**Capítulo QUATRO**

**Interfaces**

**Objetivos do Exame**

Desenvolver código que declare, implemente e/ou estenda interfaces e use a anotação @Override.

**O que é uma interface?**

Na primeira vez que você olha para uma interface, você provavelmente vai pensar que ela é como uma classe com apenas definições de métodos:



E, em termos práticos, você está certo.

Uma interface é um tipo de dado que apenas define métodos (abstratos) que uma classe deve implementar.

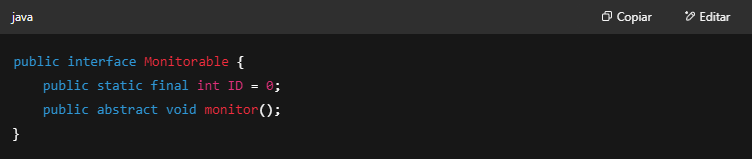
Embora, conceitualmente, seja mais interessante do que isso, porque isso permite que você defina o que uma classe pode fazer sem dizer como fazer. É por isso que se diz que uma interface é um contrato.

Qualquer classe que implemente uma interface deve fornecer uma implementação para todos os métodos da interface, caso contrário, a classe deve ser marcada como abstract.

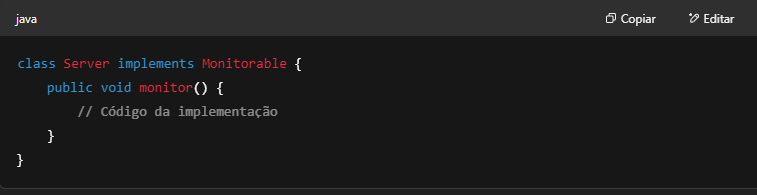
Assim como uma classe é definida com a palavra-chave class, uma interface é definida com a palavra-chave interface.

Se uma classe quiser implementar uma interface, ela deve especificá-la com a palavra-chave implements.

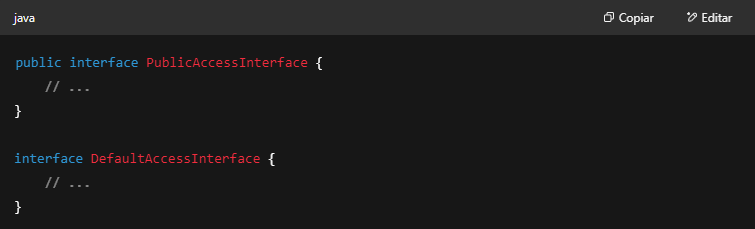
**Definindo uma interface**



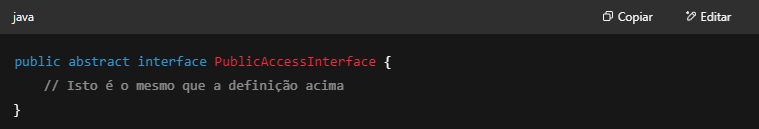
**Implementando uma interface**



Assim como uma classe, uma interface tem acessibilidade public ou padrão (default):



Interfaces são abstratas por padrão (você não precisa especificar isso):

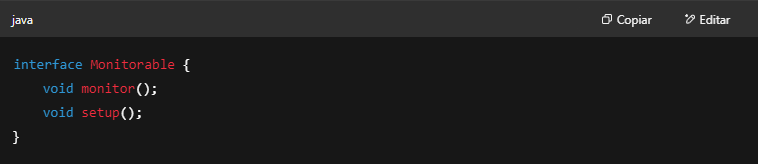


Isso significa duas coisas:

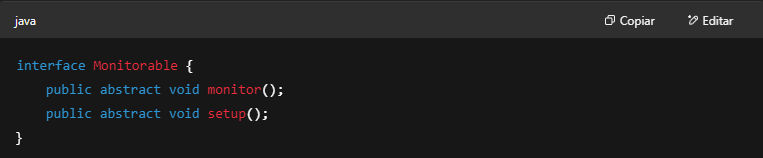
* Você não pode instanciar uma interface diretamente, ela deve ser implementada por uma classe para ser usada.
* Uma interface **não pode** ser marcada como final.

Os métodos definidos em uma interface são por padrão PUBLIC e ABSTRACT, o compilador os tratará como tais mesmo que você não os especifique.

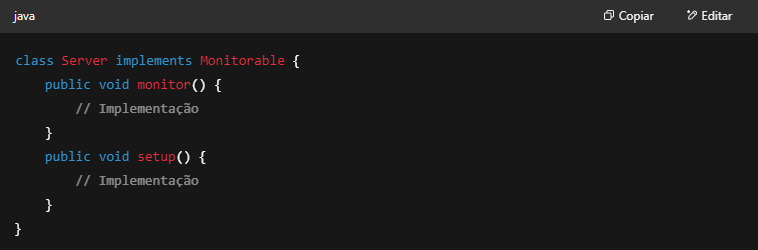
Então, mesmo que você defina uma interface assim:



Para o compilador, a interface será assim:

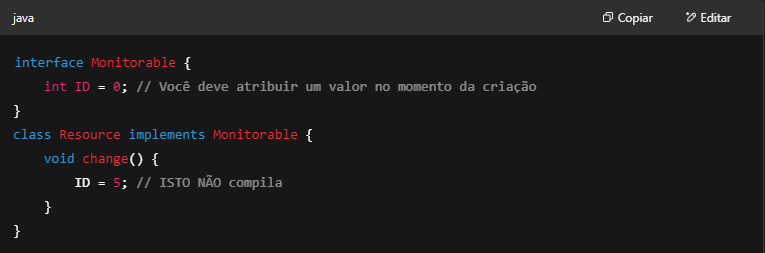


Essa é a razão pela qual você **deve marcar o método como public** quando o implementa:

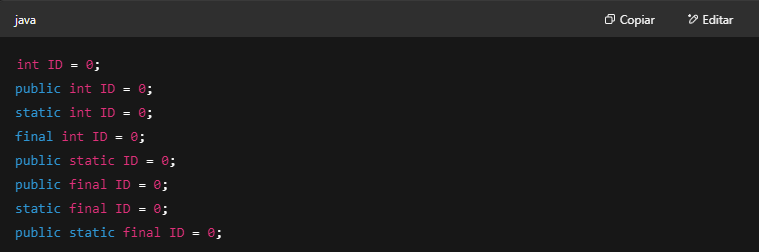


Campos declarados em uma interface são por padrão PUBLIC, STATIC e FINAL. Assim como os métodos, o compilador os tratará como tais mesmo que você não os especifique.

