31 DE MAIO DE 2023 / #JAVASCRIPT

O manual de JavaScript para iniciantes

Tradutor: Rafael B. Pires

Autor: Flavio Copes (em inglês)

Artigo original: The JavaScript Beginner's Handbook

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

Creio que seja uma ótima escolha para sua primeira linguagem de programação.

Usamos JavaScript principalmente para criar:

- sites da web
- aplicações web
- aplicações do lado do servidor (*server-side*), usando Node.js

O JavaScript, porém, não se limita a isso, sendo também usado para:

- criar aplicações para dispositivos móveis usando ferramentas como o React Native
- criar programas para microcontroladores e para a internet das coisas
- criar aplicações para relógios *smartwatch*

Ele, basicamente, pode fazer qualquer coisa. É tão popular que qualquer novidade que surja terá algum tipo de integração com JavaScript em algum momento.

O JavaScript é uma linguagem de programação que é:

- **de alto nível**: fornece abstrações que permitem ignorar os detalhes da máquina em que está sendo executada. Ele gerencia a memória automaticamente por meio de um coletor de lixo, para que você possa se concentrar no código em vez de gerenciar a memória, como outras linguagens – como o C – demandariam. Além disso, fornece várias construções que permitem lidar com variáveis e objetos muito poderosos.
- **dinâmica**: ao contrário de linguagens de programação estáticas, uma linguagem dinâmica processa em tempo de execução (*runtime*) muito do que uma linguagem estática realiza durante o tempo de compilação. Isto tem prós e contras, nos dando características poderosas como tipagem dinâmica, vinculação tardia (*late binding*), reflexão, programação funcional, alteração do tempo de execução de objetos, *closures* e muito mais. Não se preocupe se essas características forem desconhecidos para você – você entenderá todas elas até o final do artigo.
- **dinamicamente tipada**: uma variável não impõe um tipo. Você pode reatribuir qualquer tipo a uma variável, por exemplo, atribuindo um número inteiro a uma variável que contenha uma *string*.
- **fracamente tipada**: ao contrário das linguagens fortemente tipadas, as linguagens frouxamente (ou fracamente) tipadas não reforçam o tipo de um objeto. Isso permite maior flexibilidade, mas nos nega a segurança e verificação de um sistema de tipos (algo que o TypeScript – uma camada adicional sobre o JavaScript – fornece)
- **interpretada**: é comumente conhecida como uma linguagem interpretada, o que significa que não é necessária uma etapa 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 questões de desempenho, mas isso acontece de modo transparente – não há nenhuma etapa adicional envolvida.

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

Caso você esteja se perguntando, *JavaScript nada tem a ver com Java*. Trata-se apenas de uma péssima escolha de nome com a qual temos que conviver.

Resumo do manual

> Atualização: agora, você pode ter uma versão em PDF ou ePub deste Manual de JavaScript para iniciantes (em inglês).

Um pouco de história

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

Foi a primeira linguagem de *script* nativamente suportada por navegadores da web. Graças a isso, ganhou uma vantagem competitiva sobre as demais linguagens e, ainda hoje, é a única linguagem de script que podemos usar para construir aplicações para a web.

Existem outras linguagens, mas todas devem ser compiladas para JavaScript – ou, mais recentemente, para WebAssembly, mas essa é outra história.

No início, o JavaScript não era tão potente quanto hoje e era usado principalmente para animações elaboradas e para a maravilha conhecida na época como *HTML Dinâmico*.

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

O JavaScript agora também é amplamente utilizado fora do navegador. A ascensão do Node.js nos últimos anos desbloqueou o desenvolvimento para *back-end*, antes dominado por Java, Ruby, Python, PHP e linguagens mais tradicionais do lado do servidor.

O JavaScript agora também é a linguagem que alimenta bancos de dados e muitas outras aplicações. É possível, até, desenvolver aplicações embarcadas, aplicações para dispositivos móveis ou TVs e muito mais. O que começou como uma pequena linguagem dentro do navegador é agora a linguagem mais popular do mundo.

Apenas JavaScript

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

Por exemplo, o comando `console.log()` que você encontra em muitos exemplos de código não é JavaScript. Em vez disso, ele faz parte da vasta biblioteca de APIs que o navegador nos fornece.

Do mesmo modo, no servidor, às vezes pode ser difícil separar os recursos do JavaScript das APIs fornecidas pelo Node.js.

Determinado recurso é disponibilizado pelo React ou pelo Vue? Ou é "JavaScript puro" – ou "*Vanilla JavaScript*", como é comumente chamado?

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

Uma breve introdução à sintaxe do JavaScript

Nesta breve introdução, quero apresentar 5 conceitos:

- espaços em branco
- maiúsculas e minúsculas
- *literals*
- identificadores
- comentários

Espaços em branco

O JavaScript não dá qualquer significado aos espaços em branco. Espaços e quebras de linha podem ser acrescentados do modo que você desejar, pelo menos *em teoria*.

Na prática, você provavelmente manterá um estilo bem definido, aderindo ao que as pessoas comumente usam, e aplicará isso usando um *linter* ou uma ferramenta de estilo, como o *Prettier*.

Por exemplo, eu sempre uso 2 caracteres de espaço para cada identação.

Maiúsculas e minúsculas

O JavaScript diferencia maiúsculas e minúsculas. Uma variável chamada `something` é diferente de `Something`.

O mesmo vale para qualquer identificador.

Literals

Definimos ***literal*** como um valor que está escrito no código-fonte, por exemplo, um número, uma string, um booleano, ou também construções mais avançadas, como objetos ou *arrays* literais:

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

Identificadores

Um **identificador** é uma sequência de caracteres que pode ser usada para identificar uma variável, uma função ou um objeto. Pode começar com uma letra, com um cifrão `$` ou um sublinhado `_` e pode conter dígitos. Usando Unicode, uma letra pode ser qualquer caractere permitido, por exemplo, um emoji 😄.

