**Parte DOIS**

**Genéricos e Coleções**

**Capítulo SEIS**

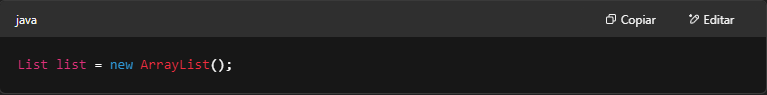
**Genéricos**

**Objetivos do Exame**

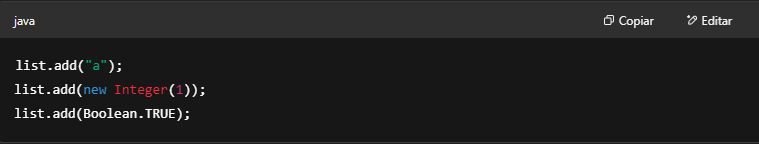
Criar e usar uma classe genérica.

**Genéricos**

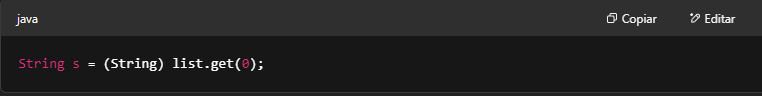
Sem genéricos, você pode declarar uma List assim:



Como uma List, por padrão, aceita objetos de qualquer tipo, você pode adicionar elementos de tipos diferentes a ela:

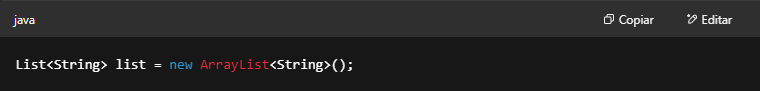


E obter valores assim:

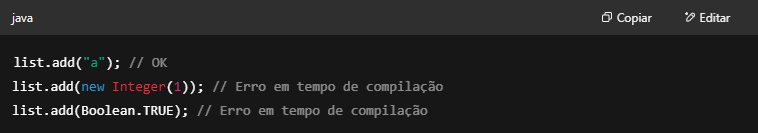


Isso pode levar a erros feios em tempo de execução e mais complexidade. Por causa disso, os genéricos foram adicionados no Java 5 como um mecanismo de verificação de tipo.

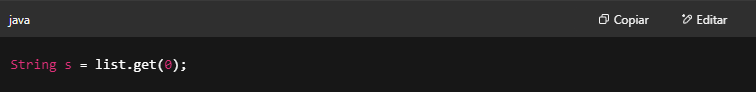
Um genérico é um tipo declarado entre sinais de menor e maior (<>), após o nome da classe. Por exemplo:



Ao adicionar o tipo genérico à List, estamos dizendo ao COMPILADOR para verificar se apenas valores do tipo String podem ser adicionados à lista:



Como agora só temos valores de um tipo, podemos obter elementos com segurança sem fazer casting:



É importante enfatizar que genéricos são uma coisa do compilador. Em tempo de execução, o Java **não sabe** sobre genéricos.

Nos bastidores, o compilador insere todas as verificações e castings para você, mas em tempo de execução, um tipo genérico é visto pelo Java como um tipo java.lang.Object.

Em outras palavras, o compilador verifica se você está trabalhando com o tipo correto e, então, gera código com o tipo Object.

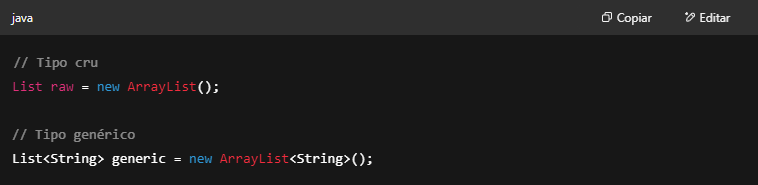
Esse processo de substituir todas as referências a tipos genéricos por Object é chamado de **apagamento de tipo** (*type erasure*).

Por causa disso, em tempo de execução, List<String> e List<Integer> são o mesmo, porque a informação de tipo foi apagada pelo compilador (elas são vistas apenas como List).