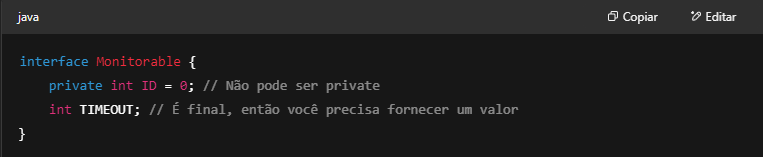
Isso significa que os campos são **CONSTANTES** ao invés de **VARIÁVEIS**:



Isso também significa que as seguintes declarações são todas equivalentes dentro de uma interface:



Portanto, cuidado com declarações que **não vão compilar**, como estas:

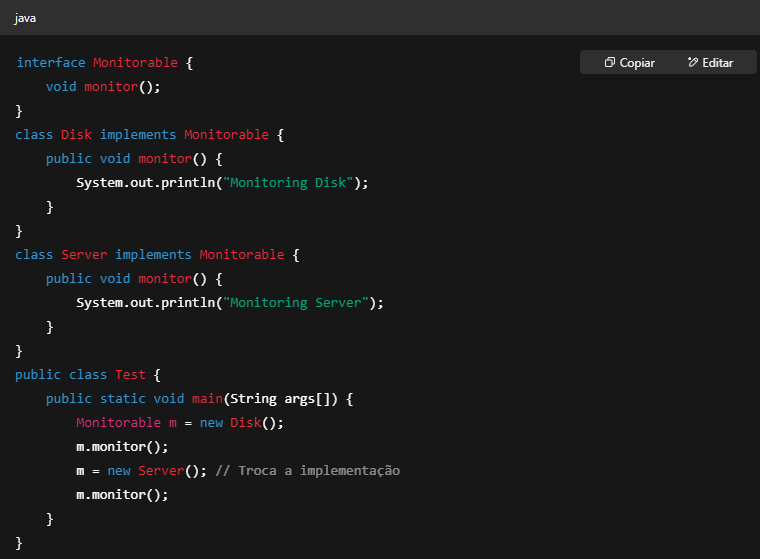


**Há duas regras quanto à herança e interfaces:**

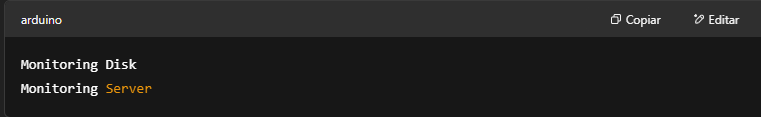
1. Uma **classe** pode **implementar** (não estender) **qualquer número de interfaces**.
2. Uma **interface** pode **estender qualquer número de interfaces**, mas **não pode estender uma classe**.

**Implementar uma interface é um tipo de herança.**

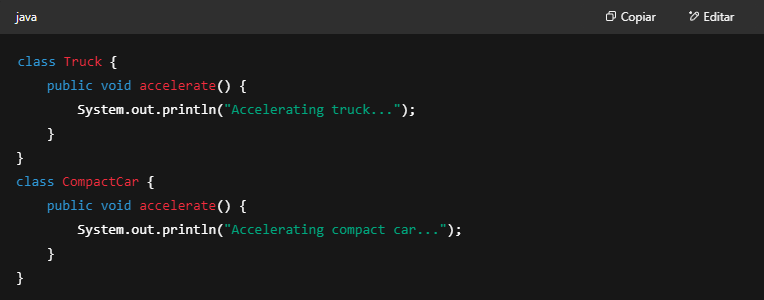
Quando uma classe implementa uma interface, podemos usá-la assim para tirar proveito do **polimorfismo**:



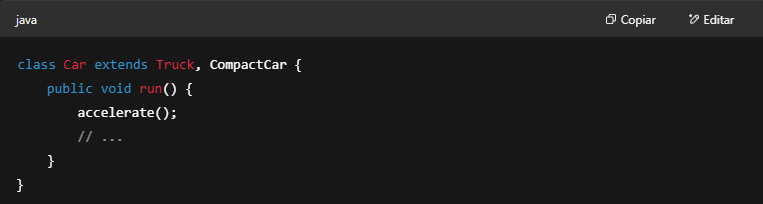
A saída será:



Uma classe **não pode estender** mais de uma classe, mas **pode implementar** mais de uma interface. Você pode entender o motivo com um exemplo. Considere:



Se fosse possível herdar de múltiplas classes:



Qual método accelerate() o Java escolheria?

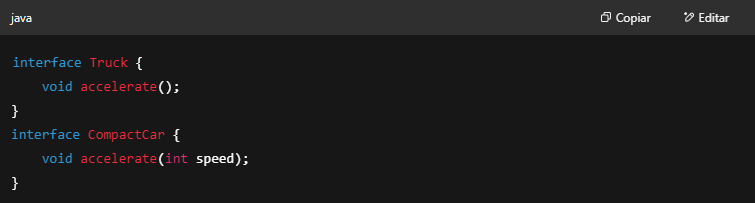
Este é um problema que os projetistas do Java decidiram evitar **não permitindo herança múltipla**.

