



O Manual JS

Índice

- [Índice](#)
- [1. Introdução ao JavaScript](#)
- [2. Um pouco de história](#)
- [3. Apenas JavaScript](#)
- [4. Uma breve introdução à sintaxe do JavaScript](#)
 - [4.1. Espaço em branco](#)
 - [4.2. Sensível a maiúsculas e minúsculas](#)
 - [4.3. Literais](#)
 - [4.4. Identificadores](#)
 - [4.5. Comentários](#)
- [5. Ponto e vírgula](#)
- [6. Valores](#)
- [7. Variáveis](#)
- [8. Tipos](#)
 - [8.1. Tipos primitivos](#)
 - [8.2. Tipos de objeto](#)
- [9. Expressões](#)
- [10. Operadores](#)
 - [10.1. O operador de adição \(+\)](#)
 - [10.2. O operador de subtração \(-\)](#)
 - [10.3. O operador da divisão \(/\)](#)
 - [10.4. O restante operador \(%\)](#)
 - [10.5. O operador de multiplicação \(*\)](#)
 - [10.6. O operador de exponenciação \(**\)](#)
- [11. Regras de precedência](#)

St Adobe Stock

Get 10 Free Images
From Adobe Stock.
Start Now.

ADS VIA CARBON

- [12. Operadores de comparação](#)
- [13. Condicionais](#)
- [13.1. Senão](#)
- [14. Matrizes](#)
- [14.1. Como adicionar um item a uma matriz](#)
- [14.2. Como remover um item de uma matriz](#)
- [14.3. Como unir dois ou mais arrays](#)
- [14.4. Como localizar um item específico na matriz](#)
- [15. Cordas](#)
- [16. Loops](#)
- [16.1. enquanto](#)
- [16.2. para](#)
- [16.3. para... de](#)
- [17. Funções](#)
- [18. Funções de seta](#)
- [19. Objetos](#)
- [19.1. Propriedades do objeto](#)
- [19.2. Métodos de objeto](#)
- [20. Aulas](#)
- [21. Herança](#)
- [22. Programação assíncrona e retornos de chamada](#)
- [23. Promessas](#)
- [24. Assíncrono e Aguarde](#)
- [25. Âmbito das variáveis](#)

1. Introdução ao JavaScript

JavaScript é uma das linguagens de programação mais populares do mundo.

Eu acredito que é uma ótima linguagem para ser sua primeira linguagem de programação de todos os tempos.

Usamos principalmente JavaScript para criar

- sites
- aplicações web
- aplicativos do lado do servidor usando o Node.js

mas JavaScript não está limitado a essas coisas, e também pode ser usado para

- criar aplicativos móveis usando ferramentas como o React Native
- criar programas para microcontroladores e a internet das coisas
- criar aplicativos de smartwatch

Pode basicamente fazer qualquer coisa. É tão popular que tudo de novo que aparece terá algum tipo de integração JavaScript em algum momento.

JavaScript é uma linguagem de programação que é:

- **alto nível**: ele fornece abstrações que permitem que você ignore os detalhes da máquina em que está sendo executado. Ele gerencia a memória automaticamente com um coletor de lixo, para que você possa se concentrar no código em vez de gerenciar a memória como outras linguagens como C precisariam, e fornece muitas construções que permitem lidar com variáveis e objetos altamente poderosos.
- **dynamic**: ao contrário das linguagens de programação estáticas, uma linguagem dinâmica executa em tempo de execução muitas das coisas que uma linguagem estática faz em tempo de compilação. Isso tem prós e contras, e nos dá recursos poderosos como digitação dinâmica, ligação tardia, reflexão, programação funcional, alteração de tempo de execução de objetos, fechamentos e muito mais. Não se preocupe se essas coisas são desconhecidas para você - você saberá todas elas no final do curso.
- **dinamicamente tipado**: uma variável não impõe um tipo. Você pode reatribuir qualquer tipo a uma variável, por exemplo, atribuindo um inteiro a uma variável que contém uma cadeia de caracteres.
- **vagamente digitado**: ao contrário da digitação forte, as linguagens vagamente (ou fracamente) digitadas não impõem o tipo de um objeto, permitindo mais flexibilidade, mas negando-nos a segurança do tipo e a verificação de tipos (algo que o TypeScript - que se baseia no JavaScript - fornece)
- **interpretado**: é comumente conhecido como uma linguagem interpretada, o que significa que não precisa de um estágio de compilação antes que um programa possa ser executado, ao contrário de C, Java ou Go, por exemplo. Na prática, os navegadores compilam JavaScript antes de

executá-lo, por razões de desempenho, mas isso é transparente para você: não há nenhuma etapa adicional envolvida.

■ **multiparadigma:** a linguagem não impõe nenhum paradigma de programação em particular, ao contrário do Java, por exemplo, que força o uso de programação orientada a objetos, ou C, que força a programação imperativa. Você pode escrever JavaScript usando um paradigma orientado a objetos, usando protótipos e a nova sintaxe de classes (a partir de ES6). Você pode escrever JavaScript em um estilo de programação funcional, com suas funções de primeira classe, ou mesmo em um estilo imperativo (C-like).

Caso você esteja se perguntando, *JavaScript não tem nada a ver com Java*, é uma má escolha de nome, mas temos que viver com isso.

2. Um pouco de história

Criado em 1995, o JavaScript percorreu um longo caminho desde o seu humilde início.

Foi a primeira linguagem de script que foi suportada nativamente por navegadores da web, e graças a isso ganhou uma vantagem competitiva sobre qualquer outra linguagem e hoje ainda é a única linguagem de script que podemos usar para construir aplicativos Web.

Outras linguagens existem, mas todas devem compilar para JavaScript - ou mais recentemente para WebAssembly, mas esta é outra história.

No início, o JavaScript não era tão poderoso quanto é hoje, e era usado principalmente para animações extravagantes e a maravilha conhecida na época como *HTMLs dinâmico*.

Com as crescentes necessidades que a plataforma web exigiu (e continua a exigir), o JavaScript teve a responsabilidade de crescer também, para acomodar as necessidades de um dos ecossistemas mais utilizados do mundo.

JavaScript agora é amplamente utilizado também fora do navegador. A ascensão do Node.js nos últimos anos desbloqueou o desenvolvimento de

backend, uma vez que o domínio de Java, Ruby, Python, PHP e linguagens mais tradicionais do lado do servidor.

