 This page was translated from English by the community. Learn more and join the MDN Web Docs community.

# Sintaxe e tipos

Este capítulo trata sobre a sintaxe básica do JavaScript, declarações de variáveis, tipos de dados e literais.

## Sintaxe básica

JavaScript pega emprestado a maior parte de sua sintaxe do Java, mas também é influenciado por Awk, Perl e Python.

JavaScript é **case-sensitive** e usa o conjunto de caracteres **Unicode**. Por exemplo, a palavra Früh (que significa "cedo" em Alemão) pode ser usada como nome de variável.

```
var Früh = "foobar";
```



Mas a variável `früh` não é a mesma que `Früh` porque JavaScript é case sensitive.

No JavaScript, instruções são chamadas de [declaração](#) e são separadas por um ponto e vírgula (;). Espaços, tabulação e uma nova linha são chamados de espaços em branco. O código fonte dos scripts em JavaScript são lidos da esquerda para a direita e são convertidos em uma sequência de elementos de entrada como símbolos, caracteres de controle, terminadores de linha, comentários ou espaço em branco. ECMAScript também define determinadas palavras-chave e literais, e tem regras para inserção automática de ponto e vírgula ([ASI](#)) para terminar as declarações. No entanto, recomenda-se sempre adicionar ponto e vírgula no final de suas declarações; isso evitará alguns imprevistos. Para obter mais informações, consulte a referência detalhada sobre a [gramática léxica](#) do JavaScript.

# Comentários

A sintaxe dos comentários em JavaScript é semelhante como em C++ e em muitas outras linguagens:

```
// comentário de uma linha

/* isto é um comentário longo
   de múltiplas linhas.
  */

/* Você não pode, porém, /* aninhar comentários */ SyntaxError */
```



## Declarações

Existem três tipos de declarações em JavaScript.

**var**

Declara uma variável, opcionalmente, inicializando-a com um valor.



**let**

Declara uma variável local de escopo do bloco, opcionalmente, inicializando-a com um valor.



**const**

Declara uma constante de escopo de bloco, apenas de leitura.

## Variáveis

Você usa variáveis como nomes simbólicos para os valores em sua aplicação. O nome das variáveis, chamados de [identificadores](#), obedecem determinadas regras.

Um identificador JavaScript deve começar com uma letra, underline ( `_` ), ou cifrão ( `$` ); os caracteres subsequentes podem também ser números (0-9). Devido JavaScript ser case-sensitive, letras incluem caracteres de "A" a "Z" (maiúsculos) e caracteres de "a" a "z" (minúsculos).

Você pode usar a ISO 8859-1 ou caracteres Unicode tal como os identificadores `å` e `ü`.

Você pode também usar as [sequências de escape Unicode](#) como caracteres e identificadores.

Alguns exemplos de nomes legais são `Numeros_visitas`, `temp99`, e `_nome`.

## Declarando variáveis

Você pode declarar uma variável de três formas:

- Com a palavra chave `var`. Por exemplo, `var x = 42`. Esta sintaxe pode ser usada para declarar tanto variáveis locais como variáveis globais.
- Por simples adição de valor. Por exemplo, `x = 42`. Isso declara uma variável global. Essa declaração gera um aviso de advertência no JavaScript. Você não deve usar essa variante.
- Com a palavra chave `let`. Por exemplo, `let y = 13`. Essa sintaxe pode ser usada para declarar uma variável local de escopo de bloco. Veja **escopo de variável** abaixo.

## Classificando variáveis

Uma variável declarada usando a declaração `var` ou `let` sem especificar o valor inicial tem o valor [undefined](#).