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

O cifrão é normalmente usado para fazer referência a elementos do DOM.

Alguns nomes são reservados para uso interno do JavaScript, não podendo ser usados como identificadores.

Comentários

Os comentários são uma das partes mais importantes de qualquer programa, em qualquer linguagem de programação. Eles são importantes porque nos permitem anotar o código e acrescentar informações importantes que, de outro modo, não estariam disponíveis para outras pessoas (ou para nós mesmos) durante sua leitura.

Em JavaScript, podemos escrever um comentário em uma única linha usando `//`. Tudo que vem depois do `//` não é considerado código pelo interpretador de JavaScript.

Assim:

```
// um comentário
true //outro comentário
```

Comentários em português: “um comentário”, “outro comentário”.

Outro tipo de comentário é um comentário de múltiplas linhas. Ele começa com `/*` e termina com `*/`.

Tudo que estiver entre eles não é considerado código:

```
/* Aqui temos
um tipo de
comentário

*/
```

Comentário em português: “algum tipo de comentário”;

Ponto-e-vírgula

Cada linha de um programa em JavaScript é terminada **opcionalmente** com um ponto-e-vírgula.

Digo opcionalmente porque o interpretador do JavaScript é inteligente o suficiente para introduzir pontos-e-vírgulas para você.

Na maioria dos casos, você pode omitir completamente o ponto-e-vírgula de seus programas sem sequer pensar nisso.

Este fato é muito controverso. Alguns desenvolvedores sempre usarão ponto-e-vírgula, outros nunca usarão, e você sempre encontrará código que usa ponto-e-vírgula e código que não usa.

Minha preferência pessoal é evitar pontos-e-vírgulas. Portanto, meus exemplos neste livro não os incluirão.

Valores

Uma string `hello` é um **valor**.
Um número, como `12`, é um **valor**.

`hello` e `12` são valores. `string` e `number` são os **tipos** desses valores.

O **tipo** é a espécie do valor, sua categoria. Temos muitos tipos diferentes em JavaScript e falaremos sobre eles em detalhes mais adiante. Cada tipo tem características próprias.

Quando precisamos ter uma referência de um valor, nós o atribuímos a uma **variável**. A variável pode ter um nome. O valor é o que está armazenado em uma variável, para que possamos acessá-lo posteriormente por meio do nome da variável.

Variáveis

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

Isso ocorre porque o JavaScript é **fracamente tipado**, um conceito que você vai ouvir falar com frequência.

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

Temos dois modos principais de declarar variáveis. O primeiro é usando `const`:

```
const a = 0
```

A segunda maneira é usando `let`:

```
let a = 0
```

Qual é a diferença?

`const` define uma referência constante a um valor. Isso significa que a referência não pode ser alterada. Você não pode reatribuir um novo valor a ela.

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

Por exemplo, você não pode fazer isto:

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

Você receberá um erro: `TypeError: Assignment to constant variable.` (Erro de Tipo: atribuição à variável constante.).

Por outro lado, você pode fazer isto usando `let`:

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

`const` não significa "constante" no mesmo sentido de outras linguagens como C. Em especial, isso não significa que o valor não pode ser alterado – significa que não pode ser reatribuído. Se a variável aponta para um objeto ou array (veremos mais sobre objetos e arrays mais tarde), o conteúdo do objeto ou array pode mudar livremente.

Variáveis `const` devem ser inicializadas no momento da declaração:

```
const a = 0
```

Variáveis `let`, por outro lado, podem ser inicializadas depois:

```
let a
a = 0
```

Você pode declarar múltiplas variáveis de uma vez com um só comando:

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

Não é possível, porém, redeclarar a mesma variável mais de uma vez:

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

Você receberá um erro de "declaração duplicada".

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

Agora que vimos como trabalhar com `const` e `let`, quero mencionar `var`.

Até 2015, `var` era a única maneira de se declarar uma variável em JavaScript. Hoje em dia, uma base de código moderna provavelmente só usará `const` e `let`. Existem algumas diferenças fundamentais que eu detalho neste artigo (em inglês), mas se você está apenas começando, talvez elas não sejam importantes para você. Basta usar `const` e `let`.

Tipos

Variáveis em JavaScript não possuem nenhum tipo definido.

Elas são *não tipadas*.

Uma vez que você atribua um valor de algum tipo a uma variável, você pode reatribuir posteriormente a variável para receber um valor de qualquer outro tipo sem problemas.

Em JavaScript, temos duas espécies principais de tipos: **tipos primitivos** e **objetos**.

Tipos primitivos

Tipos primitivos são

- números

- *strings*
- booleanos
- símbolos

Também há dois tipos especiais: `null` e `undefined`.

Objetos

Qualquer valor que não seja de um tipo primitivo (uma *string*, um número, um booleano, *null* ou *undefined*) é um **objeto**.

Os tipos objeto possuem **propriedades** e também têm **métodos** que podem agir sobre essas propriedades.

Falaremos mais sobre objetos adiante.

Expressões

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

As expressões podem variar em complexidade.

Partimos das mais simples, chamadas de expressões primárias:

```
2
0.02
'something'
true
false
this //o escopo atual
undefined
i //onde i é uma variável ou uma constante
```

Comentários em português: “o escopo atual”, “onde i é uma variável ou constante”.

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

```
1 / 2
i++
```

```
i -= 2
i * 2
```

Expressões de *string* são expressões que resultam em uma *string*:

```
'Uma ' + 'string'
```

Expressões lógicas fazem uso de operadores lógicos e resultam em um valor booleano:

```
a && b
a || b
!a
```

Expressões mais avançadas envolvem objetos, funções e *arrays*. Vou apresentá-las mais tarde.