JavaScript agora também é a linguagem que alimenta bancos de dados e muitos outros aplicativos, e é até possível desenvolver aplicativos incorporados, aplicativos móveis, aplicativos de aparelhos de TV e muito mais. O que começou como uma pequena linguagem dentro do navegador é agora a linguagem mais popular do mundo.

3. Apenas JavaScript

Às vezes, é difícil separar o JavaScript dos recursos do ambiente em que ele é usado.

Por exemplo, a linha que você pode encontrar em muitos exemplos de código não é JavaScript. Em vez disso, faz parte da vasta biblioteca de APIs fornecidas a nós no navegador. Da mesma forma, no servidor, às vezes pode ser difícil separar os recursos da linguagem JavaScript das APIs fornecidas pelo `Node.js.console.log()`

Um recurso específico é fornecido pelo React ou Vue? Ou é "JavaScript simples" ou "JavaScript baunilha", como muitas vezes chamado?

Neste livro falo sobre JavaScript, a linguagem, sem complicar o seu processo de aprendizagem com coisas que estão fora dele, e fornecidas por ecossistemas externos.

4. Uma breve introdução à sintaxe do JavaScript

Nesta pequena introdução quero falar sobre 5 conceitos:

- espaço em branco
- sensibilidade maiúscula e min
- Literais
- identifiers
- comments

4.1. White space

JavaScript does not consider white space meaningful. Spaces and line breaks can be added in any fashion you might like, even though this is *in theory*.

In practice, you will most likely keep a well-defined style and adhere to what people commonly use, and enforce this using a linter or a style tool such as *Prettier*.

For example, I like to always use 2 characters to indent.

4.2. Case sensitive

JavaScript is case sensitive. A variable named `isDifferent` is different from `.somethingSomething`

The same goes for any identifier.

4.3. Literals

We define as **literal** a value that is written in the source code, for example, a number, a string, a boolean or also more advanced constructs, like Object Literals or Array Literals:

```
5
'Test'
true
['a', 'b']
{color: 'red', shape: 'Rectangle'}
```

4.4. Identifiers

An **identifier** is a sequence of characters that can be used to identify a variable, a function, or an object. It can start with a letter, the dollar sign or an underscore `_`, and it can contain digits. Using Unicode, a letter can be any allowed char, for example, an emoji  `:$_`

```
Test  
test  
TEST  
_test  
Test1  
$test
```

The dollar sign is commonly used to reference DOM elements.

Some names are reserved for JavaScript internal use, and we can't use them as identifiers.

4.5. Comments

Comments are one of the most important parts of any program. In any programming language. They are important because they let us annotate the code and add important information that otherwise would not be available to other people (or ourselves) reading the code.

In JavaScript, we can write a comment on a single line using `//`. Everything after is not considered as code by the JavaScript interpreter.`////`

Like this:

```
// a comment  
true //another comment
```

Another type of comment is a multi-line comment. It starts with `/*` and ends with `*/`

Everything in between is not considered as code:

```
/* some kind  
of  
comment
```

```
*/
```

5. Semicolons

Every line in a JavaScript program is optionally terminated using semicolons.

I said optionally, because the JavaScript interpreter is smart enough to introduce semicolons for you.

In most cases, you can omit semicolons altogether from your programs.

This fact is very controversial, and you'll always find code that uses semicolons and code that does not.

My personal preference is to always avoid semicolons unless strictly necessary.

6. Valores

Uma cadeia de caracteres é um **valor**. Um número como é um **valor**.hello12

hello e são valores. e são os **tipos** desses valores.12stringnumber

O tipo é o **tipo** de valor, sua categoria. Temos muitos tipos diferentes em JavaScript, e falaremos sobre eles em detalhes mais tarde. Cada tipo tem suas próprias características.

Quando precisamos ter uma referência a um valor, nós a atribuímos a uma **variável**. A variável pode ter um nome, e o valor é o que está armazenado em uma variável, para que possamos acessar esse valor posteriormente por meio do nome da variável.

7. Variáveis

Uma variável é um valor atribuído a um identificador, para que você possa fazer referência e usá-lo posteriormente no programa.

Isso ocorre porque o JavaScript é **vagamente digitado**, um conceito que você ouvirá falar com frequência.

Uma variável deve ser declarada antes que você possa usá-la.

Temos 2 maneiras principais de declarar variáveis. A primeira é usar:`const`

```
const a = 0
```

A segunda maneira é usar:`let`

```
let a = 0
```

Qual a diferença?

`const` define uma referência constante a um valor. Isso significa que a referência não pode ser alterada. Não é possível reatribuir um novo valor a ele.

Usando você pode atribuir um novo valor a ele.`let`

Por exemplo, você não pode fazer isso:

```
const a = 0  
a = 1
```

Porque você receberá um erro: `.TypeError: Assignment to constant variable.`

Por outro lado, você pode fazê-lo usando:`let`

```
let a = 0  
a = 1
```

const não significa "constante" da mesma forma que algumas outras linguagens como C significam. Em particular, isso não significa que o valor não possa mudar - significa que ele não pode ser reatribuído. Se a variável apontar para um objeto ou uma matriz (veremos mais sobre objetos e matrizes mais tarde), o conteúdo do objeto ou da matriz poderá ser alterado livremente.

As variáveis Const devem ser inicializadas no momento da declaração:

```
const a = 0
```

mas os valores podem ser inicializados posteriormente:
`let`

```
let a  
a = 0
```

Você pode declarar várias variáveis de uma só vez na mesma instrução:

```
const a = 1,  
      b = 2  
let c = 1,  
    d = 2
```

Mas você não pode redeclarar a mesma variável mais de uma vez:

```
let a = 1  
let a = 2
```

ou você receberia um erro de "declaração duplicada".

Meu conselho é sempre usar e usar somente quando você souber que precisará reatribuir um valor a essa variável. Por que? Porque quanto menos poder o nosso código tiver, melhor. Se soubermos que um valor não pode ser reatribuído, é uma fonte a menos para bugs.

Agora que vimos como trabalhar com e , quero mencionar .constletvar

Até 2015, era a única maneira de declarar uma variável em JavaScript. Hoje, uma base de código moderna provavelmente usará apenas e . Existem algumas diferenças fundamentais que eu detalhei [neste post](#), mas se você está apenas começando, você pode não se importar com elas. Basta usar e .varconstletconstlet

8. Tipos

As variáveis em JavaScript não têm nenhum tipo anexado.

Eles *não são tipificados*.

Depois de atribuir um valor com algum tipo a uma variável, você pode reatribuir posteriormente a variável para hospedar um valor de qualquer outro tipo, sem qualquer problema.

