# Typescript

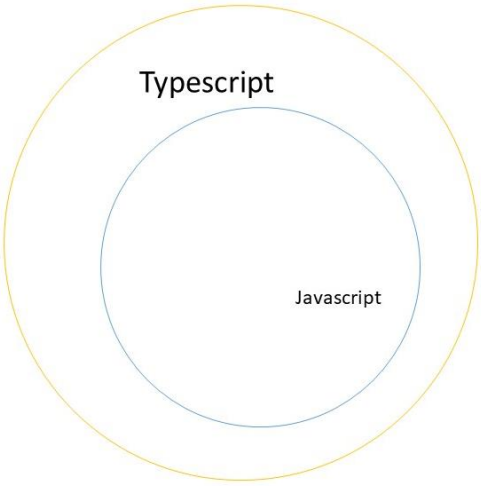
## Visão geral

O JavaScript foi adotado como uma linguagem de programação para o chamado cliente-side (lado do cliente) – ou, ainda – front-end. O desenvolvimento do Node.js também marcou o JavaScript como uma tecnologia emergente do lado do servidor (sever-side) – ou, ainda – back-end middleware. No entanto, à medida que o código JavaScript cresceu, ficou relativamente mais “confuso”. Essa percepção dificulta a manutenção e a reutilização do código. Além disso, sua “falha” em adotar os recursos de Orientação a objetos (OOP), forte verificação de tipagem de dado e verificação de erro em tempo de compilação impede que o JavaScript seja bem-sucedido, naquilo que se conheço como nível corporativo, como uma tecnologia completa do servidor (servr-side). **O TypeScript** foi apresentado para preencher essa lacuna.

## O que é o TypeScript?

Por definição, "TypeScript é JavaScript para desenvolvimento de aplicações / ou aplicativos em escala".

É uma linguagem compilada, fortemente tipada e orientada a objetos. Foi projetada por **Anders Hejlsberg** (designer do C #) na Microsoft. O TypeScript é uma linguagem e um conjunto de ferramentas que podemos utilizar para trabalhar com projetos web. É, também, considerada um superconjunto de JavaScript compilado em JavaScript. Em outras palavras, o TypeScript é JavaScript, acoplada de recursos adicionais.




## Recursos do TypeScript

**TypeScript é apenas JavaScript** . Começa com JavaScript e termina com JavaScript. Adota os blocos de construção básicos do seu programa a partir do JavaScript. Portanto, você só precisa saber JavaScript para usar o TypeScript. Todo o código TypeScript é convertido em seu equivalente em JavaScript para fins de execução.

**O TypeScript suporta outras bibliotecas JS** . É compilado, pode ser consumido a partir de qualquer código JavaScript. O JavaScript gerado pelo TypeScript pode reutilizar todas as estruturas, ferramentas e bibliotecas JS existentes.

**JavaScript é TypeScript** . Isso significa que qualquer arquivo **.js** válido pode ser renomeado para **.ts** e compilado com outros arquivos TypeScript.

**TypeScript é portavel** . O TypeScript é portável em navegadores, dispositivos e sistemas operacionais. Pode ser executado em qualquer ambiente em que o JavaScript seja executado. Diferentemente de suas contrapartes, o TypeScript não precisa de uma VM dedicada ou de um ambiente de tempo de execução específico para executar.

### TypeScript e ECMAScript

A especificação ECMAScript é uma especificação padronizada de uma linguagem de script. Existem seis edições do ECMA-262 publicadas. A versão 6 da norma possue o codinome "Harmony". O TypeScript está alinhado com a especificação ECMAScript6.


TypeScript adota seus recursos básicos de linguagem da especificação ECMAScript5, ou seja, a especificação oficial para JavaScript. Os recursos da linguagem TypeScript, como módulos e orientação baseada em classe, estão alinhados com a especificação EcmaScript 6. Além disso, o TypeScript também inclui recursos como genéricos e anotações de tipo que não fazem parte da especificação EcmaScript6.

### Por que usar o TypeScript?

O TypeScript é superior a seus outros equivalentes, como as linguagens de programação CoffeeScript e Dart, de uma maneira que o TypeScript é um JavaScript “estendido”. Por outro lado, linguagens como Dart, CoffeeScript são novas linguagens em si mesmas e exigem um ambiente de execução específico.

Os benefícios do TypeScript incluem:

- **Compilação** - JavaScript é uma linguagem interpretada. Portanto, ele precisa ser executado para testar se é válido. Isso significa que você escreve todos os códigos apenas para não encontrar saída, caso haja um erro. Portanto, você precisa passar horas tentando encontrar erros no código. O ***transpilador*** TypeScript fornece o recurso de verificação de erros. O TypeScript irá compilar o código e gerar erros de compilação, se encontrar algum tipo de erro de sintaxe. Isso ajuda a destacar erros antes da execução do script.

