**Parte TRÊS**

Expressões Lambda

**Capítulo OITO**

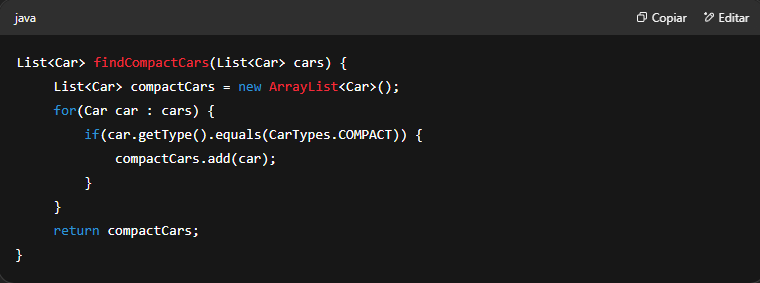
Interfaces Funcionais

**Objetivos do Exame**  
Criar e usar expressões Lambda.

**Simplificando as coisas**

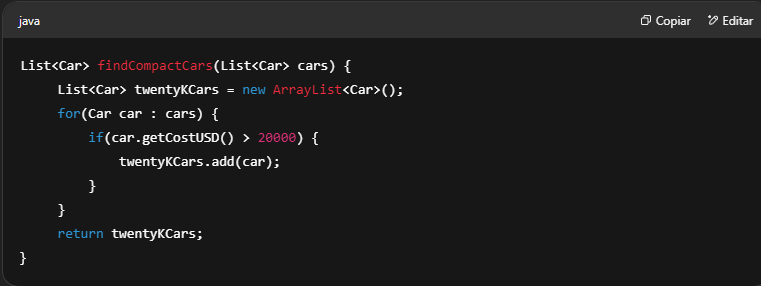
Suponha que temos um programa que possui uma lista de carros e precisamos buscar todos os carros compactos.

Podemos ter algo assim para fazer o trabalho:



Fácil. Mas no dia seguinte, os usuários percebem que também precisam buscar por carros que custem mais de 20.000 dólares.

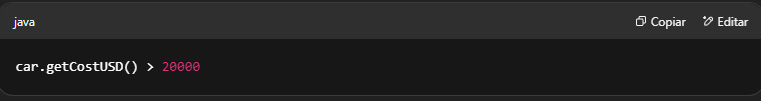
Então criamos algo assim:



Agora olhe para o código. É praticamente o mesmo. As únicas diferenças são as linhas:



E:



E se precisarmos filtrar por outra condição no futuro?

É errado ter código duplicado ou copiado e colado, não é muito flexível para mudanças e propenso a erros.

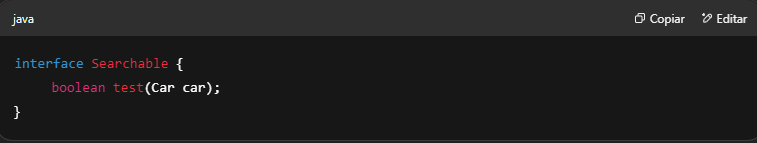
**A Abordagem Orientada a Objetos**

Como o Java é orientado a objetos, vamos aproveitar isso resolvendo o problema usando um padrão de projeto popular, Strategy (Estratégia).

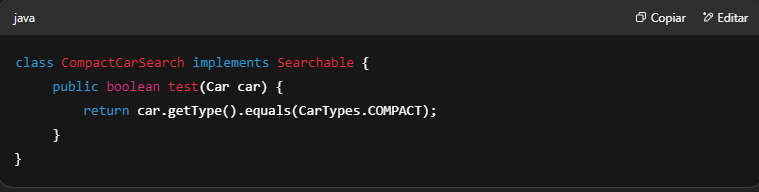
Esse padrão faz exatamente o que precisamos: encapsula o comportamento que varia (algoritmo) e torna todos esses comportamentos intercambiáveis.

A forma recomendada de implementar o padrão Strategy em Java é com uma interface, para que possamos criar implementações para cada algoritmo (neste caso, as condições a serem testadas).