Genéricos funcionam apenas com objetos. Algo como o seguinte **não compilará**:

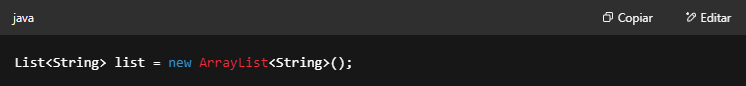


Finalmente, uma classe que aceita genéricos mas é declarada sem um tipo é dita como usando um **tipo cru** (*raw type*):

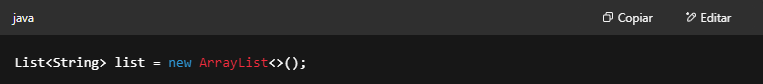


**Operador Diamante**

Observe novamente esta declaração genérica:

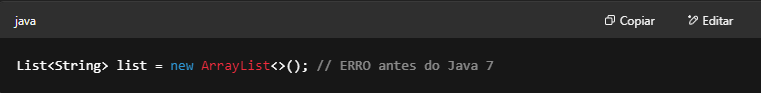


É um pouco repetitiva, não é? Podemos simplificar isso com o operador **diamante**:



O compilador entende que o lado direito precisa ser um ArrayList<String>, então ele assume o tipo.

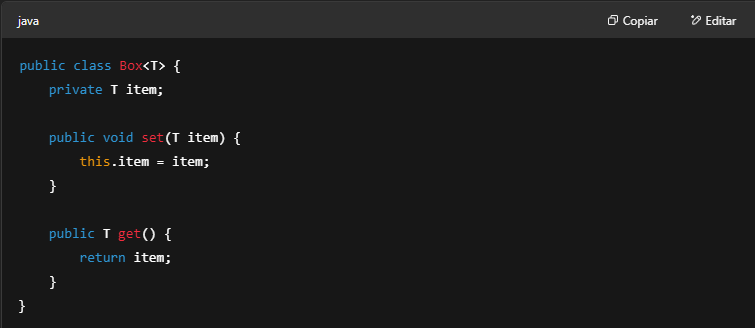
No entanto, **o operador diamante funciona apenas a partir do Java 7**.  
Antes disso, isso não compila:



Você ainda pode usar o operador diamante com classes genéricas que **você mesmo criar**, como veremos agora.

**Criando Suas Próprias Classes Genéricas**

Criar uma classe genérica é fácil.  
Você adiciona um **tipo de parâmetro entre < >** após o nome da classe.  
Por convenção, **usamos letras maiúsculas** para os parâmetros de tipo.  
O mais comum é T, de *Type*, mas outros como E, K, V também são usados frequentemente.



Agora podemos criar objetos Box que guardam tipos específicos:

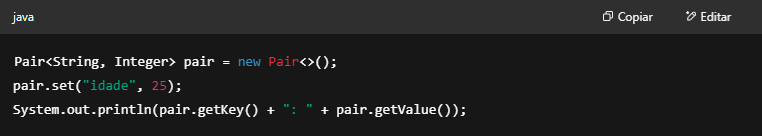


Note que a classe Box só precisa ser escrita **uma vez** — você pode instanciá-la com qualquer tipo depois.

Também é possível ter múltiplos parâmetros de tipo:



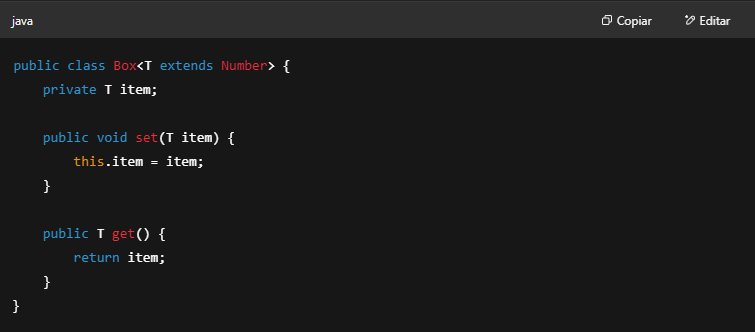
E usá-la assim:



**Limites de Tipo (Bounded Types)**

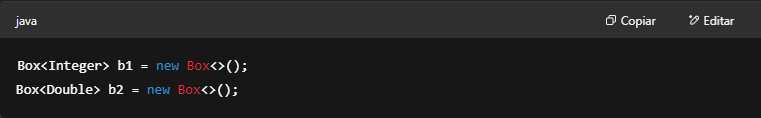
Suponha que queremos restringir os tipos genéricos para aceitar apenas **subtipos de uma determinada classe**, como Number.

Podemos fazer isso com a palavra-chave extends:

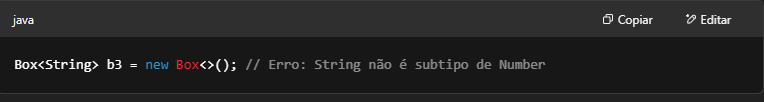


Agora a classe Box **só aceitará tipos que sejam subclasses de Number**, como Integer, Double, Float, etc.

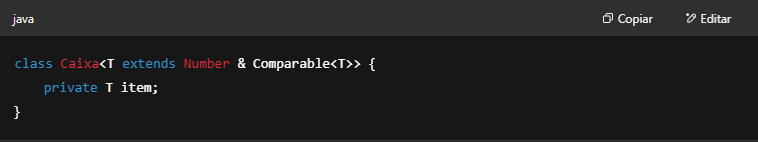
Por exemplo:



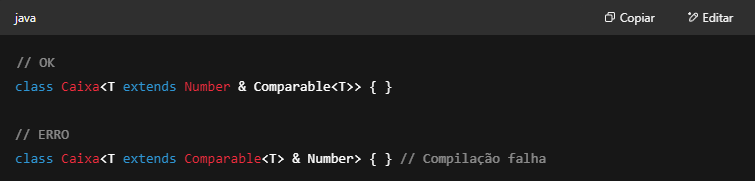
Mas se tentarmos usar um tipo como String, teremos erro de compilação:



Se você quiser impor **múltiplas restrições**, pode usar a sintaxe com **interface + classe**:



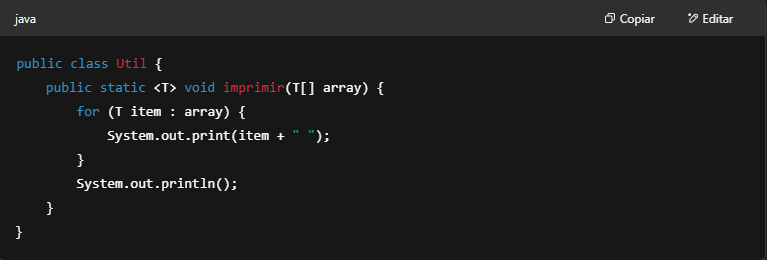
A ordem importa!  
Se você usar tanto **uma classe quanto interfaces**, a classe **deve vir primeiro**:



**Métodos Genéricos**

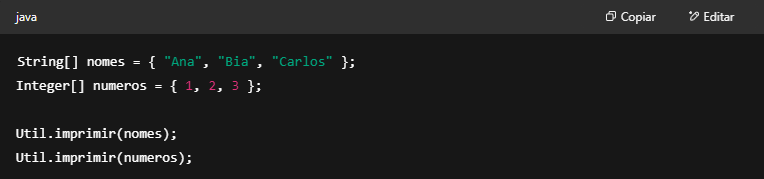
Assim como podemos criar classes genéricas, também podemos criar **métodos genéricos**.

A principal diferença é que, no caso de métodos, o parâmetro de tipo é declarado **antes do tipo de retorno**:



Observe que <T> aparece **antes de void**, e que o método é static. Isso não é obrigatório, mas é comum para métodos utilitários.