Operadores

Os operadores permitem pegar duas expressões simples e combiná-las 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 trabalha com 2 operandos. Apenas um operador trabalha com 3 operandos.

Nesta primeira introdução a operadores, apresentaremos aqueles com os quais você está mais familiarizado: operadores com 2 operandos.

Já introduzi um ao falar sobre variáveis: o operador de atribuição `=`. Você usa `=` para atribuir um valor a uma variável:

```
let b = 2
```

Vamos, agora, apresentar outro conjunto de operadores binários que você já conhece da matemática básica.

O operador de adição (+)

```
const tres = 1 + 2
const quatro = tres + 1
```

O operador `+` também faz a concatenação de strings se você usar strings, então preste atenção:

```
const tres = 1 + 2
tres + 1 // 4
'tres' + 1 // tres1
```

O operador de subtração (-)

```
const dois = 4 - 2
```

O operador de divisão (/)

Retorna o quociente entre o primeiro operador e o segundo:

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

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

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

O operador de resto (%)

O resto é um cálculo muito útil em muitos casos:

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

O restante de uma divisão por zero é sempre `NaN`, um valor especial que significa "Not a Number" (inglês para "não é um número"):

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

O operador de multiplicação (*)

Multiplica dois números

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

O operador exponencial (**)

Eleva o primeiro operando à potência do segundo operando

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

Regras de precedência

Cada expressão complexa com vários operadores na mesma linha apresentará problemas de precedência.

Tome 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 outras. As regras de precedência estão listadas nesta tabela:

OPERADOR	DESCRIÇÃO
`*` `/` `%`	multiplicação/divisão/resto
`+` `-`	adição/subtração
`=`	atribuição

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

Seguindo essas regras, a operação acima pode ser resolvida da seguinte 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
```

Operadores de comparação

Depois dos operadores de atribuição e dos matemáticos, o terceiro conjunto de operadores que quero apresentar são os operadores condicionais.

Você pode usar os seguintes operadores para comparar dois números ou duas *strings* (os operadores de comparação sempre devolvem um booleano, ou seja, um valor que é `true` ou `false`).

Estes são os **operadores de comparação de desigualdade**:

- `<` significa "menor que"
- `<=` significa "menor 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 usar apenas `===` e `!==`, pois eles podem evitar alguns problemas sutis.

Condicionais

Com os operadores de comparação estabelecidos, podemos falar sobre condicionais.

Uma declaração `if` (se) é usada para que o programa siga um ou outro caminho, dependendo do resultado da avaliação de uma expressão.

Este é o exemplo mais simples, que sempre é executado:

```
if (true) {
  //faça algo
}
```

ao contrário, isto jamais é executado:

```
if (false) {
  //faça algo (? nunca ?)
}
```

Comentário em português: “faça algo (? nunca ?)”.

A condicional verifica a expressão que você passar a ela em busca de um valor verdadeiro ou falso. Se você passar um número, este sempre será avaliado como verdadeiro, a menos que seja 0. Se você passar uma *string*, esta sempre será avaliada como verdadeira, a menos que seja uma *string* vazia. Essas são as regras gerais de conversão de tipos para um booleano.

Você notou o espaço entre as chaves? Isso é chamado de **bloco**. É usado para agrupar uma lista de diferentes instruções.

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

```
if (true) facaAlgo()
```

Eu sempre gosto de usar chaves para ser mais claro.

Você pode fornecer uma segunda parte para a instrução `if`: `else` (caso contrário).

Ou seja, você inclui uma instrução que será executada se a condição `if` for falsa:

```
if (true) {
  //faça algo
} else {
  //faça outra coisa
}
```

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

```
if (a === true) {
  //faça algo
} else if (b === true) {
  //faça outra coisa
} else {
  //faça uma terceira coisa se nenhuma das outras opções for válida
}
```

Arrays

Um *array* é uma coleção de elementos.

Arrays em JavaScript não são um *tipo* próprio.

Arrays são **objetos**.

Podemos inicializar um *array* vazio de 2 maneiras diferentes:

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

A primeira usa uma **sintaxe literal**. A segunda utiliza uma função construtora.

Você pode pré-preencher o *array* usando esta sintaxe:

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

Um *array* pode conter qualquer valor, mesmo valores de tipos diferentes:

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

Como podemos incluir um *array* dentro de um *array*, podemos criar *arrays* multidimensionais, que possuem aplicações muito úteis (por exemplo, uma matriz):

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

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

Você pode acessar qualquer elemento do *array*, fazendo referência ao seu índice, que inicia a partir de zero:

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

Você pode inicializar um novo *array* com um conjunto de valores usando esta sintaxe, que primeiro inicializa um *array* de 12 elementos e então preenche cada elemento com o número `0`:

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

Você pode obter o número de elementos no *array*, verificando sua propriedade `length` (tamanho):

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

Observe que você pode definir o comprimento do *array*. Se você atribui um número maior do que a capacidade atual do *array*, nada acontece. Se você atribui um número menor, o *array* é cortado naquela posição:

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

Como adicionar um item a um *array*

Podemos adicionar um elemento ao final de um *array* usando o método `push()`:

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

Podemos adicionar um elemento ao início de um *array* usando o método `unshift()`:

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

Como remover um item de um *array*

Podemos remover um item do final de um *array* usando o método `pop()`:

```
a.pop()
```

Podemos remover um item do início de um *array* usando o método `shift()`:

```
a.shift()
```

Como unir dois ou mais *arrays*

Você pode unir vários *arrays* 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 **spread** (`...`), deste modo:

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

Como encontrar um item específico no *array*

Você pode usar o método `find()` de um array:

```
a.find((elemento, indice, array) => {
  //retorna true ou false
})
```

Comentário em português: “retorna verdadeiro ou falso”.

Ele devolve o primeiro item que retornar verdadeiro e devolve `undefined` se o elemento não for encontrado.

Uma sintaxe comumente usada é:

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

A linha acima retornará o primeiro elemento do *array* que possui `id === my_id`.

`findIndex()` funciona de forma semelhante ao `find()`, mas devolve o índice do primeiro elemento que retornar verdadeiro. Se este não for encontrado, retorna `undefined`:

```
a.findIndex((elemento, indice, array) => {
  //retorna true ou false
})
```

Comentário em português: “retorna verdadeiro ou falso”.

Outro método é `includes()`:

```
a.includes(valor)
```

Ele retorna verdadeiro se `a` contém `valor`.

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

Retorna verdadeiro se `a` contém `valor` após a posição `i`.

Strings

Uma *string* é uma sequência de caracteres.

Ela também pode ser definida como uma *string* literal, contida entre aspas simples ou aspas duplas:

```
'Uma string'
"Outra string"
```

Pessoalmente, prefiro usar aspas simples o tempo todo. Uso aspas duplas apenas em HTML para definir atributos.

Você atribui um valor de *string* a uma variável assim:

```
const nome = 'Flavio'
```

Você pode determinar o comprimento de uma *string* usando a propriedade `length` (tamanho):

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

Esta é uma string vazia: `''`. Sua propriedade comprimento é 0:

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

Duas *strings* podem ser unidas usando operador `+`:

```
"Uma " + "string"
```

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

```
const nome = 'Flavio'
"Meu nome é " + nome //Meu nome é Flavio
```

Outra maneira de definir *strings* é usar *template literals*, que são definidos entre acentos graves. Eles são especialmente úteis para tornar *strings* de várias linhas muito mais simples. Com aspas simples ou duplas, você não pode definir uma *string* de múltiplas linhas facilmente – você precisaria usar caracteres de escape.

Uma vez que um *template literal* é aberto com um acento grave, basta apertar enter para criar uma linha, sem caracteres especiais, e ela é renderizada nos mesmos termos:

```
const string = `Olá
essa

string
é incrível!`
```

Em português: "Ei, essa string é incrível!".

Template literals também são ótimos porque fornecem uma maneira fácil de interpolar variáveis e expressões em *strings*.

Você faz isso usando a sintaxe `${...}`:

```
const var = 'teste'
const string = `algo ${var}`
//algo teste
```

Dentro do `${}` você pode incluir qualquer coisa, até expressões:

```
const string = `algo ${1 + 2 + 3}`
const string2 = `algo
  ${foo() ? 'x' : 'y'}`
```

Laços

Laços de repetição são uma das principais estruturas de controle do JavaScript.

Com um laço, podemos automatizar e repetir um bloco de código quantas vezes quisermos que ele seja executado, mesmo que indefinidamente.

O JavaScript fornece muitas maneiras de se iterar por meio de laços.

Eu quero me concentrar em 3 maneiras:

- laço while
- laço for
- laço for..of

`while`

O laço do tipo while é a estrutura de repetição mais simples que o JavaScript nos fornece.

Adicionamos uma condição após a palavra-chave `while` e fornecemos um bloco que é executado até que a condição seja avaliada como `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 laço `while` usando a palavra-chave `break`, assim:

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

Se você decidir que no meio de um laço você quer pular a iteração atual, pode avançar para a próxima iteração usando `continue`:

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

  //Faça outra coisa
}
```

Muito parecido com `while`, temos o laço `do..while`. É basicamente o mesmo que `while`, mas a condição é avaliada *depois* que o bloco de código é executado.

Isso significa que o bloco sempre é 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)
```

`for`

A segunda estrutura de repetição muito importante em JavaScript é o laço **for**.

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

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 nos laços `while`, você pode interromper um laço `for` usando `break` e avançar rapidamente para a próxima iteração usando `continue`.

`for...of`

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

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

for (const valor of list) {
  console.log(valor) //a b c
}
```

Funções

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

As funções são uma parte central e essencial do JavaScript.

O que é uma função?

Uma função é um bloco de código, contido em si mesma.

Aqui está a **declaração de uma função**:

```
function obterDados() {
  // Faça algo
}
```

Uma função pode ser executada a qualquer momento, basta invocá-la, assim:

```
obterDados()
```

Uma função pode ter um ou mais argumentos:

```
function obterDados() {
  //Faça algo
}

function obterDados(cor) {
  //Faça algo
}

function obterDados(cor, idade) {
  //Faça algo
}
```

Quando podemos passar um argumento, chamamos a função passando parâmetros:

```
function obterDados(cor, idade) {
  //Faça algo
}

obterDados('verde', 24)
obterDados('preto')
```

Observe que, na segunda chamada, eu passei como parâmetro a string `preto` para o argumento `cor`, mas não informei nenhum parâmetro para `idade`. Nesse caso, `idade` dentro da função permanece `undefined`.

Podemos verificar se um valor não é *undefined* usando esta condicional:

```
function obterDados(cor, idade) {
  //Faça algo
  if (typeof idade !== 'undefined') {
    //...
  }
}
```

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

Você também pode verificar deste modo:

```
function obterDados(cor, idade) {
  //Faça algo
  if (idade) {
    //...
```

```
  }
}
```

A condicional também será verdadeira se `idade` for `null`, `0` ou uma string vazia.

Você pode definir valores padrão (*default*, em inglês) para parâmetros, caso eles não sejam passados:

```
function obterDados(cor = 'preto', idade = 25) {
  //Faça algo
}
```

Você pode passar qualquer valor como parâmetro: números, strings, booleanos, arrays, objetos e funções.

