

<b>JAVASCRIPT E ELEMENTOS HTML.....</b>	<b>1</b>
<b>ONDE ESTA O JAVASCRIPT .....</b>	<b>2</b>
<b>EXIBIÇÃO JAVASCRIPT.....</b>	<b>3</b>
USANDO INNERHTML .....	3
USANDO DOCUMENT.WRITE() .....	4
USANDO WINDOW.ALERT() .....	4
USANDO CONSOLE.LOG() .....	4
<b>DECLARAÇÕES JAVASCRIPT .....</b>	<b>5</b>
PONTO E VÍRGULA; .....	5
ESPAÇOS EM BRANCO.....	5
LARGURA DE LINHA E QUEBRA DE LINHA .....	6
BLOCOS DE CÓDIGO .....	6
PALAVRAS-CHAVE.....	6
<b>SINTAXE JAVASCRIPT .....</b>	<b>7</b>
VALORES .....	7
EXPRESSÕES.....	7
COMENTÁRIOS .....	7
IDENTIFICADORES / NOMES .....	8
OUTRAS SINTAXES .....	8
<b>VARIÁVEIS JAVASCRIPT.....</b>	<b>9</b>
IDENTIFICADORES.....	9
DECLARAÇÃO E ATRIBUIÇÃO.....	9
UMA DECLARAÇÃO, MUITAS VARIÁVEIS .....	10
VALUE = UNDEFINED .....	10
<b>VARIÁVEIS DECLARADAS COM LET .....</b>	<b>10</b>
ESCOPO DE BLOCO (BLOCK SCOPE) .....	10
LET HOISTING (IÇAMENTO).....	11
<b>VARIÁVEIS DECLARADAS COM CONST.....</b>	<b>11</b>
QUANDO USAR CONST? .....	11
CONSTANT OBJECTS E ARRAYS.....	11

ESCOPO DE BLOCO (BLOCK SCOPE) .....	12
<b>TIPOS DE DADOS.....</b>	<b>12</b>
STRING.....	12
NUMBER .....	12
BOOLEAN.....	12
UNDEFINED.....	13
NULL .....	13
<b>OPERAÇÕES ARITMÉTICAS .....</b>	<b>13</b>
OPERADORES .....	13
<b>OPERADORES DE COMPARAÇÃO.....</b>	<b>14</b>
<b>OPERADORES LÓGICOS.....</b>	<b>15</b>
<b>OPERADOR CONDICIONAL (TERNÁRIO).....</b>	<b>15</b>
<b>OPERADORES BITWISE .....</b>	<b>16</b>
<b>OPERADORES DE ATRIBUIÇÃO.....</b>	<b>17</b>
<b>typeof.....</b>	<b>18</b>
<b>CONVERSÃO DE TIPO .....</b>	<b>19</b>
CONVERTER EM NUMBER .....	19
CONVERTER EM STRING.....	19
<b>ALERT(), CONFIRM() E PROMPT() .....</b>	<b>20</b>
WINDOW ALERT() .....	20
WINDOW CONFIRM().....	20
WINDOW PROMPT().....	21
<b>INSTRUÇÕES DE CONDIÇÕES IF, ELSE E ELSE IF.....</b>	<b>21</b>
INSTRUÇÃO CONDICIONAL IF .....	21
INSTRUÇÃO CONDICIONAL ELSE .....	21
INSTRUÇÃO CONDICIONAL ELSE IF .....	22
<b>INSTRUÇÃO CONDICIONAL SWITCH .....</b>	<b>22</b>
SINTAXE: .....	23
KEYWORDS BREAK E DEFAULT .....	23
<b>ESTRUTURAS DE REPETIÇÃO .....</b>	<b>23</b>

FOR LOOP .....	24
WHILE LOOP .....	24
DO WHILE LOOP.....	25
FOR IN LOOP.....	25
FOR OF LOOP .....	26
<b>ARRAYS .....</b>	<b>26</b>
SINTAXE .....	26
ELEMENTOS DO ARRAY .....	26
MÉTODOS E PROPRIEDADES DA ARRAY .....	27
<b>FUNÇÕES.....</b>	<b>28</b>
SINTAXE .....	28
RETURN.....	28
OPERADOR () .....	28
ARROW FUNCTIONS.....	29
<b>OBJECTS .....</b>	<b>29</b>
CRIAÇÃO DE OBJETOS .....	30
USANDO OBJECT LITERAL.....	30
USANDO KEYWORD NEW .....	30
OBJETOS SÃO MUTÁVEIS.....	30
OBJECT PROPERTIES .....	31
OBJECT METHODS .....	32
OBJECT CONSTRUCTORS.....	32
<b>THIS.....</b>	<b>33</b>
THIS EM UM MÉTODO.....	33
ALTERANDO CONTEXTO DO THIS .....	34
THIS PRECEDÊNCIA .....	34
<b>DATAS.....</b>	<b>35</b>
NEW DATE().....	35
NEW DATE( YEAR, MONTH, ... ).....	35
NEW DATE(MILLISECONDS).....	35
NEW DATE( DATESTRING ).....	35
GET DATE METHODS.....	36
SET DATE METHODS .....	36