Mas e se estivermos usando interfaces?

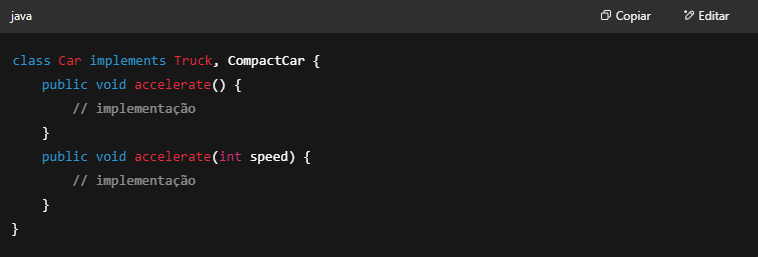


Como interfaces **não fornecem uma implementação**, há apenas uma (a da classe Car), então **não há conflito** e o problema é evitado completamente!

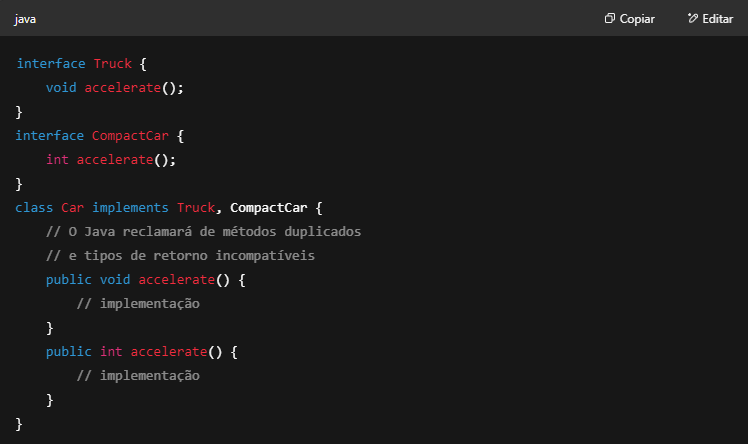
E se os métodos tiverem o mesmo nome mas parâmetros diferentes?



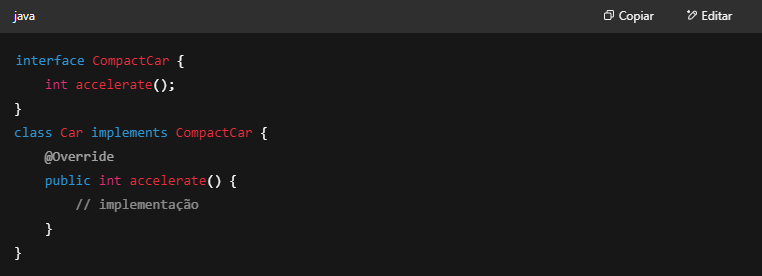
Eles são considerados **dois métodos diferentes** porque a assinatura do método é diferente. A classe implementadora terá que implementar **ambas** as versões:



Contudo, quando os métodos só diferem no tipo de retorno, como o tipo de retorno **não faz parte da assinatura do método**, o compilador do Java gerará um erro:



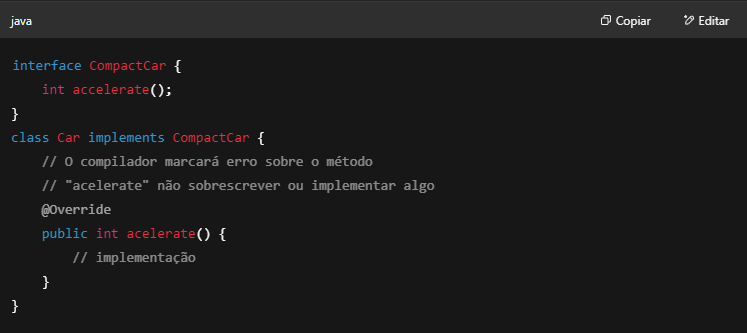
Opcionalmente, para tornar as coisas mais claras, podemos usar a anotação @Override:



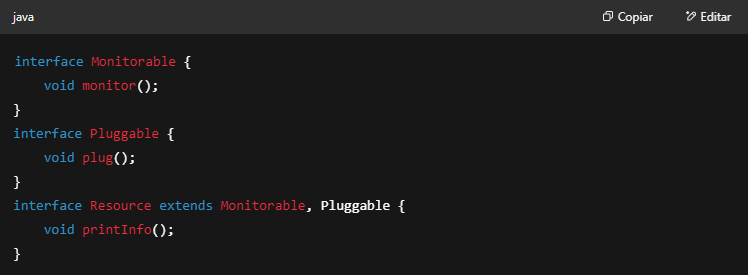
A anotação @Override indica que um método **sobrescreve** uma declaração de método em um supertipo, seja em uma interface ou em uma classe-pai.

Se o método anotado **não sobrescrever ou implementar corretamente** o método, o compilador gerará um erro.

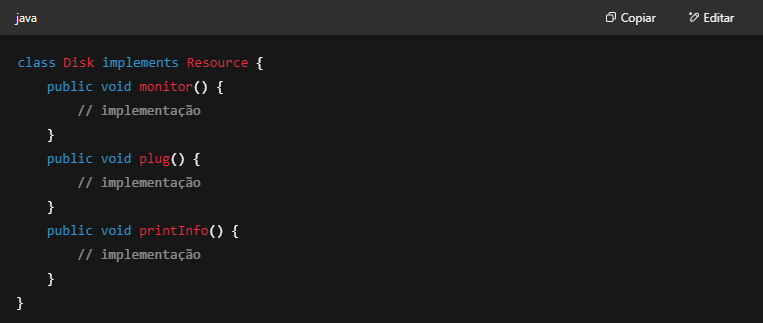
Essa é a única função da anotação. É útil em casos como quando há um erro **não tão óbvio** causado por um erro de digitação:



Por fim, uma interface **pode estender apenas outras interfaces**.