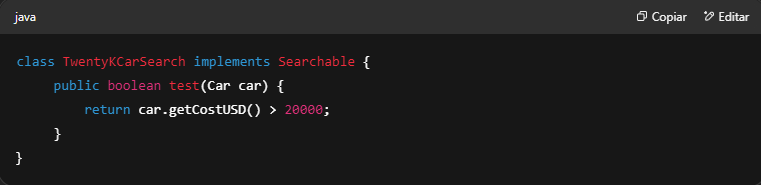
Então vamos codificar uma interface que pode conter a condição de busca que varia, por exemplo:



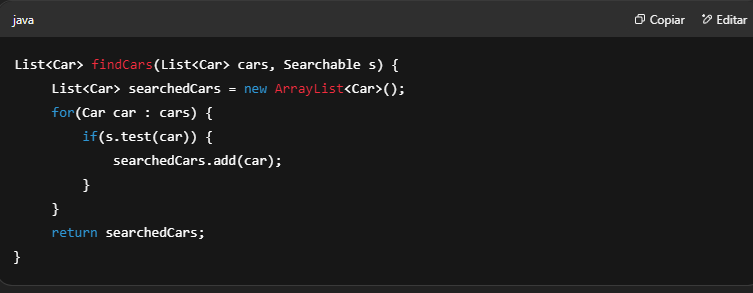
E colocamos o código que varia em implementações dessa interface:



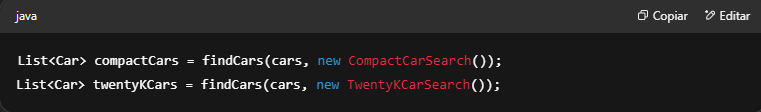
E:



Dessa forma, podemos tornar o método que faz a busca real um pouco mais geral:



E chamar a função desta forma:



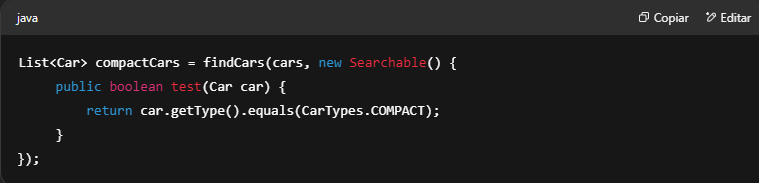
Não temos mais código duplicado.

Se o usuário quiser outra forma de buscar por um carro, em vez de copiar e colar um método de busca, agora apenas implementamos outra classe com uma condição.

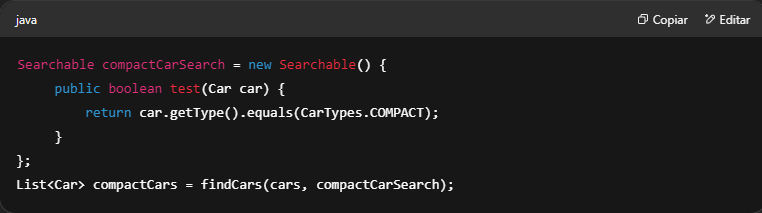
Agora somos mais flexíveis para mudar.

Mas ainda é uma solução COMPLEXA. Em vez de dois métodos, temos duas classes e uma interface.

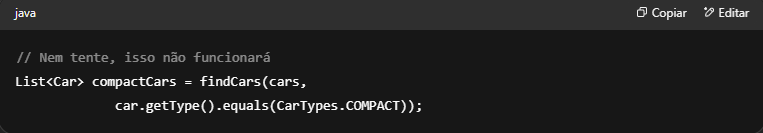
E se transformássemos as implementações em classes anônimas? Por exemplo:



Um pouco melhor, podemos até atribuir a classe a uma variável, se quisermos usá-la em outras partes do programa:



Nada mal, mas não seria ótimo apenas passar a condição ao método? Como isto, por exemplo:



**Um novo tipo de valor**

Basicamente, em Java, trabalhamos com dois tipos de valor: valores primitivos e referências para objetos.

Ambos podem ser usados como argumentos de um método:



Então, se quisermos passar um trecho de código para um método, temos que encapsulá-lo em um objeto (como no exemplo anterior).

Mas no Java 8, podemos passar esse trecho de código diretamente através do uso de uma **expressão lambda**.