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

```
function obterDados() {
  // Faça algo
  return 'Oi!'
}
```

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

```
function obterDados() {
  // Faça algo
  return 'Oi!'
}

let resultado = obterDados()
```

`result` agora contém uma *string* com o valor `Oi!`.

Você só pode retornar um valor.

Para retornar vários valores, você pode retornar um objeto, ou um *array*, assim:

```
function obterDados() {
  return ['Flavio', 37]
}

let [nome, idade] = obterDados()
```

Funções podem ser definidas dentro de outras funções:

```
const obterDados = () => {
  const fazerAlgo = () => {}
  fazerAlgo()
  return 'teste'
}
```

Porém, a função aninhada só pode ser invocada de dentro da função que a envolve.

Você também pode retornar uma função de uma função.

Arrow functions

Arrow functions foram introduzidas recentemente ao JavaScript.

Elas são frequentemente usadas no lugar de funções "regulares", aquelas que descrevi no capítulo anterior. Você encontrará ambas as formas sendo utilizadas em todo lugar.

Visualmente, elas permitem a escrita de funções com uma sintaxe mais curta, saindo de:

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

para:

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

Entretanto... observe que não temos um nome aqui.

Arrow functions são anônimas. Precisamos atribuí-las a uma variável.

Podemos atribuir uma função regular a uma variável, assim:

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

Ao fazer isso, podemos remover o nome da função:

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

e invocar a função usando o nome da variável:

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

É a mesma coisa que fazemos com *arrow functions*:

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

Se o corpo da função contiver apenas uma única instrução, você pode omitir as chaves e escrever tudo em uma única linha:

```
const obterDados = () => console.log('Oi!')
```

Os parâmetros são passados entre os parênteses:

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

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

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

Arrow functions permitem que você tenha um retorno implícito – os valores são retornados sem a necessidade de usar a palavra-chave `return`.

Isso funciona quando existe uma instrução de uma linha no corpo da função:

```
const obterDados = () => 'teste'

obterDados() //'teste'
```

Como nas funções regulares, podemos ter valores padrão para os parâmetros caso eles não sejam passados:

```
const obterDados = (cor = 'preto',
                    idade = 2) => {
  //Faça algo
}
```

Comentário em português: "faça algo".

Assim como ocorre com as funções regulares, podemos retornar apenas um valor.

Arrow functions também podem conter outras *arrow functions*, e até funções regulares.

Os dois tipos de funções são muito semelhantes. Então, você pode se perguntar por que as *arrow functions* foram introduzidas. A grande diferença para as funções regulares aparece quando elas são usadas como métodos de objetos. Isso é algo que veremos em breve.

Objetos

Qualquer valor que não seja de um tipo primitivo (uma string, um número, um booleano, um símbolo, *null* ou *undefined*) é um **objeto**.

Veja como definimos um objeto:

```
const carro = {

}
```

Essa é a **notação literal** de um objeto, que é uma das coisas mais legais em JavaScript.

Você também pode usar a sintaxe `new Object`:

```
const carro = new Object()
```

Outra sintaxe possível é usar `Object.create()`:

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

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

```
function Carro(marca, modelo) {
  this.marca = marca
  this.modelo = modelo
}
```

Inicializamos um novo objeto usando:

```
const meuCarro = new Carro('Ford', 'Fiesta')
meuCarro.marca //'Ford'
```

```
meuCarro.modelo //'Fiesta'
```

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

Se você atribuir a uma variável o mesmo valor de outra, se for um tipo primitivo, como um número ou uma *string*, eles serão passados como valor:

Veja este exemplo:

```
let idade = 36
let minhaIdade = idade
minhaIdade = 37
idade //36
```

```
const carro = {
  cor: 'azul'
}
const outroCarro = carro
outroCarro.cor = 'amarelo'
carro.cor //'amarelo'
```

Mesmo *arrays* ou funções são objetos por debaixo dos panos. Então, é muito importante entender como eles funcionam.

Propriedades dos objetos

Os objetos possuem **propriedades**, que são compostas por um nome (ou chave) associado a um valor.

O valor de uma propriedade pode ser de qualquer tipo, o que significa que pode ser um *array*, uma função e até mesmo um objeto, pois objetos podem ser aninhados em outros objetos.

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

```
const carro = {

}
```

Podemos definir uma propriedade `cor` dessa maneira:

```
const carro = {
  cor: 'azul'
}
```

Aqui temos um objeto `carro` com uma propriedade chamada `cor`, de valor `azul`.

Os nomes ou chaves que rotulam as propriedades podem ser qualquer *string*, mas tome cuidado com caracteres especiais – se eu quisesse incluir um caractere inválido para nomes de variáveis na chave da propriedade, teria que colocar o nome entre aspas:

```
const carro = {
  cor: 'azul',
  'a cor': 'azul'
}
```

Caracteres inválidos para nomes de variáveis incluem espaços, hifens e outros caracteres especiais.

Como você pode ver, 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 ponto**:

```
carro.cor //'azul'
```

A segunda (que é a única que podemos usar para propriedades com nomes inválidos), é usar a notação de colchetes:

```
carro['a cor'] //'azul'
```

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

```
carro.marca //undefined
```

Como já mencionado anteriormente, objetos podem ter outros objetos aninhados como propriedades:

```
const carro = {
  marca: {
    nome: 'Ford'
  },
  cor: 'azul'
}
```

Neste exemplo, você pode acessar o nome da marca usando

```
carro.marca.nome
```

ou

```
carro['marca']['nome']
```

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

Você, no entanto, sempre pode atualizá-lo mais tarde:

```
const carro = {
  cor: 'azul'
}

carro.cor = 'amarelo'
carro['cor'] = 'vermelho'
```

Você também pode adicionar novas propriedades a um objeto:

```
carro.modelo = 'Fiesta'

carro.modelo //'Fiesta'
```