Em JavaScript temos 2 tipos principais de tipos: tipos **primitivos e tipos de objeto**.

8.1. Tipos primitivos

Os tipos primitivos são

- Números
- Strings
- Booleanos
- Símbolos

E dois tipos especiais: e .nullundefined

8.2. Tipos de objeto

Qualquer valor que não seja de um tipo primitivo (uma cadeia de caracteres, um número, um booleano, nulo ou indefinido) é um **objeto**.

Os tipos de objeto têm propriedades e também têm **métodos** que podem atuar nessas **propriedades**.

Falaremos mais sobre objetos mais adiante.

9. Expressões

Uma expressão é uma única unidade de código JavaScript que o mecanismo JavaScript pode avaliar e retornar um valor.

As expressões podem variar em complexidade.

Partimos das muito simples, chamadas expressões primárias:

```
2  
0.02  
('something')  
true  
false  
this //the current scope  
undefined  
i //where i is a variable or a constant
```

Expressões aritméticas são expressões que tomam uma variável e um operador (mais sobre operadores em breve) e resultam em um número:

```
1 / 2  
i++  
i -= 2  
i * 2
```

Expressões de cadeia de caracteres são expressões que resultam em uma cadeia de caracteres:

```
'A' + 'string'
```

Expressões lógicas fazem uso de operadores lógicos e resolvem para um valor booleano:

```
a && b
```

```
a || b
```

```
!a
```

Expressões mais avançadas envolvem objetos, funções e matrizes, e vou apresentá-las mais tarde.

10. Operadores

Os operadores permitem que você obtenha duas expressões simples e as combine para formar uma expressão mais complexa.

Podemos classificar os operadores com base nos operandos com os quais eles trabalham. Alguns operadores trabalham com 1 operando. A maioria com 2 operandos. Apenas um operador trabalha com 3 operandos.

Nesta primeira introdução aos operadores, apresentaremos os operadores com os quais você provavelmente está familiarizado: operadores binários.

Eu já apresentei um ao falar sobre variáveis: o operador de atribuição . Use para atribuir um valor a uma variável:==

```
let b = 2
```

Vamos agora apresentar outro conjunto de operadores binários com os quais você já está familiarizado, a partir da matemática básica.

10.1. O operador de adição (+)

```
const three = 1 + 2  
const four = three + 1
```

O operador também serve como concatenação de cadeia de caracteres se você usar cadeias de caracteres, portanto, preste atenção:+

```
const three = 1 + 2  
three + 1 // 4  
'three' + 1 // three1
```

10.2. O operador de subtração (-)

```
const two = 4 - 2
```

10.3. O operador da divisão (/)

Devolve o quociente do primeiro operador e do segundo:

```
const result = 20 / 5 //result === 4  
const result = 20 / 7 //result === 2.857142857142857
```

Se você dividir por zero, o JavaScript não gerará nenhum erro, mas retornará o valor (ou se o valor for negativo).Infinity-Infinity

```
1 / 0 - //Infinity  
1 / 0 // -Infinity
```

10.4. O restante operador (%)

O restante é um cálculo muito útil em muitos casos de uso:

```
const result = 20 % 5 //result === 0  
const result = 20 % 7 //result === 6
```

Um restante por zero é sempre , um valor especial que significa "Não é um número":NaN

```
1 % 0 //NaN
```

10.5. O operador de multiplicação (*)

Multiplique dois números

```
1 * 2 //2  
1 * -2 //-2
```

10.6. O operador de exponenciação (**)

Elevar o primeiro operando à potência do segundo operando

```
1 ** 2 //1  
2 ** 1 //2  
2 ** 2 //4  
2 ** 8 //256  
8 ** 2 //64
```

11. Regras de precedência

Cada instrução complexa com vários operadores na mesma linha introduzirá problemas de precedência.

Veja este exemplo:

```
let a = 1 * 2 + ((5 / 2) % 2)
```

O resultado é 2,5, mas por quê?

Quais operações são executadas primeiro e quais precisam esperar?

Algumas operações têm mais precedência do que as outras. As regras de precedência estão listadas nesta tabela:

Operador	Descrição
* / %	multiplicação/divisão
+ -	adição/subtração
=	designação

Operações no mesmo nível (como e) são executadas na ordem em que são encontradas, da esquerda para a direita.+-

Seguindo estas regras, a operação acima pode ser resolvida desta forma:

```
let a = 1 * 2 + ((5 / 2) % 2)
let a = 2 + ((5 / 2) % 2)
let a = 2 + (2.5 % 2)
let a = 2 + 0.5
let a = 2.5
```

12. Operadores de comparação

Depois dos operadores de atribuição e matemática, o terceiro conjunto de operadores que quero introduzir são os operadores de comparação.

Você pode usar os operadores a seguir para comparar dois números ou duas cadeias de caracteres.

Os operadores de comparação sempre retornam um booleano, um valor que é ou).truefalse

Esses são **operadores de comparação de desigualdade**:

- < significa "menos de"
- <= significa "menos que, ou igual a"
- > significa "maior que"
- >= significa "maior ou igual a"

Exemplo:

```
let a = 2  
a >= 1 //true
```

Além desses, temos 4 **operadores de igualdade**. Eles aceitam dois valores e retornam um booleano:

- === verifica a igualdade
- !== verifica a desigualdade

Note que também temos e em JavaScript, mas sugiro vivamente usar apenas e porque podem evitar alguns problemas subtis.`==`!`=====`!`==`

13. Condicionais

Com os operadores de comparação em vigor, podemos falar sobre condicionais.

Uma instrução é usada para fazer com que o programa tome uma rota, ou outra, dependendo do resultado de uma avaliação de expressão.`if`

Este é o exemplo mais simples, que sempre executa:

```
if (true) {  
    //do something  
}
```

pelo contrário, isso nunca é executado:

```
if (false) {  
    //do something (? never ?)  
}
```

A condicional verifica a expressão que você passa para ela em busca de um valor verdadeiro ou falso. Se você passar um número, isso sempre será avaliado como verdadeiro, a menos que seja 0. Se você passar uma cadeia de caracteres, ela sempre será avaliada como true, a menos que seja uma cadeia de caracteres vazia. Essas são regras gerais de tipos de fundição para um booleano.

Você notou os aparelhos encaracolados? Isso é chamado de **bloco** e é usado para agrupar uma lista de instruções diferentes.

