**Capítulo QUATORZE**

Classe Optional

**Objetivos do Exame:**  
Desenvolver código que use a classe Optional.

**O problema com null**

A maioria das linguagens de programação tem um tipo de dado para representar a ausência de um valor, e ele é conhecido por muitos nomes:

**NULL**, **nil**, **None**, **Nothing**

O tipo *null* foi introduzido em ALGOL W por Tony Hoare em 1965, e é considerado um dos piores erros da ciência da computação. Nas próprias palavras de Tony Hoare:

Eu o chamo de meu erro de um bilhão de dólares. Foi a invenção da referência nula em 1965. Naquele tempo, eu estava projetando o primeiro sistema de tipos abrangente para referências em uma linguagem orientada a objetos (ALGOL W). Meu objetivo era garantir que todo uso de referências fosse absolutamente seguro, com verificação realizada automaticamente pelo compilador. Mas eu não resisti à tentação de incluir uma referência nula, simplesmente porque era muito fácil de implementar. Isso levou a inúmeros erros, vulnerabilidades e falhas de sistema, que provavelmente causaram um bilhão de dólares em dor e prejuízo nos últimos quarenta anos.  
– Tony Hoare

Ainda assim, alguns podem estar se perguntando, qual é o problema com *null*?

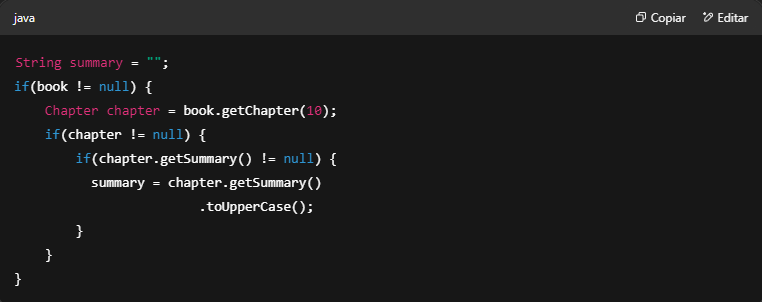
Bem, se você está um pouco preocupado com os problemas que esse código pode causar, você já sabe a resposta:



O problema com esse código é que, se qualquer um desses métodos retornar uma referência *null* (por exemplo, se o livro não tiver um décimo capítulo), uma **NullPointerException** (a exceção mais comum em Java) será lançada em tempo de execução, interrompendo o programa.

O que podemos fazer para evitar essa exceção?

Talvez, a maneira mais fácil seja verificar *null*. Aqui está uma forma de fazer isso:



Você não sabe se algum objeto nessa hierarquia pode ser *null*, então você verifica cada objeto por esse valor. Obviamente, essa não é a melhor solução; ela não é muito prática e prejudica a legibilidade.

Pode haver outro problema. Verificar *null* é realmente desejável? Quero dizer, e se esses objetos nunca deveriam ser *null*? Ao verificar *null*, estaremos escondendo o erro e não lidando com ele.

Claro, isso também é uma questão de design. Por exemplo, se um capítulo ainda não tem um resumo, o que seria melhor usar como valor padrão? Uma string vazia ou *null*?

Para lidar com esse problema, o Java 8 introduziu a classe **java.util.Optional<T>**.

**A classe Optional**

A função desta classe é **ENCAPSULAR** um valor opcional, um objeto que pode ser *null*.

Usando o exemplo anterior, se sabemos que nem todos os capítulos têm um resumo, em vez de modelar a classe assim:



Podemos usar a classe Optional:

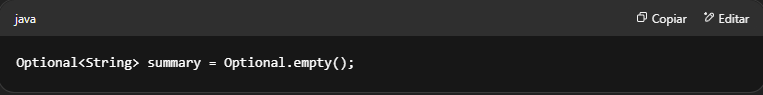


Assim, se houver um valor, a classe **Optional** apenas o encapsula. Caso contrário, um valor vazio é representado pelo método Optional.empty(), que retorna uma instância singleton de Optional.