- **Forte Static Typing** - JavaScript não é fortemente tipado. O TypeScript é fornecido com um sistema estático opcional de typing e inferência de tipos através do TLS (TypeScript Language Service - Serviço de Linguagem TypeScript). O tipo de uma variável, declarada sem definição de tipo, pode ser inferida pelo TLS com base em seu valor.
- O TypeScript **suporta definições de tipo** para bibliotecas JavaScript existentes. O arquivo de definição TypeScript (com extensão **.d.ts** ) fornece definição para bibliotecas JavaScript externas. Portanto, o código TypeScript pode conter essas bibliotecas.
- O TypeScript **suporta** conceitos de **programação orientada a objetos** , como classes, interfaces, herança etc.

## Componentes do TypeScript

No coração, o TypeScript possui os três componentes a seguir -

- **Idioma** - É composto pela sintaxe, pelas palavras-chave e pelas anotações de tipo.
- **O compilador TypeScript** - O compilador TypeScript (tsc) converte as instruções escritas em TypeScript em seu equivalente em JavaScript.
- **O TypeScript Language Service - serviço de linguagem TypeScript** - O "Serviço de linguagem" expõe uma camada adicional ao redor do pipeline do compilador principal que são aplicativos semelhantes a editor. O serviço de idioma suporta o conjunto comum de operações típicas de um editor, como conclusão de instruções, ajuda de assinatura, formatação e estrutura de códigos, colorização, etc.

## Configuração do ambiente local

O TypeScript é uma tecnologia de código aberto (Open Source). Pode ser executado em qualquer navegador, host e sistema operacional. Você precisará das seguintes ferramentas para escrever e testar um programa Typescript:

### Um editor de texto

O editor de texto ajuda você a escrever seu código fonte. Exemplos de alguns editores aceito na maioria dos sistemas operacionais: Notepad, Notepad ++, Visual Studio Code, entre outros.

Os arquivos de origem geralmente são nomeados com a extensão **.ts**

### O compilador TypeScript

O compilador TypeScript é ele próprio um arquivo **.ts** compilado no arquivo JavaScript (.js). O TSC (TypeScript Compiler) é um compilador fonte a fonte (transcompiler / transpiler).

O *TSC* gera uma versão JavaScript do arquivo ***.ts*, que é** passado para ele. Em outras palavras, o TSC produz um código-fonte JavaScript equivalente - sempre a partir do arquivo Typescript fornecido, como uma entrada, para ele –tsc . Este processo é denominado como ***transpilação***.

No entanto, o compilador rejeita qualquer arquivo JavaScript bruto que seja passado. Para o compilador podemos passar apenas arquivos ***ts*** ou ***.d.ts****.*

Para instalar o *typescript* basta inserir o comando:

```
npm install typescript
```

Também é possível gerar um arquivo *tsconfig.json* – caso seja necessário fazer algum tipo de configuração adicional. Para gerar est arquivo, basta inserir a instrução abaixo em seu prompt de comando (no caminho da pasta onde se encontram ou onde salvar os projetos *typescript*):

```
tsc –init
```

Se tudo correr bem, o resultado do arquivo tsconfig.json é este indicado abaixo:


## Sintaxe básica

Sintaxe define um conjunto de regras para escrever programas. Toda especificação de idioma define sua própria sintaxe. Um programa TypeScript é composto por -

- Módulos
- Funções
- Variáveis
- Declarações e expressões
- Comentários

### Compilar e executar um programa TypeScript

Vamos ver como compilar e executar um programa TypeScript usando o Visual Studio Code. Siga as etapas abaixo:

**Etapa 1** - Salve o arquivo com extensão .ts. Vamos salvar o arquivo como teste.ts. O editor de código marca erros, caso eles existam, enquanto você o salva.

**Etapa 2** - Clique com o botão direito do mouse no arquivo TypeScript na opção Arquivos de Trabalho no Painel de Exploração do VS Code. Selecione Abrir na opção Prompt de Comando (Command Prompt) ou Abrir no Terminal (Open Terminal).

**Etapa 3** - Para compilar o arquivo, use o seguinte comando na janela do terminal.

```
tsc teste.ts
```

**Etapa 4** - O arquivo é compilado para teste.js. Para executar o programa – agora, convertido no formato *.js* -, digite a seguinte instrução no terminal que você abriu:

```
node teste.js
```

Agora, implemente o código abaixo como se segue:

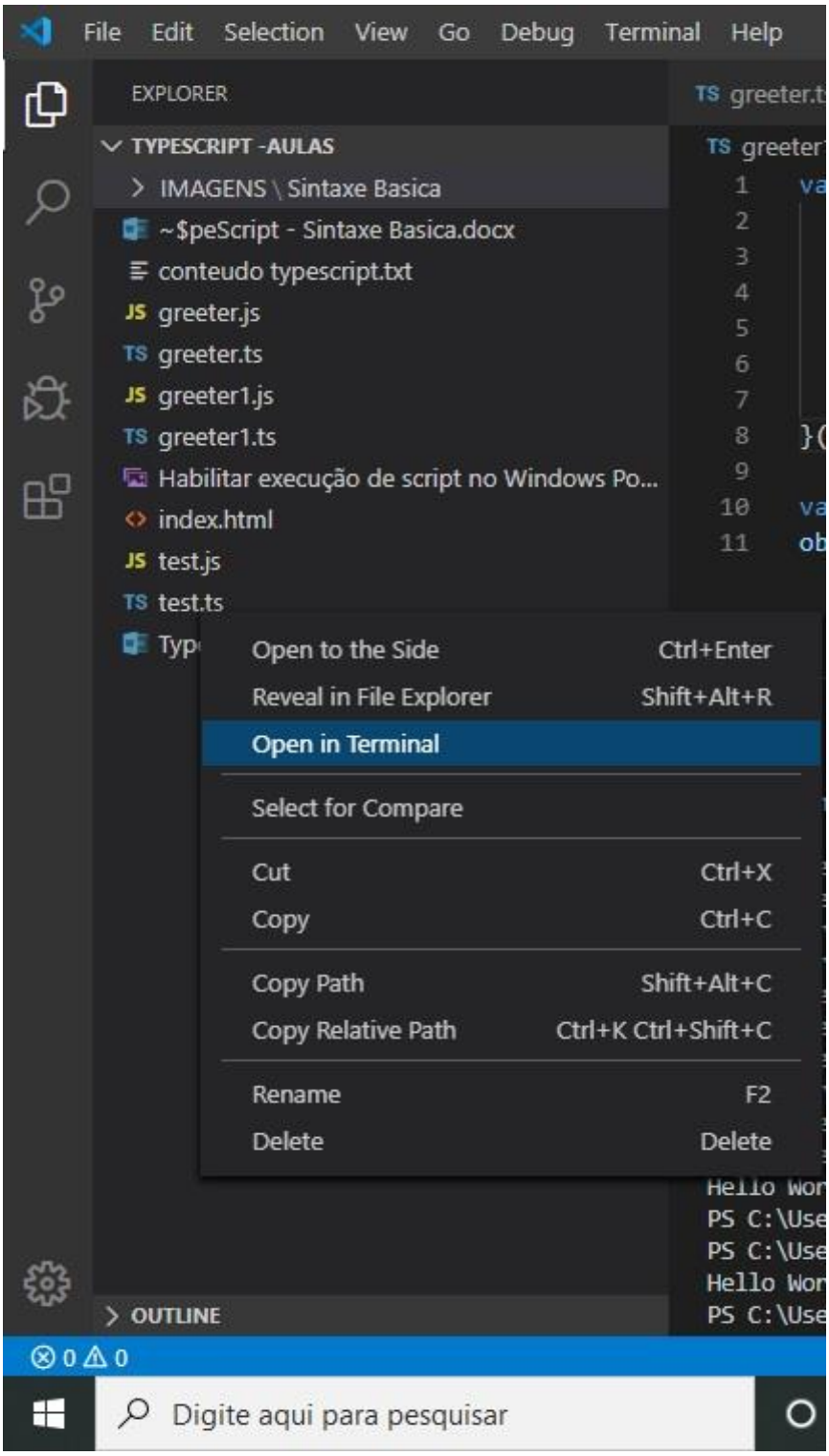
**Seu primeiro código TypeScript**

```
var mensagem:string = "Hello Mundo"
console.log(mensagem)
```

Ao compilar, ele gera o seguinte código JavaScript.

```
var mensagem = "Hello Mundo";
console.log(mensagem);
```

- A linha 1 declara uma variável pela mensagem de nome. Variáveis são um mecanismo para armazenar valores em um programa.
- A linha 2 imprime o valor da variável no prompt. Aqui, console refere-se à janela do terminal. O *log da* função *()* é usado para exibir o texto na tela.