Uma tentativa de acessar uma variável não declarada resultará no lançamento de uma exceção [ReferenceError](#):

```
var a;  
console.log("O valor de a é " + a); // saída "O valor de a é undefined"  
console.log("O valor de b é " + b); // executa uma exception de erro de referência
```

Você pode usar `undefined` para determinar se uma variável tem um valor. No código a seguir, não é atribuído um valor de entrada na variável e a declaração `if` será avaliada como verdadeira (`true`).

```
var input;  
if(input === undefined){  
    facaIsto();  
} else {
```

```
facaAquila();  
}
```

O valor `undefined` se comporta como falso ( `false` ), quando usado em um contexto booleano. Por exemplo, o código a seguir executa a função `myFunction` devido o elemento `myArray` ser `undefined`:

```
var myArray = [];  
if (!myArray[0]) myFunction();
```



O valor `undefined` converte-se para `NaN` quando usado no contexto numérico.

```
var a;  
a + 2; // Avaliado como NaN
```



Quando você avalia uma variável nula, o valor nulo se comporta como 0 em contextos numéricos e como falso em contextos booleanos. Por exemplo:

```
var n = null;  
console.log(n * 32); // a saída para o console será 0.
```



## Escopo de variável

Quando você declara uma variável fora de qualquer função, ela é chamada de variável *global*, porque está disponível para qualquer outro código no documento atual. Quando você declara uma variável dentro de uma função, é chamada de variável *local*, pois ela está disponível somente dentro dessa função.

JavaScript antes do ECMAScript 6 não possuía escopo de [declaração de bloco](#); pelo contrário, uma variável declarada dentro de um bloco de uma *função* é uma variável local (ou contexto *global*) do bloco que está inserido a função. Por exemplo o código a seguir exibirá 5, porque o escopo de `x` está na função (ou contexto global) no qual `x` é declarado, não o bloco, que neste caso é a declaração `if`.

```
if (true) {  
  var x = 5;  
}  
console.log(x); // 5
```



```
console.log(x); // 5
```

Esse comportamento é alterado, quando usado a declaração `let` introduzida pelo ECMAScript 6.

```
if (true) {  
  let y = 5;  
  
}  
console.log(y); // ReferenceError: y não está definido
```

## Variável de elevação

Outra coisa incomum sobre variáveis em JavaScript é que você pode utilizar a variável e declará-la depois, sem obter uma exceção. Este conceito é conhecido como **hoisting**; variáveis em JavaScript são num sentido "hoisted" ou lançada para o topo da função ou declaração. No entanto, as variáveis que são "hoisted" retornarão um valor `undefined`. Então, mesmo se você usar ou referir a variável e depois declará-la e inicializá-la, ela ainda retornará `undefined`.

```
/**  
 * Exemplo 1  
 */  
console.log(x === undefined); // exibe "true"  
var x = 3;  
  
/**  
 * Exemplo 2  
 */  
// retornará um valor undefined  
var myvar = "my value";  
  
(function() {  
  console.log(myvar); // undefined  
  var myvar = "local value";  
})();
```

Os exemplos acima serão interpretados como:

```
/**
```

```

^ Exemplo 1
*/
var x;
console.log(x === undefined); // exibe "true"
x = 3;

/**
 * Exemplo 2
 */
var myvar = "um valor";

(function() {
  var myvar;
  console.log(myvar); // undefined
  myvar = "valor local";
})();

```

Devido o hoisting, todas as declarações `var` em uma função devem ser colocadas no início da função. Essa recomendação de prática deixa o código mais legível.

## Variáveis Globais

Variáveis globais são propriedades do *objeto global*. Em páginas web o objeto global é a `window`, assim você pode configurar e acessar variáveis globais utilizando a sintaxe `window.variavel`.

Consequentemente, você pode acessar variáveis globais declaradas em uma janela ou frame ou frame de outra janela. Por exemplo, se uma variável chamada `phoneNumber` é declarada em um documento, você pode consultar esta variável de um frame como `parent.phoneNumber`.