<b>WINDOW OBJECT .....</b>	<b>37</b>
<b>DOM (DOCUMENT OBJECT MODEL) .....</b>	<b>37</b>
MÉTODOS E PROPRIEDADES DOM.....	38
LOCALIZANDO ELEMENTOS HTML DOM .....	38
ALTERANDO ELEMENTOS HTML DOM.....	40
<b>DOM EVENTS.....</b>	<b>41</b>
<b>DOM EVENTLISTENER.....</b>	<b>43</b>
SINTAXE .....	43
<b>THIS KEYWORD .....</b>	<b>44</b>
THIS EM UM MÉTODO.....	44
THIS SOZINHO.....	44
THIS EM UMA FUNÇÃO (DEFAULT).....	44
THIS EM UMA FUNÇÃO (STRICT).....	44
THIS EM EVENT HANDLERS.....	44
EMPRÉSTIMO DE FUNÇÃO BIND() .....	45
ORDEM DE PRECEDÊNCIA THIS.....	45
<b>BIBLIOTECA MATH .....</b>	<b>45</b>
MATH PROPERTIES (CONSTANTS) .....	45
MATH METHODS.....	46
NUMEROS INTEIROS .....	47
<b>JSON .....</b>	<b>47</b>
JSON PARSE .....	47
JSON STRINGIFY.....	48
<b>LOCAL STORAGE .....</b>	<b>48</b>
SINTAXE .....	48
<b>JAVASCRIPT TIMING EVENTS .....</b>	<b>49</b>
SETTIMEOUT .....	49
SETINTERVAL.....	50
<b>HTML DOM EVENTS REFERENCE .....</b>	<b>51</b>
<b>OBJETOS DE EVENTO HTML DOM.....</b>	<b>55</b>
<b>PROPRIEDADES E MÉTODOS DO EVENTO HTML DOM.....</b>	<b>55</b>

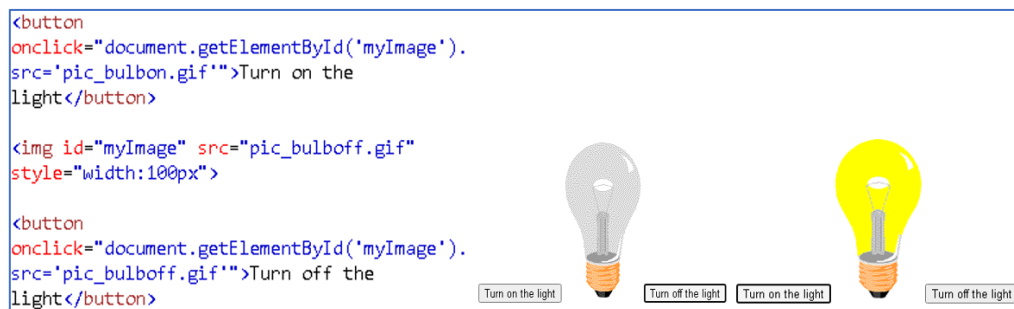
<b>JS STRICT MODE .....</b>	<b>59</b>
POR QUE USAR STRICT MODE .....	59
NÃO PERMITIDO NO STRICT MODE .....	59
<b>ARROW FUNCTION .....</b>	<b>60</b>
ARROW FUNCTION THIS.....	60
<b>JS ASYNC .....</b>	<b>61</b>
CALLBACK.....	61
ASYNCHRONOUS JAVASCRIPT .....	62
PROMISES .....	64
ASYNC/AWAIT .....	64
<b>ARRAY ITERATION .....</b>	<b>65</b>
FILTER() .....	65
MAP() .....	66
REDUCE() .....	66
<b>ARRAY REFERÊNCIA METHODS AND PROPERTIES.....</b>	<b>66</b>
<b>SPREAD OPERATOR .....</b>	<b>67</b>
<b>JAVASCRIPT ERRORS.....</b>	<b>68</b>
TRY CATCH.....	68
THROW .....	68
FINALLY .....	69
<b>FETCH .....</b>	<b>70</b>