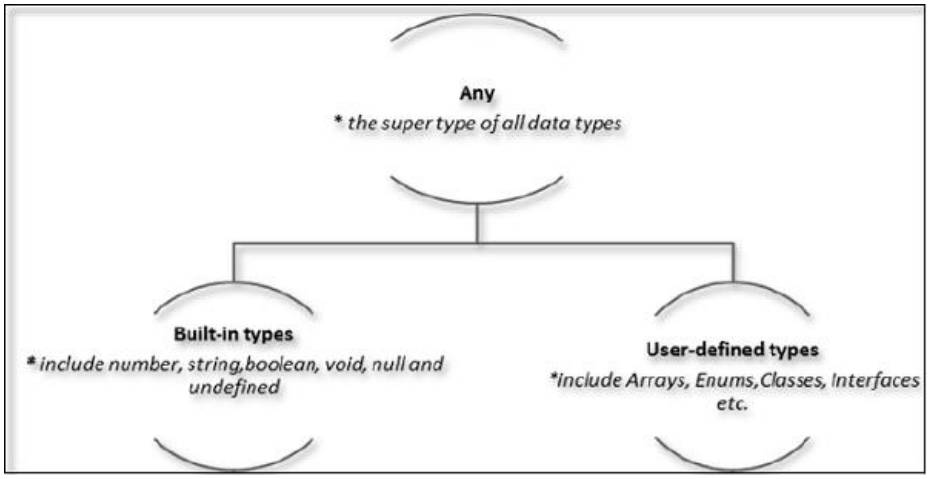



## Tipos

O sistema de tipos representa os diferentes tipos de valores suportados pela linguagem *TypeScript.* O sistema de tipos verifica a validade dos valores fornecidos, antes de serem armazenados ou manipulados pelo programa. Isso garante que o código se comporte conforme o esperado. O *Type System*

também permite dicas de código mais avançadas e documentação automatizada.

O *TypeScript* fornece tipos de dados como parte de seu sistema de tipos opcional. A classificação do tipo de dados é a seguinte:




### Tipo any

O *any data type* é o supertipo de todos os tipos no *TypeScript*. Denota um tipo dinâmico. Usar o tipo *any* é equivalente a desativar a verificação de tipo para uma variável.

### Null (Null) e Undefined (undefined) — São iguais?

Os tipos de dados ***null*** e ***undefined*** são frequentemente uma fonte de confusão. O null e undefined não podem ser usados para referenciar o tipo de dado de uma variável. Eles só podem ser atribuídos como valores a uma variável.

No entanto, *null e undefined não são a mesma coisa*. Uma variável inicializada com undefined significa que a variável não tem valor ou objeto atribuído a ela, enquanto null significa que a variável foi configurada para um objeto cujo valor é undefined.

### Tipos definidos pelo usuário

Tipos definidos pelo usuário incluem enumerations (enums), classes, interfaces, matrizes e tuplas.

## Variáveis

Uma variável, por definição, é "um espaço nomeado na memória" que armazena valores. Em outras palavras, ele atua como um contêiner para valores em um programa. As variáveis TypeScript devem seguir as regras de nomenclatura JavaScript:

- Os nomes de variáveis podem conter alfabetos e dígitos numéricos.
- Eles não podem conter espaços e caracteres especiais, exceto o sublinhado (_) e o sinal de dólar ($).

- Os nomes de variáveis não podem começar com um dígito.

Uma variável deve ser declarada antes de ser usada. Use a palavra-chave **var** para declarar variáveis.

### Declaração variável no TypeScript

A sintaxe do tipo para declarar uma variável no TypeScript é incluir dois pontos (:) após o nome da variável, seguido por seu tipo. Assim como no JavaScript, usamos a palavra-chave **var** para declarar uma variável.

Ao declarar uma variável, você tem quatro opções:

- Declare seu tipo e valor em uma instrução.


- Declara o tipo, mas sem valor. Nesse caso, a variável será definida como undefined.


- Declara seu valor, mas nenhum tipo. O tipo de variável será definido para o tipo de dado do valor atribuído.


- Não declara nem o valor nem o tipo. Nesse caso, o tipo de dado da variável será any e será inicializado como indefinido.


A tabela a seguir ilustra a sintaxe válida para a declaração de variável, conforme discutido acima:

| Sintaxe e descrição da declaração variável |
| --- |
| **var nome:string = ‘Ivonete”**<br>A variável armazena um valor do tipo *string* |
| **var nome:string;**<br>A variável é uma variável string. O valor da variável é definido como *undefined* por padrão |
| **var nome = ”Ivonete”** |

| | |
|---|---|
| *string* | O tipo da variável é inferido a partir do tipo de dado do valor. Aqui, a variável é do tipo |
| | **var nome;**<br>O tipo de dado da variável é *any*. Seu valor é definido como indefinido por padrão. |

### Inferência de tipo

Dado o fato de que o TypeScript é fortemente tipado, esse recurso é opcional. O TypeScript também incentiva a digitação dinâmica de variáveis. Isso significa que, o TypeScript incentiva a declaração de uma variável sem um tipo. Nesses casos, o compilador determinará o tipo da variável com base no valor atribuído a ela. O TypeScript encontrará o primeiro uso da variável no código, determinará o tipo para o qual ela foi definida inicialmente e assumirá o mesmo tipo para essa variável no restante do seu bloco de código.

O mesmo é explicado no seguinte *snippet* de código:

**Exemplo**

```
var umNumero = 2; //infeirndo o tipo do dado da variavel
console.log("valor da variavel umNumero é :"+ umNumero);
```

No *snippet* de código acima:

- O código declara uma variável e define seu valor como 2. Observe que a declaração da variável não especifica o tipo de dado. Portanto, o programa usa *typing inferido* para determinar o tipo de dados da variável, ou seja, atribui o tipo do primeiro valor ao qual a variável está configurada. Nesse caso, ***umNumero*** é definido como *type number (tipo number)*.
- Quando o código tenta definir o valor da variável como string, o compilador gera um erro, já que o tipo da variável já está definido como número.