Um bloco pode ser colocado onde quer que você possa ter uma única instrução. E se você tiver uma única instrução para executar após as condicionais, você pode omitir o bloco e apenas escrever a instrução:

```
if (true) doSomething()
```

Mas eu sempre gosto de usar chaves para ser mais claro.

13.1. Senão

Você pode fornecer uma segunda parte para a declaração: `.ifelse`

Você anexa uma instrução que será executada se a condição for falsa:`:if`

```
if (true) {  
    //do something  
} else {  
    //do something else  
}
```

Como aceita uma instrução, você pode aninhar outra instrução if/else dentro dela: else

```
if (a === true) {  
    //do something  
} else if (b === true) {  
    //do something else  
} else {  
    //fallback  
}
```

14. Matrizes

Uma matriz é uma coleção de elementos.

Matrizes em JavaScript não são um *tipo* por conta própria.

Matrizes são **objetos**.

Podemos inicializar uma matriz vazia destas 2 maneiras diferentes:

```
const a = []  
const a = Array()
```

A primeira é usar a **sintaxe literal da matriz**. O segundo usa a função interna Array.

Você pode preencher previamente a matriz usando esta sintaxe:

```
const a = [1, 2, 3]  
const a = Array.of(1, 2, 3)
```

Uma matriz pode conter qualquer valor, até mesmo valor de diferentes tipos:

```
const a = [1, 'Flavio', ['a', 'b']]
```

Como podemos adicionar uma matriz a uma matriz, podemos criar matrizes multidimensionais, que têm aplicações muito úteis (por exemplo, uma matriz):

```
const matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
]

matrix[0][0] //1
matrix[2][0] //7
```

Você pode acessar qualquer elemento da matriz fazendo referência ao seu índice, que começa do zero:

```
a[0] //1
a[1] //2
a[2] //3
```

Você pode inicializar uma nova matriz com um conjunto de valores usando essa sintaxe, que primeiro inicializa uma matriz de 12 elementos e preenche cada elemento com o número:0

```
Array(12).fill(0)
```

Você pode obter o número de elementos na matriz verificando sua propriedade: length

```
const a = [1, 2, 3]
a.length //3
```

Observe que você pode definir o comprimento da matriz. Se você atribuir um número maior do que a capacidade atual do array, nada acontecerá. Se você atribuir um número menor, a matriz será cortada nessa posição:

```
const a = [1, 2, 3]
a // [ 1, 2, 3 ]
a.length = 2
a // [ 1, 2 ]
```

14.1. Como adicionar um item a uma matriz

Podemos adicionar um elemento no final de uma matriz usando o método:push()

```
a.push(4)
```

Podemos adicionar um elemento no início de uma matriz usando o método:unshift()

```
a.unshift(0)
a.unshift(-2, -1)
```

14.2. Como remover um item de uma matriz

Podemos remover um item do final de uma matriz usando o método:pop()

```
a.pop()
```

Podemos remover um item do início de uma matriz usando o método:shift()

```
a.shift()
```

14.3. Como unir dois ou mais arrays

Você pode unir várias matrizes usando:concat()

```
const a = [1, 2]
const b = [3, 4]
const c = a.concat(b) // [1,2,3,4]
a // [1,2]
b // [3,4]
```

Você também pode usar o operador **de propagação** () desta maneira:...

```
const a = [1, 2]
const b = [3, 4]
const c = [...a, ...b]
c // [1,2,3,4]
```

14.4. Como localizar um item específico na matriz

Você pode usar o método de uma matriz: `find()`

```
a.find((element, index, array) => {
  // return true or false
})
```

Esse método retorna o primeiro item que retorna na função de retorno de chamada fornecida. Ele retorna indefinido se nada retornar "true".`true`

É sua responsabilidade definir o corpo da função de retorno de chamada, para que você possa dizer o que está procurando.`find()`

Uma sintaxe comumente usada é:

```
const my_id = 3

a.find((x) => x.id === my_id)
```

A linha acima retornará o primeiro elemento na matriz que tem igual a , o valor de .id3my_id

findIndex() é outro método de matriz que funciona de forma semelhante ao , mas retorna o índice do primeiro item que retorna true e, se não for encontrado, ele retorna:find()undefined

```
a.findIndex((element, index, array) => {  
    //return true or false  
})
```

Outro método útil é:includes()

```
a.includes(value)
```

Retorna true se contiver .a value

```
a.includes(value, i)
```

Retorna true se contiver após a posição .a valuei

15. Cordas

Uma cadeia de caracteres é uma sequência de caracteres.

Também pode ser definido como um literal de cadeia de caracteres, que é colocado entre aspas ou aspas duplas:

```
'A string'  
'Another string'
```

Eu pessoalmente prefiro aspas simples o tempo todo, e uso aspas duplas apenas em HTML para definir atributos.

Você atribui um valor de cadeia de caracteres a uma variável como esta:

```
const name = 'Flavio'
```

Você pode determinar o comprimento de uma cadeia de caracteres usando a propriedade dela:`length`

```
'Flavio'.length //6  
const name = 'Flavio'  
name.length //6
```

Esta é uma cadeia de caracteres vazia: `.`. Sua propriedade `length` é 0: `''`

```
''.length //0
```

Duas cadeias de caracteres podem ser unidas usando o operador:`+`

```
'A ' + 'string'
```

Você pode usar o operador para *interpolar* variáveis:`+`

```
const name = 'Flavio'  
'My name is ' + name //My name is Flavio
```

Outra maneira de definir cadeias de caracteres é usar uma sintaxe especial chamada **literais de modelo**, definida dentro de backticks. Eles são especialmente úteis para tornar as cadeias de caracteres de várias linhas muito mais simples. Com aspas simples ou duplas, você não pode definir uma cadeia de caracteres de várias linhas facilmente: você precisaria usar caracteres de escape.

Depois que um literal de modelo é aberto com o backtick, basta pressionar enter para criar uma nova linha, sem caracteres especiais, e ela é renderizada como está:

```
const string = `Hey  
this  
  
string  
is awesome!`
```

Os literais de modelo também são ótimos porque fornecem uma maneira fácil de interpolar variáveis e expressões em cadeias de caracteres.

Para fazer isso, use a sintaxe: `${...}`

```
const var = 'test'  
const string = `something ${var}`  
//something test
```

dentro do você pode adicionar qualquer coisa, até mesmo expressões: `${}`

```
const string = `something ${1 + 2 + 3}`  
const string2 = `something  
 ${a > b ? 'a' : 'b'}`
```

16. Loops

Loops são uma das principais estruturas de controle do JavaScript.