## JAVASCRIPT E ELEMENTOS HTML

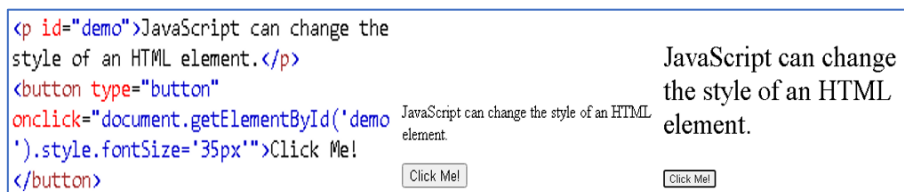
O Javascript pode mudar o conteúdo de um elemento HTML através do método “getElementById()”. O exemplo abaixo vai “procurar” no HTML o elemento com o id=”demo”, e mudar o conteúdo desse elemento (innerHTML) para “Hello JavaScript” quando o botão for clicado(onclick):



O Javascript pode mudar os atributos de um elemento HTML. No próximo exemplo os valores de atributo src(source) de um elemento <img> são alterados quando um dos botões for clicado:



O Javascript pode mudar os atributos de estilo de um elemento HTML.



O Javascript pode ocultar ou mostrar um elemento HTML. Através da alteração da propriedade de estilo “display”:

```

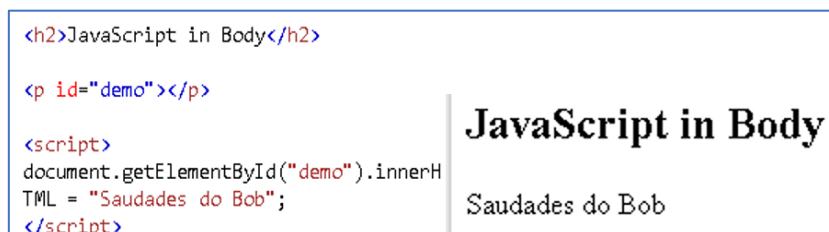
document.getElementById("demo").style.display = "none";

document.getElementById("demo").style.display = "block";

```

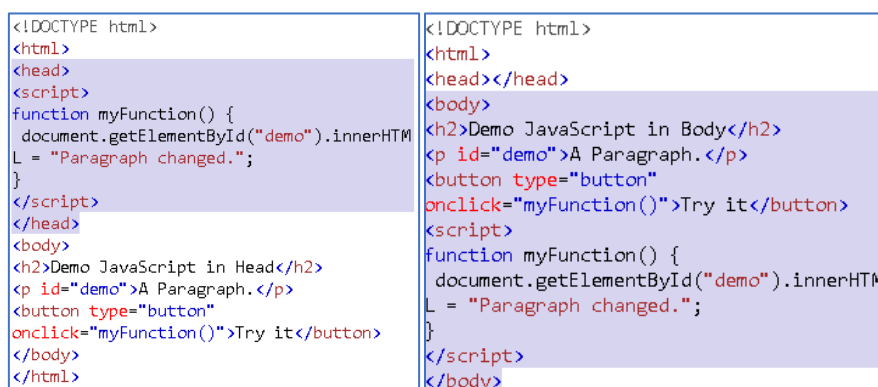
## ONDE ESTA O JAVASCRIPT

No HTML, O código JavaScript é inserido entre as tags `<script>` e `</script>`.



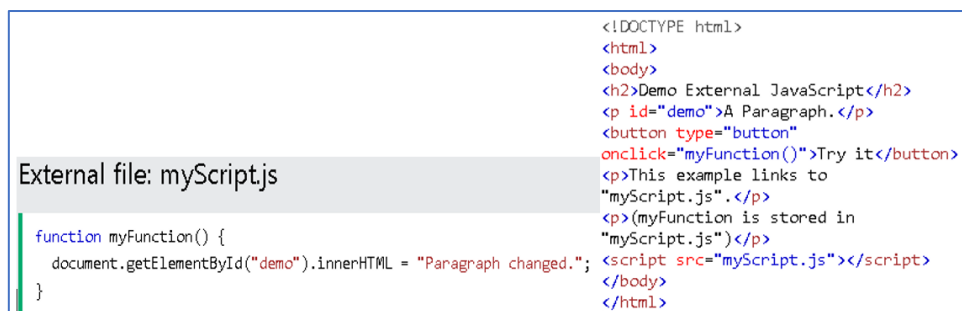
É possível inserir qualquer quantidade de scripts em um documento HTML. Scripts podem ser inseridos dentro da seção `<body>`, ou dentro da seção `<head>`, ou em ambos ao mesmo tempo.