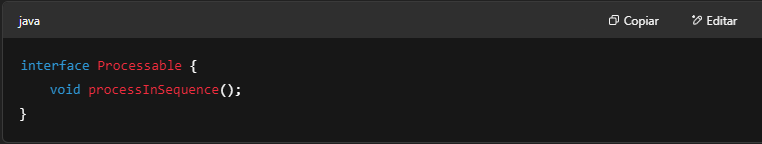


Uma classe **não abstrata** que implementa Resource deve implementar **todos os métodos** das três interfaces:

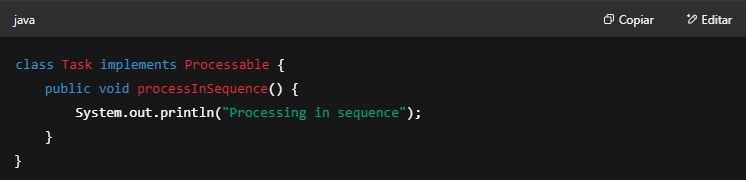


**O que há de novo no Java 8?**

Suponha que temos uma interface como esta:



E uma implementação:

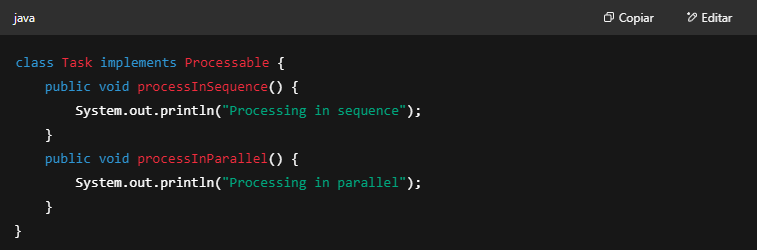


Sabemos que quando uma classe implementa uma interface, **a menos que a classe seja marcada como abstract**, ela deve implementar **TODOS** os métodos dessa interface.

Então se adicionarmos outro método a Processable, por exemplo:



Teremos que atualizar a classe para evitar um erro de compilação:



Isso foi fácil. Mas pense no seguinte:

* E se tivermos centenas de classes implementando Processable?
* E se não pudermos atualizar ou não tivermos acesso ao código por algum motivo?
* E se o novo método **não for necessário** ou **não fizer sentido** para algumas implementações?

Esses são problemas reais, às vezes difíceis de resolver.

Contudo, o **Java 8** nos dá os **métodos default**. Não precisamos fornecer implementações para eles porque **são métodos não abstratos**.

Em outras palavras, interfaces agora permitem métodos **com CORPO**. E isso **não é tão simples quanto parece**.

**Métodos default**

A principal razão para adicionar métodos default às interfaces foi **permitir a evolução das interfaces**, adicionando nova funcionalidade e, ao mesmo tempo, **garantindo compatibilidade** com o código escrito em versões anteriores.

Existem dois efeitos colaterais dignos de nota:

1. Agora podemos projetar **métodos opcionais**. Podemos ter métodos com funcionalidade limitada ou padrão, e as classes que implementam as interfaces podem decidir se mantêm essa funcionalidade ou fornecem outra.
2. Podemos ter **métodos utilitários diretamente na interface**. Métodos que obtêm ou criam recursos, por exemplo, feitos apenas por conveniência e possivelmente implementados em termos de métodos não-default da interface.

Contudo, agora que interfaces podem fornecer comportamento, a diferença entre elas e **classes abstratas** não é muito clara em alguns casos. Ainda assim, há duas diferenças significativas:

* Uma **classe** pode estender **somente UMA** classe abstrata, mas pode **implementar MÚLTIPLAS interfaces**.
* Uma classe abstrata pode ter estado por meio de **variáveis de instância (campos)**. Uma **interface NÃO PODE**.

**Definindo um método default**

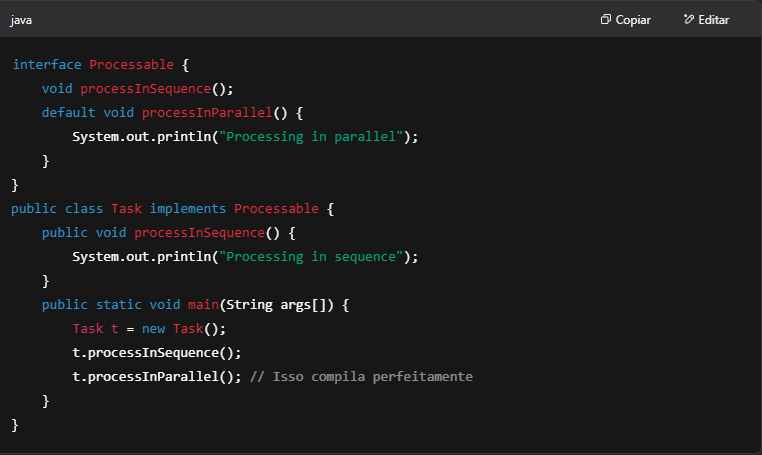


Uma interface pode ter **qualquer número** de métodos abstratos e métodos default.

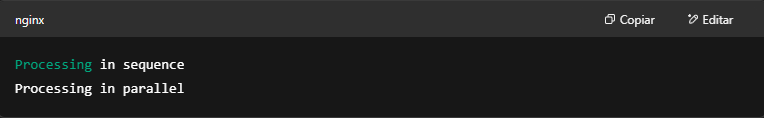
Todos os métodos com a palavra-chave default devem ter um **corpo**.