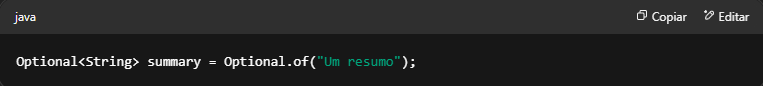
Usando esta classe em vez de *null*, primeiro, declaramos explicitamente que o atributo *summary* é opcional. Depois, podemos evitar **NullPointerExceptions** e ainda contar com os métodos úteis de Optional, que veremos a seguir.

**Criando instâncias de Optional**

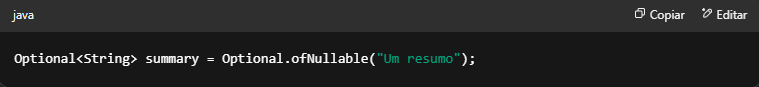
Para obter um objeto Optional vazio, use:



Se você tem certeza de que um objeto **não** é *null*, pode encapsulá-lo em um Optional assim:



Uma **NullPointerException** será lançada se o objeto for *null*. No entanto, você pode usar:

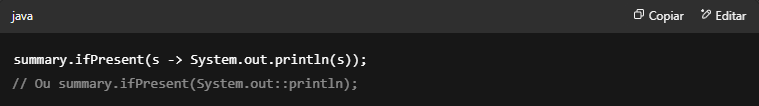


Isso retorna uma instância Optional com o valor especificado, se ele não for *null*. Caso contrário, retorna um Optional vazio.

**Verificando se Optional contém valor**

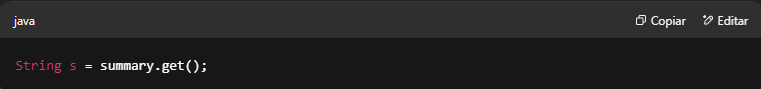


Ou, de forma mais funcional:



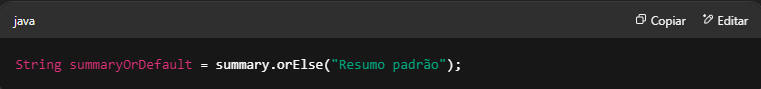
O método ifPresent() recebe um Consumer<T> como argumento, que é executado apenas se o Optional contiver um valor.

**Obtendo o valor de um Optional**

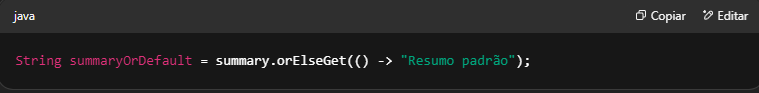


Porém, esse método lançará uma **java.util.NoSuchElementException** se o Optional não contiver valor. Portanto, é melhor usar o método ifPresent().

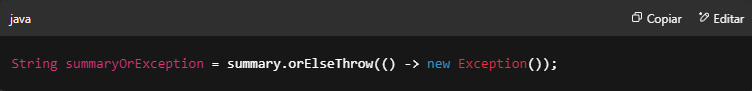
**Métodos alternativos para valores padrão ou exceção**



orElse() retorna o argumento (que deve ser do tipo T, neste caso uma String) se o Optional estiver vazio. Caso contrário, retorna o valor encapsulado.



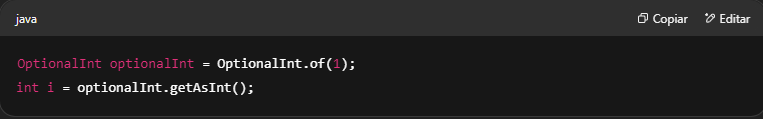
orElseGet() recebe um Supplier<? extends T> que retorna um valor quando o Optional está vazio. Caso contrário, retorna o valor encapsulado.



orElseThrow() recebe um Supplier<? extends X>, onde X é o tipo da exceção a ser lançada quando o Optional estiver vazio. Caso contrário, retorna o valor encapsulado.