## Constantes

Você pode criar uma constante apenas de leitura por meio da palavra-chave `const`. A sintaxe de um identificador de uma constante é semelhante ao identificador de uma variável: deve começar com uma letra, sublinhado ou cifrão e pode conter caractere alfabético, numérico ou sublinhado.

```
const PI = 3.14;
```



Uma constante não pode alterar seu valor por meio de uma atribuição ou ser declarada novamente enquanto o script está em execução. Deve ser inicializada com um valor.

As regras de escopo para as constantes são as mesmas para as variáveis `let` de escopo de bloco. Se a palavra-chave `const` for omitida, presume-se que o identificador represente uma variável.

Você não pode declarar uma constante com o mesmo nome de uma função ou variável que estão no mesmo escopo. Por exemplo:

```
// Isto irá causar um erro
function f() {};
const f = 5;

// Isto também irá causar um erro.
function f() {
  const g = 5;
  var g;

  //declarações
}
```



## Estrutura de dados e tipos

### Tipos de dados

O mais recente padrão ECMAScript define sete tipos de dados:

- Seis tipos de dados são os chamados [primitivos](#):
  - [Boolean](#). `true` e `false`.
  - [null](#). Uma palavra-chave que indica valor nulo. Devido JavaScript ser case-sensitive, `null` não é o mesmo que `Null`, `NULL`, ou ainda outra variação.
  - [undefined](#). Uma propriedade superior cujo valor é indefinido.
  - [Number](#). `42` ou `3.14159`.
  - [String](#). `"Howdy"`

- [Symbol](#) (novo em ECMAScript 6). Um tipo de dado cuja as instâncias são únicas e imutáveis.
- e [Object](#)

Embora esses tipos de dados sejam uma quantidade relativamente pequena, eles permitem realizar funções úteis em suas aplicações. [Objetos](#) e [funções](#) são outros

elementos fundamentais na linguagem. Você pode pensar em objetos como recipientes para os valores, e funções como métodos que suas aplicações podem executar.

## Conversão de tipos de dados

JavaScript é uma linguagem dinamicamente tipada. Isso significa que você não precisa especificar o tipo de dado de uma variável quando declará-la, e tipos de dados são convertidos automaticamente conforme a necessidade durante a execução do script. Então, por exemplo, você pode definir uma variável da seguinte forma:

```
var answer = 42;
```

E depois, você pode atribuir uma string para a mesma variável, por exemplo:

```
answer = "Obrigado pelos peixes...";
```

Devido JavaScript ser dinamicamente tipado, essa declaração não gera uma mensagem de erro.

Em expressões envolvendo valores numérico e string com o operador +, JavaScript converte valores numérico para strings. Por exemplo, considere a seguinte declaração:

```
x = "A resposta é " + 42 // "A resposta é 42"  
y = 42 + " é a resposta" // "42 é a resposta"
```

Nas declarações envolvendo outros operadores, JavaScript não converte valores numérico para strings. Por exemplo:

```
"37" - 7 // 30  
"37" + 7 // "377"
```



## Convertendo strings para números

No caso de um valor que representa um número está armazenado na memória como uma string, existem métodos para a conversão.

- [parseInt\(\)](#)
- [parseFloat\(\)](#)

`parseInt` irá retornar apenas números inteiros, então seu uso é restrito para a casa dos decimais. Além disso, é uma boa prática ao usar `parseInt` incluir o parâmetro da base. O parâmetro da base é usado para especificar qual sistema numérico deve ser usado.

Uma método alternativo de conversão de um número em forma de string é com o operador `+` (operador soma):

```
"1.1" + "1.1" = "1.11.1"  
(+"1.1") + (+"1.1") = 2.2  
// Nota: Os parênteses foram usados para deixar mais legível o código, ele
```

## Literais

Você usa literais para representar valores em JavaScript. Estes são valores fixados, não variáveis, que você literalmente insere em seu script. Esta seção descreve os seguintes tipos literais:

- [Array literal](#)
- [Literais boolean](#)
- [Literais de ponto flutuante](#)
- [Inteiros](#)
- [Objeto literal](#)
- [String literal](#)

### Array literal

Um array literal é uma lista de zero ou mais expressões, onde cada uma delas representam um elemento do array, inseridas entre colchetes ( `[]` ). Quando você cria um array usando um array literal, ele é inicializado com os valores especificados como seus elementos, e seu comprimento é definido com o número de elementos especificados.

