



## Programação cliente com TypeScript

Prof. Sérgio Assunção Monteiro

### Descrição

Apresentação dos aspectos essenciais da linguagem de programação TypeScript.

### Propósito

A linguagem de programação TypeScript é uma extensão da linguagem JavaScript. Ambas têm o mesmo objetivo: desenvolver aplicações para Web. Como era de se esperar, o TypeScript oferece recursos mais avançados do que o JavaScript. Portanto, vamos conhecer esses recursos e explorá-los por meio de diversos exemplos.

### Preparação

Tudo o que você precisa para rodar os exemplos é de um computador com acesso à Internet e de um navegador instalado, como o Google Chrome, por exemplo. Vamos rodar os exemplos no seguinte endereço eletrônico: [typescriptrlang.org](https://typescriptrlang.org)

## Objetivos

## Módulo 1

## Conceitos gerais e Javascript

Reconhecer conceitos gerais e similaridades com o JavaScript.

## Módulo 2

## Funções com TypeScript

Analisar os tipos e narrowing.

## Módulo 3

## Tipos e narrowing

Descrever funções com TypeScript.

## Módulo 4

## Classes com TypeScript

Identificar classes com TypeScript.



## Introdução

Os sistemas computacionais fazem parte da nossa vida. Por meio deles, podemos acessar diversos tipos de serviços, dos mais básicos até os mais sofisticados. Muitos desses serviços

estão disponíveis na Web por sistemas que nos permitem realizar interações. Para desenvolver esses sistemas, utilizamos linguagens de programação para implementar estruturas lógicas que representam processos.

A compra de determinado produto em uma loja virtual exige que o usuário informe seus dados e avance em etapas até finalizar a compra (lógica do negócio). Além da própria lógica do processo, ainda há a questão da interação gráfica com o usuário. Esses dois tipos de desenvolvimento são chamados, respectivamente, de BackEnd e FrontEnd.

Uma das linguagens de programação que se popularizou bastante tanto com desenvolvimento BackEnd como FrontEnd foi o JavaScript, sendo a tecnologia mais importante para implementação de projetos Web.

No entanto, como era de se esperar, havia muitas questões que precisavam ser melhoradas no JavaScript e, para supri-las, foi criada a linguagem TypeScript, que possui os mesmos recursos de sua progenitora, mas a supera com o suporte para uso de tipos e programação orientada a objetos.

Dominar o TypeScript é ficar por dentro das tecnologias mais modernas para desenvolvimento Web e, claro, é mais uma forma de se destacar em um mercado com grandes demandas por profissionais qualificados.



## 1 - Conceitos gerais e Javascript

Ao final deste módulo, você será capaz de reconhecer conceitos gerais e similaridades com o JavaScript.

# Introdução ao TypeScript

TypeScript (TS) é uma linguagem de programação baseada na linguagem de programação JavaScript (JS). A principal característica dela é a possibilidade de utilizar tipos estáticos, classes e interfaces. Portanto, já conseguimos distinguir alguns aspectos fundamentais da linguagem, sendo elas:



Uma linguagem fortemente tipada;



Orientada a objetos;



Compilada.

O TS foi desenvolvido pela Microsoft como resultado do trabalho de Anders Hejlsberg – que projetou a linguagem de programação C#. Uma das definições muito comuns sobre o TS é que ele é um superconjunto tipado e compilado de JS que, na prática, é o JavaScript com recursos adicionais.

Em especial para os desenvolvedores de aplicações Web, utilizar o TS traz vantagens no sentido de construir aplicações mais tolerantes a falhas. Além disso, é um software de código aberto.

## Semelhanças entre JavaScript e TypeScript

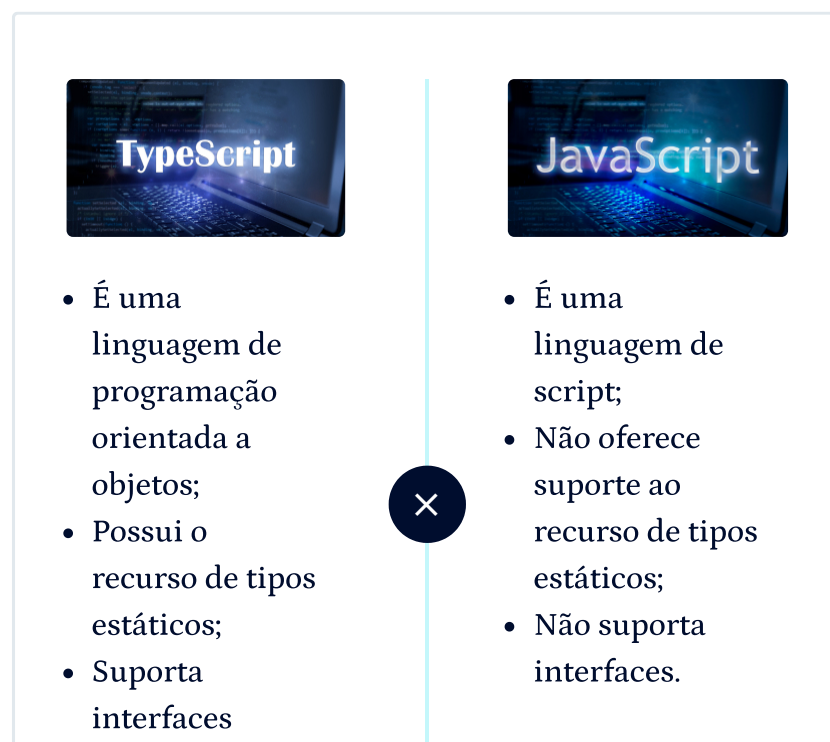
A origem do TypeScript está nas **limitações** do JavaScript, pois o JavaScript foi desenvolvido como uma linguagem de programação do lado do cliente que passou a ser usada como uma linguagem de programação do lado do servidor. No entanto, ficou evidente que, à

medida que o código ia crescendo, ele se tornava mais complexo e difícil de gerenciar. Isso impediu que o JavaScript tivesse sucesso como uma tecnologia do lado do servidor. Por causa dessas questões, houve a necessidade de criar o TypeScript. Ambas as linguagens, naturalmente, têm muitas semelhanças das quais podemos destacar (FREEMAN, 2019):

- O código de um projeto do TypeScript é convertido em código JavaScript simples, pois os browsers não podem interpretá-lo de forma nativa. Isso significa que o código escrito em TypeScript é compilado e convertido para JavaScript;
- Qualquer código escrito em JavaScript pode ser convertido em TypeScript alterando a extensão de .js (JavaScript) para .ts (TypeScript), o que significa que JavaScript é TypeScript. Qual é o porquê disso? Devemos lembrar que o TS é um superconjunto de JS;
- Como consequência dos dois primeiros itens, o TypeScript pode ser compilado para ser executado em qualquer navegador, dispositivo ou sistema operacional, ou seja, ele não é específico para nenhum ambiente;
- O TypeScript pode ser utilizado pelos desenvolvedores para usar código JavaScript já existente e incorporar bibliotecas.

## Diferenças entre JavaScript e TypeScript

Apesar das semelhanças entre as duas linguagens de programação, obviamente existem algumas **diferenças marcantes**. Entre elas, estão as seguintes:



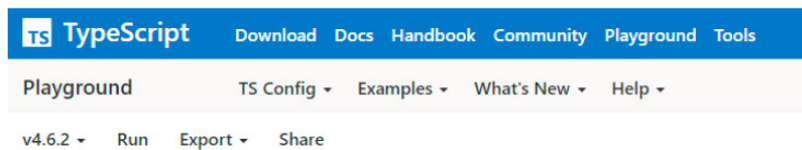
(recurso de  
programação  
orientada a  
objetos)

## Como testar aplicações com TypeScript

Existem algumas formas de testar uma aplicação desenvolvida com TypeScript. Uma delas é fazer a **instalação local via npm**. O passo a passo para esse tipo de instalação pode ser encontrado no site: [typescriptlang.org](https://typescriptlang.org) (download).

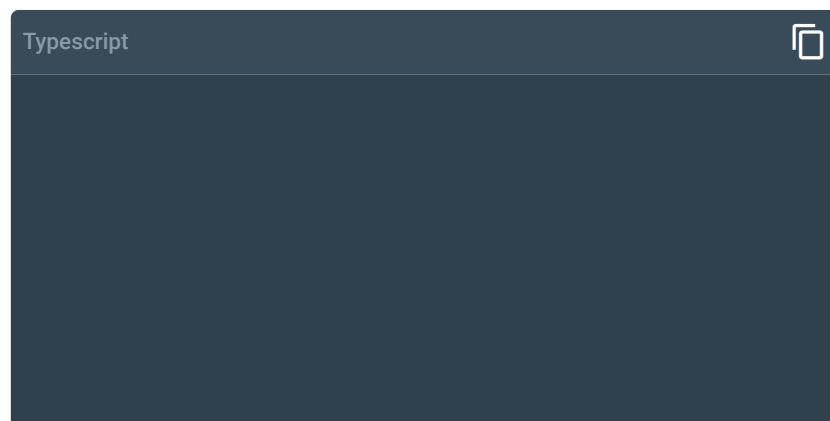
Neste trabalho, vamos executar nosso script on-line com o compilador oficial. Para isso, acessamos o site do [typescriptlang.org](https://typescriptlang.org) (play).

Ao acessarmos esse endereço, veremos uma tela semelhante à da imagem a seguir:



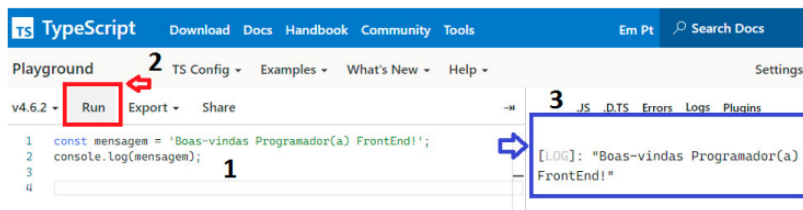
Ambiente de desenvolvimento on-line do TypeScript.

Será nesse ambiente que vamos desenvolver todos os nossos exemplos. A seguir, apresentamos o código de “boas-vindas” no TypeScript:



Agora, vamos selecionar esse código e copiá-lo para o ambiente on-line de desenvolvimento do TypeScript e, em seguida, pressionar o botão

“Run”, conforme na figura:

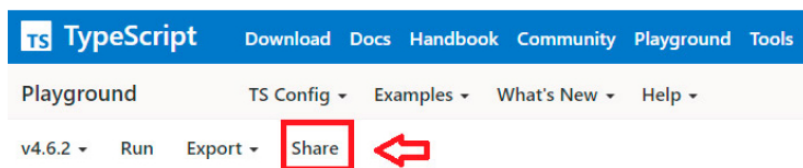


Primeiro programa em TypeScript.

Na imagem, mostramos a sequência de passos, sendo:

- Passo 1: escrever o código em TypeScript;
- Passo 2: pressionar o botão “Run” para executar o código;
- Passo 3: é apresentada a saída da execução.

De fato, esse ambiente é excelente para aprender a programar sem a necessidade de fazer instalações e configurações locais. Além disso, temos a opção de compartilhar o projeto com outras pessoas. Para isso, basta pressionar o botão “Share”, copiar o endereço que ele fornece e enviar para outras pessoas. Na imagem a seguir, mostramos como esse processo ocorre.



Como compartilhar código on-line do TypeScript.

A seguir, os nossos próximos passos serão desenvolver vários exemplos que comparam as linguagens de programação JavaScript e TypeScript.

# Comparações entre JavaScript e TypeScript

## Manipulação de variáveis String

Vamos iniciar o nosso processo de aprendizado com a utilização de variáveis do tipo String que são cadeias de caracteres. Primeiro, vamos ao código em JavaScript:

```
JavaScript
```

Que produz como saída o resultado:

```
[LOG]: "Resultados com JavaScript"
```

```
[LOG]: "Isto é um teste."
```

Agora, vamos implementar uma versão semelhante para TypeScript

```
Typescript
```

Cuja saída é:

```
[LOG]: "Resultados com TypeScript"
```

```
[LOG]: "Isto é um teste."
```

As saídas dos dois códigos são semelhantes. Qual foi, então, a principal diferença entre ambos?



A diferença principal está na declaração da variável usada para armazenar a mensagem com os resultados.

No caso do TypeScript, tivemos que indicar que ela é do tipo "string".

## Manipulação de variáveis numéricas

O nosso próximo passo é entender como utilizar as variáveis numéricas. A seguir, apresentamos um código em JavaScript com comentários explicativos:

```
JavaScript
```

Ao executar o código, obtemos o resultado abaixo:

```
[LOG]: "Resultados com JavaScript"
[LOG]: "número real: 10"
[LOG]: "número hexadecimal: 5066"
[LOG]: "número octal: 442"
[LOG]: "número binário: 45"
```

Agora, vamos implementar uma versão semelhante para TypeScript usando os tipos estáticos, conforme o exemplo a seguir:

```
Typescript
```

Cuja saída é:

```
[LOG]: "Resultados com TypeScript"
```

```
[LOG]: "número real: 10"
```

```
[LOG]: "número hexadecimal: 5066"
```

```
[LOG]: "número octal: 442"
```

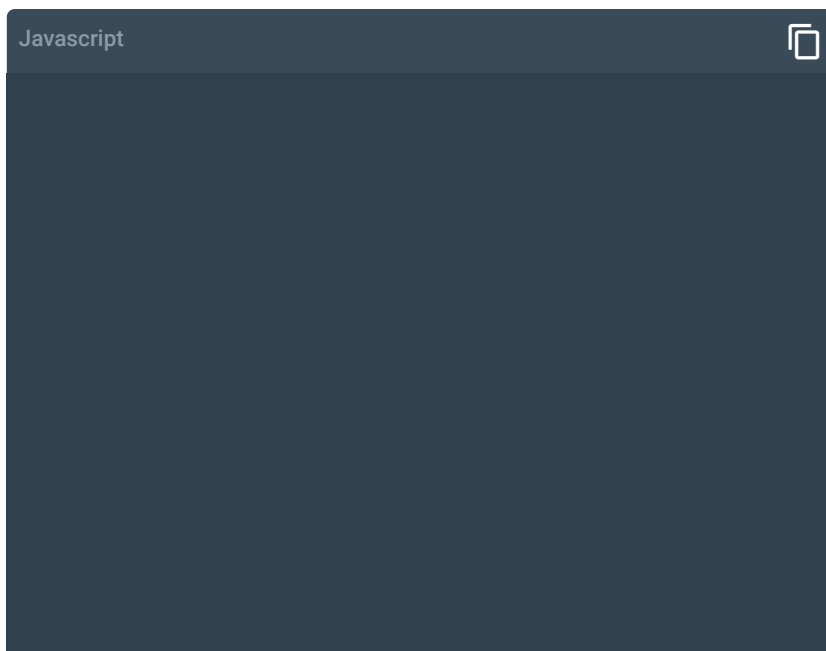
```
[LOG]: "número binário: 45"
```

Novamente, temos saídas semelhantes. A principal distinção entre os códigos está na declaração dos tipos das variáveis.