Métodos default são **implicitamente públicos**, como qualquer outro método de uma interface.

Ao tornar processInParallel() um método default, a classe que o implementa **o herda automaticamente**. Veja o exemplo completo:



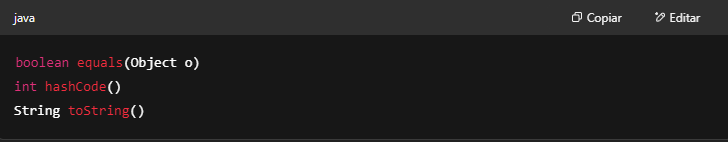
A saída será:



Este é o cenário mais simples, onde a classe implementadora **herda o método default**.

Antes de apresentar cenários mais complexos, vejamos **quais são as restrições** ao usar métodos default.

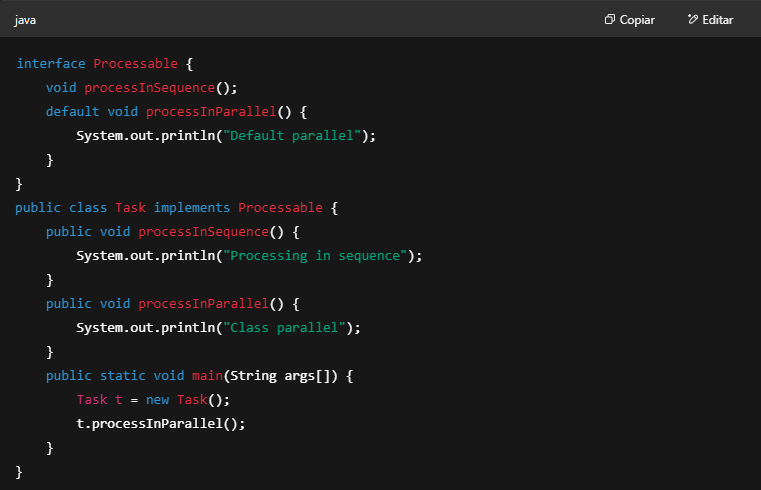
**Restrições dos métodos default**

* Métodos default **não podem ser final**.  
  Se um método é final, não pode ser sobrescrito pelas classes implementadoras, o que contraria o objetivo principal dos métodos default.
* Métodos default **não podem ser synchronized**.  
  Essa foi uma decisão deliberada dos projetistas da linguagem. Se um método fosse synchronized na interface, significaria que todas as classes que o implementassem herdariam esse comportamento. Mas essa decisão deve pertencer à implementação; **a interface não tem base razoável para definir a política de sincronização**.
* Métodos default são sempre public.  
  Como qualquer outro método de uma interface. Diferente de uma classe abstrata, onde você pode escolher a visibilidade.
* Você **não pode ter métodos default para métodos da classe Object**.  
  Uma interface **não pode fornecer implementações default para**:

Se uma interface contiver métodos com essas assinaturas, o **compilador lançará um erro**.  
O motivo é que esses métodos dizem respeito ao **estado do objeto**. Como **interfaces não têm estado**, esses métodos devem estar nas **classes implementadoras**.

**Classe sobrescreve método default**

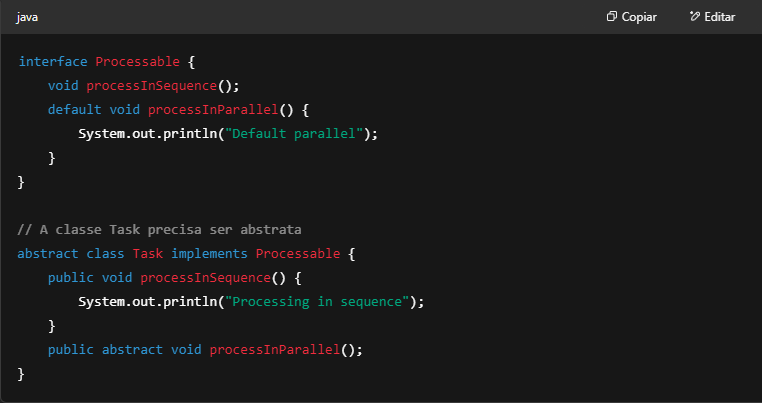
As **classes sempre VENCEM** as interfaces.  
Se uma classe sobrescreve um método default, o **método da classe será o usado**. Por exemplo:



A saída será:



Isso é verdade **mesmo que a classe redefina o método default como abstract**:

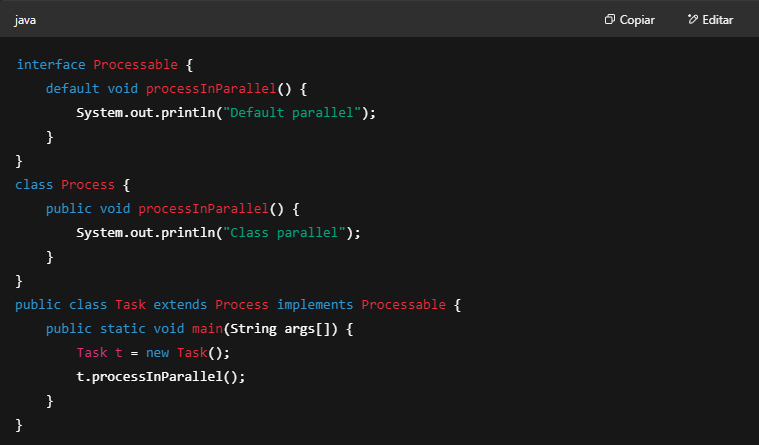


Se por algum motivo, você precisar chamar a **implementação default** do método, pode fazer isso com o **nome da interface seguido de super**:

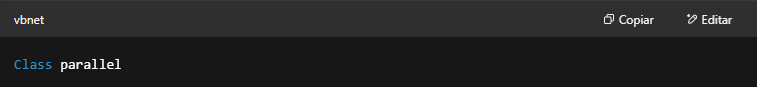


Isso **só funciona com métodos default**.  
Tentar chamar um método **não-default** dessa forma resulta em erro de compilação.  
Além disso, super deve ser usado com uma interface **diretamente herdada** pela classe.