Dado o objeto

```
const carro = {
  cor: 'azul',
  marca: 'Ford'
}
```

você pode excluir uma propriedade dele usando

```
delete carro.marca
```

Métodos de objetos

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 `ligar` tem uma função atribuída e podemos invocá-la usando a sintaxe de ponto que usamos para propriedades, com os parênteses no final:

```
const carro = {
  marca: 'Ford',
  modelo: 'Fiesta',
  ligar: function() {
    console.log('Ligado')
  }
}

carro.ligar()
```

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

No exemplo a seguir, temos acesso aos valores das propriedades `marca` e `modelo` usando `this.marca` e `this.modelo`:

```
const carro = {
  marca: 'Ford',
  modelo: 'Fiesta',
  ligar: function() {
    console.log(`${this.marca} ${this.modelo} ligado!`)
  }
}

car.ligar()
```

É importante observar essa diferença entre funções regulares e *arrow functions* – não temos acesso ao `this` ao usar uma *arrow function*:

```
const carro = {
  marca: 'Ford',
  modelo: 'Fiesta',
  ligar: () => {
    console.log(`${this.marca} ${this.modelo} ligado!`) //não vai funcionar
  }
}

carro.ligar()
```

Isso acontece porque as ***arrow functions* não estão vinculadas ao objeto**.

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

Métodos podem aceitar parâmetros, como funções regulares:

```
const carro = {
  marca: 'Ford',
  modelo: 'Fiesta',
  irPara: function(destino) {
    console.log(`Indo para ${destino}`)
  }
}

carro.irPara('Roma')
```

Classes

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

Neste capítulo, subiremos um degrau ao introduzirmos as classes.

O que são classes? Elas são uma maneira de definir um padrão comum para múltiplos objetos.

Vamos pegar um objeto `pessoa`:

```
const pessoa = {
  nome: 'Flavio'
}
```

Podemos criar uma classe chamada `Pessoa` (observe o `P` maiúsculo, uma convenção ao usar classes), que possui uma propriedade `nome`:

```
class Pessoa {
  nome
}
```

Agora, a partir dessa classe, inicializamos um objeto `flavio` assim:

```
const flavio = new Pessoa()
```

`flavio` é chamado de uma *instância* da classe Pessoa.

Podemos definir o valor da propriedade `nome`:

```
flavio.nome = 'Flavio'
```

e podemos acessá-lo usando

```
flavio.nome
```

assim como fazemos para as propriedades do objeto.

As classes podem conter propriedades, como `nome`, e métodos.

Os métodos são definidos desta maneira:

```
class Pessoa {
  ola() {
    return 'Olá, meu nome é Flavio'
  }
}
```

e podemos invocar métodos em uma instância da classe:

```
class Pessoa {
  ola() {
    return 'Olá, meu nome é Flavio'
  }
}
const flavio = new Pessoa()
flavio.ola()
```

Existe um método especial chamado `constructor()`, que podemos usar para inicializar as propriedades da classe quando criamos uma instância de objeto.

Funciona assim:

```
class Pessoa {
  constructor(nome) {
    this.nome = nome
  }

  ola() {
    return 'Olá, meu nome é ' + this.nome + '.'
  }
}
```

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

Agora, podemos instanciar um novo objeto da classe, passar uma *string* e, quando chamarmos `ola`, receberemos uma mensagem personalizada:

```
const flavio = new Pessoa('flavio')
flavio.ola() //'Olá, meu nome é flavio.'
```

Quando o objeto é inicializado, o método `constructor` é chamado com qualquer parâmetro passado.

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

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

```
class Pessoa {
  static olaGenerico() {
    return 'Olá'
  }
}

Pessoa.olaGenerico() //Olá
```

Isso é muito útil, às vezes.

Herança

Uma classe pode ser uma **extensão** de outra classe de modo que os objetos inicializados usando essa classe herdam todos os métodos de ambas as classes.

Suponha que temos uma classe `Pessoa`:

```
class Pessoa {
  ola() {
    return 'Olá, eu sou uma Pessoa'
  }
}
```

Podemos definir uma nova classe, `Programador`, que estende `Pessoa`:

```
class Programador extends Pessoa {

}
```

Agora, se instanciarmos um novo objeto com a classe `Programador`, ele tem acesso ao método `ola()` de `Pessoa`:

```
const flavio = new Programador()
flavio.ola() //'Olá, eu sou uma Pessoa'
```

Dentro de uma classe filha, você pode referenciar a classe pai chamando `super()`:

```
class Programador extends Pessoa {
  ola() {
    return super.ola() +
      '. Eu também sou um programador.'
  }
}

const flavio = new Programador()
flavio.ola()
```

O código acima imprime na tela *Olá, eu sou uma Pessoa. Eu também sou um programador*.

Programação assíncrona e *callbacks*

Na maioria das vezes, o código JavaScript é executado de modo síncrono.

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

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

Todavia, há momentos em que você não pode esperar que uma linha de código seja executada.

Você simplesmente não pode esperar 2 segundos para carregar um arquivo grande e parar o programa completamente enquanto isso.

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

O JavaScript resolve esse problema usando ***callbacks***.

Um dos exemplos mais simples de como *callbacks* são usadas é com temporizadores (*timers*). Os temporizadores não fazem parte do JavaScript, mas são fornecidos pelo navegador e pelo Node.js. Deixe-me falar sobre um pouco sobre os *timers* que temos: `setTimeout()`.

A função `setTimeout()` aceita 2 argumentos: uma função e um número. O número são os milissegundos que devem passar antes que a função seja executada.

Exemplo:

```
setTimeout(() => {
  // executa após 2 segundos
  console.log('dentro da função')
}, 2000)
```

Comentário em português: "executa após 2 segundos".

A função que contém a linha `console.log('dentro da função')` será executada após 2 segundos.