## Manipulação de vetores

Agora, veremos uma estrutura de dados fundamental em uma linguagem de programação: **vetores**. Com vetores, podemos trabalhar com diversos elementos agrupados em uma única variável. Para acessarmos um elemento específico, precisamos fazer referência para a posição dele dentro do vetor, é o que chamamos de indexação. A seguir, apresentamos o código em JavaScript com os devidos comentários:

```
JavaScript
```

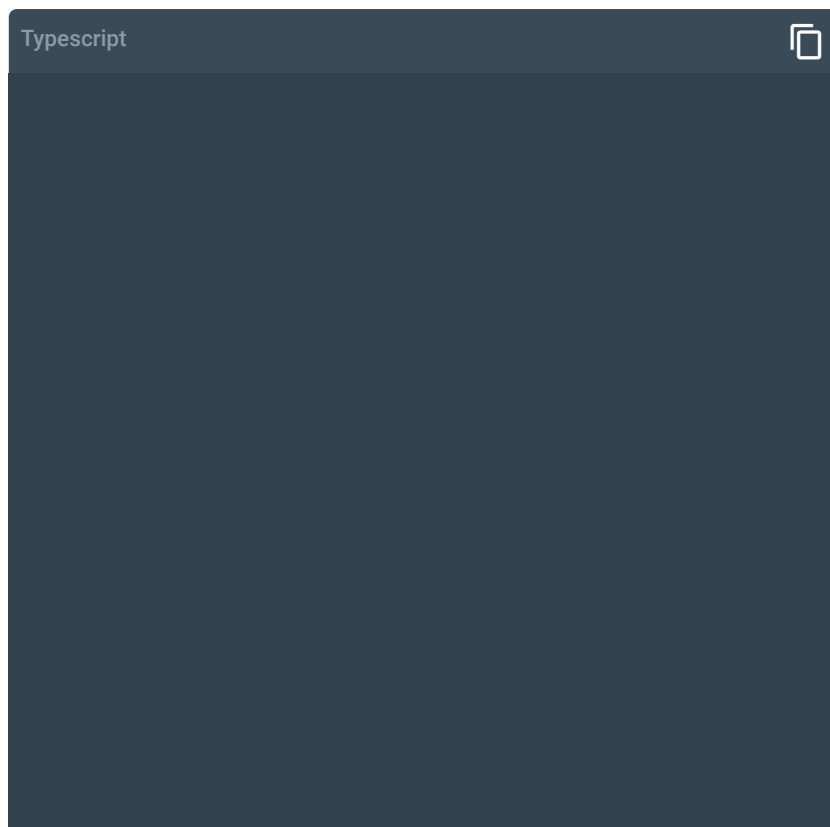




Que produz como saída o resultado:

```
[LOG]: "Resultados com JavaScript"
[LOG]: "x[0]= 7"
[LOG]: "x[1]= 5"
[LOG]: "x[2]= 2"
[LOG]: "x= 7,5,2,1,1,3"
[LOG]: "x= 9,7,5,2,1,1,3"
[LOG]: "x= 1976,7,5,2,1,1,3"
[LOG]: "elemento removido:3"
[LOG]: [1976, 7, 5, 2, 1, 1]
[LOG]: "elemento removido: 1976"
[LOG]: [7, 5, 2, 1, 1]
[LOG]: "o comprimento do vetor é: 5"
[LOG]: "vetor ordenado: 1,1,2,5,7"
[LOG]: "Posição do elemento dentro do vetor: 4"
```

Agora, vamos implementar uma versão semelhante para TypeScript:





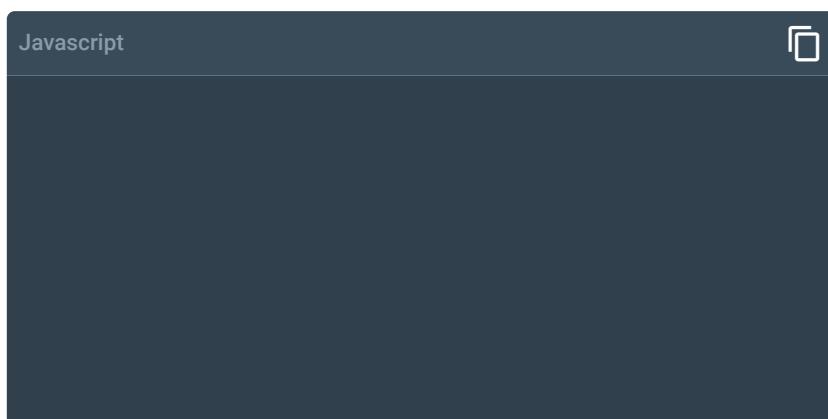
Cuja saída é:

```
[LOG]: "Resultados com TypeScript"
[LOG]: "x[0]= 7"
[LOG]: "x[1]= 5"
[LOG]: "x[2]= 2"
[LOG]: "x= 7,5,2,1,1,3"
[LOG]: "x= 9,7,5,2,1,1,3"
[LOG]: "x= 1976,7,5,2,1,1,3"
[LOG]: "elemento removido:3"
[LOG]: [1976, 7, 5, 2, 1, 1]
[LOG]: "elemento removido: 1976"
[LOG]: [7, 5, 2, 1, 1]
[LOG]: "o comprimento do vetor é: 5"
[LOG]: "vetor ordenado: 1,1,2,5,7"
[LOG]: "Posição do elemento dentro do vetor: 4"
```

Nos dois códigos, fizemos a inserção e remoção de elementos, além de ordená-los. No caso do TypeScript, tivemos que declarar o **tipo do dado** e utilizar **colchetes** para indicar que se trata de um vetor.

## Manipulação de variáveis lógicas

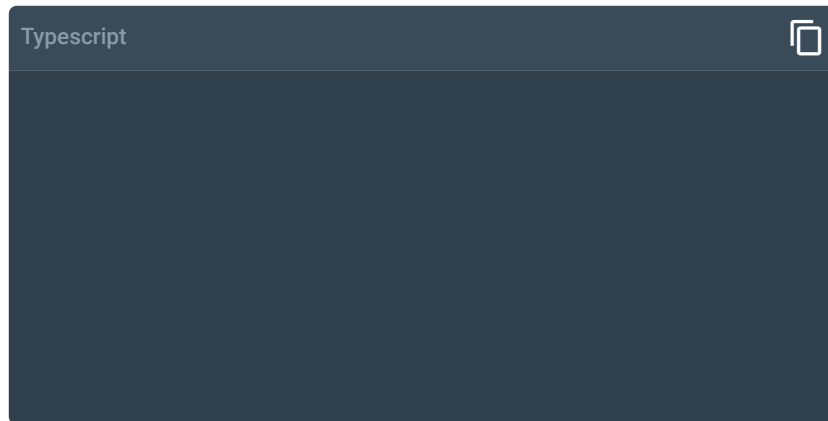
Um tipo de dado muito importante é o lógico, também conhecido como booleano. Uma variável lógica pode assumir apenas um dos seguintes valores em dado momento: **true (verdade)** ou **false (falso)**. Esse tipo de variável é muito útil para realizarmos controles ao longo da execução do programa, como veremos mais adiante na parte de comandos condicionais. Primeiro, vamos ao código em JavaScript:



Que produz como saída o resultado:

[LOG]: "O tipo da variável é: boolean"

Agora, vamos implementar uma versão semelhante para TypeScript:



Cuja saída é:

[LOG]: "O tipo da variável é: boolean"

No caso do TypeScript, tivemos que **explicitar** que a variável é do tipo "booleanos". Além disso, **iniciamos** a variável com true.

## Tipos de operadores

### Trabalhando com operadores aritméticos

Um aspecto básico de qualquer linguagem de programação são os operadores aritméticos. No caso do TypeScript, ele utiliza os seguintes **operadores**:

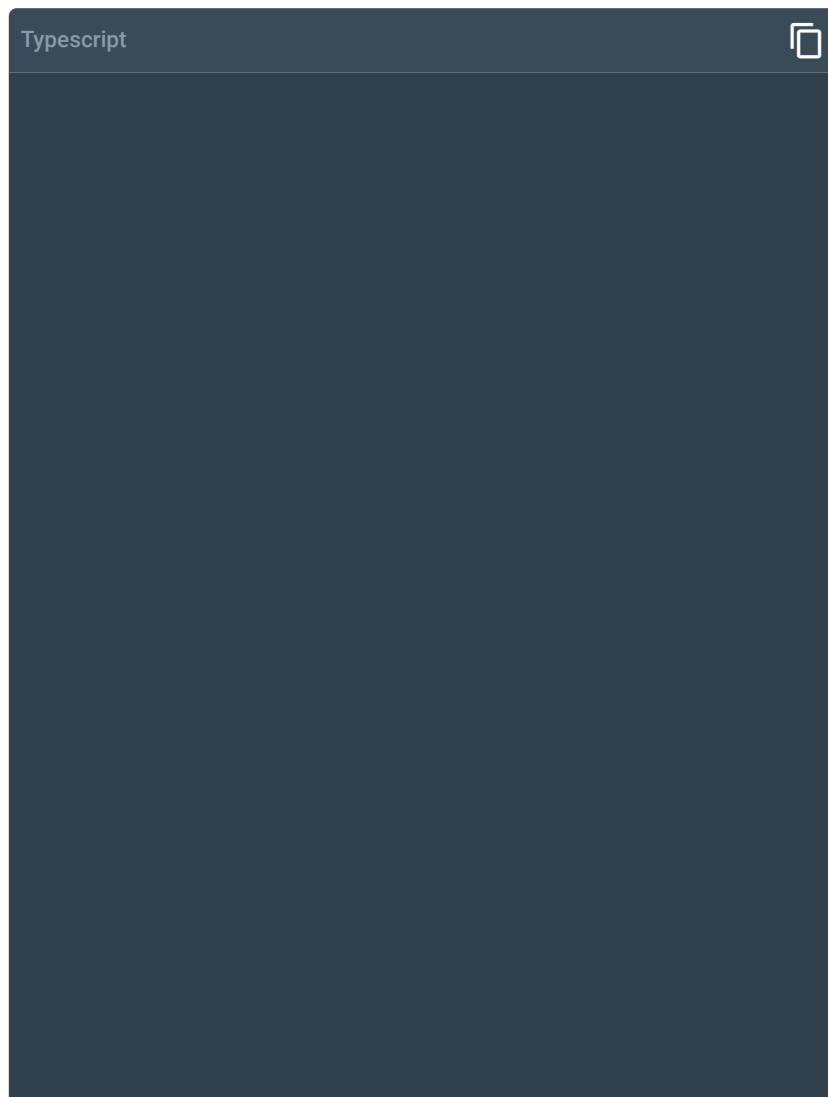
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
++	Incremento
--	Decremento
%	Resto de divisão inteira

**	Potência
+=	Soma e atribuição
-=	Subtração e atribuição
*=	Multiplicação e atribuição
=	Divisão e atribuição

Operadores aritméticos do TypeScript.  
Sérgio Assunção Monteiro

---

Abaixo, apresentamos um exemplo em TypeScript que utiliza os operadores aritméticos:



Abaixo, apresentamos os resultados da execução do programa:

```
[LOG]: "a + b: 30"  
[LOG]: "a - b: -10"  
[LOG]: "a * b: 200"  
[LOG]: "a / b: 0.5"
```

```
[LOG]: "a**2 : 100"
```

```
[LOG]: "a % b: 10"
```

```
[LOG]: "valor += a : 15"
```

```
[LOG]: "valor -= a : -5"
```

```
[LOG]: "valor *= a : 50"
```

```
[LOG]: "valor /= a : 0.5"
```

## Trabalhando com operadores lógicos

Agora, vamos aprender a utilizar os operadores lógicos no TypeScript.

Basicamente, temos os operadores: **&& (E lógico)**, **|| (OU lógico)** e **!**

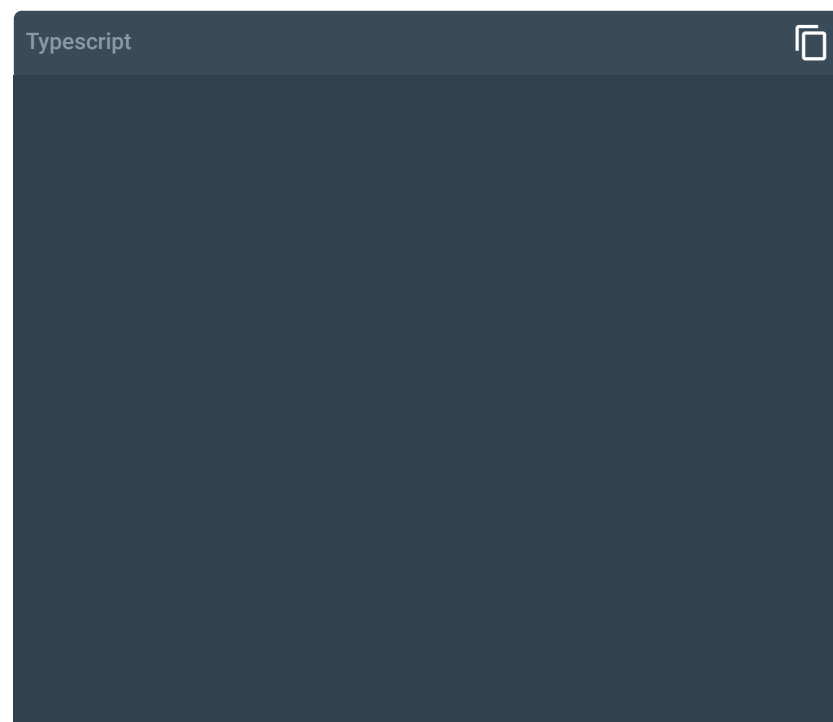
(**negação**). Na tabela, podemos ver como esses operadores funcionam por meio da tabela-verdade:

p	q	p&&q
false	false	false
false	true	false
true	false	false
true	true	true

Operadores lógicos e tabela-verdade.