### Criando varáveis com diferentes definções de tipo

Observe o código abaixo:

```
var nome:string = "Chespirito"
var pontuacao:number = 110
var pontuacao2:number = 11.35
var soma = pontuacao + pontuacao2
console.log("nome: "+nome)
console.log("primeira pontuação: "+pontuacao)
console.log("segunda pontuação: "+pontuacao2)
console.log("soma das pontuações: "+soma)
```

Ao compilar, ele gera o seguinte código JavaScript.

```
var nome = "Chespirito";
var pontuacao = 110;
var pontuacao2 = 11.35;
```

```
var soma = pontuacao + pontuacao2;
console.log("nome: " + nome);
console.log("primeira pontuação: " + pontuacao);
console.log("segunda pontuação: " + pontuacao2);
console.log("soma das pontuações: " + soma);
```

A saída do programa acima é dada abaixo -

**nome: Chespirito**

**primeira pontuação: 110**

**segunda pontuação: 11.35**

**soma das pontuações: 121.35**

O compilador TypeScript gerará erros, se tentarmos atribuir um valor a uma variável que não é do mesmo tipo. Portanto, o TypeScript segue a Strong Typing. A sintaxe de Strong Typing garante que os tipos especificados em ambos os lados do operador de atribuição (=) sejam os mesmos. É por isso que o código a seguir resultará em um erro de compilação -

```
var num:number = "typescript"
```

*a instrução acima retornará um erro de compilação*

### Declaração *let*

Para resolver problemas com declarações ***var***, o ES6 introduziu dois novos tipos de declarações de variáveis no JavaScript, usando as palavras-chave ***let*** e ***const***. O *TypeScript,* sendo um *superconjunto* de *JavaScript*, também suporta esses novos tipos de declarações de variáveis. Observe o código abaixo:

```
var nomeFuncionario = "Miguelito";

ou

let nomeFuncionario = "Miguelito";
```

As declarações *let* seguem a mesma sintaxe que as declarações var. Diferente das variáveis declaradas com var, as variáveis declaradas com *let* possuem um escopo de bloco. Isso significa que o escopo das variáveis *let* é limitado ao seu bloco de contenção. Considere o seguinte exemplo.

### Exemplo

```
let numeroUm:number = 1;
```

```
function declarandoLet() {
    let numeroDois:number = 2;

    if (numeroUm > numeroDois) {
        let numeroTres: number = 3;
        numeroTres++;
        console.log(numeroTres);
    }

    while(numeroUm < numeroDois) {
        let numeroQuatro: number = 4;
        numeroUm++;
        console.log(numeroUm);
    }

    console.log(numeroUm); //OK
    console.log(numeroDois); //OK
    //console.log(numeroTres); //Compiler Error: Cannot fin
d name 'numeroTres'
    //console.log(numeroQuatro); //Compiler Error: Cannot f
ind name 'numeroQuatro'
}

declarandoLet();
```

No exemplo acima, todas as variáveis são declaradas usando *let*. A variável *numeroTres* é declarada no bloco *if*, portanto, seu escopo é limitado ao bloco *if* e não pode ser acessado fora do bloco. Da mesma maneira, *numeroQuatro* é declarado no bloco *while* para que não possa ser acessado fora do bloco. Assim, ao acessar *numeroTres* e *numeroQuatro* fora dos respectivos blocos um erro será exibido no compilador.

O mesmo exemplo com a declaração var é compilado sem erro.

**Exemplo: escopo de variáveis usando var**

```
// criando e declarando variaveis com a palavra let
var numeroUm: number = 1

// criando a função para trabalhar com let
function declarandoLet(){
    var numeroDois: number = 2
    var numeroTres: number = 3
    // bloco if
```

```
    if(numeroUm < numeroDois){
        numeroTres++

    }
    var numeroQuatro:number = 4
    // bloco while
    while(numeroUm < numeroDois){
        numeroQuatro++
        numeroUm++
        }

       // chamando algumas propriedades
    console.log(numeroUm)
    console.log(numeroDois)
    console.log(numeroTres)
    console.log(numeroQuatro)
}
declarandoLet()
```

**Vantagens de usar let em relação a var**

As variáveis com escopo no bloco não podem ser lidas ou gravadas antes de serem declaradas.

No exemplo acima, o compilador TypeScript emitirá um erro se usarmos variáveis antes de declará-las usando let, enquanto que não ocorrerá um erro ao usar variáveis antes de declará-las usando var.

**Permitir que variáveis não possam ser declaradas novamente**

O compilador *TypeScript* emitirá um erro quando variáveis com o mesmo nome (com distinção entre maiúsculas e minúsculas) são declaradas várias vezes no mesmo bloco usando *let*.

