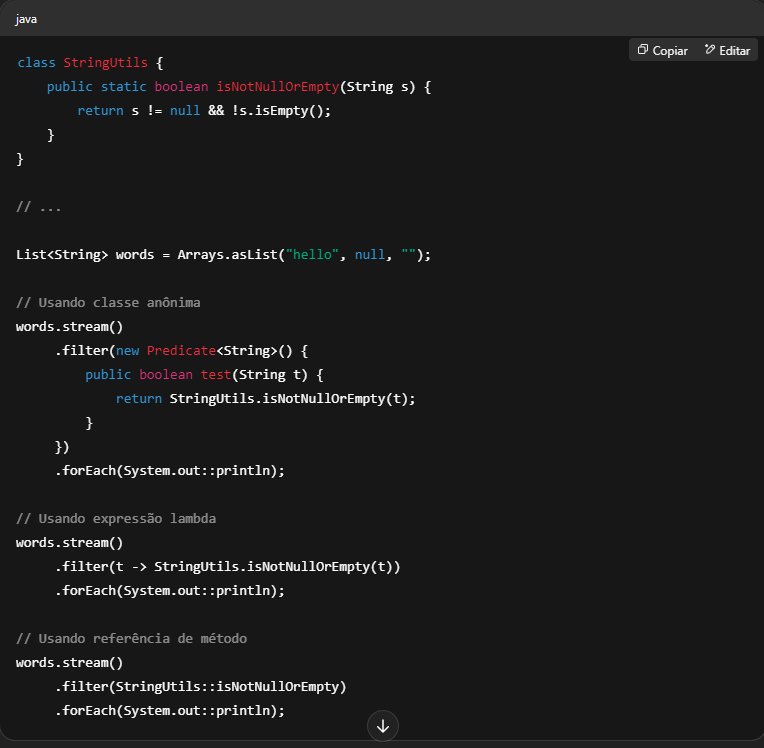
Como este método retorna um stream, ele representa uma operação intermediária, o que basicamente significa que você pode encadear qualquer número de filtros ou outras operações intermediárias:



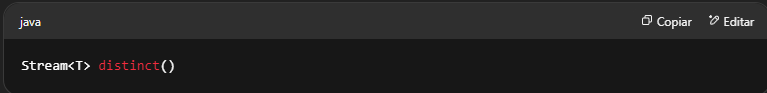
Claro, o resultado da execução desse código é:



Você também pode criar um método (ou usar um já existente) em alguma classe para fazer isso com uma referência de método, apenas por questão de clareza:



A interface Stream também possui o método distinct para filtrar elementos duplicados, de acordo com o método Object.equals(Object):

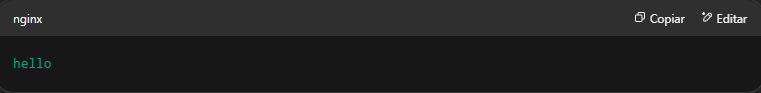


Novamente, como ele retorna um novo stream, essa é uma operação intermediária. Como ele precisa conhecer os valores dos elementos para identificar duplicatas, essa operação também é com estado (*stateful*).

Aqui está um exemplo:

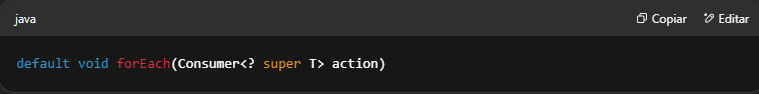


O resultado é:

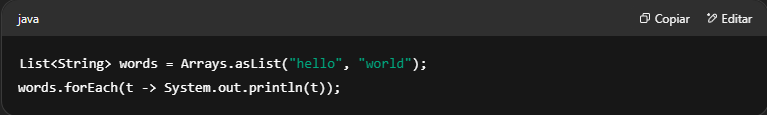


**Pontos-Chave**

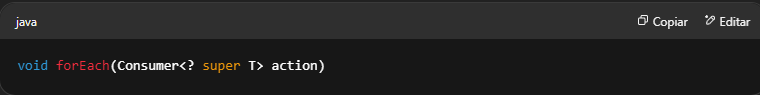
O Java 8 adiciona o seguinte método à interface Iterable como outra opção para iterar sobre implementações dessa interface (como listas):



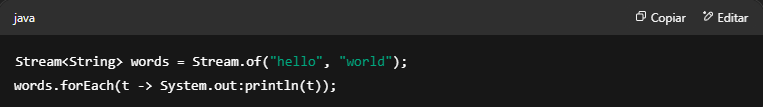
Por exemplo:



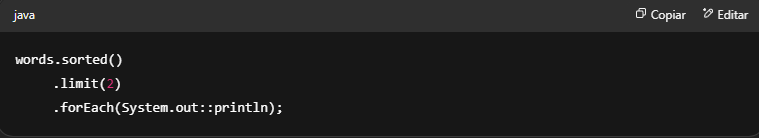
A interface Stream também possui esse método:



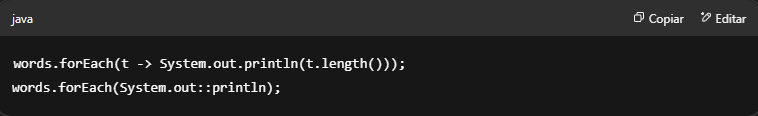
Essa é uma operação terminal. Aqui está um exemplo:



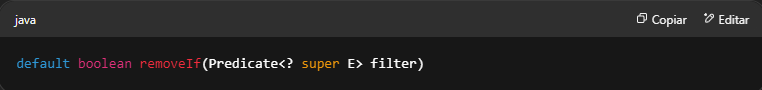
Claro, a vantagem de usar streams é que você pode encadear operações, por exemplo:



Mas como se trata de uma operação terminal, você **não pode** fazer coisas como:

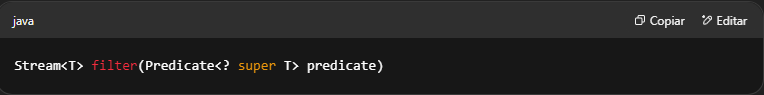


Para filtragem, no lado das coleções, temos um novo método:



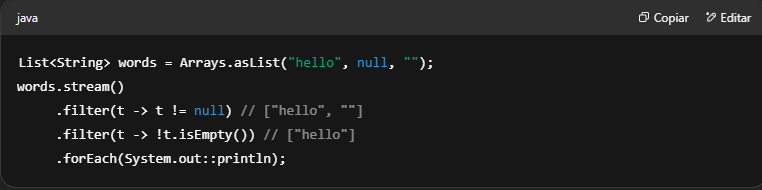
Que remove todos os elementos da coleção que satisfaçam o predicado fornecido.

Na interface Stream, temos:

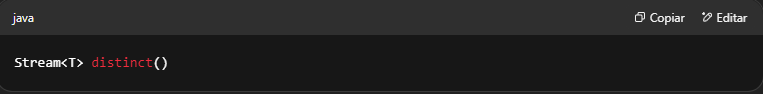


Que retorna um novo stream composto pelos elementos que satisfazem o predicado fornecido.

Como este método retorna um stream, ele representa uma operação intermediária, o que significa que você pode encadear qualquer número de filtros ou outras operações intermediárias:



A interface Stream também possui o método distinct para filtrar elementos duplicados, conforme o método Object.equals(Object):



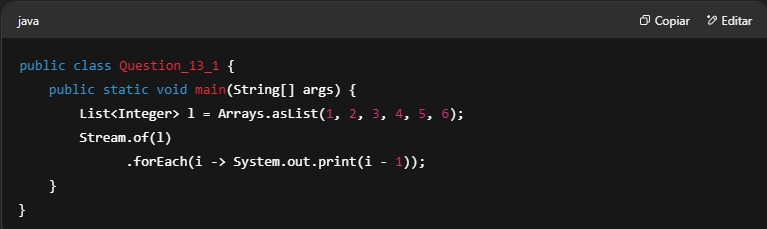
Essa é uma operação intermediária e, como precisa conhecer os valores dos elementos para identificar os duplicados, essa operação também é com estado (*stateful*).

Aqui está um exemplo:



**Autoavaliação**

**1. Dado:**



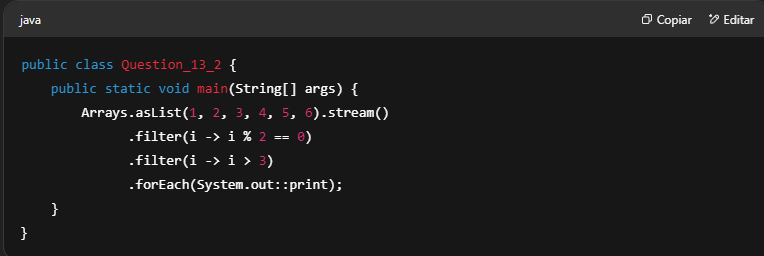
Qual é o resultado?

A. 123456  
B. 012345  
C. 543210  
D. Erro de compilação  
E. Uma exceção é lançada

**2. Qual das seguintes afirmações é verdadeira?**

A. filter é uma operação terminal.  
B. filter é uma operação com estado (*stateful*).  
C. Stream.forEach recebe uma implementação da interface funcional Consumer como argumento.  
D. Você pode encadear mais de uma operação forEach em um pipeline de stream.

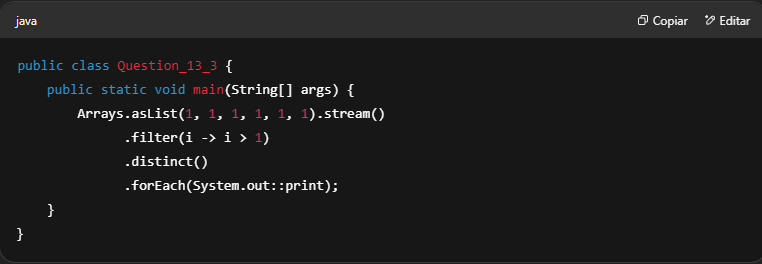
**3. Dado:**



Qual é o resultado?

A. 246  
B. 46  
C. 1  
D. 5  
E. Erro de compilação

**4. Dado:**



Qual é o resultado?

A. 1  
B. 0  
C. Nada é impresso  
D. Erro de compilação  
E. Uma exceção é lançada