Adaptado de Developer.mozilla.org

A seguir, apresentamos um código desenvolvido com TypeScript que utiliza esses operadores lógicos:





E agora, mostramos o resultado da execução do programa:

[LOG]: "p1 && p2: false"  
[LOG]: "p1 || p2: true"  
[LOG]: "NOT p1: false"

## Trabalhando com operadores relacionais

Outro aspecto fundamental em uma linguagem de programação é a utilização dos **operadores relacionais** que usamos para fazer comparações entre variáveis. Na tabela a seguir, mostramos os operadores relacionais do TypeScript:

==	igual
!=	diferente
>	maior que
>=	maior ou igual a
<	menor que
<=	menor ou igual a

Tabela 3: Operadores relacionais do TypeScript.  
Adaptado de Developer.mozilla.org

Agora, mostramos um código em TypeScript que utiliza os operadores relacionais. Ele calcula uma média ponderada de duas notas e, dependendo do resultado, exibe uma mensagem para o usuário:

Typescript





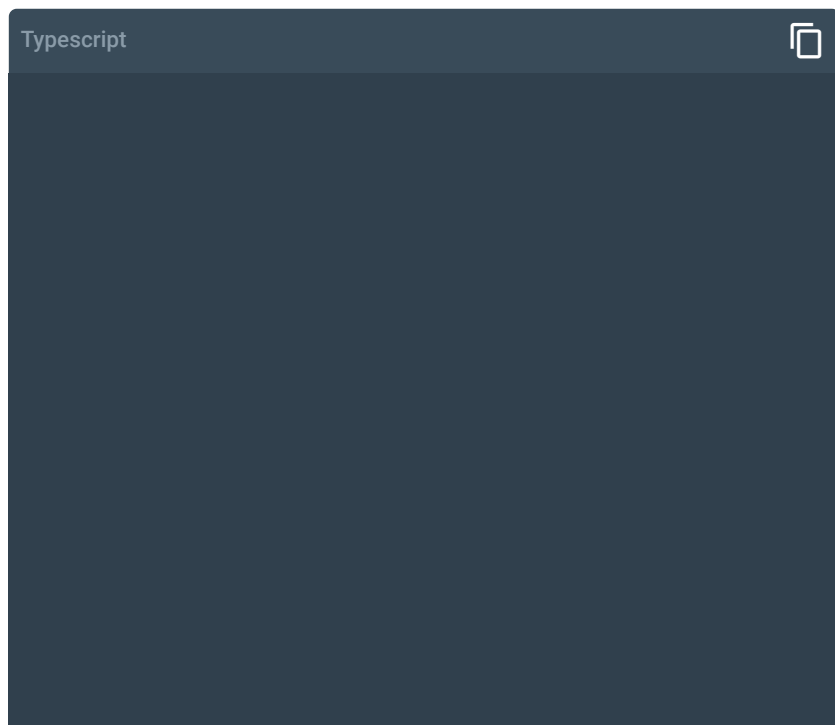
A saída do programa é:

```
[LOG]: "primeira nota: 8.5 ,segunda nota: 6.5"  
[LOG]: "média ponderada: 7.699999999999999"  
[LOG]: "(media == 10): ", false  
[LOG]: "(media>9)&&(media<=9.5): ", false  
[LOG]: "(media>=7)&&(media<9.5): ", true  
[LOG]: "(media>=5)|| (media>9) ", true
```

## Tipos de comandos

### Trabalhando com comandos condicionais

Agora que já aprendemos como declarar variáveis com tipos e utilizar os operadores lógicos e relacionais no TypeScript, vamos estudar como utilizar os **comandos condicionais if-else**. A ideia é bem simples: basicamente, testamos se uma condição é verdadeira; se ela realmente for verdadeira, então o programa passa a executar o bloco de comandos associados ao comando if; caso contrário, ele vai para o comando else se este for utilizado no programa (ABREU, 2017). A seguir, mostramos um exemplo de como utilizar esses comandos condicionais:



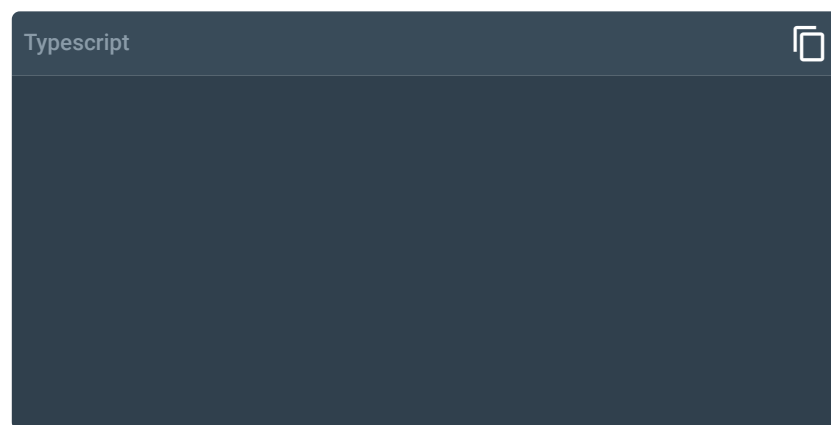
A saída resultante da execução do código é:

[LOG]: "Parabéns!"

No TypeScript, podemos ainda utilizar o operador ternário, que tem a seguinte sintaxe:

```
x = teste? valor1 : valor2;
```

Ou seja, se teste for verdadeiro, então a variável x recebe o valor1, caso contrário, a variável x recebe valor2. A seguir, mostramos um exemplo de código que utiliza o operador ternário no TypeScript:



Após a execução do código, obtemos a seguinte saída:

[LOG]: "já pode votar"

## Trabalhando com comandos iterativos

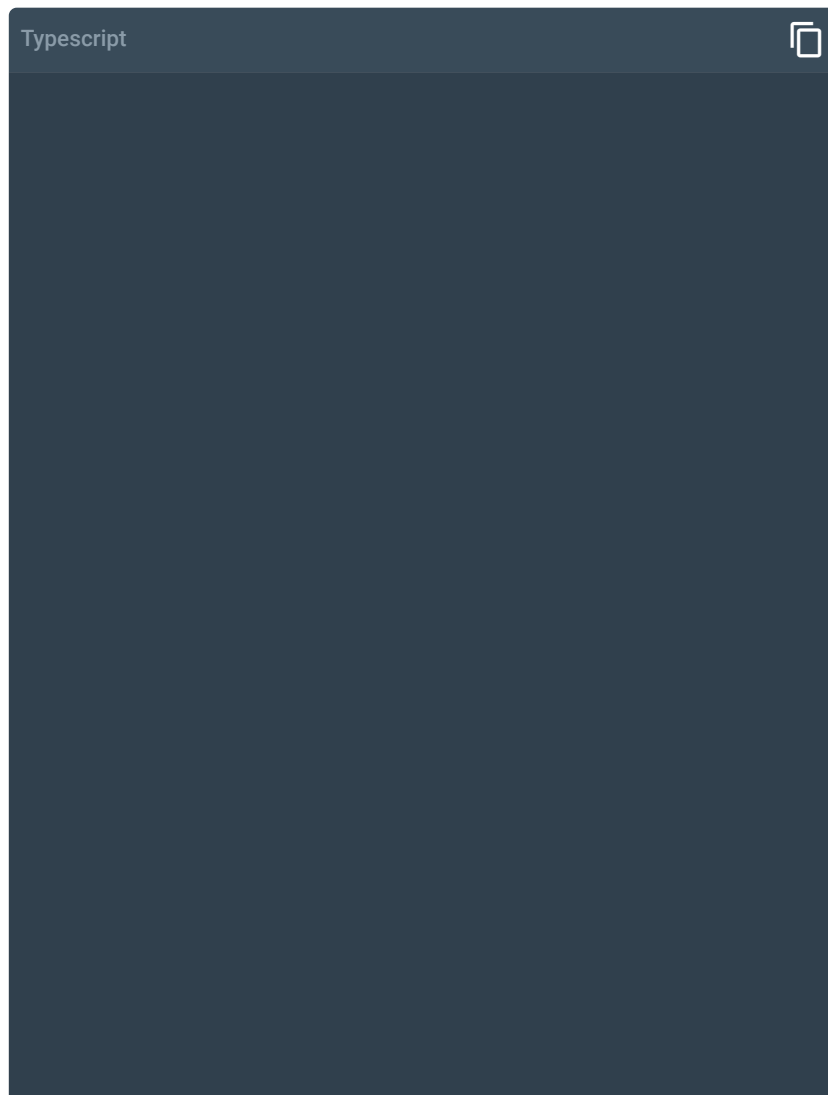
Quando trabalhamos com vetores ou listas, é normal que tenhamos que manipulá-los. Para isso, utilizamos os comandos iterativos. O TypeScript possui dois comandos iterativos: **for** e **while**. O comando **for**, em especial, pode ser usado de diferentes modos. No modo tradicional, ele possui a seguinte sintaxe:

```
for (início; condição; incremento ou decremento)
```

Mas há outra versão baseada na teoria dos conjuntos com a seguinte sintaxe:

```
for (elemento of conjunto)
```

Nesse exemplo, podemos ver as duas versões do comando "for":



A execução do programa produz a seguinte saída:

[LOG]: "Forma 01: O maior elemento do vetor é:40"

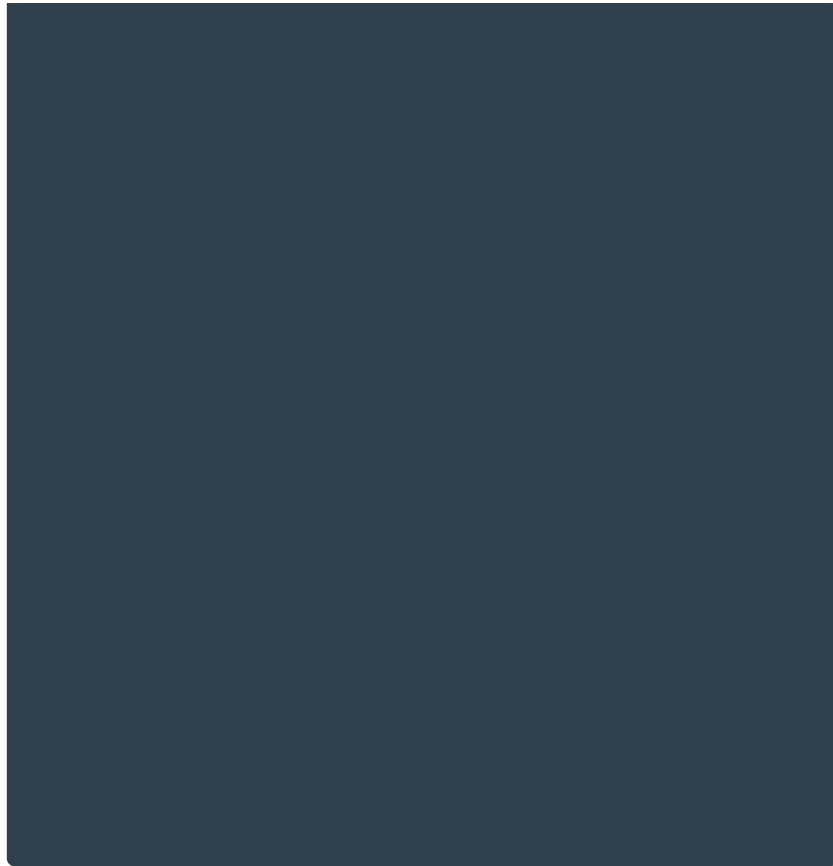
LOG]: "Forma 02: O maior elemento do vetor é:40"

Agora, vamos analisar o comando **while**. Ele também tem duas versões:

- uma em que o teste é realizado antes de entrar no bloco de comandos associados a ele;
- outra em que o teste só é realizado após a execução do bloco de comandos.

A seguir, mostramos um exemplo com as duas versões do comando **while**:





A saída da execução desse código é dada por:

[LOG]: "Forma 01: O maior elemento do vetor é:70"

[LOG]: "Forma 02: O maior elemento do vetor é:70"

Aqui, fechamos os conhecimentos básicos do TypeScript. Não deixe de testar todos esses códigos e fazer pequenas modificações para analisar o comportamento deles. Mais adiante, vamos aprender alguns assuntos mais avançados do TypeScript. Bons estudos!



## Conceitos gerais e similaridades do TypeScript com o JavaScript

Neste vídeo, são apresentados os principais conceitos e similaridades do TypeScript com o JavaScript.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

## Questão 1

O TypeScript é uma linguagem de programação baseada no JavaScript. Ambas podem ser utilizadas em contextos semelhantes. Então, é natural nos questionarmos por que devemos utilizar o TypeScript em vez do JavaScript. Nesse sentido, selecione a opção correta que justifica essa escolha:

**A**

O TypeScript oferece recursos que permitem construir códigos reutilizáveis e mais fáceis de dar manutenção.

**B**

O TypeScript é mais moderno que o JavaScript.

**C**

O JavaScript é mais lento do que o TypeScript.

**D**

O JavaScript, diferentemente do TypeScript, não dá suporte para utilização de vetores.

**E**

O JavaScript, diferentemente do TypeScript, não é adequado para aplicações de BackEnd.

**Parabéns! A alternativa A está correta.**

O TypeScript é baseado em JavaScript com o incremento de importantes melhorias, tais como o uso de tipos de dados e suporte à programação orientada a objetos. Dessa forma, é muito mais simples fazer a manutenção de um código em TypeScript do que em JavaScript.

## Questão 2

Os comandos iterativos, tais como o “for”, fazem parte da sintaxe do TypeScript. Por exemplo, vamos considerar o trecho de código abaixo:





Ao executá-lo, o desenvolvedor obteve como resultado "0" e ficou surpreso. Nesse sentido, selecione a opção correta a respeito do código:

**A**

De fato, o desenvolvedor deveria mudar a operação "x+=a," por "x=x+a".

**B**

Está correto, pois faz a soma dos números pares.