O exemplo a seguir altera um parágrafo invocando (chamando) uma função quando o botão for clicado, o script pode ser inserido tanto no `<head>`, quanto `<body>` do documento HTML.



O elemento HTML `<head>` providencia informações gerais (metadados) sobre o documento HTML, incluindo seu título e link, o elemento `<body>` do HTML representa o conteúdo de um documento HTML. Inserir scripts na parte inferior do elemento `<body>` melhora a velocidade de exibição, já que, posicionar ele no `<head>` faz o navegador interpretar esse script primeiro, tornando a exibição mais lenta.

Os scripts também podem ser posicionados em arquivos externos, scripts externos são mais práticos quando alguns códigos são usados por diferentes páginas web. Arquivos Javascript possuem a extensão `“.js”`. Para utilizar um script externo, deve ser inserido o nome do arquivo do script no atributo `src(source)` da tag `<script>`:



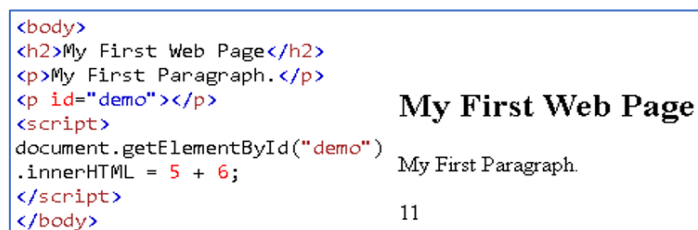
Colocar scripts em arquivos externos tem algumas vantagens: separa HTML e código, torna o HTML e o JavaScript mais fáceis de ler e manter, arquivos JavaScript em cache podem acelerar o carregamento da página. Um script externo pode ser referenciado de 3 maneiras diferentes: Com um URL completo (um endereço web completo); com um caminho de arquivo (como /js/); sem nenhum caminho (como no exemplo acima).

## EXIBIÇÃO JAVASCRIPT

O JavaScript pode “exibir” dados de diferentes formas: Escrevendo dentro de um elemento HTML, usando “innerHTML”; Escrevendo dentro da saída do HTML usando “document.write()”; Escrevendo em uma caixa de alerta no navegador, usando “window.alert()”; Escrevendo no console do navegador, usando “console.log()”, entre outras maneiras.

### Usando Innerhtml

Para acessar um elemento HTML, o Javascript pode usar o método “document.getElementById(id)”. O atributo “id” define o elemento HTML, a propriedade “innerHTML” define o conteúdo desse elemento. Alterar a propriedade “innerHTML” de um elemento HTML é uma maneira comum de exibir dados em HTML.





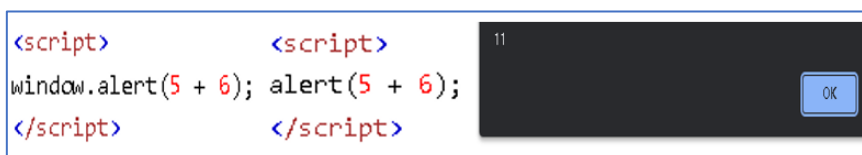
## Usando Document.Write()

Para propósitos de testes, é conveniente usar o método `document.write()`, pois, utilizá-lo após um documento HTML ser carregado, vai deletar todo o HTML existente.



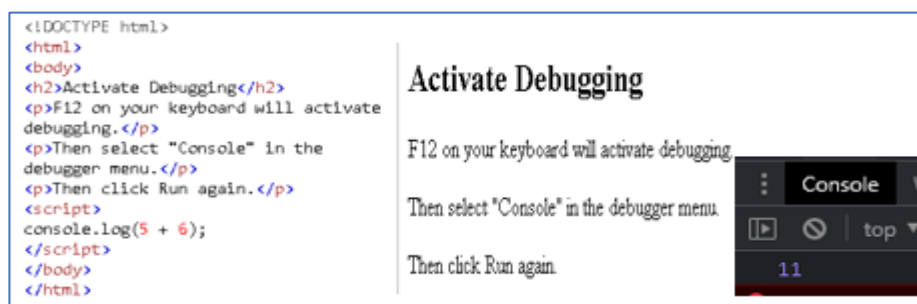
## Usando Window.Alert()

É possível utilizar uma caixa de alerta no navegador para exibir dados, o uso da palavra-chave “window” é opcional, já que, o objeto “window” é por padrão um objeto de escopo global, isso significa que variáveis, propriedades, e métodos por padrão pertencem ao objeto “window”.