O exemplo a seguir cria um array `coffees` com três elementos e um comprimento de três:

```
var coffees = ["French Roast", "Colombian", "Kona"];
```



**Nota :** Um array literal é um tipo de inicializador de objetos. Veja [Usando inicializadores de Objetos](#).

Se um array é criado usando um literal no topo do script, JavaScript interpreta o array cada vez que avalia a expressão que contém o array literal. Além disso, um literal usado em uma função é criado cada vez que a função é chamada.

Array literal são também um array de objetos. Veja [Array](#) e [Coleções indexadas](#) para detalhes sobre array de objetos.

### Vírgulas extras em array literal

Você não precisa especificar todos os elementos em um array literal. Se você colocar duas vírgulas em uma linha, o array é criado com `undefined` para os elementos não especificados. O exemplo a seguir cria um array chamado `fish`:

```
var fish = ["Lion", , "Angel"];
```



Esse array tem dois elementos com valores e um elemento vazio ( `fish[0]` é "Lion", `fish[1]` é `undefined`, e `fish[2]` é "Angel" ).

Se você incluir uma vírgula à direita no final da lista dos elementos, a vírgula é ignorada. No exemplo a seguir, o comprimento do array é três. Não há nenhum `myList[3]`. Todas as outras vírgulas na lista indicam um novo elemento.

**Nota :** Vírgulas à direita podem criar erros em algumas versões de navegadores web antigos. É recomendável removê-las.

arrays, e recomendamos removê-los.

```
var myList = ['home', , 'school', ];
```



No exemplo a seguir, o comprimento do array é quatro, e `myList[0]` e `myList[2]` são `undefined`.

```
var myList = [ , 'home', , 'school'];
```



No exemplo a seguir, o comprimento do array é quatro, e `myList[1]` e `myList[3]` são `undefined`. Apenas a última vírgula é ignorada.

```
var myList = ['home', , 'school', , ];
```



Entender o comportamento de vírgulas extras é importante para a compreensão da linguagem JavaScript, no entanto, quando você escrever seu próprio código: declarar explicitamente os elementos em falta como `undefined` vai aumentar a clareza do código, e conseqüentemente na sua manutenção.

## Literais Boolean

O tipo Boolean tem dois valores literal: `true` e `false`.

Não confunda os valores primitivos Boolean `true` e `false` com os valores `true` e `false` do objeto Boolean. O objeto Boolean é um invólucro em torno do tipo de dado primitivo.

Veja [Boolean](#) para mais informação.

## Inteiros

Inteiros podem ser expressos em decimal (base 10), hexadecimal (base 16), octal (base 8) e binário (base 2).

- Decimal inteiro literal consiste em uma sequência de dígitos sem um 0 (zero).
- 0 (zero) em um inteiro literal indica que ele está em octal. Octal pode incluir somente os dígitos 0-7.
- 0x (ou 0X) indica um hexadecimal. Inteiros hexadecimais podem incluir dígitos (0-9) e as letras a-f e A-F.

- 0b (ou 0B) indica um binário. Inteiros binário podem incluir apenas os dígitos 0 e 1.

Alguns exemplos de inteiros literal são:

```
0, 117 and -345 (decimal, base 10)
015, 0001 and -077 (octal, base 8)
0x1123, 0x00111 and -0xF1A7 (hexadecimal, "hex" or base 16)
0b11, 0b0011 and -0b11 (binário, base 2)
```

Para maiores informações, veja [Literais numérico na referência Léxica](#).