Coincidentemente, para usar uma expressão lambda em vez de um objeto no exemplo anterior, temos que começar com a mesma interface:



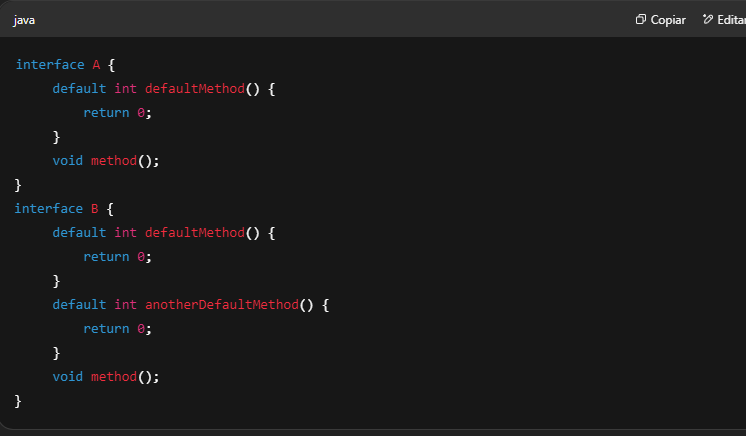
**Uma interface funcional**

O ponto de partida para aprender sobre expressões lambda é aprender sobre interfaces funcionais.

Uma **interface funcional** é qualquer interface que tenha **exatamente UM MÉTODO ABSTRATO**.

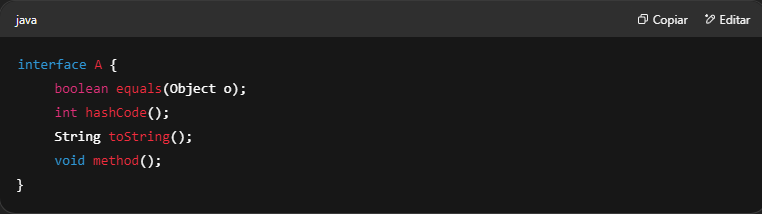
Essa é uma definição complicada. Muitas pessoas pensam que apenas ter um método torna uma interface funcional (já que métodos de interface são abstratos por padrão), mas não é bem assim.

Veja, por exemplo, métodos default. Como métodos default possuem implementação, eles **não são abstratos**. Portanto, interfaces como as seguintes são consideradas funcionais:

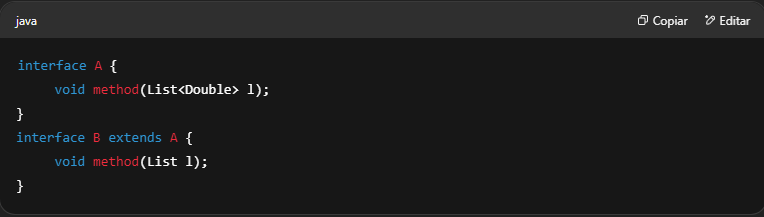


Se uma interface declara um método abstrato com a assinatura de um dos métodos de java.lang.Object, ele **não conta** para a contagem de métodos abstratos da interface funcional, já que qualquer implementação da interface terá uma implementação do método (pois todas as classes estendem java.lang.Object).

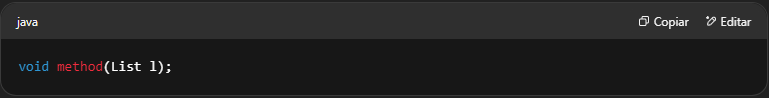
Portanto, uma interface como a seguinte é considerada funcional:



Um cenário mais confuso é quando uma interface herda um método que é equivalente, mas não idêntico a outro:



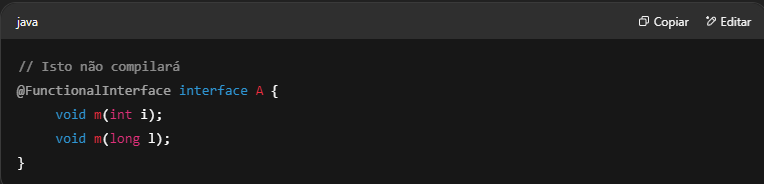
Nesse caso, o método é o mesmo, então é considerado como um único método. A classe que implementar B terá que implementar o método que pode legalmente sobrescrever todos os métodos abstratos:



Aliás, uma interface vazia **não** é considerada funcional.

A chave aqui é ter EXATAMENTE UM MÉTODO ABSTRATO; é por isso que essas interfaces também são chamadas de **interfaces de método abstrato único (SAM - Single Abstract Method)**.

Para facilitar, o Java 8 também introduziu a anotação @FunctionalInterface, que gera um erro de compilação quando a interface anotada não é uma interface funcional válida.



Interfaces Java que possuem apenas um método declarado em sua definição agora são anotadas como interfaces funcionais. Alguns exemplos são:

* java.lang.Runnable
* java.util.Comparator
* java.util.concurrent.Callable
* java.awt.event.ActionListener

Mas lembre-se, essa anotação serve apenas para ajudar; tê-la não torna uma interface funcional.