## Usando Console.Log()

Para propósito de depuração de erros (debugging), é possível chamar o método “`console.log()`” em um navegador para exibir dados no console.



## DECLARAÇÕES JAVASCRIPT

Um programa de computador é uma lista de “instruções” a serem “executadas” por um computador. Em uma linguagem de programação, essas instruções são chamadas de statements (declarações). Um programa JavaScript é uma lista de declarações programáveis. Programas JavaScript e declarações JavaScript são frequentemente chamados por códigos JavaScript. No HTML, os programas JavaScript são executados pelo navegador.

As declarações JavaScript podem ser compostas por: Valores, Operadores, Expressões, Keywords e comentários. Muitos programas Javascript contém muitas declarações Javascript, essas declarações (instruções) são executadas, uma por uma, seguindo a ordem em que elas foram escritas.

### Ponto E Vírgula;

O sinal gráfico ponto e vírgula separa as declarações JavaScript, ele deve ser adicionado no final de cada declaração executável.

```
let a, b, c; // Declare 3 variables
a = 5;      // Assign the value 5 to a
b = 6;      // Assign the value 6 to b
c = a + b;  // Assign the sum of a and b to c
```

Quando separadas por ponto e vírgula, múltiplas declarações podem ser alocadas em uma única linha para facilitar a leitura.

### Espaços Em Branco

O JavaScript ignora múltiplos espaços, é possível adicioná-los para tornar o script mais legível. Os seguintes exemplos são equivalentes, porém o primeiro apresenta um código mais limpo.

```
let person = "Hege";
let person="Hege";
```

Também é uma boa prática adicionar espaços ao redor dos operadores (= + = \* /).

## Largura De Linha E Quebra De Linha

Para tornar os códigos mais legíveis, programadores frequentemente evitam linhas de códigos maiores do que 80 caracteres. Se uma declaração JavaScript não couber em uma linha, a melhor maneira é quebra-la após um operador.

## Blocos De Código

As declarações JavaScript podem ser agrupadas juntas em blocos de código dentro de chaves {...}. O propósito dos blocos de código é definir as declarações que vão ser executadas juntas.

## Palavras-Chave

As declarações JavaScript frequentemente começam após uma palavra-chave (**keyword**) para o JavaScript identificar a ação que está sendo realizada.

Exemplo de algumas Keywords:

- **var**: Declara uma variável
- **let**: Declara uma variável de bloco
- **const**: Declara uma constante de bloco
- **if**: Marca um bloco de instruções a serem executadas em uma condição
- **switch**: Marca um bloco de instruções a serem executadas em diferentes casos
- **for**: Marca um bloco de instruções a serem executadas em um loop
- **function**: Declara uma função
- **return**: Sai de uma função
- **try**: Implementa o tratamento de erros para um bloco de instruções

## SINTAXE JAVASCRIPT

A sintaxe em JavaScript é um conjunto de regras que define como os programas em JavaScript devem ser construídos.

### Valores

Existem dois tipos de valores em JavaScript: Valores fixos e Valores variáveis. Valores fixos são chamados de literais (Literals) e valores variáveis chamados de variáveis (Variables). As regras mais importantes para valores fixos (Literals) são: números são escritos com ou sem o seu decima; strings são texto, escrito com aspas duplas ou aspas simples. Em uma linguagem de programação, variáveis (variables) são usadas para armazenar valores. JavaScript usa as keywords `var`, `let` e `const` para declarar variáveis. O sinal gráfico de igualdade (`=`) é usado para atribuir valores a uma variável. No seguinte exemplo, `x` é definido como uma variável, após isso, é atribuído (“`x` recebe”) o valor 6:

```
let x;  
x = 6;
```

### Expressões

Uma expressão é uma combinação de valores, variáveis e operadores que calculam um valor. Na computação isso é chamado de evaluation (estimativa ou avaliação). Como por exemplo, cinco vezes dez é igual a cinquenta, ou em uma expressão: `5 * 10`. Expressões também podem conter valores de variáveis: `x * 10`. Os valores podem ser de vários tipos como números e strings, por exemplo, a expressão `“John” + “ “ + “Doe”`, ser igual a `“John Doe”`.

### Comentários

Nem todas as declarações são “executadas”, códigos escritos após barras duplas `//` ou entre `/*` e `*/` são considerados como comentários. Comentários são ignorados e não será possível executá-los.

## Identificadores / Nomes

Identificadores são nomes em JavaScript, eles são usados para nomear variáveis, palavras-chave e funções. As regras para nomes permitidos são as mesmas para a maioria das linguagens de programação. Em JavaScript o nome deve começar com:

- Uma letra (A-Z ou a-z)
- Um sinal de cifrão (\$)
- Ou um sublinhado (\_)

Números não podem ser alocados para o primeiro caractere em um nome, para facilitar o JavaScript distinguir um número de um nome.