## Literais de ponto flutuante

Um literal de ponto flutuante pode ter as seguintes partes:

- Um inteiro decimal que pode ter sinal (precedido por "+" ou "-"),
- Um ponto decimal ("."),
- Uma fração (outro número decimal),
- Um expoente.

O expoente é um "e" ou "E" seguido por um inteiro, que pode ter sinal (precedido por "+" ou "-"). Um literal de ponto flutuante deve ter no mínimo um dígito e um ponto decimal ou "e" (ou "E").

Mais sucintamente, a sintaxe é:

```
[(+|-)][digitos][.digitos][(E|e)[(+|-)]digitos]
```

Por exemplo:

```
3.1415926
-.123456789
-3.1E+12
.1e-23
```

## Objeto literal

Um objeto literal é uma lista de zero ou mais pares de nomes de propriedades e valores associados de um objeto, colocado entre chaves ( `{ }` ). Você não deve usar um objeto

literal no início de uma declaração. Isso levará a um erro ou não se comportará conforme o esperado, porque o `{` será interpretado como início de um bloco.

Segue um exemplo de um objeto literal. O primeiro elemento do objeto `carro` define uma propriedade, `meuCarro`, e atribui para ele uma nova string, "Punto"; o segundo elemento, a propriedade `getCarro`, é imediatamente atribuído o resultado de chamar uma função

(`tipoCarro("Fiat")`); o terceiro elemento, a propriedade `especial`, usa uma variável existente (`vendas`).

```
var vendas = "Toyota";

function tipoCarro(nome) {
  if (nome == "Fiat") {
    return nome;
  } else {
    return "Desculpa, não vendemos carros " + nome + ".";
  }
}

var carro = { meuCarro: "Punto", getCarro: tipoCarro("Fiat"), especial: vendas };

console.log(carro.meuCarro); // Punto
console.log(carro.getCarro); // Fiat
console.log(carro.especial); // Toyota
```

Além disso, você pode usar um literal numérico ou string para o nome de uma propriedade ou aninhar um objeto dentro do outro. O exemplo a seguir usa essas opções.

```
var carro = { carros: {a: "Saab", "b": "Jeep"}, 7: "Mazda" };

console.log(carro.carros.b); // Jeep
console.log(carro[7]); // Mazda
```

Nomes de propriedades de objeto podem ser qualquer string, incluindo uma string vazia. Caso o nome da propriedade não seja um [identificador](#) JavaScript ou número, ele deve ser colocado entre aspas. Nomes de propriedades que não possuem identificadores válidos, também não podem ser acessadas pela propriedade de ponto (`.`), mas podem ser

acessadas e definidas com a notação do tipo array ("[]").

```
var unusualPropertyNames = {  
  "": "Uma string vazia",  
  "!": "Bang!"  
}  
  
console.log(unusualPropertyNames.""); // SyntaxError: string inesperada  
console.log(unusualPropertyNames[""]); // Um string vazia  
  
console.log(unusualPropertyNames.!); // SyntaxError: símbolo ! inesperado  
console.log(unusualPropertyNames["!"]); // Bang!
```

Observe:

```
var foo = {a: "alpha", 2: "two"};  
console.log(foo.a); // alpha  
console.log(foo[2]); // two  
//console.log(foo.2); // Error: missing ) after argument list  
//console.log(foo[a]); // Error: a não está definido  
console.log(foo["a"]); // alpha  
console.log(foo["2"]); // two
```

## Expressão Regex Literal

Um regex literal é um padrão entre barras. A seguir um exemplo de regex literal.

```
var re = /ab+c/;
```

## String Literal

Uma string literal são zero ou mais caracteres dispostos em aspas duplas (") ou aspas simples ('). Uma sequência de caracteres deve ser delimitada por aspas do mesmo tipo; ou seja, as duas aspas simples ou ambas aspas duplas. A seguir um exemplo de strings literais.

```
"foo"  
'bar'  
"1234"  
"um linha \n outra linha"
```

```
"John's cat"
```