Se você adicionar um `console.log('antes')` antes da função e um `console.log('depois')` depois dela, teremos:

```
console.log('antes')
setTimeout(() => {
  // executa após 2 segundos
  console.log('dentro da função')
}, 2000)
console.log('depois')
```

Você verá isto no seu console:

```
antes
depois
```

```
dentro da função
```

A função de *callback* é executada de modo assíncrono.

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

Todas as coisas que mencionei não são JavaScript "básico". Portanto, não são explicadas neste manual, mas você encontrará muitos exemplos em meus outros manuais disponíveis em https://flaviocopes.com.

Veja como podemos implementar *callbacks* em nosso código.

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

Quando o código estiver pronto para invocar a *callback*, nós a invocamos passando o resultado:

```
const fazerAlgo = callback => {
  //Faça coisas
  //Faça coisas
  const resultado = /* .. */
  callback(resultado)
}
```

Um código que utiliza essa função, a usaria assim:

```
fazerAlgo(resultado => {
  console.log(resultado)
})
```

Promises

Promises, ou promessas, são uma maneira alternativa de lidar com código assíncrono.

Como vimos no capítulo anterior, com *callbacks*, estaríamos passando uma função para outra chamada de função, que seria invocada quando a função terminasse de ser processada.

Assim:

```
fazerAlgo(resultado => {
  console.log(resultado)
})
```

Quando o código `fazerAlgo()` termina, ele chama a função recebida como parâmetro:

```
const fazerAlgo = callback => {
  //Faça coisas
  //Faça coisas
  const resultado = /* .. */
  callback(resultado)
}
```

O principal problema dessa abordagem é que, se precisarmos usar o resultado dessa função no restante do nosso código, todo o nosso código deve estar aninhado dentro da função de *callback* e, se tivermos que fazer 2 a 3 *callbacks*, entramos no que geralmente chamamos de "*callback hell*" (em português, inferno das callbacks) com muitos níveis de funções aninhadas em outras funções:

```
fazerAlgo(resultado => {
  fazerOutraCoisa(outroResultado => {
    fazerMaisUmaCoisa(maisUmResultado => {
      console.log(resultado)
    })
  })
})
```

Promises são uma maneira de lidar com isso.

Em vez de fazer:

```
fazerAlgo(resultado => {
  console.log(resultado)
})
```

Chamamos uma função baseada em uma *promise*, desta maneira:

```
fazerAlgo()
  .then(resultado => {
    console.log(resultado)
  })
```

Primeiro chamamos a função. Em seguida, temos o método `then()`, que é chamado quando a função termina.

O recuo não importa, mas você frequentemente usará esse estilo de identação para maior clareza.

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

```
fazerAlgo()
  .then(resultado => {
    console.log(resultado)
  })
  .catch(erro => {
    console.log(erro)
  })
```

Agora, para poder usar essa sintaxe, a implementação da função `fazerAlgo()` deve ser um pouco especial. Ela deve usar a API *Promises*.

Em vez de declará-la como uma função normal:

```
const fazerAlgo = () => {

}
```

Nós a declaramos como um objeto *promise*:

```
const fazerAlgo = new Promise()
```

e passamos uma função no construtor de *Promise*:

```
const fazerAlgo = new Promise(() => {

})
```

Essa função recebe 2 parâmetros. O primeiro é uma função que chamamos para resolver a *promise*; o segundo é uma função que chamamos para rejeitá-la.

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

})
```

Resolver uma *promise* significa completá-la com sucesso (o que resulta na chamada do método `then()` em qualquer lugar que o utilize).

Rejeitar uma *promise* significa encerrá-la com um erro (o que resulta na chamada do método `catch()` em qualquer lugar que o utilize).

Veja como:

```
const fazerAlgo = new Promise(
  (resolve, reject) => {
    //um código
    const sucesso = /* ... */
    if (sucesso) {
      resolve('ok')
    } else {
      reject('um erro ocorreu')
    }
  }
)
```

Podemos passar um parâmetro para as funções *resolve* e *reject* de qualquer tipo que desejarmos.

Async e await

As funções assíncronas são uma abstração de alto nível das *promises*.

Uma função assíncrona retorna uma *promise*, como neste exemplo:

```
const obterDados = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() =>
      resolve('dados'), 2000)
  })
}
```

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

```
const dados = await obterDados()
```

Ao fazer isso, todos os dados retornados pela *promise* serão atribuídos à variável `dados`.

No nosso caso, os dados são apenas a string "dados".

Há uma ressalva: toda vez que usamos a palavra-chave `await`, precisamos fazer isso dentro de uma função definida como `async`.

Assim:

```
const fazerAlgo = async () => {
  const dados = await obterDados()
  console.log(dados)
}
```

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 bem mais simples. Compare-o ao código usando *promises* ou funções de *callback*.

Esse é um exemplo bem simples. Os maiores benefícios surgirão quando o código for muito mais complexo.

A título de exemplo, veja como você receberia um recurso em JSON usando a API *Fetch* e o analisaria, usando *promises*:

```
const pegarDadosDoPrimeiroUsuario = () => {
  // obter a lista de usuarios
  return fetch('/usuarios.json')
    // fazer o parsing do JSON
    .then(response => resposta.json())
    // pegar o primeiro usuario
    .then(usuarios => usuarios[0])
    // obter dados do usuario
    .then(usuario =>
      fetch(`/users/${usuario.nome}`))
    // fazer o parsing do JSON
    .then(respostaDoUsuario => resposta.json())
}

pegarDadosDoPrimeiroUsuario()
```

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