**C**

O uso do comando "for" está incorreto.

**D**

Está faltando definir o tipo da variável "a".

**E**

O operador "%" retorna o resultado da divisão de "a" por "2" em termos percentuais.

**Parabéns! A alternativa B está correta.**

O trecho de código está correto. O que está ocorrendo é que ele está somando apenas os números que são divisíveis por 2, ou seja, que são pares. Como o vetor "v" possui apenas números ímpares, o resultado da soma é "0".



## 2 - Funções com TypeScript

Ao final deste módulo, você será capaz de analisar os tipos e narrowing.

# Introdução às funções

## A importância das funções

As funções constituem unidades lógicas básicas de um código. É por meio delas que garantimos que um programa seja sustentável e reutilizável. Quando as utilizamos corretamente, podemos construir bibliotecas que poderão ser utilizadas, inclusive, por vários programas. Uma consequência positiva imediata disso é a divisão de trabalhos de uma equipe, pois pessoas com diferentes habilidades podem trabalhar em partes específicas do desenvolvimento.

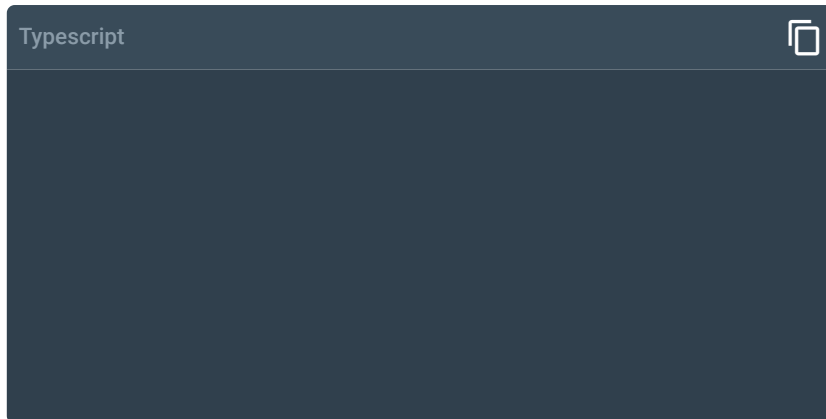
- Definir os argumentos de uma função;
- Identificar a saída que a função retorna, se é que ela deve retornar algum valor.

Portanto, o usuário que vai consumir a função a enxerga como uma caixa preta, ficando, desse modo, sob a responsabilidade do desenvolvedor garantir que funcione corretamente e com o melhor desempenho possível.

## Sintaxe básica

O TypeScript oferece suporte para funções. A sintaxe básica de uma função com o TypeScript é, (TYPESCRIPT HANDBOOK, 2022):

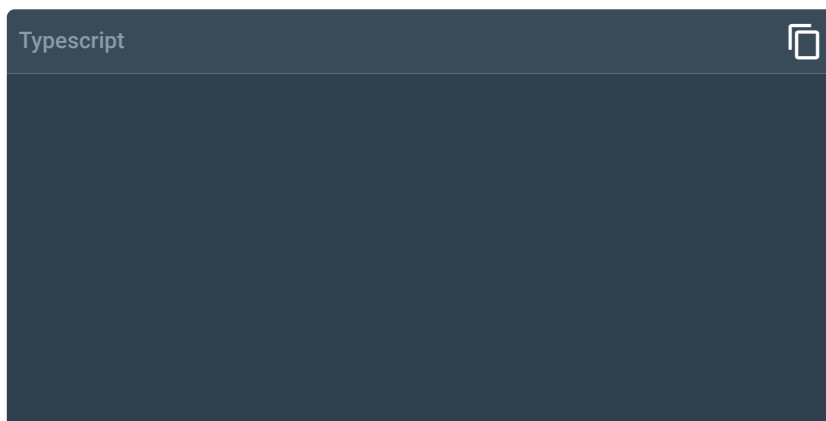




Apesar de essa ser a sintaxe mais comum, existem outras formas de programar uma função como veremos mais adiante. A seguir, vamos conhecer as funções que não possuem retorno.

## Funções sem retorno

Há situações em que não estamos interessados que a função tenha retorno. Nesses casos, a função utiliza o tipo “void” para informar que não terá retorno. A seguir, apresentamos um código com um exemplo de função que não possui retorno:



A função “imprimir\_ts” recebe dois parâmetros, sendo um deles do tipo string e o outro do tipo number. Ela simplesmente imprime os conteúdos dos parâmetros “msg” e “num”, como podemos ver abaixo:

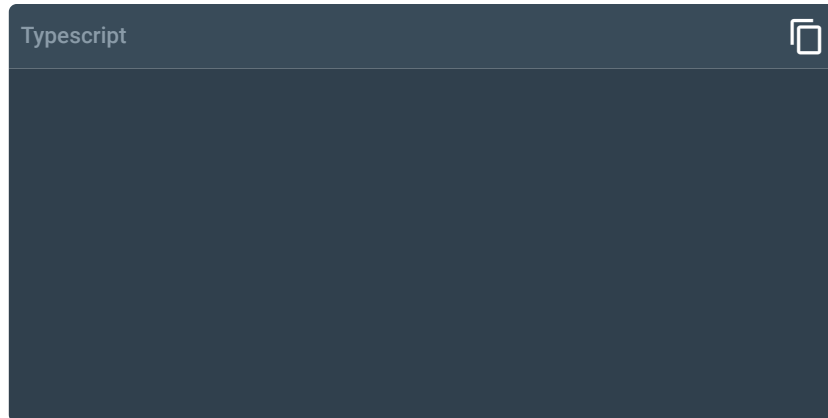
[LOG]: "O conteúdo da variável mensagem é este é um teste e da variável numérica é 10"

A seguir, estudaremos a forma mais comum de utilizar uma função que é a aplicação de retorno.

## Funções com retorno

Uma função omite do usuário a complexidade da lógica de programação. Dessa forma, o usuário deve se preocupar apenas com a passagem de parâmetros para a função e utilizar o retorno que ela produz. Nesse caso, precisamos informar qual o tipo de retorno da

função e utilizar, explicitamente, o comando “return”. A seguir, apresentamos um exemplo de função com retorno: |

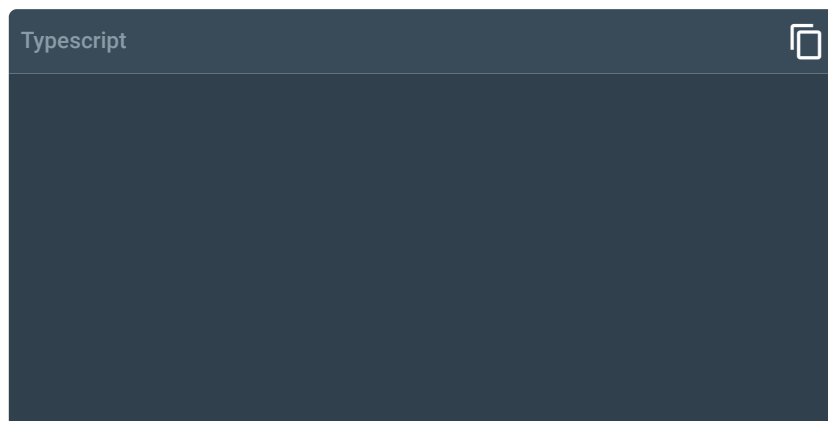


No caso, a função recebe dois parâmetros do tipo number, faz a adição deles e, em seguida, retorna o resultado da adição para quem fez a chamada da função. Agora, apresentamos a saída do código do exemplo:

[LOG]: "A soma de 10 com 20 é: 30"

## Funções com parâmetros do tipo string

Vamos ver mais um exemplo de função com retorno. Nesse caso, a função recebe dois parâmetros do tipo String e retorna a concatenação deles, como podemos ver no seguinte código :



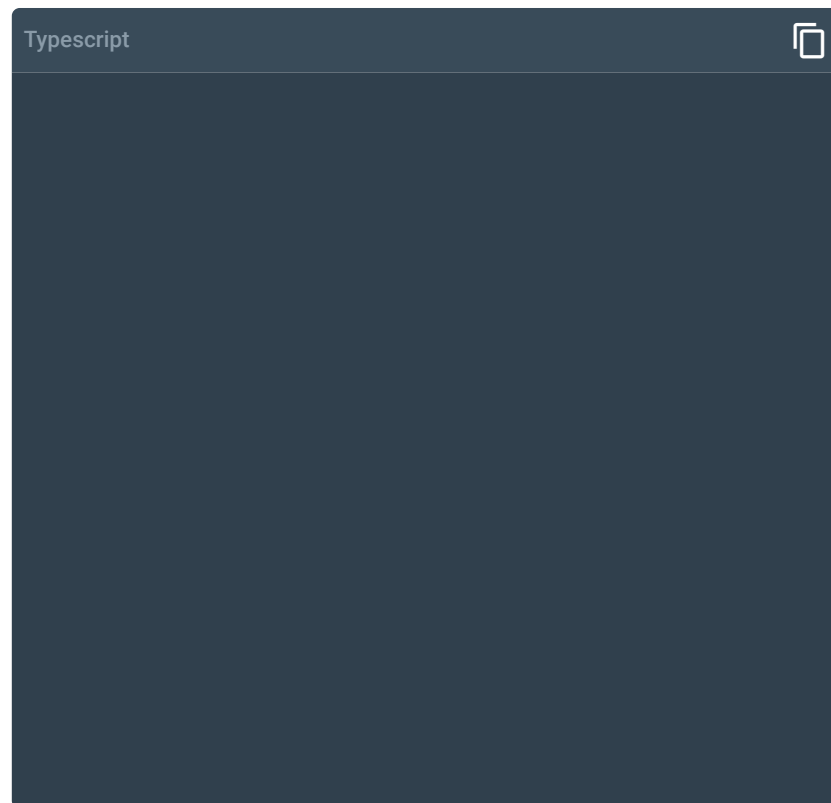
O resultado da execução do programa é dado a seguir:

[LOG]: "A concatenação de primeiro com segundo é: primeiro, segundo"

## Funções para manipulação de vetores e recursivas

## Funções para manipular vetores

Como vimos, vetores são estruturas de dados que permitem manipularmos diversos elementos por meio da mesma variável com o uso de índices. Logo, é natural trabalharmos com vetores nas funções. Passar um vetor como parâmetro para uma função é semelhante ao que fizemos com os outros tipos de elementos, ou seja, precisamos colocar o tipo de dado e utilizar os colchetes. Apresentamos agora um exemplo de como passar um vetor como parâmetro de uma função:



A função “somatorio” faz a soma de todos os elementos do vetor numérico. Para acessar cada elemento do vetor dentro da função, utilizamos o comando iterativo “for”. O resultado da execução do exemplo é dado por:

[LOG]: "O resultado do somatório dos elementos do vetor 1,2,3,4,5 é: 15"

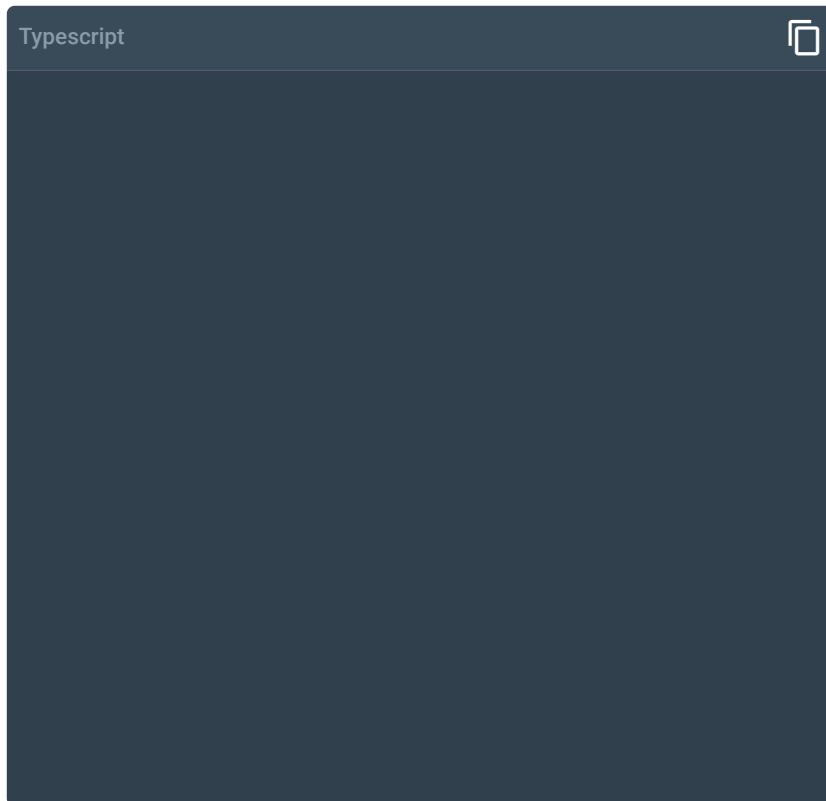
## Funções recursivas

Há situações que podemos descrever um problema em termos de si mesmo. O exemplo clássico para esse tipo de situação ocorre com o cálculo do fatorial de um número  $n$  que é dado por:

$$\text{fatorial}(n) = 0, \text{ caso } n = 0, \text{ ou } n = 1$$

$$\text{fatorial}(n) = n * \text{fatorial}(n-1)$$

A seguir, apresentamos a implementação para calcular o fatorial de um número  $n$  por meio de uma função no TypeScript:



A saída da execução desse código é dada por:

[LOG]: "O fatorial de 5 é: 120"

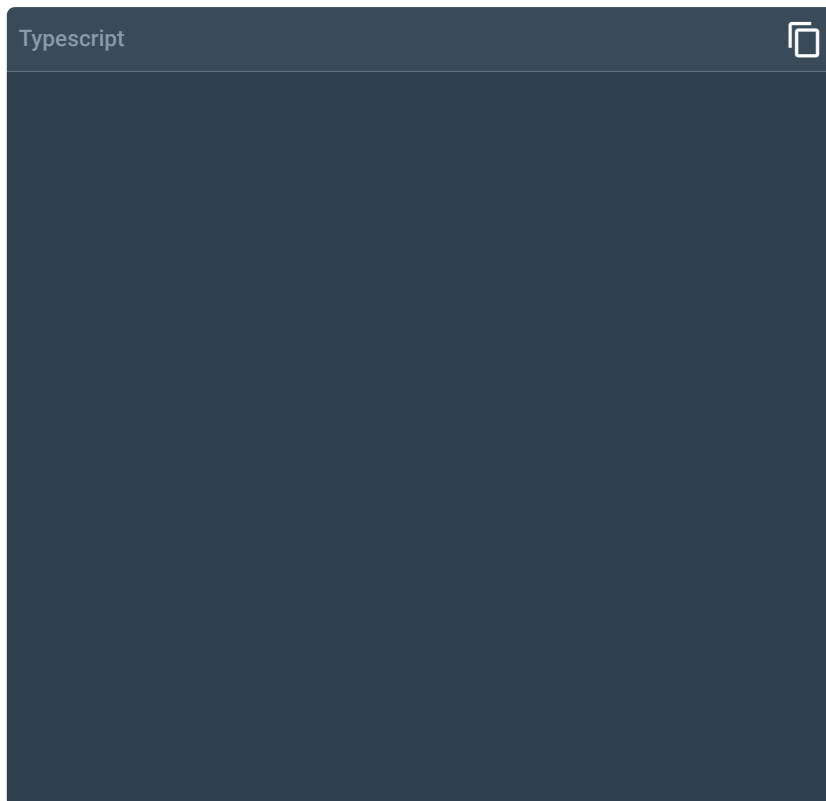
Aqui, cabe uma observação: utilizar recursividade é muito útil em algumas situações, mas sempre devemos **ficar atentos** com o **caso base**, pois é ele que garante que a função vai parar de fazer chamadas a si mesma. Na próxima seção, vamos conhecer mais sobre funções no TypeScript.