## Outras Sintaxes

JavaScript é uma linguagem de programação case sensitive, ou seja, ela identifica como diferentes os caracteres maiúsculos e caracteres minúsculos. No exemplo a seguir as variáveis `lastName` e `lastname` são diferentes variáveis:

```
let lastname, lastName;  
lastName = "Doe";  
lastname = "Peterson";
```

Historicamente, programadores têm usado diferentes maneiras de adicionar múltiplas palavras em um único nome de uma variável. Uma das maneiras é utilizando o sinal de hífen (-), porém em JavaScript esse sinal é reservado para ser utilizado como sinal de subtração. Algumas das maneiras são: utilização do underscore (\_) "first\_name"; Camel Case maiúsculo (Pascal case) "FirstName"; Camel Case minúsculo "firstName". Programadores JavaScript tendem a utilizar camel case iniciado com letras minúsculas.

JavaScript usa o conjunto de caracteres Unicode. Unicode cobre (quase) todos os caracteres, pontuações e símbolos do mundo.

[https://www.w3schools.com/charsets/ref\\_html\\_utf8.asp](https://www.w3schools.com/charsets/ref_html_utf8.asp)

## VARIÁVEIS JAVASCRIPT

Variáveis são containers para armazenamento de dados (armazenar valores de dados). Existem 4 maneiras de declarar uma variável em JavaScript, usando “var”, usando “let”, usando “const” ou não usando nada. Porém uma variável deve sempre ser declarada usando var, let ou const. A keyword “var” é usada em códigos JavaScript de 1995 até 2015. As keywords “let” e “const” foram adicionadas em 2015, códigos que devem rodar em navegadores mais antigos devem usar “var”.

### Identificadores

Todas as variáveis JavaScript devem ser identificadas com nomes exclusivos. Esses nomes exclusivos são chamados de identificadores. Os identificadores podem ser nomes curtos (como x e y) ou nomes mais descritivos (idade, soma, volume total). As regras gerais para construir nomes para variáveis (identificadores exclusivos) são:

- Os nomes podem conter letras, dígitos, sublinhados e cifrões.
- Os nomes devem começar com uma letra
- Os nomes também podem começar com \$ e \_ (mas não o usaremos neste tutorial)
- Os nomes diferenciam maiúsculas de minúsculas (y e Y são variáveis diferentes)
- Palavras reservadas (como palavras-chave JavaScript) não podem ser usadas como nomes

### Declaração E Atribuição

A ação de criar uma variável em JavaScript é chamada de “declarar” uma variável. Uma variável é declarada com as keywords var ou let (var carName; ou let carName;). Após a declaração, a variável não terá um valor (tecnicamente ela tem um valor “undefined”). Para atribuir um valor a uma variável é utilizado o sinal de igual (carName = "Volvo";). Em JavaScript, o sinal de igual (=) é um operador de “atribuição”, não um sinal de “igual a”. É possível também atribuir valor à variável quando ela for declarada (let carName = "Volvo";), para declarar uma variável com const deve-se sempre atribuir valores durante a declaração (const price1 = 5;). É uma boa prática de programação sempre declarar as variáveis no início do script.

## Uma Declaração, Muitas Variáveis

É possível declarar muitas variáveis em uma única declaração, apenas iniciando a declaração com “let” e separando as seguintes com ponto e vírgula:

```
let person = "John Doe", carName = "Volvo", price = 200;
```

A declaração também pode ser feita em múltiplas linhas para uma leitura mais agradável do código.

## Value = Undefined

Na programação, variáveis são frequentemente declaradas sem um valor. O valor pode ser algo que ainda vai ser calculado, ou fornecido mais tarde, como por exemplo, através de input do usuário. Uma variável declarada sem um valor vai ter um valor “undefined”.

## VARIÁVEIS DECLARADAS COM LET

A keyword let foi introduzida no ES6 (2015), variáveis definidas com let não podem ser redeclaradas, devem ser declaradas antes do uso e possuem escopo de bloco (Block Scope). Uma variável declarada com let pode ter seu valor reatribuído, porém ela não pode ser redeclarada.