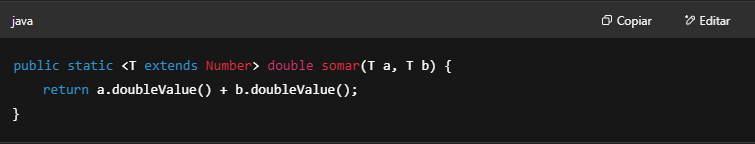
Podemos chamar o método assim:



Saída:



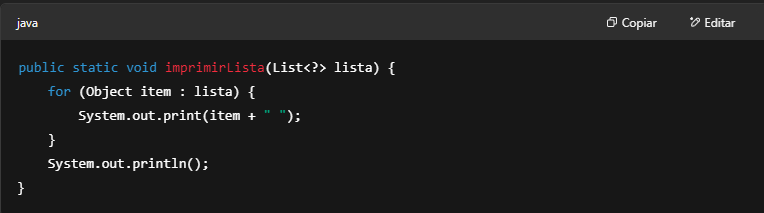
Você pode também usar limites nos métodos genéricos:



Aqui, T deve ser um subtipo de Number, e usamos o método doubleValue() disponível na classe Number.

**Curingas (?) e Tipos Genéricos Coringa**

Em alguns casos, você quer que um método **aceite diferentes tipos genéricos**, mas **não se importe com o tipo exato**.  
Para isso, usamos o **curinga ?**.



Esse método aceita uma List<?>, ou seja, **qualquer lista de qualquer tipo**.  
O curinga ? significa “algum tipo desconhecido”.

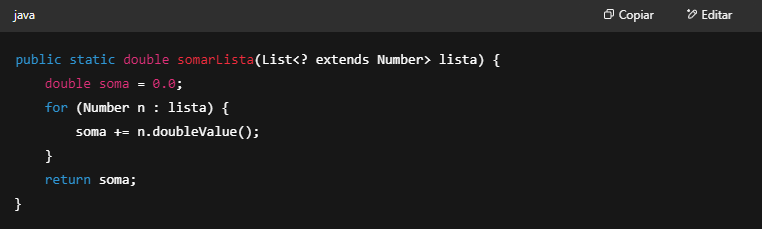
Você pode chamá-lo com diferentes listas:



Contudo, quando usamos <?>, **não podemos adicionar novos elementos à lista** dentro do método (exceto null).  
Isso acontece porque o compilador **não sabe qual é o tipo da lista**, então não pode garantir a segurança de tipo.

**Curinga com extends**

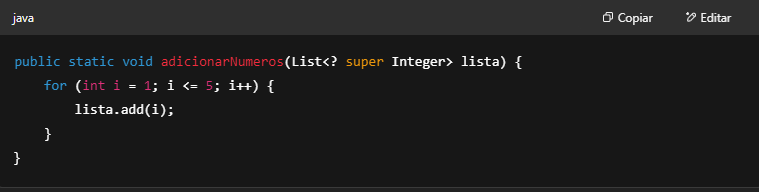
Às vezes, queremos aceitar **qualquer subtipo de um tipo específico**.  
Usamos <? extends T> para isso:



Isso permite que você passe List<Integer>, List<Double>, etc.  
Mas **não pode adicionar elementos à lista**, pelo mesmo motivo de antes — o compilador **não sabe o tipo exato** da lista.

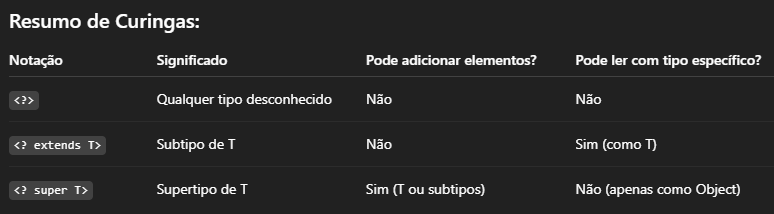
**Curinga com super**

Às vezes, queremos aceitar **qualquer supertipo** de um tipo específico.  
Usamos <? super T> para isso:



Nesse caso, você pode adicionar Integer à lista, porque sabemos que a lista aceita **Integer ou supertipos** (como Number ou Object).