Com um loop, podemos automatizar e repetir indefinidamente um bloco de código, por quantas vezes queremos que ele seja executado.

O JavaScript fornece muitas maneiras de iterar através de loops.

Quero me concentrar em 3 maneiras:

- enquanto loops
- para loops
- durante.. de loops

16.1. while

O loop while é a estrutura de looping mais simples que o JavaScript nos fornece.

Adicionamos uma condição após a palavra-chave e fornecemos um bloco que é executado até que a condição seja avaliada como `while(true)`

Exemplo:

```
const list = ['a', 'b', 'c']
let i = 0
while (i < list.length) {
    console.log(list[i]) //value
    console.log(i) //index
    i = i + 1
}
```

Você pode interromper um loop usando a palavra-chave, desta forma:`whilebreak`

```
while (true) {
    if (somethingIsTrue) break
}
```

e se você decidir que, no meio de um loop, deseja ignorar a iteração atual, poderá pular para a próxima iteração usando:`continue`

```
while (true) {
    if (somethingIsTrue) continue

    //do something else
}
```

Muito semelhante a , temos loops. É basicamente o mesmo que , exceto que a condição é avaliada *depois* que o bloco de código é executado.`while` do .. `while` `while`

Isso significa que o bloco é sempre executado *pelo menos uma vez*.

Exemplo:

```
const list = ['a', 'b', 'c']
let i = 0
do {
    console.log(list[i]) //value
    console.log(i) //index
    i = i + 1
} while (i < list.length)
```

16.2. for

A segunda estrutura de looping muito importante em JavaScript é o **loop for**.

Usamos a palavra-chave e passamos um conjunto de 3 instruções: a inicialização, a condição e a parte de incremento.`for`

Exemplo:

```
const list = ['a', 'b', 'c']

for (let i = 0; i < list.length; i++) {
    console.log(list[i]) //value
    console.log(i) //index
}
```

Assim como com loops, você pode interromper um loop usando e você pode avançar rapidamente para a próxima iteração de um loop usando `.while` `for` `break` `for` `continue`

16.3. `for...of`

Esse loop é relativamente recente (introduzido em 2015) e é uma versão simplificada do `loop:for`

```
const list = ['a', 'b', 'c']

for (const value of list) {
    console.log(value) //value
}
```

17. Funções

Em qualquer programa JavaScript moderadamente complexo, tudo acontece dentro das funções.

As funções são uma parte essencial e central do JavaScript.

O que é uma função?

Uma função é um bloco de código, autossuficiente.

Aqui está uma **declaração de função**:

```
function getData() {
    // do something
}
```

Uma função pode ser executada a qualquer momento que você quiser, invocando-a, assim:

```
getData()
```

Uma função pode ter um ou mais argumentos:

```
function getData() {  
    //do something  
}  
  
function getData(color) {  
    //do something  
}  
  
function getData(color, age) {  
    //do something  
}
```

Quando podemos passar um argumento, invocamos os parâmetros de passagem de função:

```
function getData(color, age) {  
    //do something  
}  
  
getData('green', 24)  
getData('black')
```

Observe que na segunda invocação eu passei o parâmetro string como o argumento, mas não . Neste caso, dentro da função é .blackcolorageageundefined

Podemos verificar se um valor não está indefinido usando esta condicional:

```
function getData(color, age) {  
    //do something  
    if (typeof age !== 'undefined') {  
        //...  
    }  
}
```

`typeof` é um operador unário que nos permite verificar o tipo de uma variável.

Você também pode verificar desta forma:

```
function getData(color, age) {  
    //do something  
    if (age) {  
        //...  
    }  
}
```

embora, neste caso, a condicional será falsa se for , ou uma cadeia de caracteres vazia.`agenull0`

Você pode ter valores padrão para parâmetros, caso eles não sejam passados:

```
function getData(color = 'black', age = 25) {  
    //do something  
}
```

Você pode passar qualquer valor como um parâmetro: números, cadeias de caracteres, booleanos, matrizes, objetos e também funções.

Uma função tem um valor de retorno. Por padrão, uma função retorna , a menos que você adicione uma palavra-chave com um `valor:undefinedreturn`

```
function getData() {  
    // do something  
    return 'hi!'  
}
```

Podemos atribuir esse valor de retorno a uma variável quando invocamos a função:

```
function getData() {  
    // do something  
    return 'hi!'  
}  
  
let result = getData()
```

result agora contém uma cadeia de caracteres com o valor.hi!

Você só pode retornar um valor.

Para retornar vários valores, você pode retornar um objeto ou uma matriz, como este:

```
function getData() {  
    return ['Flavio', 37]  
}  
  
let [name, age] = getData()
```

As funções podem ser definidas dentro de outras funções:

```
const getData = () => {  
    const dosomething = () => {}  
    dosomething()  
    return 'test'  
}
```

A função aninhada não pode ser chamada do lado de fora da função de delimitação.

Você também pode retornar uma função de uma função.

18. Funções de seta

As funções de seta são uma introdução recente ao JavaScript.

Eles são muito frequentemente usados em vez de funções "regulares", a que descrevi no capítulo anterior. Você encontrará os dois formulários usados em todos os lugares.

Visualmente, eles permitem que você escreva funções com uma sintaxe mais curta, de:

```
function getData() {  
    //...  
}
```

Para

```
;() => {  
    //...  
}
```

Mas.. observe que não temos um nome aqui.

As funções de seta são anônimas. Devemos atribuí-los a uma variável.

Podemos atribuir uma função regular a uma variável, como esta:

```
let getData = function getData() {  
    //...  
}
```

Quando fazemos isso, podemos remover o nome da função:

```
let getData = function () {  
    //...  
}
```

e invoque a função usando o nome da variável:

```
let getData = function () {  
    //...  
}  
getData()
```

É a mesma coisa que fazemos com as funções de seta:

```
let getData = () => {  
    //...  
}  
getData()
```

Se o corpo da função contiver apenas uma única instrução, você poderá omitir os parênteses e escrever tudo em uma única linha:

```
const getData = () => console.log('hi!')
```

Os parâmetros são passados entre parênteses:

```
const getData = (param1, param2) => console.log(param1, param2)
```

Se você tiver um (e apenas um) parâmetro, poderá omitir completamente os parênteses:

```
const getData = (param) => console.log(param)
```

As funções de seta permitem que você tenha um retorno implícito: os valores são retornados sem precisar usar a palavra-chave `return`

Ele funciona quando há uma instrução on-line no corpo da função:

```
const getData = () => 'test'
```

```
getData() // 'test'
```

Como com funções regulares, podemos ter parâmetros padrão:

Você pode ter valores padrão para parâmetros, caso eles não sejam passados:

```
const getData = (color = 'black', age = 2) => {  
    // do something  
}
```

e só podemos retornar um valor.