Você pode chamar qualquer um dos métodos do objeto string em uma string literal - JavaScript automaticamente converte a string literal para um objeto string temporário, chama o método, em seguida, descarta o objeto string temporário. Você também pode usar a propriedade `String.length` com uma string literal:

```
console.log("John's cat".length)
// Irá exibir a quantidade de caracteres na string incluindo o espaço
// Nesse caso, 10 caracteres.
```

Você deve usar string literal, a não ser que você precise usar um objeto string. Veja [String](#) para detalhes sobre objetos de strings.

### Uso de caracteres especiais em string

Além dos caracteres comuns, você também pode incluir caracteres especiais em strings, como mostrado no exemplo a seguir.

```
"uma linha \n outra linha"
```

A tabela a seguir lista os caracteres especiais que podem ser usados em strings no JavaScript.

**Tabela: Caracteres especiais no JavaScript**

Caracter	Descrição
\0	Byte nulo
\b	Backspace
\f	Alimentador de formulário
\n	Nova linha
\r	Retorno do carro

\t Caracter	Tabulação Descrição
\v	Tabulação vertical
\'	Apóstrofo ou aspas simples
\"	Aspas dupla


\\	Caractere de barra invertida
\XXX	Caractere com a codificação Latin-1 especificada por três dígitos octal XXX entre 0 e 377. Por exemplo, \251 é sequência octal para o símbolo de direitos autorais.
\xXX	Caractere com a codificação Latin-1 especificada por dois dígitos hexadecimal XX entre 00 e FF. Por exemplo, \xA9 é a sequência hexadecimal para o símbolo de direitos autorais.
\uXXXX	Caractere Unicode especificado por quatro dígitos hexadecimal XXXX. Por exemplo, \u00A9 é a sequência Unicode para o símbolo de direitos autorais. Veja <a href="#">sequências de escape Unicode</a> .

### Caracteres de escape

Para caracteres não listados na tabela, se precedidos de barra invertida ela é ignorada, seu uso está obsoleto e deve ser ignorado.

Você pode inserir uma aspa dentro de uma string precedendo-a com uma barra invertida. Isso é conhecido como *escaping* das aspas. Por exemplo:

```
var quote = "Ele lê \"The Cremation of Sam McGee\" de R.W. Service.";
console.log(quote);
```





O resultado disso seria:

```
Ele lê "The Cremation of Sam McGee" de R.W. Service.
```

Para incluir uma barra invertida dentro de uma string, você deve escapar o caractere de barra invertida. Por exemplo, para atribuir o caminho do arquivo `c:\temp` para uma string, utilize o seguinte:

```
var home = "c:\\temp";
```



Você também pode escapar quebras de linhas, precedendo-as com barra invertida. A barra invertida e a quebra de linha são ambas removidas da string.

```
var str = "esta string \
está quebrada \
em várias\
linhas."
console.log(str);    // esta string está quebrada em várias linhas.
```



Embora JavaScript não tenha sintaxe "heredoc", você pode adicionar uma quebra de linha e um escape de quebra de linha no final de cada linha:

```
var poema =
"Rosas são vermelhas\n\
Violetas são azuis,\n\
Esse seu sorriso\n\
é o que me seduz. (Lucas Pedrosa)"
```



## Mais informação

Este capítulo focou na sintaxe básica das declarações e tipos. Para saber mais sobre a linguagem JavaScript, veja também os seguintes capítulos deste guia:

- [Controle de fluxo e manipulação de erro](#)
- [Laços e iteração](#)
- [Funções](#)
- [Expressões e operadores](#)

No próximo capítulo, veremos a construção de controle de fluxos e manipulação de erro.

**Last modified:** 12 de set. de 2021, [by MDN contributors](#)