**Exemplo: Várias variáveis com o mesmo nome**

```
let numero:number = 1; // OK
let Numero:number = 2; // OK

let numero:number = 5;// Compiler Error: Cannot redeclared
block-scoped variable 'numero'
```

```
let Numero:number = 6;// Compiler Error: Cannot redeclared
block-scoped variable 'Numero'
```

No exemplo acima, o compilador *TypeScript* trata os nomes das variáveis como distinção entre maiúsculas e minúsculas; No entanto, ocorrerá um erro para as variáveis com o mesmo nome e caso.

**Declaração Const**

As variáveis podem ser declaradas usando *const* semelhante às declarações *var* ou *let*. A const torna uma variável uma constante em que seu valor não pode ser alterado. As variáveis *const* possuem as mesmas regras de escopo que as variáveis let.

**Exemplo: Variável Const**

```
const numeroConst:number = 100;
numeroConst = 200; //Compiler Error: Cannot assign to 'nume
roConst' because it is a constant or read-only property
```

As variáveis *const* devem ser declaradas e inicializadas em uma única instrução. Declaração e inicialização separadas não são suportadas; esse tipo de variável permite alterar as subpropriedades de um objeto, mas não a estrutura do objeto. Observe o código abaixo:

**Exemplo: const Object**

```
// implementando a primeira constante
const idParticipantes = {
    participanteA: 1,
    participanteB: 2,
    participanteC: 3,
    participanteD: 4
}
console.log(idParticipantes)
// acessar uma propriedade do objeto const
idParticipantes.participanteC = 167

// exibindo o valor alterado da propriedade do objeto
console.log(idParticipantes)

console.log(idParticipantes.participanteC)
```

Mesmo se a tentativa for de alteração da estrutura do objeto, o compilador apontará esse erro. Observe o código abaixo:

```
const idParticipantes = {
    participanteA : 1,
    participanteB : 2,
    participanteC : 3,
    participanteD : 4
};
idParticipantes.participanteC = 105; // OK

idParticipantes = {     //Compiler Error: Cannot assign to
playerCodes because it is a constant or read-only
    participanteA : 50,
    participanteB : 10,
    participanteC : 13,
    participanteD : 20
};
```

Isso não deve ser confundido com a ideia de que os valores a que se referem são imutáveis. Observe o próximo bloco de código:

```
let vidaGatos = 9

//criando o objeto literal
const dadosGato = {
    nome: 'Mila Burns',
    qtdeVidas: vidaGatos
}
// exibir o objeto
console.log(dadosGato)

// aplicando modificações nas propriedades do objeto
dadosGato.nome = 'Frajola Jenkins'
dadosGato.qtdeVidas = 67

// exibindo as alterações nas propriedades do objeto
console.log('Quantidade de vidas de ', dadosGato.nome, 'é i
gual a ', dadosGato.qtdeVidas)

// exibindo o objeto
console.log(dadosGato)
```

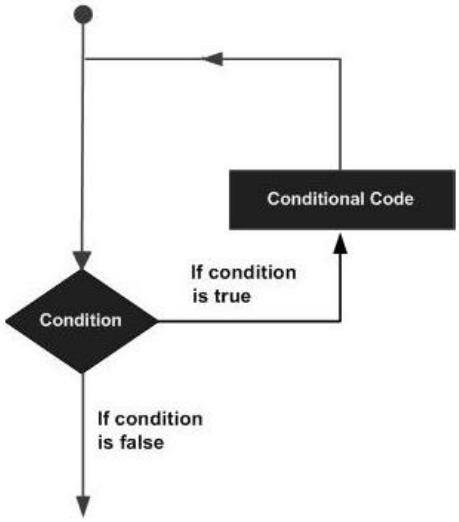
A menos que se tome medidas específicas para evitá-lo, o estado interno de uma *const* ainda pode ser modificado. Felizmente, o *TypeScript* permite especificar os membros de um objeto *readonly*.

## Loops

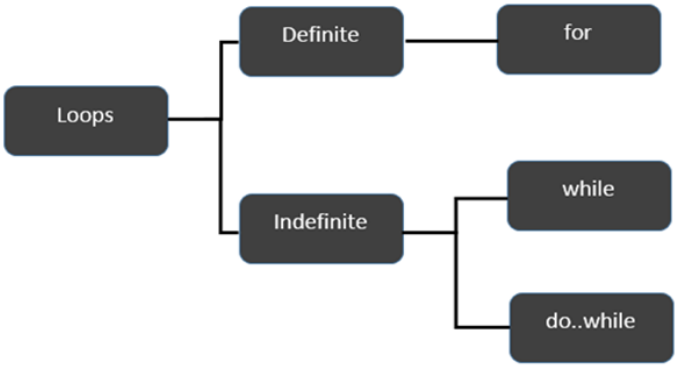
Quando o cenário necessita da aplicação de loops (laços) significa que, em geral, as instruções são executadas sequencialmente: A primeira instrução em uma função é executada primeiro, seguida pela segunda e assim por diante.