As funções de seta podem conter outras funções de seta ou também funções regulares.

Eles são muito semelhantes, então você pode perguntar por que eles foram introduzidos? A grande diferença com funções regulares é quando elas são usadas como métodos de objeto. Isso é algo que em breve analisaremos.

19. Objetos

Qualquer valor que não seja de um tipo primitivo (uma cadeia de caracteres, um número, um booleano, um símbolo, nulo ou indefinido) é um **objeto**.

Veja como definimos um objeto:

```
const car = {}
```

Essa é a sintaxe **literal** do objeto, que é uma das coisas mais legais do JavaScript.

Você também pode usar a sintaxe: `new Object`

```
const car = new Object()
```

Outra sintaxe é usar :Object.create()

```
const car = Object.create()
```

Você também pode inicializar um objeto usando a palavra-chave antes de uma função com uma letra maiúscula. Essa função serve como um construtor para esse objeto. Lá, podemos inicializar os argumentos que recebemos como parâmetros, para configurar o estado inicial do objeto:`new`

```
function Car(brand, model) {  
    this.brand = brand  
    this.model = model  
}
```

Inicializamos um novo objeto usando

```
const myCar = new Car('Ford', 'Fiesta')  
myCar.brand // 'Ford'  
myCar.model // 'Fiesta'
```

Os objetos são **sempre passados por referência**.

Se você atribuir a uma variável o mesmo valor de outra, se for um tipo primitivo como um número ou uma cadeia de caracteres, elas serão passadas pelo valor:

Veja este exemplo:

```
let age = 36  
let myAge = age  
myAge = 37  
age // 36
```

```
const car = {
  color: 'blue',
}

const anotherCar = car
anotherCar.color = 'yellow'
car.color // 'yellow'
```

Mesmo matrizes ou funções são, sob o capô, objetos, por isso é muito importante entender como eles funcionam.

19.1. Propriedades do objeto

Os objetos têm **propriedades**, que são compostas por um rótulo associado a um valor.

O valor de uma propriedade pode ser de qualquer tipo, o que significa que pode ser uma matriz, uma função e pode até ser um objeto, pois os objetos podem aninhar outros objetos.

Esta é a sintaxe literal do objeto que vimos no capítulo anterior:

```
const car = {}
```

Podemos definir uma propriedade desta forma:

```
const car = {
  color: 'blue',
}
```

aqui temos um objeto com uma propriedade chamada , com o valor .carcolorblue

Os rótulos podem ser qualquer cadeia de caracteres, mas cuidado com caracteres especiais: se eu quisesse incluir um caractere não válido como um

nome de variável no nome da propriedade, eu teria que usar aspas em torno dele:

```
const car = {  
    color: 'blue',  
    'the color': 'blue',  
}
```

Caracteres de nome de variável inválidos incluem espaços, hífens e outros caracteres especiais.

Como você vê, quando temos várias propriedades, separamos cada propriedade com uma vírgula.

Podemos recuperar o valor de uma propriedade usando 2 sintaxes diferentes.

A primeira é **a notação de pontos**:

```
car.color // 'blue'
```

O segundo (que é o único que podemos usar para propriedades com nomes inválidos), é usar colchetes:

```
car['the color'] // 'blue'
```

Se você acessar uma propriedade inexistente, obterá o valor:`undefined`

```
car.brand // undefined
```

Como dito, os objetos podem ter objetos aninhados como propriedades:

```
const car = {  
    brand: {  
        name: 'Ford',  
    },  
}
```

```
},  
color: 'blue',  
}
```

Neste exemplo, você pode acessar o nome da marca usando

```
car.brand.name
```

ou

```
car['brand']['name']
```

Você pode definir o valor de uma propriedade ao definir o objeto.

Mas você sempre pode atualizá-lo mais tarde:

```
const car = {  
  color: 'blue',  
}
```

```
car.color = 'yellow'  
car['color'] = 'red'
```

E você também pode adicionar novas propriedades a um objeto:

```
car.model = 'Fiesta'  
  
car.model // 'Fiesta'
```

Dado o objeto

```
const car = {  
  color: 'blue',
```

```
    brand: 'Ford',  
}
```

você pode excluir uma propriedade deste objeto usando

```
delete car.brand
```

19.2. Métodos de objeto

Eu falei sobre funções em um capítulo anterior.

As funções podem ser atribuídas a uma propriedade de função e, nesse caso, são chamadas **de métodos**.

Neste exemplo, a propriedade tem uma função atribuída e podemos invocá-la usando a sintaxe de pontos que usamos para propriedades, com os parênteses no final:

```
const car = {  
  brand: 'Ford',  
  model: 'Fiesta',  
  start: function () {  
    console.log('Started')  
  },  
}  
  
car.start()
```

Dentro de um método definido usando uma sintaxe, temos acesso à instância do objeto fazendo referência a `.function() {}this`

No exemplo a seguir, temos acesso aos valores e propriedades usando e `:brandmodelthis.brandthis.model`

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  start: function () {
    console.log(`Started
${this.brand} ${this.model}`)
  },
}

car.start()
```

É importante notar esta distinção entre funções regulares e funções de seta:
não temos acesso a se usarmos uma função de seta: this

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  start: () => {
    console.log(`Started
${this.brand} ${this.model}`) //not going to work
  },
}

car.start()
```

Isso ocorre porque **as funções de seta não estão vinculadas ao objeto.**

Esta é a razão pela qual as funções regulares são frequentemente usadas como métodos de objeto.

Os métodos podem aceitar parâmetros, como funções regulares:

```
const car = {
  brand: 'Ford',
  model: 'Fiesta',
  goTo: function (destination) {
```

```
    console.log(`Going to ${destination}`)
  },
}

car.goTo( 'Rome' )
```

20. Aulas

Falamos sobre objetos, que são uma das partes mais interessantes do JavaScript.

Neste capítulo, subiremos um nível, introduzindo classes.

O que são aulas? Eles são uma maneira de definir um padrão comum para vários objetos.

Vamos pegar um objeto de pessoa:

```
const person = {
  name: 'Flavio',
}
```

Podemos criar uma classe chamada (observe a maiúscula, uma convenção ao usar classes), que tem uma propriedade: PersonPname

```
class Person {
  name
}
```

Agora, a partir dessa classe, inicializamos um objeto como este: flavio

```
const flavio = new Person()
```

flavio é chamada de *instância* da classe Person.