## Mais alguns tópicos sobre funções no TypeScript

### Arrow Functions

Nos códigos mais avançados do TypeScript, é muito comum utilizarmos **funções lambda**, que são estruturas lógicas de programação funcional. A sintaxe desse tipo de função utiliza uma seta que é a sintaxe abreviada para definir a função anônima. Esse tipo de função é conhecida como função seta, ou, de forma mais comum, como **arrow function**. Entre as implicações práticas de utilizá-las estão a eliminação

da necessidade de escrever a palavra-chave "function" e fazer a associação da função a uma variável. Na sequência, apresentamos dois exemplos de como utilizar uma arrow function:



Na função "soma\_anom\_ts\_1", utilizamos a delimitação de blocos com os símbolos de abre parênteses e fecha parênteses, além de utilizarmos a palavra-chave return. No caso da função "soma\_anom\_ts\_2", não fizemos uso dos delimitadores de bloco nem da palavra-chave return e, ainda assim, obtivemos os mesmos resultados, como podemos ver a seguir:

[LOG]: "Teste 01: a soma de 10 com 20 é: 30"

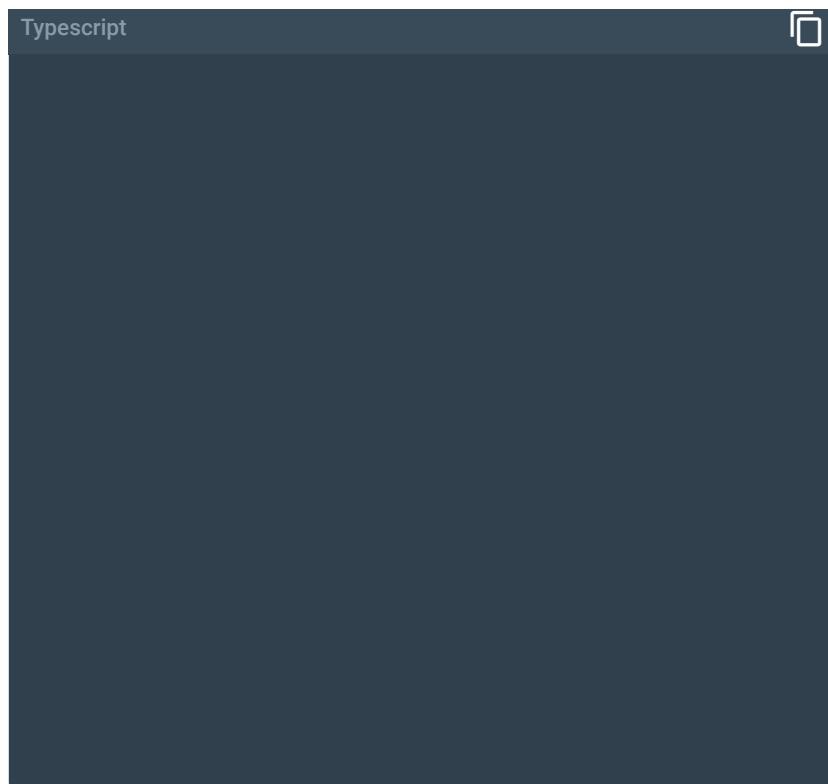
[LOG]: "Teste 02: a soma de 10 com 20 é: 30"

## Funções com expressões regulares

Agora, vamos aprender como utilizar expressões regulares. Esse é um **conhecimento essencial** quando trabalhamos com desenvolvimento de aplicações Web, como para buscar por padrões em textos. Por meio dessas funções, podemos procurar por **substrings** dentro de outras. A seguir, mostramos um exemplo de uso de expressões regulares no TypeScript:

### Substring

É uma sequência de caracteres vizinhos dentro de uma string. Por exemplo, "variáveis com tipos" é uma substring da string "Typescript possui variáveis com tipos".



Para utilizar uma expressão regular, precisamos indicar que o tipo de dado da variável é "RegExp". No caso do exemplo, o padrão que estamos procurando é a palavra "te" dentro do texto de teste. A seguir, apresentamos o resultado da execução do programa:

[LOG]: "O resultado da execução do regex foi: Foi detectado o padrão dentro do texto"

## Funções com expressões regulares mais interessantes

O que vimos anteriormente sobre o uso de expressões regulares por si só já é muito útil, pois é muito comum precisarmos fazer esse tipo de busca. Mas podemos fazer muito mais do que isso com expressões regulares. Por exemplo, gostaríamos de saber quantas vezes determinado padrão foi encontrado, como no código seguinte:



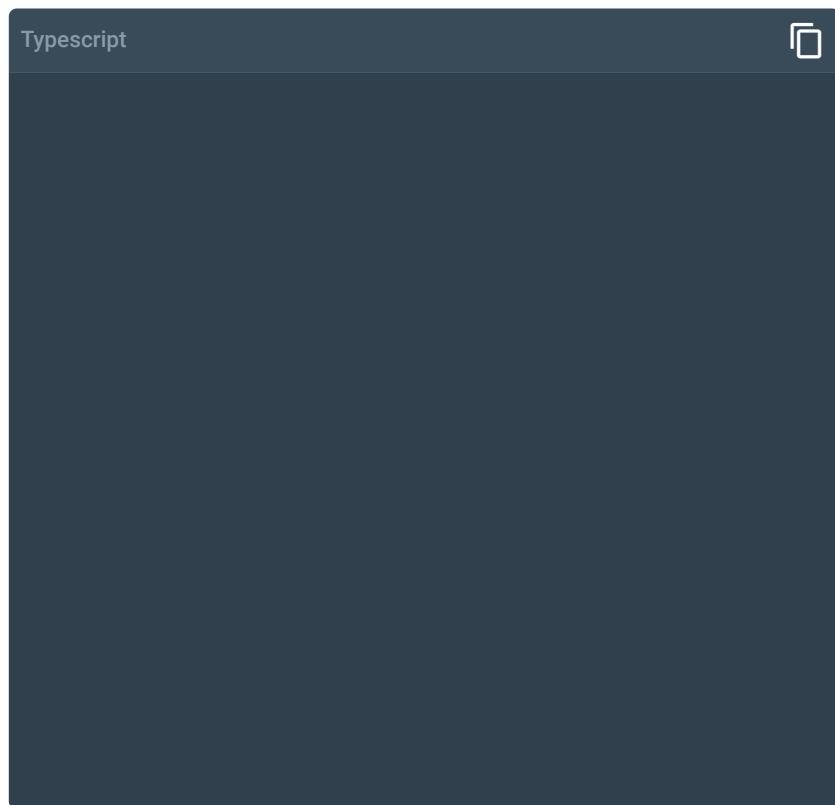


No código, procuramos pelo padrão de cinco dígitos, separados por um traço e seguidos de exatamente três dígitos. Observe a saída da execução:

[LOG]: "Das 4 palavras, o total que satisfaz o padrão `\d{5}-\d{3}/` é de: 2"

## Funções com parâmetros Rest

Esse tipo de função permite receber uma quantidade não definida de parâmetros. Ela é muito útil para aplicações Web. A sintaxe dela é caracterizada pelo uso de três pontos logo depois de algum parâmetro. Agora, apresentamos um exemplo da utilização tradicional desse tipo de função:



Cuja saída é dada por:

[LOG]: "Primeira saudação: Bom dia! Alunos, Alunas!"

[LOG]: "Segunda saudação: Bom dia! Alunos, Alunas, Professoras, Professores!"

Vejamos outra versão que vai resolver a mesma situação com a utilização de arrow function que mostramos no seguinte código:





Cuja saída é dada por:

[LOG]: "Primeira saudação: Bom dia! Alunos, Alunas!" [LOG]: "Segunda saudação: Bom dia! Alunos, Alunas, Professoras, Professores!"

As saídas das duas versões são semelhantes, como era de se esperar. O ponto fundamental que precisamos notar é que utilizamos as mesmas chamadas da função, mas aplicamos **quantidades diferentes** de parâmetros. Isso se aplica muito bem em situações nas quais queremos generalizar tratamentos de dados sem a necessidade de implementar diversas funções para cada situação. Mas há um problema aqui também, pois se essa utilização não for bem-organizada, será difícil dar manutenção no código.

Bem, agora que já conhecemos os aspectos fundamentais do TypeScript e sabemos as diversas maneiras de criarmos e utilizarmos funções, vamos nos aprofundar em aspectos mais avançados da linguagem como o conceito de Narrowing, que oferece mais flexibilidade na programação. Antes, porém, fica a sugestão: execute todos os códigos que você viu até agora. Faça pequenas alterações e estude os resultados. Também crie situações para gerar erros, pois é uma boa forma de aprender mais sobre a linguagem. Depois de dominar bem o que viu até agora, avance para a próxima etapa. Bons Estudos!



## Tipos e narrowing da linguagem de programação



# TypeScript

Neste vídeo, serão apresentados os principais conceitos de tipos e narrowing da linguagem de programação TypeScript.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

### Questão 1

O TypeScript dá suporte ao uso de funções. Os parâmetros dessas funções possuem tipos, e a própria função também utiliza tipos para definir o retorno dela. Nesse sentido, selecione a opção correta que justifica como uma vantagem a forma como o TypeScript trabalha com funções:

**A**

O uso de funções elimina a possibilidade de erros em tempo de execução.

**B**

Ao utilizar tipos para os parâmetros das funções, o TypeScript acelera o desempenho de execução de um programa.

**C**

A principal vantagem de utilizar tipos nos parâmetros das funções é a reutilização de código.

**D**

Com a utilização de tipos nos parâmetros das funções, o TypeScript possibilita trabalhar com vetores.

**E**

Usar tipos nos parâmetros auxilia na aplicação correta das operações que podem ser feitas com eles.

Parabéns! A alternativa E está correta.

A utilização de tipos nos parâmetros de uma função que o TypeScript suporta é uma vantagem no sentido de auxiliar no

controle das operações que podem ser realizadas com uma determinada variável.

## Questão 2

O TypeScript dá suporte ao uso de função anônimas, também conhecidas como arrow functions. Por exemplo, vamos considerar o trecho de código em que x deve ser, obrigatoriamente, maior do que 1:

```
let r = (x: number):number => x==1? 1:x+r(x-1);  
console.log(r(8));
```

Nesse sentido, selecione a opção em que há um código com uma lógica equivalente:

**A**

```
function r1(n: number): number {  
  return (1+n)*n/2;  
}
```

**B**

```
function r2(n: number): number {  
  return 36;  
}
```

**C**

```
let r3 = (x: number):number => 36;
```

**D**

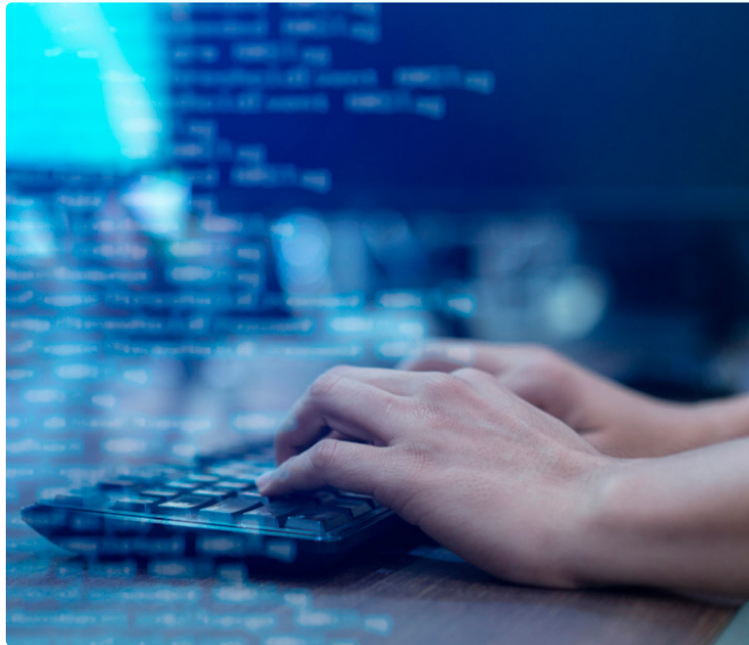
```
function r4(n: number): number {  
  if(n%2==0){  
    return 36;  
  }  
  return 0;  
}
```

**E**

```
let r5 = (x: number):number => x%2==1? 0:36;
```

**Parabéns! A alternativa A está correta.**

Apesar de todos os códigos retornarem o mesmo valor passando como parâmetro o valor "8", o único que é equivalente ao código do exercício é o da função "r1". Para verificar isso, basta testar a chamada para outros valores, com "5", "9" e "20", por exemplo. Para esses exemplos, apenas a função "r1" produzirá os mesmos valores da função "r".



### 3 - Tipos e narrowing

Ao final deste módulo, você será capaz de descrever funções com TypeScript.