### Escopo De Bloco (Block Scope)

Antes do ES6(2015), JavaScript tinha apenas os chamados “escopo global” e “escopo de função”. ES6 introduziu duas importantes novas keywords: “let” e “const”. Essas keywords providenciaram o “escopo de bloco” (**Block Scope**) no JavaScript. Variáveis declaradas dentro de um sinal de chaves {...}(bloco) não podem ser acessadas fora desse bloco. Variáveis declaradas com a keyword var não pode ter um escopo de bloco, pois elas podem ser acessadas fora do bloco.

```
{  
  var x = 2;  
}  
// x CAN be used here
```

Redeclarar uma variável com let dentro de um bloco não redeclarará a variável fora do bloco, evitando problemas que aconteceriam ao utilizar var.

<pre>let x = 10; // Here x is 10 {   let x = 2;   // Here x is 2 } // Here x is 10</pre>	<pre>var x = 2; // Allowed let x = 3; // Not allowed {   let x = 2; // Allowed   let x = 3 // Not allowed } {   let x = 2; // Allowed   var x = 3 // Not allowed }</pre>
--	--

Redeclarar uma variável com var é possível, porém, não é possível redeclarar uma variável com let dentro do mesmo bloco.

### Let Hoisting (Içamento)

Variáveis definidas com var são içadas (hoisted) para o topo e podem ser inicializadas a qualquer momento. Isso significa que é possível usar uma variável antes de ela ser declarada. Variáveis definidas com let e const também são içadas até o topo de um bloco, porém não podem ser inicializadas. Isso significa que usar uma variável (atribuir um valor) antes de ela ser declarada vai resultar em um erro (**ReferenceError**).

<pre>carName = "Volvo"; var carName;</pre>	<pre>carName = "Saab"; let carName = "Volvo"; initialization</pre>	<p>ReferenceError: Cannot access 'carName' before initialization</p>
--	--	--

## VARIÁVEIS DECLARADAS COM CONST

A keyword const foi introduzida em ES6 (2015). Variáveis definidas com const não podem ser redeclaradas. Variáveis definidas com const não podem ser reatribuídas. As variáveis definidas com const têm o escopo do bloco. As variáveis declaradas com const não podem ter seus valores reatribuídos e não pode ser redeclarada.

### Quando Usar Const?

As variáveis const devem ter seus devidos valores atribuídos durante a sua declaração. De regra geral uma variável é atribuída com const sempre que você souber que seu valor não será alterado mais tarde.

### Constant Objects E Arrays

A keyword const possui uma pegadinha, ela é uma abreviação para constant (constante), porém, ela não define um valor constante, ela define uma referência constante a um valor. Por conta disso não é possível reatribuir um



valor constante, reatribuir um array constante e reatribuir um objeto constante. Mas é possível mudar elementos de um array constante e mudar propriedades de um objeto constante.

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];
// You can change an element:
cars[0] = "Toyota";
// You can add an element:
cars.push("Audi");

const cars = ["Saab", "Volvo", "BMW"];
cars = ["Toyota", "Volvo", "Audi"]; // ERROR
```

```
// You can create a const object:
const car = {type:"Fiat", model:"500", color:"white"};
// You can change a property:
car.color = "red";
// You can add a property:
car.owner = "Johnson";

const car = {type:"Fiat", model:"500", color:"white"};
car = {type:"Volvo", model:"EX60", color:"red"}; // ERROR
```

## Escopo De Bloco (Block Scope)

É semelhante a declaração de uma variável com `const` e `let` quando se trata de escopo de bloco. No uma variável declarada com `const` dentro de um bloco não é a mesma declarada fora do bloco.

## TIPOS DE DADOS

Na programação, é importante entender o conceito dos tipos de dados para escrever expressões corretamente e evitar possíveis erros.

### String

Uma string é uma série de caracteres como "John Doe". Strings são escritas com aspas simples ou aspas duplas. É possível utilizar aspas dentro de uma string desde que elas não sejam as mesmas que definem essa string. ("He is called 'Johnny'").

### Number

JavaScript possui apenas um tipo de número, eles podem ser escritos com ou sem seus decimais, e números muito grandes podem ser escritos com notação científica (123e5, 34.00).

### Boolean

Tipos booleanos possuem apenas dois valores: `true` (verdadeiro) ou `false` (falso), eles são frequentemente usados após testes condicionais (`let x`

```
= 5; let y = 5; let z = 6; (x == y // Returns true (x == z // Returns false).
```

### Undefined

Em JavaScript, uma variável sem um valor tem o valor undefined. Qualquer variável pode ser considerada vazia configurando o valor para undefined. Isso significa que a variável ainda não foi preenchida com algum outro tipo de valor, logo um valor indefinido (undefined) preenche essa variável. Porém é importante lembrar que como os valores são indefinidos, o tipo dessa variável também é indefinido.

### Null

Em JavaScript null é “nada”. Supostamente isso torna inexistente algo com esse valor. Porém, em javascript null é considerado um objeto. É possível tornar um objeto vazio definindo-o como null. Undefined e null são iguais em valor, mas diferentes em tipo.