Contudo, **ao recuperar os elementos**, você só pode tratá-los como Object, pois o compilador **não sabe o tipo exato**.

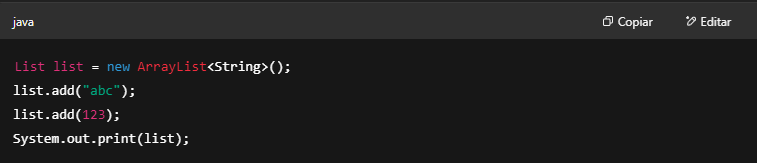


**Pontos-chave**

* Genéricos permitem **verificação de tipo em tempo de compilação**, evitando erros de ClassCastException em tempo de execução.
* A **sintaxe** para declarar um tipo genérico é usar **<T>** após o nome da classe ou antes do tipo de retorno no método.
* Você pode criar **classes genéricas** e **métodos genéricos**.
* Você pode restringir o tipo genérico com **extends**, o que permite apenas subtipos específicos.
* O compilador **apaga** as informações de tipo genérico durante a compilação. Esse processo é chamado de **type erasure**.
* Em tempo de execução, List<String> e List<Integer> são vistos como List.
* Você **não pode** usar tipos primitivos diretamente como argumentos genéricos (List<int> é inválido).
* O operador **diamante (<>)** permite omitir o tipo no lado direito da declaração — funciona a partir do Java 7.
* Os **curingas (?)** são usados quando o tipo exato **não é relevante**:
  + <?> significa **qualquer tipo desconhecido**
  + <? extends T> significa **qualquer subtipo de T**
  + <? super T> significa **qualquer supertipo de T**

**Autoavaliação (Self Test)**

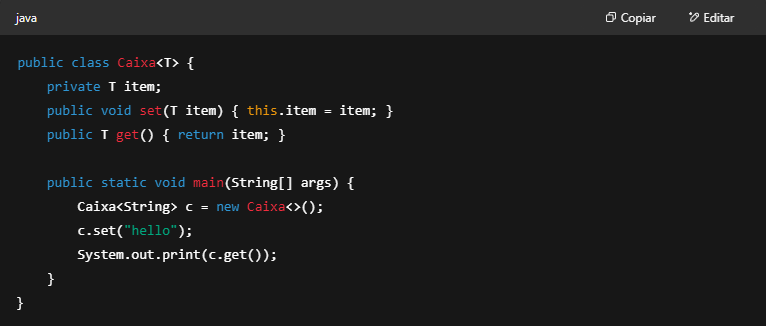
**1. Dado:**



**Qual é o resultado?**

A. [abc, 123]  
B. [abc]  
C. Erro de compilação  
D. Exceção em tempo de execução

**2. Dado:**



**Qual é o resultado?**

A. hello  
B. null  
C. Erro de compilação  
D. Exceção em tempo de execução

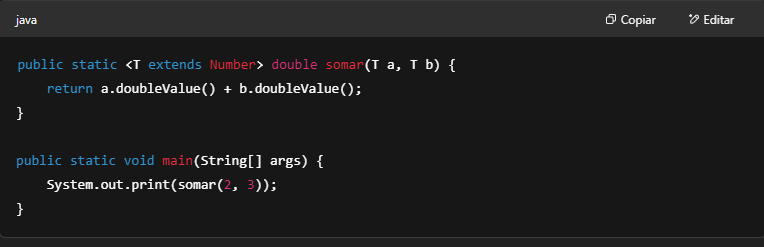
**3. Dado:**



**Qual é o resultado?**

A. 10  
B. 0  
C. Erro de compilação  
D. Exceção em tempo de execução

**4. Dado:**



**Qual é o resultado?**

A. 5  
B. 5.0  
C. Erro de compilação  
D. Exceção em tempo de execução

**5. Dado:**



**Qual é o resultado?**

A. Ana Bia  
B. [Ana, Bia]  
C. Erro de compilação  
D. Exceção em tempo de execução