**Optional para primitivos**

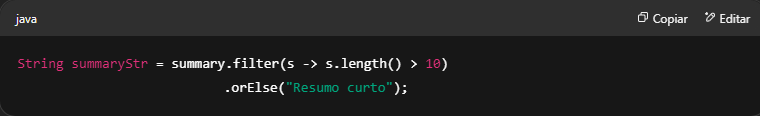
Como *streams*, há versões de Optional para tipos primitivos: OptionalInt, OptionalLong e OptionalDouble, então você pode usar OptionalInt em vez de Optional<Integer>:



No entanto, o uso dessas versões primitivas não é encorajado, principalmente porque elas não têm três métodos úteis da classe Optional: filter(), map() e flatMap(). E como Optional contém apenas um valor, o custo de *boxing/unboxing* de um primitivo não é significativo.

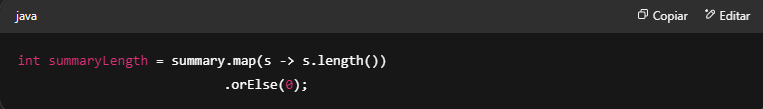
**filter()**

Retorna o Optional se um valor estiver presente e corresponder ao predicado fornecido. Caso contrário, retorna um Optional vazio:



**map()**

Geralmente usado para transformar de um tipo para outro:



**flatMap()**

Semelhante ao map(), mas recebe um Function<? super T, Optional<U>>. Se o valor estiver presente, retorna o Optional resultante da função. Caso contrário, retorna um Optional vazio.

No Capítulo 17, revisaremos com mais detalhes os métodos map() e flatMap() e como são usados com *streams*.

**Pontos-chave**

* A classe **java.util.Optional<T>** encapsula um valor opcional, ou seja, um objeto que pode ser *null*.
* Um valor vazio é representado pelo método Optional.empty().
* Você pode encapsular um objeto com Optional.of(), mas ele lançará uma **NullPointerException** se o objeto for *null*.
* O método ofNullable() retorna um Optional com o valor especificado se ele não for *null*, caso contrário, retorna vazio.
* get() retorna o valor, mas pode lançar **NoSuchElementException**. É preferível usar ifPresent().
* orElse(), orElseGet() e orElseThrow() são alternativas para fornecer um valor ou lançar uma exceção quando vazio.

**Autoavaliação**

**1. Dado:**



Qual é o resultado?

A. 1  
B. Nada é impresso  
C. Falha de compilação  
D. Uma exceção ocorre em tempo de execução

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

A. O método Optional.isPresent() recebe um Consumer<T> como argumento que é executado apenas se o Optional contiver um valor.  
B. O método Optional.of() pode criar um Optional vazio.  
C. O método Optional.of() pode lançar uma NullPointerException.  
D. O método Optional.ifPresent() recebe uma Function<T,U> como argumento.

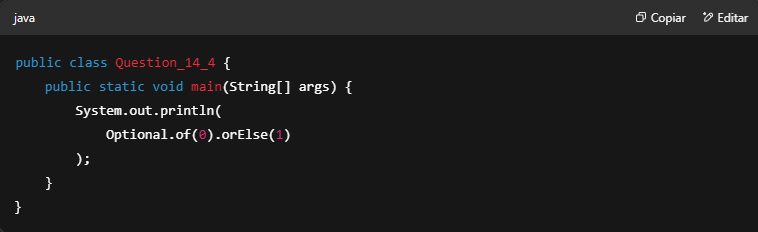
**3. Dado:**



Qual é o resultado?

A. a  
B. Optional.empty  
C. Falha de compilação  
D. Uma exceção ocorre em tempo de execução

**4. Dado:**



Qual é o resultado?

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