Outro cenário relacionado a essa regra é quando **um método herdado de uma classe** sobrescreve um **método default** de uma interface:



A saída é:

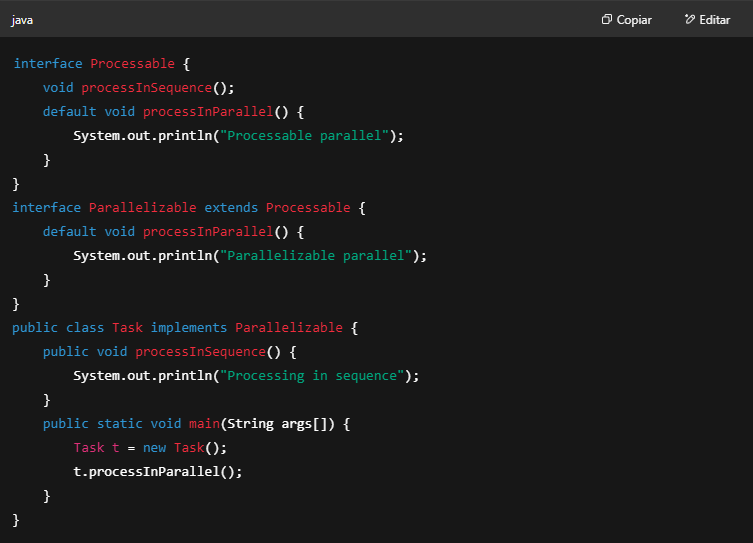


O método processInParallel() retorna a string "Class parallel" porque a classe Task herda esse método da classe Process, que **sobrescreve** o método default de mesmo nome da interface Processable.

**Herança de interface com métodos default**

**Interfaces mais específicas sempre VENCEM** interfaces menos específicas.

Os métodos default das interfaces **mais profundas** na hierarquia de herança serão usados. Por exemplo:

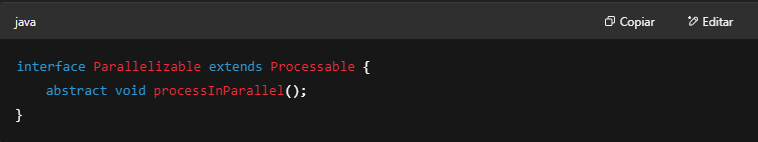


A saída será:

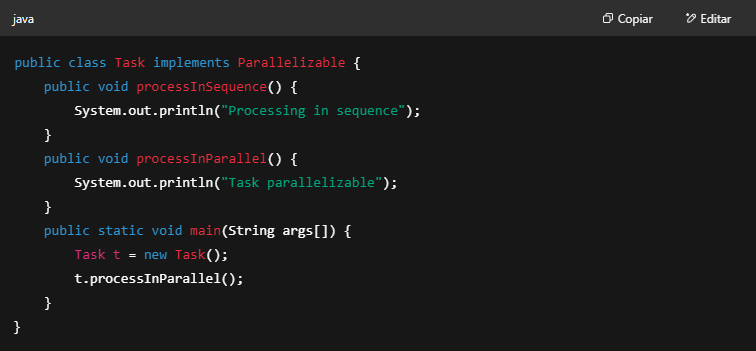


A interface Parallelizable herda o método default processInParallel(), mas como ela **o redefine**, é a sua versão que será usada pela classe Task.

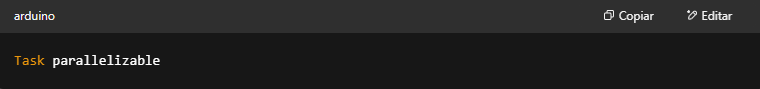
Se Parallelizable definisse processInParallel() como abstract:



Então Task **teria que implementar o método**, para não ser uma classe abstrata:



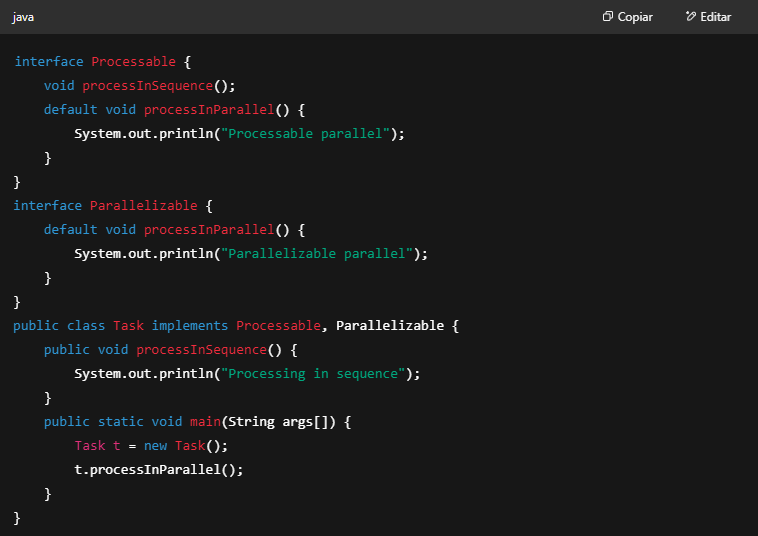
A saída seria:



**Herança múltipla de interfaces com métodos default**

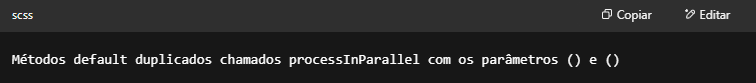
As classes podem implementar **múltiplas interfaces**.  
O que acontece quando **duas interfaces têm o mesmo método default**?  
Qual delas a classe implementadora escolhe?