Todas as linguagens de programação fornecem várias estruturas de controle que permitem caminhos de execução mais complexos ou menos complexos.

Uma declaração de loop nos permite executar uma declaração ou grupo de declarações várias vezes. Dada a seguir é a forma geral de uma instrução de loop na maioria das linguagens de programação.




O *TypeScript* fornece diferentes tipos de loops para lidar com os requisitos de loop. A figura a seguir ilustra a classificação dos loops:




### Loop definido - *for*

### O loop for ... in

Outra variação do *de* ciclo é o loop *for ... in*. O loop *for… in* pode ser usado para iterar sobre um conjunto de valores, como no caso de um array (matriz) ou de uma tupla. A sintaxe para o mesmo é fornecida abaixo -

O loop for ... in é usado para iterar através de uma lista ou coleção de valores. O tipo de dados *val* aqui deve ser string ou qualquer. A sintaxe do loop ***for..in*** é a seguinte:

**Sintaxe**

```
for (var x in colecao) {
   // declaracoes
}
```

Observe o exemplo a seguir:

**Exemplo**

```
// vamos criar nosso primeiro loop for
var y: any // essa é nossa variavel iteradora

// criando a coleção de dados
var z: any = 'a b c'

// criando o loop
for(y in z){
    console.log(z[y])
}
```

Ao compilar, ele gera o seguinte código JavaScript:

```
// vamos criar nosso primeiro loop for
var y; // essa é variavel iteradora
// criando a coleção de dados
var z = 'a b c';
// criando o loop
for (y in z) {
    console.log(z[y]);
}
```

A execução produzirá a seguinte saída:

```
a
b
c
```

**Loop for ... of**

O loop *for ... of* retorna elementos de uma coleção, por exemplo, array, lista ou tupla e, portanto, não é necessário usar o loop for tradicional.

Veja o exemplo abaixo:

```
// este é o loop for of
var y: any
//criando a coleção de dados
```

```
let umArray: Array<number> = [380, 1067, 2087, 4780, 6750]
//criando o loop
for(y of umArray){
    console.log(y)
}
```

O código acima exibirá a seguinte saída:

```
380

1067

2087

4780

6750
```

### Loop indefinido - while

Um loop indefinido é usado quando o número de iterações em um loop é indeterminado ou desconhecido.

Loops indefinidos podem ser implementados usando:

| Loops e descrição |
| --- |
| ***Loop while***<br>O loop *while* executa as instruções sempre que a condição especificada é avaliada como verdadeira. |
| ***do....while***<br>O loop *do ... while* é semelhante ao loop while, exceto que o loop do ... while não avalia a condição pela primeira vez que o loop é executado. |

### Loop While

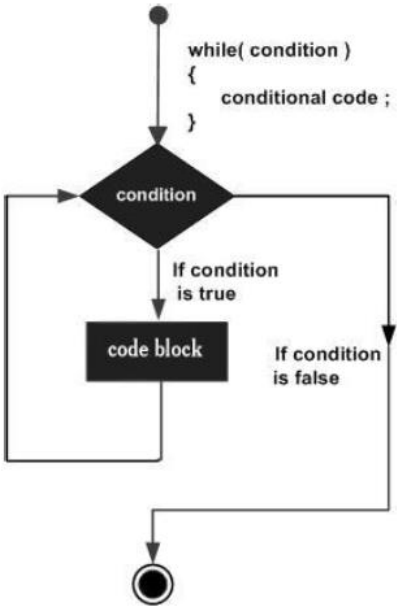
O *Loop while* executa as instruções cada vez que a condição especificada for avaliada como *True*. Em outras palavras, o loop avalia a condição antes da execução do bloco de código.

### Sintaxe

```
while(expressão_teste) {
```

```
    // declaracoes
}
```

**Diagrama de fluxo**




**Exemplo: loop While**

Observe o código abaixo. A premissa é a mesma do loop for – calculo do fatorial – agora, executado com o *loop while.*

```
console.log("O fatorial é uma operação muito importante par
a o estudo e desenvolvimento da análise combinatória")
console.log("Conhecemos como fatorial de um número natural
a multiplicação desse número por seus antecessores com exce
ção do zero")
// primeiro loop while
// vamos estabelecer as variaveis
var numero: number = 10
var fatorial: number = 1

// vamos estabelecer o loop while
while(numero >= 1){
    fatorial = fatorial * numero
    numero--
}

console.log('O valor do fatorial de 10 até 1 é : ', fatoria
l)
```

Ao compilar, a execução gera o seguinte código JavaScript:

```
console.log("O fatorial é uma operação muito importante par
a o estudo e desenvolvimento da análise combinatória");
console.log("Conhecemos como fatorial de um número natural
a multiplicação desse número por seus antecessores com exce
ção do zero");
// primeiro loop while
// vamos estabelecer as variaveis
var numero: number = 10
var fatorial: number = 1

// vamos estabelecer o loop while
while(numero >= 1){
    fatorial = fatorial * numero
    numero--

}
console.log('O valor do fatorial de 10 até 1 é : ', fatoria
l)
```