Podemos definir o valor da propriedade: name

```
flavio.name = 'Flavio'
```

e podemos acessá-lo usando

```
flavio.name
```

como fazemos para propriedades de objeto.

As classes podem conter propriedades, como , e métodos.name

Os métodos são definidos da seguinte forma:

```
class Person {  
    hello() {  
        return 'Hello, I am Flavio'  
    }  
}
```

e podemos invocar métodos em uma instância da classe:

```
class Person {  
    hello() {  
        return 'Hello, I am Flavio'  
    }  
}  
const flavio = new Person()  
flavio.hello()
```

Há um método especial chamado que podemos usar para inicializar as propriedades de classe quando criamos uma nova instância de objeto.constructor()

Funciona assim:

```
class Person {  
    constructor(name) {  
        this.name = name  
    }  
  
    hello() {  
        return 'Hello, I am ' + this.name + '.'  
    }  
}
```

Observe como usamos para acessar a instância do objeto.`this`

Agora podemos instanciar um novo objeto da classe, passando uma cadeia de caracteres, e quando ligarmos , receberemos uma mensagem personalizada:`hello`

```
const flavio = new Person('flavio')  
flavio.hello() // 'Hello, I am flavio.'
```

Quando o objeto é inicializado, o método é chamado, com quaisquer parâmetros passados.`constructor`

Normalmente, os métodos são definidos na instância do objeto, não na classe.

Você pode definir um método para permitir que ele seja executado na classe:`static`

```
class Person {  
    static genericHello() {  
        return 'Hello'  
    }  
}  
  
Person.genericHello() // Hello
```

Isso é muito útil, às vezes.

21. Herança

Uma classe pode **estender** outra classe, e os objetos inicializados usando essa classe herdam todos os métodos de ambas as classes.

Suponha que tenhamos uma classe:Person

```
class Person {  
    hello() {  
        return 'Hello, I am a Person'  
    }  
}
```

Podemos definir uma nova classe que se estende:ProgrammerPerson

```
class Programmer extends Person {}
```

Agora, se instanciarmos um novo objeto com classe, ele terá acesso ao método:Programmerhello()

```
const flavio = new Programmer()  
flavio.hello() // 'Hello, I am a Person'
```

Dentro de uma classe filha, você pode fazer referência à classe pai chamando:super()

```
class Programmer extends Person {  
    hello() {  
        return super.hello() + '. I am also a programmer.'  
    }  
}
```

```
const flavio = new Programmer()  
flavio.hello()
```

O programa acima imprime *Olá, eu sou uma pessoa. Eu também sou um programador.*.

22. Programação assíncrona e retornos de chamada

Na maioria das vezes, o código JavaScript é executado de forma síncrona.

Isso significa que uma linha de código é executada, depois a próxima é executada e assim por diante.

Tudo é como você espera, e como funciona na maioria das linguagens de programação.

No entanto, há momentos em que você não pode apenas esperar por uma linha de código para executar.

Você não pode simplesmente esperar 2 segundos para que um arquivo grande seja carregado e interromper o programa completamente.

Você não pode simplesmente esperar que um recurso de rede seja baixado antes de fazer outra coisa.

O JavaScript resolve esse problema usando **retornos de chamada**.

Um dos exemplos mais simples de como usar retornos de chamada são os temporizadores. Os temporizadores não fazem parte do JavaScript, mas são fornecidos pelo navegador e pelo Node.js. Deixe-me falar sobre um dos temporizadores que temos: `.setTimeout()`

A função aceita 2 argumentos: uma função e um número. O número é o milissegundo que deve passar antes que a função seja executada.`.setTimeout()`

Exemplo:

```
setTimeout(() => {
  // runs after 2 seconds
  console.log('inside the function')
}, 2000)
```

A função que contém a linha será executada após 2 segundos.

```
console.log('inside the function')
```

Se você adicionar um antes da função e depois dela:

```
console.log('before')console.log('after')
```

```
console.log('before')
setTimeout(() => {
  // runs after 2 seconds
  console.log('inside the function')
}, 2000)
console.log('after')
```

Você verá isso acontecendo no seu console:

```
before
after
inside the function
```

A função de retorno de chamada é executada de forma assíncrona.

Esse é um padrão muito comum ao trabalhar com o sistema de arquivos, a rede, eventos ou o DOM no navegador.

Todas as coisas que mencionei não são JavaScript "core", então elas não são explicadas neste manual, mas você encontrará muitos exemplos em meus outros manuais disponíveis em flaviocopes.com.

Veja como podemos implementar retornos de chamada em nosso código.

Definimos uma função que aceita um parâmetro, que é uma função.callback

Quando o código está pronto para invocar o retorno de chamada, nós o invocamos passando o resultado:

```
const doSomething = (callback) => {
    //do things
    //do things
    const result = /* .. */ callback(result)
}
```

O código usando essa função o usaria assim:

```
doSomething((result) => {
    console.log(result)
})
```

23. Promessas

As promessas são uma maneira alternativa de lidar com o código assíncrono.

Como vimos no capítulo anterior, com retornos de chamada estariámos passando uma função para outra chamada de função, que seria chamada quando a função terminasse de processar.

Assim:

```
doSomething((result) => {
    console.log(result)
})
```

Quando o código termina, ele chama a função recebida como um parâmetro:doSomething()

```
const doSomething = (callback) => {
    //do things
    //do things
    const result = /* .. */ callback(result)
}
```

O principal problema com essa abordagem é que, se precisarmos usar o resultado dessa função no restante do nosso código, todo o nosso código deve estar aninhado dentro do retorno de chamada, e se tivermos que fazer 2-3 retornos de chamada, entramos no que geralmente é definido como "inferno de retorno de chamada" com muitos níveis de funções recuadas em outras funções:

```
doSomething((result) => {
    doSomethingElse((anotherResult) => {
        doSomethingElseAgain((yetAnotherResult) => {
            console.log(result)
        })
    })
})
```

As promessas são uma maneira de lidar com isso.

Em vez de fazer:

```
doSomething((result) => {
    console.log(result)
})
```

Chamamos uma função baseada em promessas da seguinte maneira:

```
doSomething().then((result) => {
    console.log(result)
})
```

Primeiro chamamos a função, então temos um método que é chamado quando a função termina.`then()`

O recuo não importa, mas você geralmente usará esse estilo para obter clareza.