```
const pegarDadosDoPrimeiroUsuario = async () => {
  // obter a lista de usuarios
  const resposta = await fetch('/usuarios.json')
  // fazer o parsing do JSON
  const usuarios = await response.json()
  // pegar o primeiro usuario
  const usuario = usuarios[0]
  // obter dados do usuario
  const respostaDoUsuario =
    await fetch(`/users/${usuario.nome}`)
  // fazer o parsing do JSON
  const dadosDoUsuario = await usuario.json()
  return dadosDoUsuario
}

pegarDadosDoPrimeiroUsuario()
```

Escopos de variáveis

Quando introduzi variáveis, falei sobre usar `const`, `let`, e `var`.

Escopo é o conjunto de variáveis que estão visíveis para uma determinada parte do programa.

No JavaScript, temos um escopo global, um escopo de bloco e um 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 do

programa.

Há uma diferença muito importante entre as declarações de variáveis com `var`, `let` e `const`.

Uma variável definida como `var` dentro de uma função só fica visível dentro dessa função, de maneira semelhante aos seus argumentos.

Uma variável definida como `const` ou `let`, por outro lado, só é visível dentro do **bloco** onde foi definida.

Um bloco é um conjunto de instruções agrupadas em um par de chaves, como as que encontramos dentro de uma instrução `if`, um laço `for` ou uma função.

É importante entender que um bloco não define um novo escopo para `var`, mas o faz isso para `let` e `const`.

Isso tem implicações muito práticas.

Suponha que você defina uma variável `var` dentro de uma condicional `if` em uma função

```
function obterDados() {
  if (true) {
    var dados = 'dados'
    console.log(dados)
  }
}
```

Se você chamar essa função, terá `dados` sendo impresso no console.

Se você mover `console.log(dados)` para após o `if`, ele ainda funcionará:

```
function obterDados() {
  if (true) {
    var dados = 'dados'
  }
  console.log(dados)
}
```

Porém, se você mudar `var data` para `let data`:

```
function obterDados() {
  if (true) {
    let dados = 'dados'
  }
  console.log(dados)
}
```

Você receberá um erro: `ReferenceError: dados is not defined` (erro de referência: dados não foi definida).

Isso acontece porque `var` possui escopo de função. Há algo especial acontecendo aqui que chamamos de *hoisting* (em português, "içamento"). Resumindo, a declaração `var` é movida para o topo da função mais próxima pelo JavaScript antes de executar o código. É assim, mais ou menos, que o JS vê a função internamente:

```
function obterDados() {
  var dados
  if (true) {
    dados = 'dados'
  }
  console.log(dados)
}
```

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

```
function obterDados() {
  console.log(dados)
  if (true) {
    var dados = 'dados'
  }
}
```

Porém, se você mudar para `let`, receberá um erro `ReferenceError: dados is not defined`, porque o *hoisting* não acontece em declarações com `let`.

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

Pode ser complicado no começo, mas assim que você perceber essa diferença, verá por que usar `var` é considerada uma má prática hoje em dia em comparação com o uso do

`let` – que possui menos partes móveis e escopo limitado ao bloco, o que também torna `let` muito bom como variável para laços porque deixa de existir após o término da repetição:

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

fazerLaco()
```

Ao sair do laço, `i` será uma variável válida e com valor 10.

Se você mudar para `let`, quando usar `console.log(i)` receberá um erro `ReferenceError: i is not defined` (erro de referência: i não foi definido).

Conclusão

Muito obrigado por ter lido este livro.

Espero que ele sirva de inspiração para aprender mais sobre JavaScript.

Para saber mais sobre JavaScript, confira o blog do autor: flaviocopes.com.

> Observação: você pode baixar uma versão em PDF ou ePub deste Manual de JavaScript para Iniciantes inscrevendo-se na newsletter do autor *(em inglês)*.

Tradutor: Rafael B. Pires

Rookie web developer. Apaixonado por tecnologia e por compartilhar conhecimento. Entusiasta do mundo open-source e da filosofia colaborativa.

Autor: Flavio Copes (em inglês)

Ler mais publicações.

Se você leu até aqui, agradeça ao autor para mostrar que você se importa com o trabalho.

Agradeça

Aprenda a programar gratuitamente. O plano de estudos em código aberto do freeCodeCamp já ajudou mais de 40.000 pessoas a obter empregos como desenvolvedores. Comece agora

O freeCodeCamp é uma organização beneficente 501(c)(3), isenta de impostos e apoiada por doações (Número de identificação fiscal federal dos Estados Unidos: 82-0779546).

Nossa missão: ajudar as pessoas a aprender a programar de forma gratuita. Conseguimos isso criando milhares de vídeos, artigos e lições de programação interativas, todas disponíveis gratuitamente para o público.

As doações feitas ao freeCodeCamp vão para nossas iniciativas educacionais e ajudam a pagar servidores, serviços e a equipe.

Você pode fazer uma doação dedutível de impostos aqui.

Livros e manuais mais procurados

Nova aba em HTML
Tutorial de button onClick
Excluir pastas com o cmd
Excluir branches
Arrays vazios em JS
Provedores de e-mail
10 sites de desafios
Var, let e const em JavaScript
Fotos do Instagram
SQL em Python
Máscaras de sub-rede
Bot do Discord
Imagens em CSS
Jogo do dinossauro
Caracteres especiais
15 portfólios
Clonar branches
Axios em React
Media queries do CSS
Interpretadas x compiladas
40 projetos em JavaScript
Centralizar em CSS
25 projetos em Python
Menu iniciar
Python para iniciantes
Node.js no Ubuntu
Date now em JavaScript
ForEach em JavaScript
Fix do Live Server no VS Code
Imagens SVG em HTML e CSS

Aplicativo móvel

Download on the App Store
GET IT ON Google Play

Nossa instituição

Sobre Rede de ex-alunos Código aberto Loja Apoio Patrocinadores Honestidade acadêmica Código de conduta
Política de privacidade Termos de serviço Política de direitos de autor