## Os tipos estáticos do TypeScript e suas limitações

### Definição de Narrowing

**Narrowing**, ou como é traduzido para o português, estreitamento, é um recurso que o TypeScript oferece como um modo de seguir possíveis caminhos de execução quando uma função recebe um parâmetro, ou mesmo quando declaramos uma variável.

A ideia é que uma variável possa ter o seu tipo definido durante a execução do programa.

Para fazer isso, precisamos indicar quais são os possíveis tipos que ela pode assumir (TYPESCRIPT HANDBOOK, 2022).

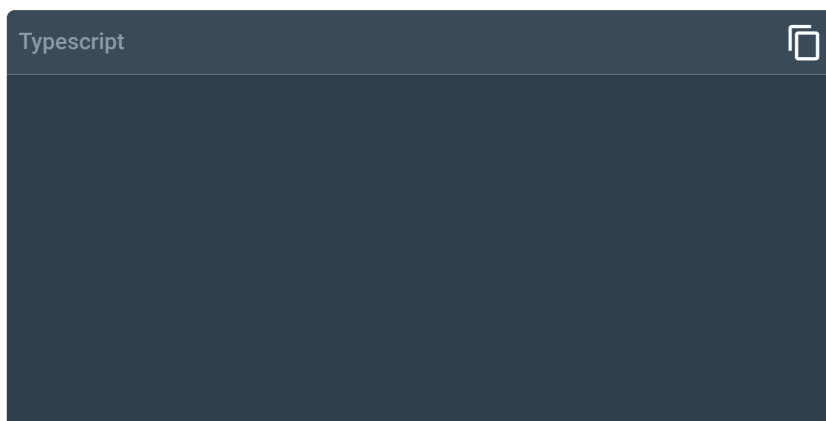
Há um ponto importante para levarmos em consideração aqui. Como os tipos das variáveis serão definidos durante a execução, precisamos verificar em que tipo a variável está operando, pois não podemos utilizá-las da mesma forma.

Para tratar essa situação, o TypeScript utiliza um recurso chamado **"Type Guard"** – é traduzido para o português como "tipos protegidos". O objetivo desse recurso é verificar qual o tipo em que a variável está operando por meio do operador `typeof`.

A seguir, vamos ver alguns exemplos de como utilizar o operador `typeof` com os tipos de variáveis.

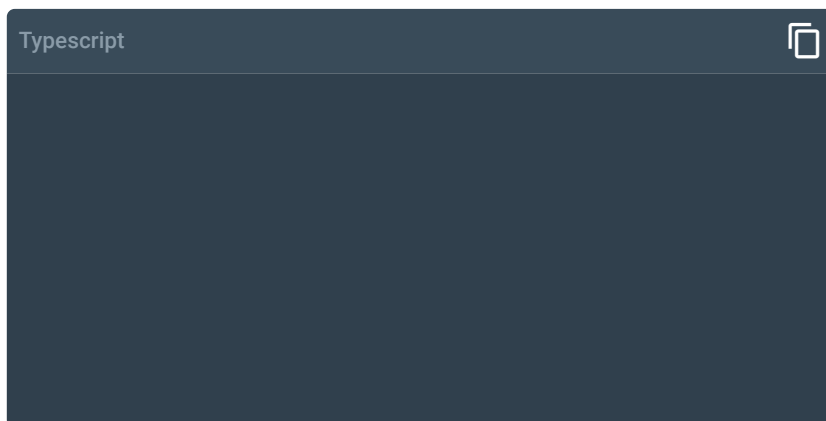
### Parâmetro String

Neste exemplo, para testar se o parâmetro é do tipo String, precisamos realizar o seguinte teste:



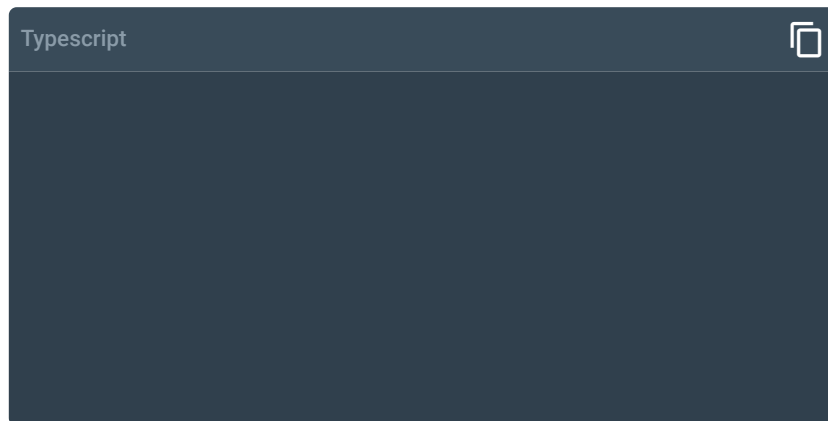
### Parâmetro Number

Utilizamos tipos numéricos com bastante frequência em aplicações desenvolvidas com TypeScript. A sintaxe é dada por:



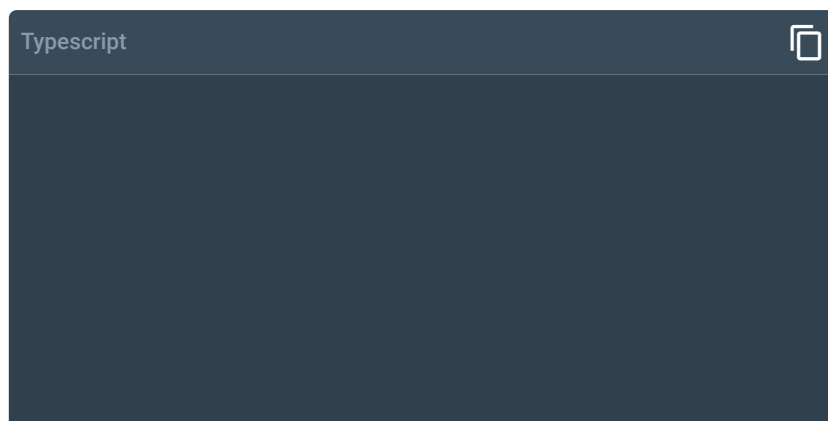
### Parâmetro BigInt

O tipo `BigInt` é utilizado em situações nas quais precisamos manipular números inteiros muito grandes, como, por exemplo, identificadores de banco de dados e situações em que precisamos manipular data e hora. A sintaxe para testar se um parâmetro é do tipo `bigint` é dada por:



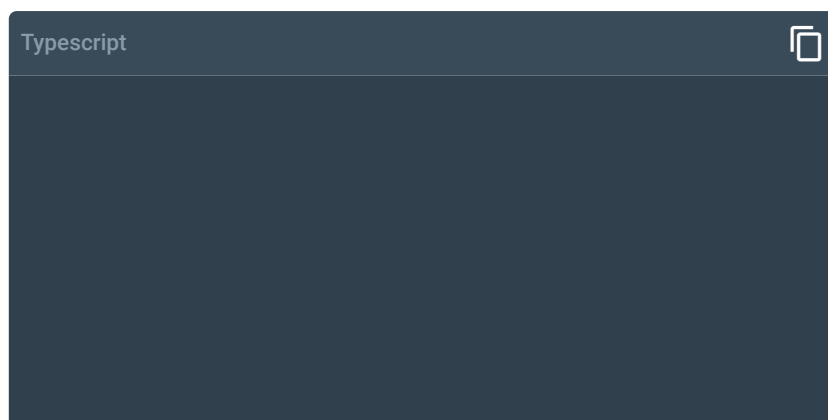
### Parâmetro Boolean

Utilizar variáveis lógicas é muito comum, pois, constantemente, deparamo-nos com situações que exigem testes. A sintaxe para testar se uma variável é do tipo lógico é dada por:



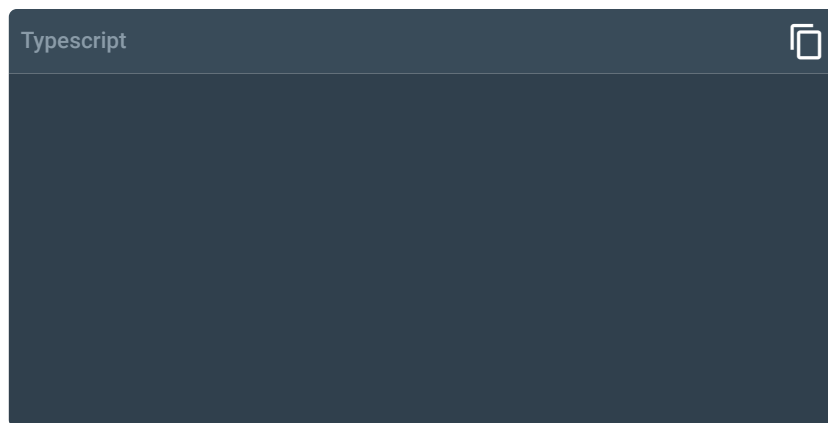
### Parâmetro Symbol

`Symbol` é um tipo primitivo do TypeScript que é útil para criarmos símbolos que serão utilizados para controlar os possíveis valores de uma variável. A sintaxe para testar se uma variável é do tipo `symbol` é dada por:

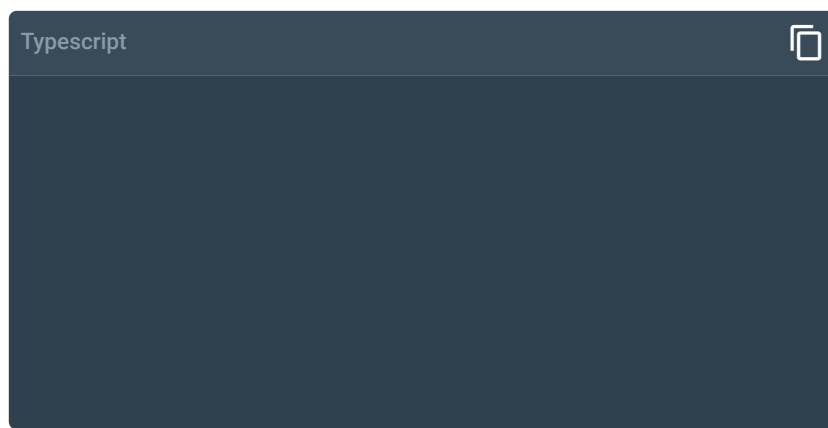


### Parâmetro Undefined

Quando um tipo não é definido, o TypeScript utiliza o Undefined. A sintaxe básica para testar se uma variável é Undefined é:

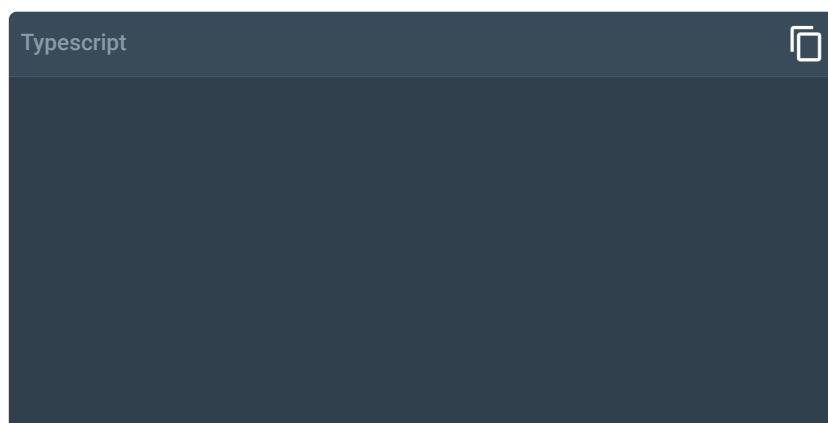


### Parâmetro Object



### Parâmetro Function

Uma das implicações de utilizar arrow function é que podemos passá-las como parâmetros para outras funções. Para testar se um parâmetro é do tipo function, utilizamos a seguinte sintaxe:



### Narrowing confiável

O TypeScript oferece diversos recursos para trabalhar com Narrowing. Um desses recursos é a utilização do símbolo "?" (interrogação) para verificar se uma variável é realmente confiável. A seguir, mostramos um exemplo de como isso funciona na prática:





O parâmetro “y” da função está escrito “y?”, que já indica para o TypeScript que deve ser verificado se ela é ou não confiável. Além disso, no retorno da função, é utilizado “??”, indicando que, se a variável “y” não for confiável, então deve ser somado o valor “10” à variável “x”. Observe a saída da execução do código:

[LOG]: "Teste 1: 10"

[LOG]: "Teste 2: 11"

[LOG]: "Teste 3: 11"

## Narrowing por meio de Type Guards

### Narrowing com if (refinamento)

A forma mais comum de utilizarmos o Narrowing em uma função é definir nos parâmetros quais são os possíveis tipos que a função pode assumir com o auxílio do símbolo “|” (barra vertical). Abaixo, apresentamos um exemplo em que o parâmetro pode assumir ou o tipo “number”, ou “string”. Além disso, utilizamos o operador “typeof” para garantir a proteção do tipo:





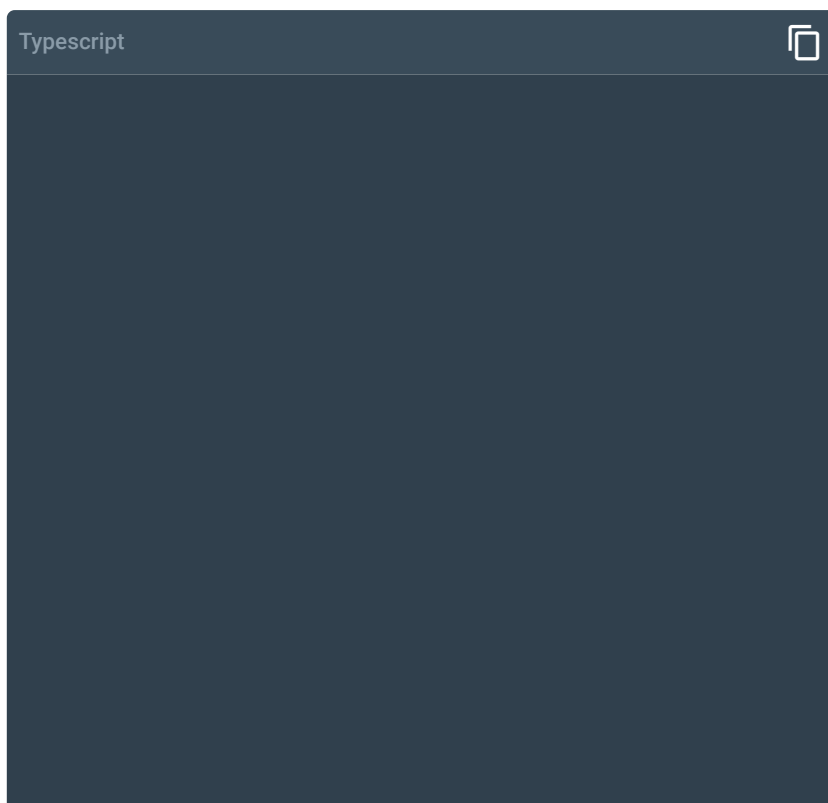


Devemos perceber, ainda, que a variável `t_msg` também utiliza explicitamente o `narrowint`. A saída, após executar o código, é dada por:

[LOG]: "O resultado da execução da função é: 15"

## Narrowing com switch

Outra forma de fazer os testes dos tipos é por meio do comando condicional `switch`. Esse comando é muito útil para testar situações que envolvem enumeração. No exemplo seguinte, podemos ver como utilizar o `switch`:



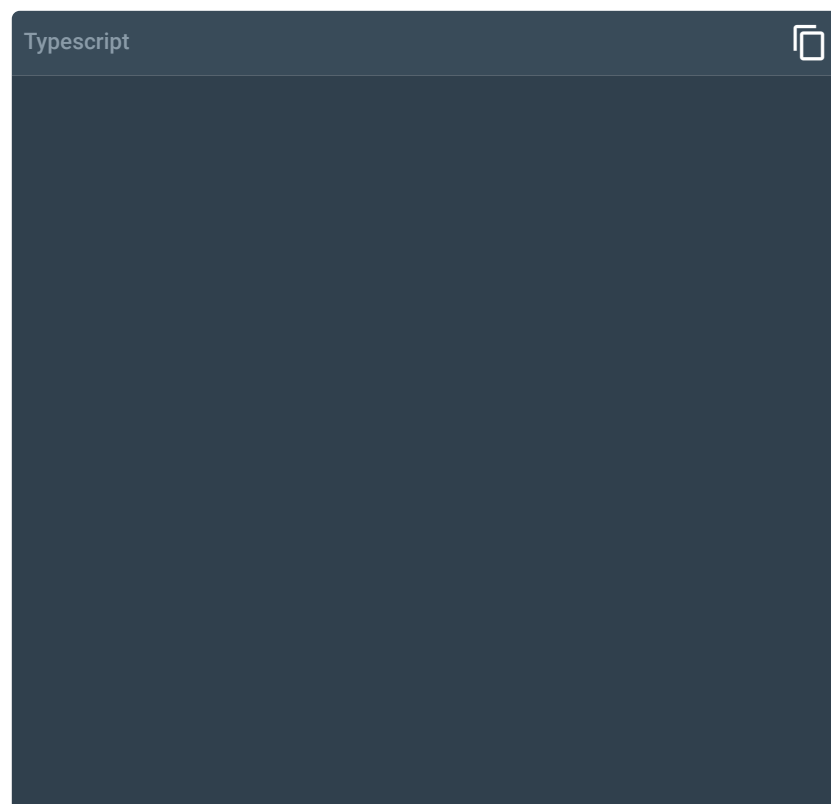
Como o parâmetro passado para a função `func_testar_ns` foi uma string, então o programa retornou o comprimento da frase:

[LOG]: "O resultado da execução da função é: 15" Teste.

# Narrowing customizado

## Narrowing com tipos definidos pelo usuário

Apesar do TypeScript oferecer muitos tipos, às vezes, é necessário que criemos os nossos próprios tipos que também podem ser utilizados com Narrowing. No exemplo a seguir, temos uma situação em que criamos um tipo que pode ser String, number, ou undefined:



Após executarmos os testes, obtemos a seguinte saída:

[LOG]: "Teste 1: 30"

[LOG]: "Teste 2: funcionou corretamente"

[LOG]: "Teste 3: Os argumentos são inválidos. Ambos devem ser números ou strings."

No primeiro teste, passamos como parâmetros dois números e tudo foi bem. No segundo teste, passamos duas strings e tudo funcionou corretamente. No terceiro teste, passamos um número e uma string e o programa nos avisou que essa não é uma entrada válida.

Com o conhecimento que adquirimos até aqui, estamos aptos a dar o próximo passo: programar orientado a objetos no TypeScript.

Novamente, é importante que façamos os testes de todos os códigos,

além de realizar modificações para adquirirmos segurança para programar nessa linguagem.



## Funções com TypeScript

Neste vídeo, serão apresentados os principais conceitos sobre uso de funções com TypeScript.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

### Questão 1

Uma das características que distingue o TypeScript de outras linguagens de programação é o suporte para Narrowing. Há situações em que utilizar o Narrowing oferece vantagens. Nesse sentido, selecione a opção que apresenta uma dessas situações:

- A** Eliminar erros de tipos de dados.
- B** Acelerar o desempenho da execução das funções.
- C** Implementar funções compatíveis com o JavaScript.
- D** Garantir a flexibilidade e reutilização de código.
- E** Trabalhar com vetores heterogêneos.

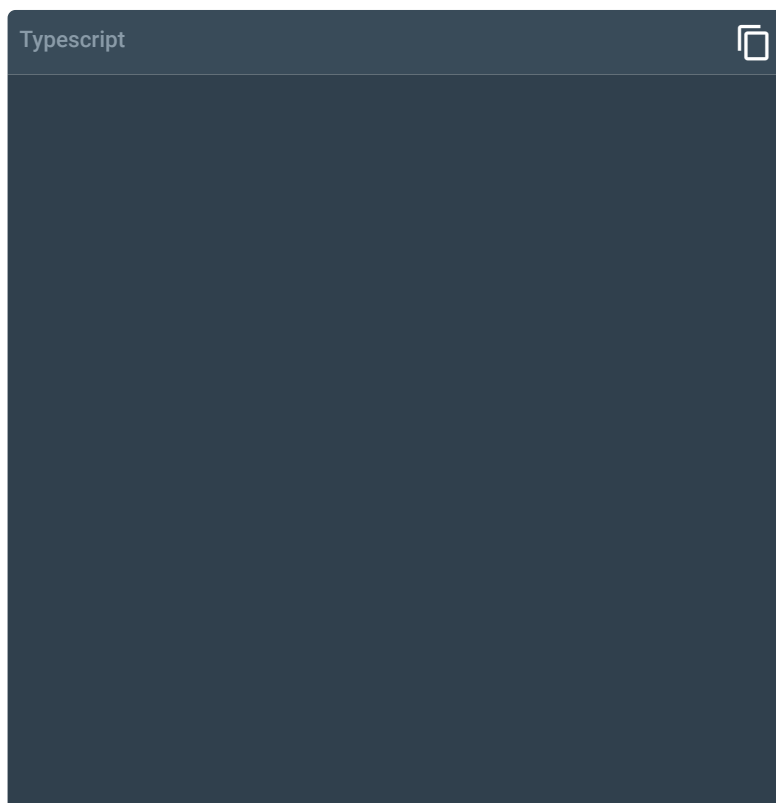
Parabéns! A alternativa D está correta.

A utilização de Narrowing no TypeScript oferece flexibilidade do código, pois é possível verificar qualquer tipo, incluindo tipos personalizados pelo desenvolvedor. Além disso, também auxilia na

reutilização das funções, pois permite que combinemos vários tipos de dados dentro da mesma função.

## Questão 2

Aplicar Narrowing pode ser bastante útil, mas é necessário manter atenção para não criar programas difíceis de entender. Por exemplo, vamos considerar o seguinte trecho de código:



Nesse sentido, selecione a opção correta a respeito do código ao fazer a chamada como `t(5)`:

**A**

Vai retornar "o resultado é: 34".

**B**

Vai retornar "o resultado é: 5".

**C**

Vai imprimir "string".

**D**

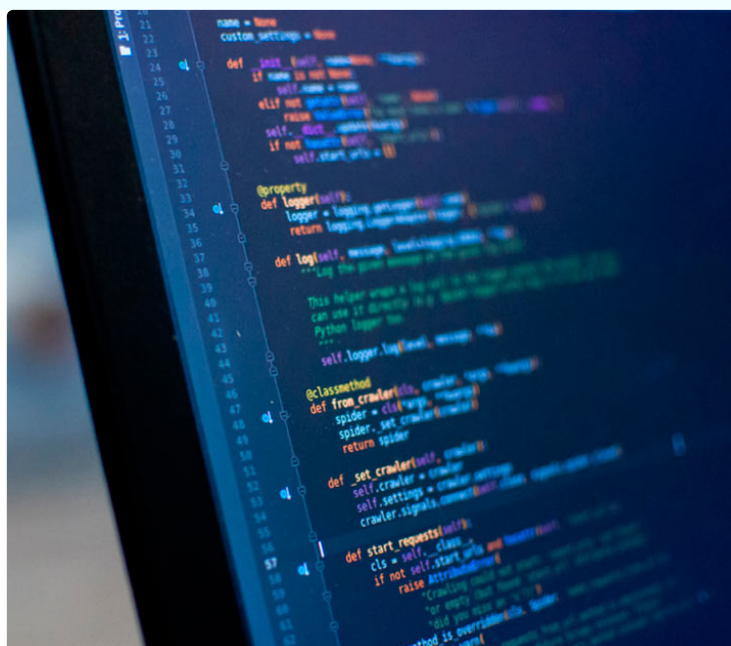
Vai imprimir "Os argumentos são inválidos".

E

Vai retornar "o resultado é: 15".

Parabéns! A alternativa B está correta.

O código do exercício aplica Narrowing por meio do tipo definido pelo desenvolvedor que, no caso, é o alfanumérico. Dentro da função, é feito o teste sobre o tipo do parâmetro. No exemplo dado, o parâmetro é do tipo numérico. Apesar do laço iterativo, não é feita nenhuma alteração do valor do parâmetro e é ele que é utilizado para retorno, resultando, portanto, na impressão de "o resultado é: 5".



## 4 - Classes com TypeScript

Ao final deste módulo, você será capaz de identificar classes com TypeScript.

# Conceitos de programação orientada a objetos

## O que é programação orientada a objetos?

A programação orientada a objetos (POO) é um paradigma de desenvolvimento de software que tem como elemento central a utilização de classes (TYPESCRIPT HANDBOOK, 2022). As classes, por sua vez, são modelos computacionais compostos por campos de dados

chamados de atributos e por funções que são chamadas de métodos. Os métodos são responsáveis por manipular os atributos da classe.

Aqui, deve ficar claro para nós que as classes são modelos. Mas para trabalharmos no programa, precisamos de instâncias dessas classes. Essas instâncias de classes são chamadas de objetos. O objeto, de fato, é quem gerencia os dados e aciona os métodos que vão manipulá-los.



Já faz alguns anos que a POO é o principal paradigma de desenvolvimento de software que existe. Muitas linguagens de programação são baseadas em POO, sendo que a mais conhecida é o Java. O TypeScript também dá suporte para esse tipo de programação e, de fato, é uma boa prática fazer uso dela, pois o programa fica muito mais legível e fácil de manter.

## Propriedades

Vimos que o elemento central da POO é a utilização de classes. Outro importante conceito é o de propriedades da POO. Por meio da aplicação delas, podemos **garantir restrição de acesso, reusabilidade de código e adaptabilidade de execução**. As três principais propriedades de POO são:

### Encapsulamento

- Privado: membros desse tipo só podem ser acessados dentro da classe em que foram definidos;
- Protegido: membros protegidos podem ser acessados dentro da classe em que foram definidos, como também pelas classes herdeiras de uma superclasse;
- Público: por padrão, todos os membros de uma classe no TypeScript são públicos. Os membros públicos podem ser acessados em qualquer lugar sem nenhum tipo de restrição.

### Herança



Permite-nos herdar todos os métodos e atributos de outra classe.

### Polimorfismo



Podemos ter métodos com mesmos nomes, mas com comportamentos distintos. O modo como o método vai se comportar dependerá de como fazemos chamadas para ele.

A seguir, vamos ver exemplos de como utilizar essas propriedades na prática.

## As propriedades da POO

### Encapsulamento

Para aplicar o encapsulamento, o TypeScript fornece três tipos de modificadores que são **private**, **protected** e **public**, que correspondem, respectivamente, aos tipos de acesso privado, protegido e público. A seguir, apresentamos um exemplo que faz uso na prática desses modificadores de acesso:

TypeScript







No final do código, definimos um objeto “pessoa” como uma classe “Pessoa”. Apesar de os nomes serem iguais, o TypeScript os diferencia, pois é sensível à utilização de letras maiúsculas e minúsculas. Esse código tem vários aspectos interessantes, entre os quais destacamos:



Utilização dos modificadores `private`, `protected` e `public`.



Aplicação de arrow functions para definir os métodos da classe.



Passagem de valores para os atributos da classe por meio do construtor.