É comum detectar erros usando um método:`catch()`

```
doSomething()  
  .then((result) => {  
    console.log(result)  
  })  
  .catch((error) => {  
    console.log(error)  
  })
```

Agora, para poder usar essa sintaxe, a implementação da função deve ser um pouco especial. Ele deve usar a API Promessas.`doSomething()`

Em vez de declará-lo como uma função normal:

```
const doSomething = () => {}
```

Nós o declaramos como um objeto promessa:

```
const doSomething = new Promise()
```

e passamos uma função no construtor Promise:

```
const doSomething = new Promise(() => {})
```

Esta função recebe 2 parâmetros. A primeira é uma função que chamamos para resolver a promessa, a segunda é uma função que chamamos para rejeitar a promessa.

```
const doSomething = new Promise((resolve, reject) => {})
```

Resolver uma promessa significa completá-la com sucesso (o que resulta em chamar o método em quem o usa).`then()`

Rejeitar uma promessa significa terminá-la com um erro (o que resulta em chamar o método em quem a usa).`catch()`

Veja como:

```
const doSomething = new Promise(  
  (resolve, reject) => {  
    //some code  
    const success = /* ... */  
    if (success) {  
      resolve('ok')  
    } else {  
      reject('this error occurred')  
    }  
  })
```

Podemos passar um parâmetro para as funções de resolução e rejeição, de qualquer tipo que quisermos.

24. Assíncrono e Aguarde

As funções assíncronas são uma abstração de nível mais alto sobre as promessas.

Uma função assíncrona retorna uma promessa, como neste exemplo:

```
const getData = () => {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => resolve('some data'), 2000)
```

```
})  
}
```

Qualquer código que queira usar essa função usará a palavra-chave `logo` antes da função: `await`

```
const data = await getData()
```

e, com isso, todos os dados retornados pela promessa serão atribuídos à variável `data`

No nosso caso, os dados são a cadeia de caracteres "alguns dados".

Com uma ressalva em particular: sempre que usamos a palavra-chave, devemos fazê-lo dentro de uma função definida como `.awaitasync`

Assim:

```
const doSomething = async () => {  
  const data = await getData()  
  console.log(data)  
}
```

A dupla `Async/await` nos permite ter um código mais limpo e um modelo mental simples para trabalhar com código assíncrono.

Como você pode ver no exemplo acima, nosso código parece muito simples. Compare-o com o código usando promessas ou funções de retorno de chamada.

E este é um exemplo muito simples, os principais benefícios surgirão quando o código for muito mais complexo.

Como exemplo, veja como você obteria um recurso JSON usando a API Fetch e o analisaria, usando promessas:

```

const getFirstUserData = () => {
  // get users list
  return (
    fetch('/users.json')
      // parse JSON
      .then((response) => response.json())
      // pick first user
      .then((users) => users[0])
      // get user data
      .then((user) => fetch(`/users/${user.name}`))
      // parse JSON
      .then((userResponse) => userResponse.json())
  )
}

getFirstUserData()

```

E aqui está a mesma funcionalidade fornecida usando await/async:

```

const getFirstUserData = async () => {
  // get users list
  const response = await fetch('/users.json')
  // parse JSON
  const users = await response.json()
  // pick first user
  const user = users[0]
  // get user data
  const userResponse = await fetch(`/users/${user.name}`)
  // parse JSON
  const userData = await userResponse.json()
  return userData
}

getFirstUserData()

```

25. Âmbito das variáveis

Quando apresentei variáveis, falei sobre o uso de , , e .constletvar

Escopo é o conjunto de variáveis que é visível para uma parte do programa.

Em JavaScript, temos um escopo global, escopo de bloco e escopo de função.

Se uma variável é definida fora de uma função ou bloco, ela é anexada ao objeto global e tem um escopo global, o que significa que está disponível em todas as partes de um programa.

Há uma diferença muito importante entre , e declarações.varletconst

Uma variável definida como dentro de uma função só é visível dentro dessa função. Da mesma forma que uma função argumenta:var

Uma variável definida como ou por outro lado só é visível dentro do **bloco** onde é definida.constlet

Um bloco é um conjunto de instruções agrupadas em um par de aparelhos encaracolados, como os que podemos encontrar dentro de uma instrução ou um loop. E uma função também.iffor

É importante entender que um bloco não define um novo escopo para , mas para e .varletconst

Isso tem implicações muito práticas.

Suponha que você defina uma variável dentro de uma condicional em uma função varif

```
function getData() {  
  if (true) {  
    var data = 'some data'  
    console.log(data)  
  }  
}
```

Se você chamar essa função, será impresso no console.some data

Se você tentar mover console.log(data) após o , ele ainda funciona:if

```
function getData() {  
  if (true) {  
    var data = 'some data'  
  }  
  console.log(data)  
}
```

Mas se você mudar para :var datalet data

```
function getData() {  
  if (true) {  
    let data = 'some data'  
  }  
  console.log(data)  
}
```

Você receberá um erro: .ReferenceError: data is not defined

Isso ocorre porque o escopo da função é definido e há uma coisa especial acontecendo aqui, chamada içamento. Em suma, a declaração é movida para o topo da função mais próxima pelo JavaScript, antes de executar o código. Mais ou menos é assim que a função se parece com JS, internamente:varvar

```
function getData() {  
  var data  
  if (true) {  
    data = 'some data'  
  }  
  console.log(data)  
}
```

É por isso que você também pode no topo de uma função, mesmo antes de ser declarada, e você obterá como um valor para essa variável:`console.log(data)``undefined`

```
function getData() {
  console.log(data)
  if (true) {
    var data = 'some data'
  }
}
```

mas se você mudar para `let`, você receberá um erro, porque o içamento não acontece com as declarações.`letReferenceError: data is not defined`

`const` segue as mesmas regras que : é com escopo de bloco.`let`

Pode ser complicado no começo, mas uma vez que você perceba essa diferença, então você verá por que é considerado uma má prática hoje em dia em comparação com: eles têm menos partes móveis, e seu escopo é limitado ao bloco, o que também os torna muito bons como variáveis de loop, porque eles deixam de existir depois que um loop terminou:`var``let`

```
function doLoop() {
  for (var i = 0; i < 10; i++) {
    console.log(i)
  }
  console.log(i)
}

doLoop()
```

Quando você sair do loop, será uma variável válida com valor 10.`i`

Se você alternar para `let`, se tentar resultará em um erro
`let``ReferenceError: i is not defined`