Considere o exemplo:



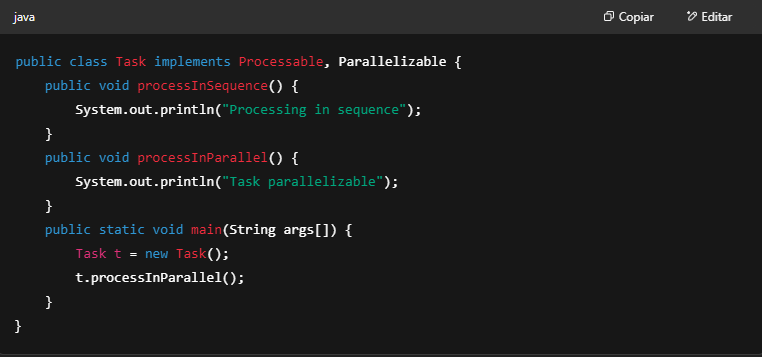
O resultado será:

**Erro de compilação:**



O compilador **não sabe qual versão escolher**, então gera um erro.

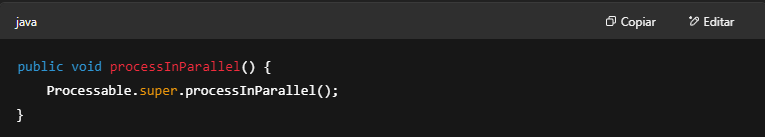
Neste caso, Task **deve fornecer sua própria implementação** (seguindo a regra de que a interface ou classe mais específica vence), sobrescrevendo os métodos default da interface e resolvendo o conflito:



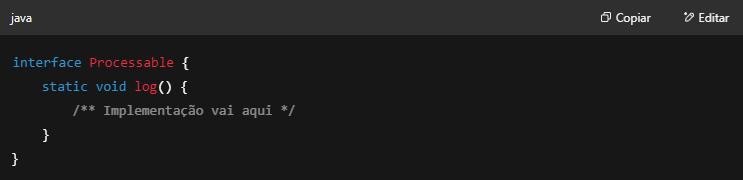
A saída agora será:



Claro, também podemos chamar uma implementação default diretamente com:



**Definindo um método static**



Métodos static em interfaces são definidos **exatamente como em classes**, com a palavra-chave static.

* Métodos static são **implicitamente públicos**, como qualquer outro método da interface.
* Uma interface pode conter **qualquer número** de métodos static.

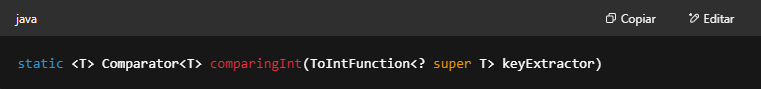
**Métodos static**

Sempre que nos referimos a algo static, estamos nos referindo a algo que pertence a uma **classe** (ou interface), e **não a uma instância ou objeto específico**.

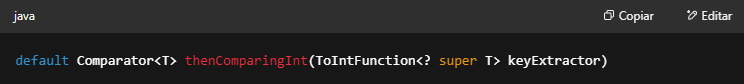
Métodos static em interfaces seguem o mesmo conceito: **pertencem à interface onde são declarados**.

Eles foram adicionados para **ajudar os métodos default** e para **organizar melhor métodos utilitários**, porque geralmente esses métodos eram definidos em outra classe (como java.util.Collections), em vez de onde **naturalmente pertencem**.

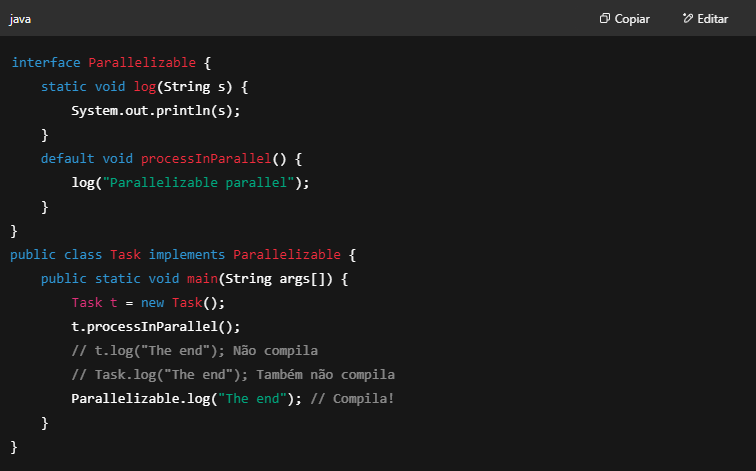
Por exemplo, a interface java.util.Comparator define o método estático:



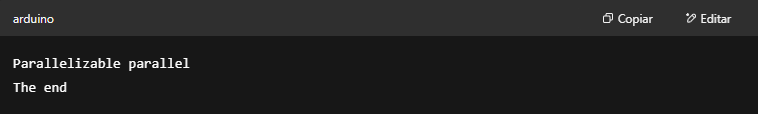
Usado por um método default:



**Métodos static de interface não são herdados**.  
Você deve prefixar o método com o nome da interface:



Saída:



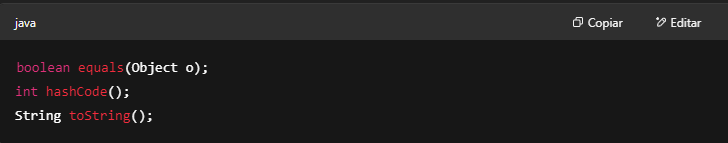
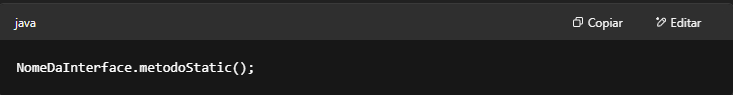
**Pontos-chave**

* Uma **interface** é um tipo de dado que apenas define métodos (**abstratos**) que uma classe deve implementar.
* Uma interface é definida com a palavra-chave interface.  
  Se uma classe quiser implementar uma interface, ela deve especificar isso com a palavra-chave implements.
* Uma interface tem acessibilidade public ou padrão (default) e é **abstrata por padrão**.
* Os métodos definidos em uma interface são, por padrão, public e abstract.  
  O compilador os tratará assim mesmo que você não os declare explicitamente.
* Os campos declarados em uma interface são, por padrão, public, static e final.  
  Assim como os métodos, o compilador os tratará assim mesmo que não sejam especificados.

Isso significa que os campos são **constantes**, não variáveis.

* Uma classe pode **implementar** (não estender) **qualquer número de interfaces**.
* Uma interface pode **estender qualquer número de interfaces**, mas **não pode estender uma classe**.
* A anotação @Override indica que um método sobrescreve uma declaração de método em um supertipo (interface ou classe-pai).

Se o método anotado não sobrescrever ou implementar corretamente o método, o compilador gerará um erro.

* O Java 8 introduziu **métodos default** em interfaces para suportar a evolução das interfaces, adicionando nova funcionalidade e, ao mesmo tempo, mantendo a compatibilidade com código legado.
* Métodos default são marcados com a palavra-chave default, e devem ter um corpo.  
  As classes que os implementam podem usar ou sobrescrever esses métodos.
* Métodos default são sempre public, mas **não são static**.  
  Eles **não podem ser synchronized nem final**.
* Você **não pode definir métodos default com a mesma assinatura dos métodos da classe Object**
* Em uma hierarquia de herança, o método **mais específico é o que será chamado**.
* Exemplo: se uma classe sobrescreve um método default, o método da classe será usado.
* Se uma interface sobrescreve um método default herdado de uma superinterface, o método da subinterface será usado.
* Se uma classe implementa **duas interfaces diferentes com o mesmo método default (mesma assinatura)**, ela **deve sobrescrever o método**. Caso contrário, ocorre um erro de compilação.
* Se você quiser chamar o método default de uma interface a partir da classe implementadora (ou de uma subinterface), faça assim:
* 
* O Java 8 também introduziu métodos static em interfaces para que elas pudessem conter métodos utilitários.
* Métodos static são marcados com a palavra-chave static, e também devem ter corpo.
* Métodos static em interfaces têm o mesmo comportamento que os de classes: **não são herdados**.
* Se quiser chamar um método static de uma interface, faça assim:

**Autoavaliação (Self Test)**

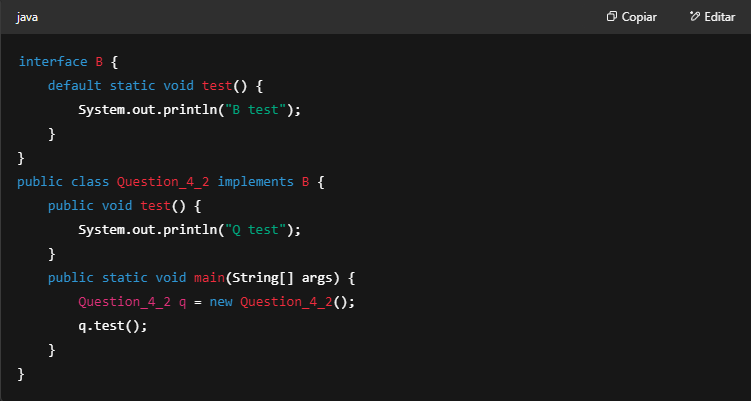
**1. Dado:**



**Qual é o resultado?**

A. 0  
B. 1  
C. Falha de compilação  
D. Uma exceção ocorre em tempo de execução

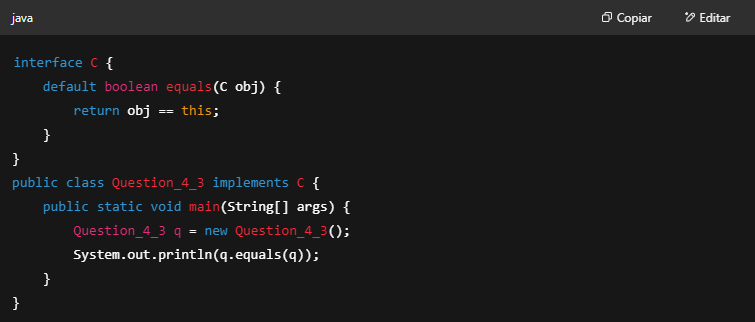
**2. Dado:**



**Qual é o resultado?**

A. B test  
B. Q test  
C. Falha de compilação  
D. Uma exceção ocorre em tempo de execução

**3. Dado:**



**Qual é o resultado?**

A. true  
B. false  
C. Falha de compilação  
D. Uma exceção ocorre em tempo de execução

**4. Dado:**



**Qual é o resultado?**

A. D  
B. E  
C. D e depois E  
D. Falha de compilação  
E. Uma exceção ocorre em tempo de execução

**5. Dado:**



**Qual é o resultado?**

A. F test  
B. Q test  
C. Falha de compilação  
D. Uma exceção ocorre em tempo de execução

**6. Dado:**



**Qual é o resultado?**

A. G - Do It  
B. Do It  
C. Falha de compilação  
D. Uma exceção ocorre em tempo de execução