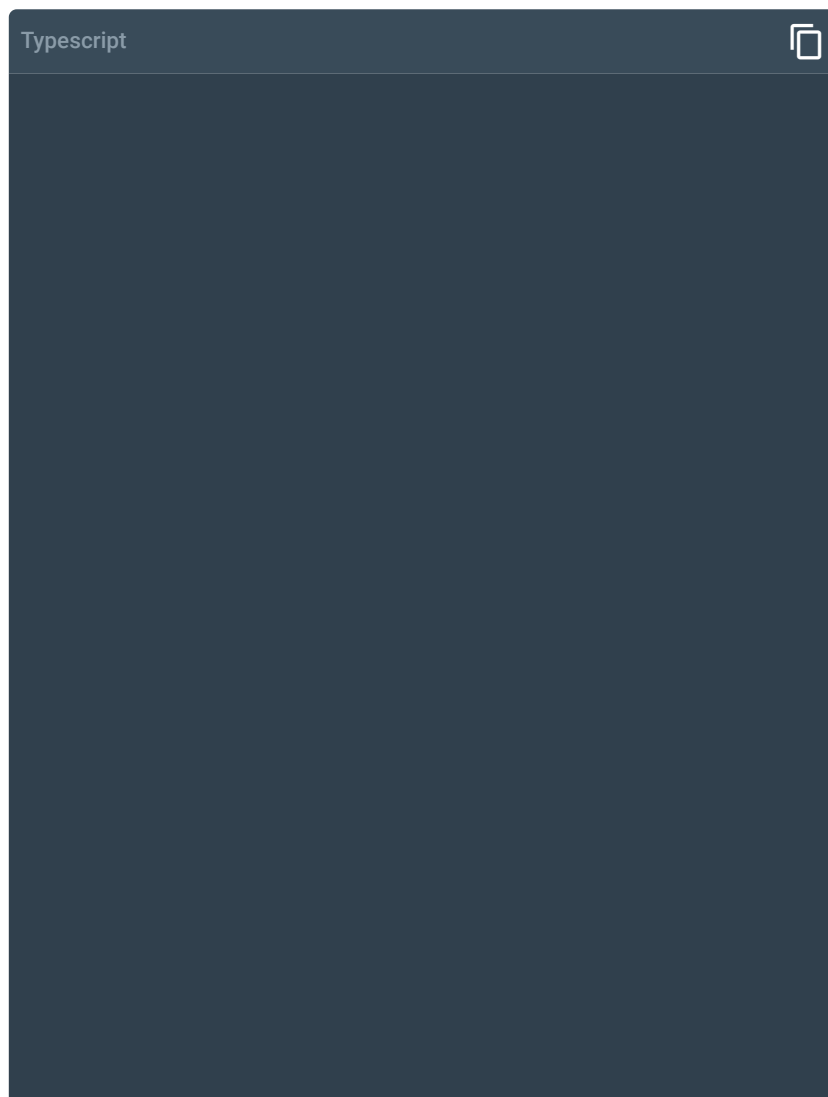
O construtor é o método responsável por estabelecer o comportamento inicial dos objetos.

Observe a saída resultante da execução do programa:

[LOG]: "Dados da pessoa: nome = Teste, endereço = rua de pedra"

## Herança

Aqui, vamos apresentar um exemplo prático da propriedade de herança. Vamos criar duas classes: `PessoaFisica` e `PessoaJuridica`. Ambas herdam as características da classe `Pessoa` que apresentamos no exemplo anterior. Para indicar a herança, o TypeScript utiliza a palavra-chave `extends`. Veja a seguir o exemplo de herança:



Os principais destaques desse exemplo são:

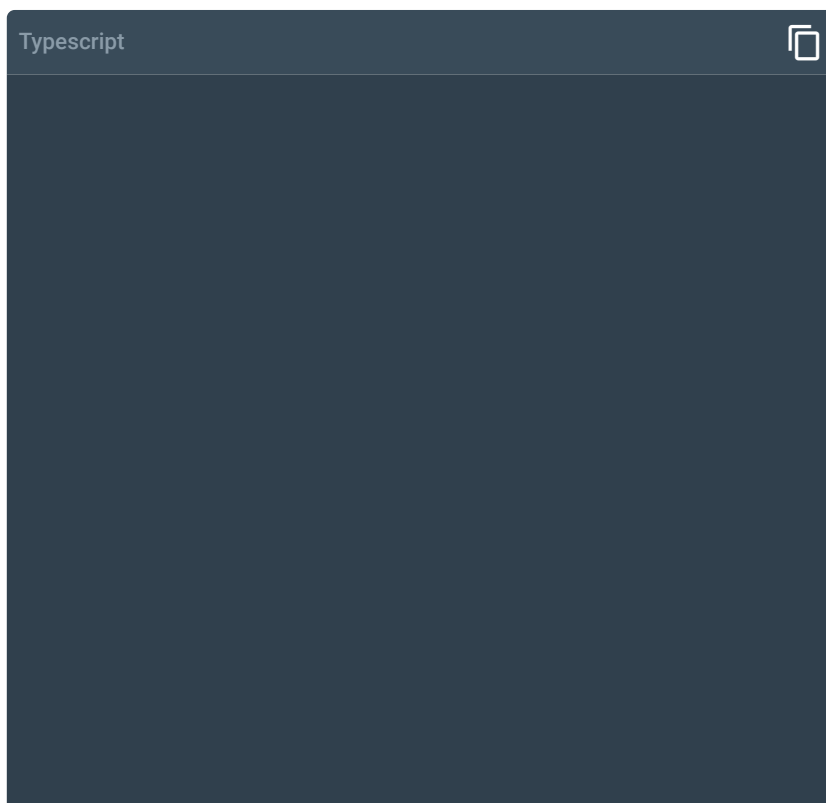
- A utilização da palavra-chave `extends` para fazer a herança das classes;
- A implementação do método `get_dados` tanto na classe `PessoaFisica`, como na classe `PessoaJuridica` com a manipulação de diferentes dados.

Veja as saídas da execução do código:

```
[LOG]: "Dados da pessoa: nome = Teste, endereço = rua de pedra"
[LOG]: "Dados da pessoa: nome = Teste, endereço = rua de pedra"
[LOG]: "Dados da pessoa física: nome = Isaac Newton, endereço = rua
das maçãs, cpf = 100001811-82" [LOG]: "Dados da pessoa jurídica: nome
= TypeScript S.A., endereço = rua dos tipos, cpf = 921.0001-07"
```

## Polimorfismo

A última propriedade que vamos estudar é o polimorfismo. No exemplo que vamos apresentar, a classe “LinguaPortuguesa” herda as características da classe “LinguaLatina”. Nas duas classes, implementamos o método “exibir\_informacao()”, mas o resultado gerado pela execução do programa é diferente, devido ao escopo no qual o método é implementado. A seguir, mostramos o exemplo completo:



Cuja saída é dada por:

```
[LOG]: "Esta classe contém informações sobre a língua latina" [LOG]:  
"Esta classe contém informações sobre a língua portuguesa"
```

Na saída, podemos perceber claramente que os métodos executados foram diferentes.

## Classe genérica

### Aspectos conceituais da classe genérica

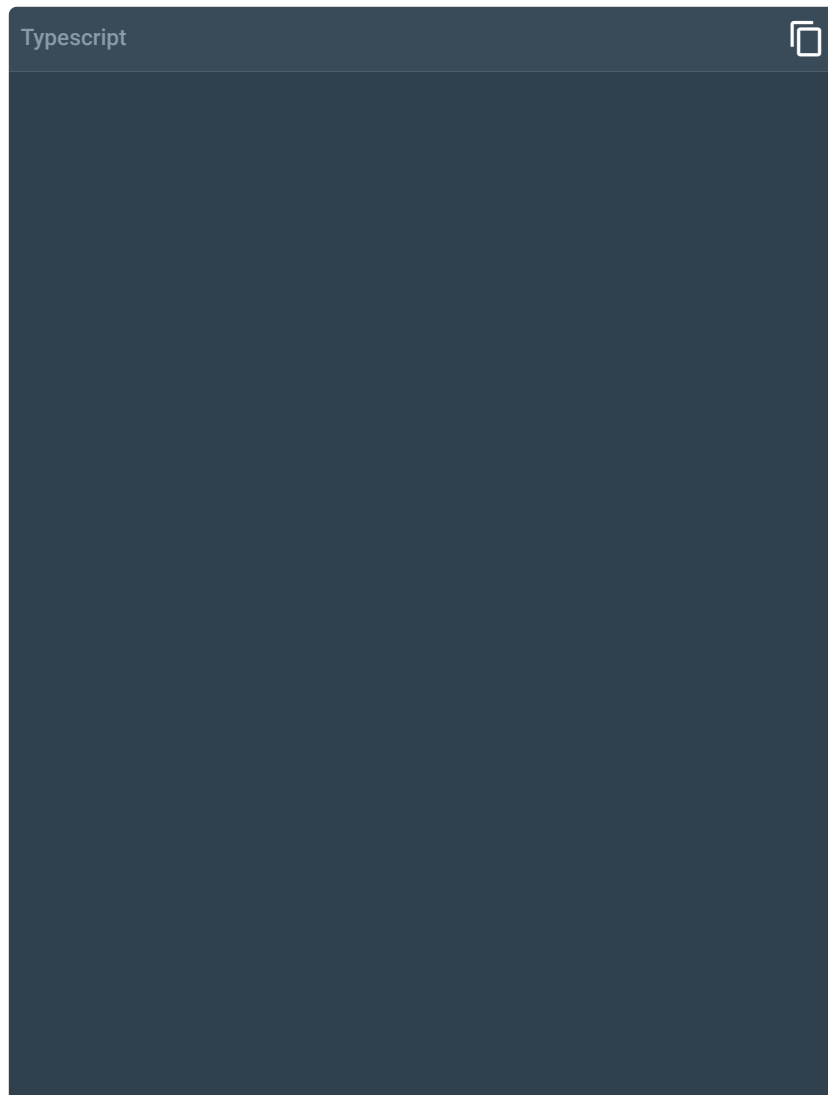
O TypeScript suporta a utilização de classes genéricas.

**Classes genéricas são estruturas lógicas que generalizam comportamentos sem ficar vinculadas a um tipo específico de dado.**

São úteis, especialmente, para a construção de componentes. Para utilizá-las no código, é necessário especificar o parâmetro de tipo genérico entre os símbolos "<" (menor que) e ">" (maior que) após o nome da classe. Uma classe genérica pode ter atributos genéricos, como métodos.

## Um exemplo de classe genérica

Agora, apresentamos um exemplo que utiliza uma classe genérica. Nesse caso, o nome da classe genérica é "Estudante" e os parâmetros genéricos são "U" e "T". Abaixo, está o código que utiliza a classe genérica:



Devemos notar que os tipos utilizados dentro da classe são definidos apenas quando um objeto é instanciado. Abaixo, apresentamos a saída da execução do código:

[LOG]: "Id = 7709, Nome = Isaac Newton"

[LOG]: "Id = 9903, Nome = Coraline"

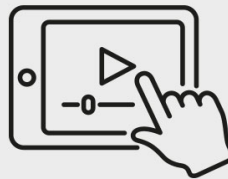
Com isso, concluímos o nosso estudo sobre a linguagem TypeScript. Apesar de já termos destacado em vários momentos a importância de executar esses códigos, voltamos a reforçar que a melhor forma de aprender uma linguagem de programação é executando os exemplos e, a partir deles, fazer modificações e forçar situações que possam levar a erros, pois, na prática, situações desse tipo vão acontecer com frequência e a melhor forma de estar preparado para tratá-las é já ter passado por casos semelhantes. Bons Estudos!



## Classes com TypeScript

Neste vídeo, serão apresentados os conceitos e práticas para desenvolver Classes com TypeScript.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

### Questão 1

Uma das características que distinguem o TypeScript do JavaScript é o suporte à programação orientada a objetos (POO). A POO é um paradigma de programação muito importante e que ganhou muita popularidade por meio das linguagens de programação como C++, C# e Java. Nesse sentido, selecione a opção que apresenta algumas das vantagens de utilizar a POO no TypeScript:

A

Qualquer código no TypeScript deve ser desenvolvido com POO.

B

Suporte para utilizar arrow functions nos métodos das classes.

C

As classes aceleram o processamento de um código.

D

A POO é a técnica mais moderna de desenvolvimento de software.

E

Reutilização e flexibilidade de código.

**Parabéns! A alternativa E está correta.**

A utilização de POO no TypeScript auxilia na depuração mais fácil do código, na reutilização dele por meio da propriedade de herança, aumenta a flexibilidade por meio do uso do polimorfismo e é mais eficaz na resolução de problemas devido à representação que uma classe oferece por meio de atributos e métodos.

## Questão 2

A utilização de programação orientada a objetos no TypeScript permite combinar atributos e métodos em uma unidade lógica chamada de classe. Por exemplo, vamos considerar o código seguinte:

Typescript



Nesse sentido, selecione a opção correta a respeito da implementação da classe Conta:

**A**

Para que a classe funcione corretamente, é necessário implementar o construtor.

**B**

A execução do comando `c.getSaldo()` retornará o valor 20.

**C**

Como a classe não possui construtor, o atributo saldo recebe o valor inicial igual a 0 (zero).

**D**

O método `setDeposito` retorna um valor igual a 20.

**E**

É necessário usar o modificador `public` antes do método `getSaldo`.

**Parabéns! A alternativa B está correta.**

A classe está implementada corretamente, e a chamada do método `getSaldo` vai retornar o valor 20, pois o atributo `saldo` recebeu o valor 10 logo que o objeto `c` foi instanciado. Quando o método construtor não é implementado explicitamente, o TypeScript utiliza uma implementação implícita do construtor.

## Considerações finais

Ao longo deste trabalho, estudamos sobre a linguagem de programação TypeScript. Vimos também as semelhanças e diferenças que ela possui em relação ao JavaScript, além de estudarmos funções, Narrowing e programação orientada a objetos.

Os recursos que o TypeScript oferece nos permite desenvolver aplicações Web tanto para o BackEnd como para o FrontEnd por meio da implementação de boas práticas de programação, que implicam



códigos mais seguros, flexíveis e adequados para receber manutenção. Especialmente no ambiente heterogêneo da Web, essas características são muito importantes, pois as aplicações tendem a evoluir rapidamente e a aumentar a sua complexidade.

É muito importante que todos os exemplos que usamos ao longo do nosso estudo sejam submetidos a testes. Por meio desses testes e de modificações dos códigos, podemos ter uma compreensão melhor de como funcionam os recursos da linguagem e quais são as suas limitações. Além disso, podemos nos deparar com situações mais complexas e aprendermos a resolvê-las por meio da leitura da documentação oficial e fóruns na Web.

Neste podcast, serão apresentados os principais aspectos da linguagem de programação.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.



## Explore +

Acesse o site oficial do TypeScript, pesquise por “download” e aprenda como instalar o TypeScript na sua máquina local por meio de uma sequência de passos.

Vá ao site oficial do nodejs e procure por “Quick Start”. Lá, você encontrará diversos exemplos de JavaScript e TypeScript que vão lhe ajudar a aprofundar mais nessas linguagens de programação.

# Referências

ABREU, L. **Typescript**: o JavaScript moderno para criação de aplicações. Lisboa: FCA, 2017.

FREEMAN, A. **Essential TypeScript**. Springer, 2019.

TYPESCRIPT HANDBOOK. Typescriptlang. Publicado em: 24 jun. 2022.  
Consultado na Internet em: 27 jun. 2022.



## Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.



Download material

O que você achou do conteúdo?



Relatar problema