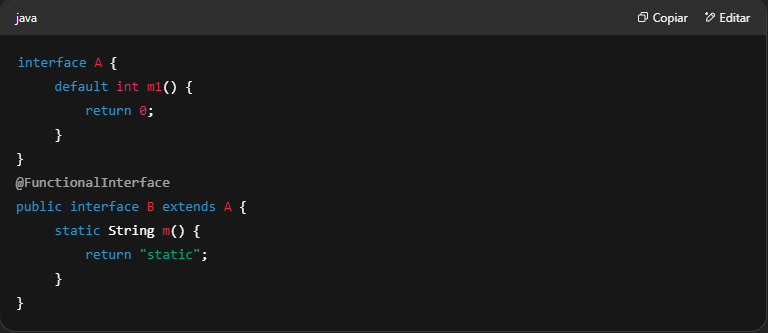
No próximo capítulo, veremos como interfaces funcionais e expressões lambda estão relacionadas.

**Pontos-chave**

* Uma interface funcional é qualquer interface que tenha exatamente um método abstrato.
* Como métodos default têm implementação, eles não são abstratos, então uma interface funcional pode ter qualquer número deles.
* Se uma interface declara um método abstrato com a assinatura de um dos métodos de java.lang.Object, ele não conta para a contagem.
* Uma interface funcional é válida mesmo que herde um método equivalente, mas não idêntico a outro.
* Uma interface vazia não é considerada funcional.
* Uma interface funcional é válida mesmo sem a anotação @FunctionalInterface.
* Interfaces funcionais são a base das expressões lambda.

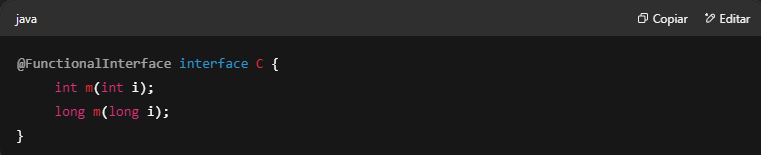
**Autoavaliação**

**1. Dado:**



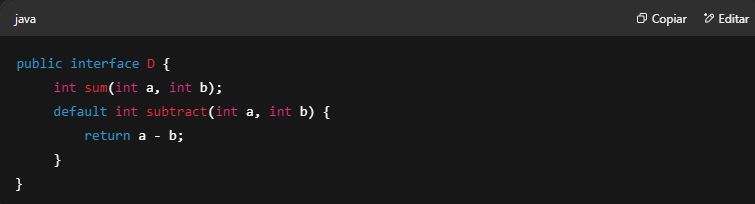
**Quais das seguintes afirmações são verdadeiras?**  
A. A compilação falha  
B. Compila com sucesso  
C. Compila somente se a interface B declarar um método default  
D. Uma exceção ocorre em tempo de execução se essa interface for implementada por uma classe

**2. Dado:**



**Quais das seguintes afirmações são verdadeiras?**  
A. Este código compila com sucesso  
B. Se removermos a anotação, este código compilará  
C. Se removermos um método, este código compilará  
D. A anotação @FunctionalInterface torna esta interface funcional

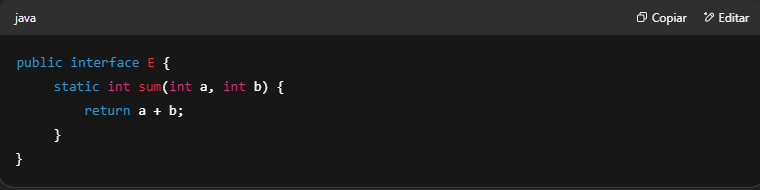
**3. Dado:**



**Quais das seguintes afirmações são verdadeiras?**  
A. Este código compila com sucesso  
B. Este código não compila  
C. Este é um exemplo de uma interface funcional  
D. Remover o método sum tornaria esta interface funcional

**4. Quais das seguintes interfaces da API Java podem ser consideradas funcionais?**  
A. java.util.concurrent.Callable  
B. java.util.Map  
C. java.util.Iterator  
D. java.lang.Comparable

**5. Dado:**



**Quais das seguintes afirmações são verdadeiras?**  
A. Este código não compila  
B. Este código compila com sucesso  
C. Este é um exemplo de interface funcional  
D. Adicionar a anotação @FunctionalInterface tornaria esta interface funcional