Produz a seguinte saída:

```
O valor do fatorial é da variavel umNumero é:  3628800
```

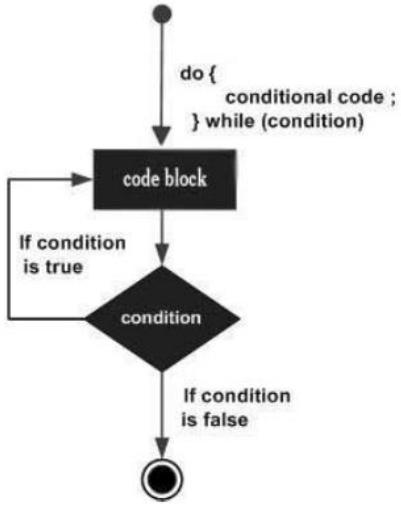
**do... while loop**

O *loop do ... while* é semelhante ao *loop while*, exceto que o *loop do ... while* não avalia a condição pela primeira vez que o loop é executado. No entanto, a condição é avaliada para as iterações subsequentes. Em outras palavras, o bloco de código será executado pelo menos uma vez em um *loop do ... while*.

**Sintaxe**

```
do {
   //declareações
} while(expressão_teste)
```

**Fluxograma**




**Exemplo: do… while**

```
// implementado o loop do...while
var novoNum: number = 13
console.log('Valores encontrados durante a iteração do loop
 do..while')

// estabelecendo o loop
do{
    console.log(novoNum)
    novoNum--
}while(novoNum >= 0)
```

Ao compilar, ele gera o seguinte código JavaScript:

```
// implementado o loop do...while
var novoNum = 13;
console.log('Valores encontrados durante a iteração do loop
 do..while');
// estabelecendo o loop
do {
    console.log(novoNum);
    novoNum--;
} while (novoNum >= 0);
```

O exemplo imprime números de 0 a 13 na ordem inversa.

```
Valores encontrados durante a iteração do loop
do..while
13
12
11
```

```
10
9
8
7
6
5
4
3
2
1
0
```

## Classes

### TypeScript e orientação a objeto

*TypeScript* é *JavaScript* orientado a objetos. A Orientação a Objetos é um paradigma de desenvolvimento de software que segue a modelagem do mundo real. A Orientação a Objetos considera um programa como uma coleção de objetos que se comunicam por meio de um mecanismo chamado métodos. O TypeScript também suporta esses componentes orientados a objetos.

- Objeto - Um objeto é uma representação em tempo real de qualquer entidade. De acordo com Grady Brooch, todo objeto deve ter três recursos -
  - Estado - descrito pelos atributos de um objeto
  - Comportamento - descreve como o objeto agirá
  - Identidade - um valor único que distingue um objeto de um conjunto de objetos semelhantes.
- Classe - Uma classe em termos de POO é um plano para criar objetos. Uma classe encapsula dados para o objeto.
- Método - Métodos facilitam a comunicação entre objetos.

### Criando uma classe

Para criar uma classe em Typescript é necessário iniciar com o uso da palavra-reservada *class* – como na maioria das linguagens de programação. Observe a sintaxe abaixo:

Sintaxe

```
class class_name {
   //class scope
}
```

**Exemplo: TypeScript e orientação a objeto**

```
//vamos criar nosso primeira class typesccript

class Saudacao{
    saudacao():void{
        console.log("Ola mundo a partir do POO TS!")
    }
}

//vamos criar nosso objeto a partir da classe Saudacao
//assim, podemos fazer uso do método saudação()
var obj = new Saudacao();
obj.saudacao();
```

O exemplo acima define uma classe *Saudacao* . A classe possui um método *saudacao()* . O método imprime a sequência "Hello World" no terminal. A palavra-chave ***new*** cria um objeto da classe (obj). O objeto chama o método *saudacao()* .

Ao compilar, ele gera o seguinte código JavaScript.

```
//vamos criar nosso primeira class typesccript
var Saudacao = /** @class */ (function () {
    function Saudacao() {
    }
    Saudacao.prototype.saudacao = function () {
        console.log("Ola mundo a partir do POO TS!");
    };
    return Saudacao;
}());
//vamos criar nosso objeto a partir da classe Saudacao
//assim, podemos fazer uso do método saudacao()
var obj = new Saudacao();
obj.saudacao();
```

A saída do programa acima é dada abaixo:

```
Ola mundo a partir do POO TS!
```

*TypeScript* suporta recursos de programação orientada a objetos, como *classes*, i*nterfaces*, etc. Uma classe em termos de OOP é um modelo para a criação de objetos. Uma classe encapsula dados para o objeto. O *Typescript* oferece suporte embutido para esse conceito chamado classe. O *JavaScript* ES5 ou anterior não suporta classes. O *Typescript* obtém esse recurso do ES6.