```
typeof undefined //undefined typeof null// object
```

## OPERAÇÕES ARITMÉTICAS

Operadores aritméticos executam cálculos aritméticos em números.

### Operadores

Os números são chamados de operandos, os símbolos que executam as operações entre os operandos são chamados de operadores.

- **ADIÇÃO (+):** O operador de adição (+) realiza uma soma entre os números.
- **SUBTRAÇÃO (-):** O operador de subtração (-) realiza a subtração dos números.
- **MULTIPLICAÇÃO (\*):** O operador de multiplicação (\*) multiplica os números
- **DIVISÃO (/):** O operador de divisão (/) realiza uma divisão entre os números.
- **RESTO (%):** O operador de módulo (%) retorna o restante da divisão dos números.

- **INCREMENTO (++)**: O operador de incremento (++) adiciona o valor 1 a um número.
- **DECREMENTO (--)**: O operador de decremento (--) retira o valor 1 de um número.
- **EXPONENCIAÇÃO (\*\*)**: O operador de exponenciação (\*\*) eleva o primeiro operando à potência do segundo operando.

## OPERADORES DE COMPARAÇÃO

Operadores de comparação são usados em declarações lógicas para determinar a igualdade ou diferença entre variáveis ou valores. No exemplo a seguir o valor 5 foi atribuído a variável x (**x = 5**), a tabela demonstra a utilização desses comparadores e a sua descrição.

OPERADOR	DESCRIÇÃO	EXEMPLO	RETORNO
==	Igual a	x == 8	false
		x == 5	true
		x == "5"	true
===	Igual valor e tipo (idêntico)	x === 5	true
		x === "5"	false
!=	Diferente	x != 8	true
!==	Diferente em valor ou tipo	x !== 5	false
		x !== "5"	true
		x !== 8	true
>	Maior que	x > 8	false
<	Menor que	x < 8	true
>=	Maior ou igual a	x >= 8	false
<=	Menor ou igual a	x <= 8	true

## OPERADORES LÓGICOS

Operadores lógicos são usados para determinar uma lógica entre variáveis ou valores. No exemplo a seguir foi atribuído o valor 6 a variável x (**x = 6**) e o valor 3 a variável y (**y = 3**), a tabela demonstra a lógica dos operadores:

OPERADOR	DESCRIÇÃO	EXEMPLOS
<b>&amp;&amp;</b>	e	(x < 10 <b>&amp;&amp;</b> y > 1) is true
<b>  </b>	ou	(x == 5 <b>  </b> y == 5) is false
<b>!</b>	negação	!(x == y) is true (!false y) is true (!true y) is false

## OPERADOR CONDICIONAL (TERNÁRIO)

JavaScript também contém um operador condicional que atribui um valor a uma variável baseado em uma condição. Esse operador é representado pelo sinal de interrogação (?) em conjunto com o sinal de dois pontos (:). Ou seja, eles podem ser interpretados como “if” e “else” (se e senão).

```
nomedavariavel = (condição) ? valor1 : valor2
```

```

<!DOCTYPE html>
<html>
<body>
<input id="age" value="18" />
<button onclick="myFunction()">Try it</button>
<p id="demo"></p>
<script>
function myFunction() {
  let age = document.getElementById("age").value;
  let voteable = (age < 18) ? "Too young":"Old enough";
  document.getElementById("demo").innerHTML = voteable + " to vote.";
}
</script>
</body>
</html>

```

18

Try it

19

Try it

Old enough to vote.

## OPERADORES BITWISE

OPERADOR	NOME	DESCRIÇÃO
&	AND (E)	Sets each bit to 1 if both bits are 1
	OR (OU)	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT (NÃO)	Inverts all the bits
<<	Zero fill left shift (Deslocamento à esquerda de preenchimento zero)	Shifts left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift (Deslocamento à direita)	Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off
>>>	Zero fill right shift (Deslocamento à direita de preenchimento zero)	Shifts right by pushing zeros in from the left, and let the rightmost bits fall off

Exemplos:

OPERAÇÃO	RESULTADO	MESMO QUE	RESULTADO
5 & 1	1	0101 & 0001	0001
5   1	5	0101   0001	0101
~ 5	10	~0101	1010
5 << 1	10	0101 << 1	1010
5 ^ 1	4	0101 ^ 0001	0100
5 >> 1	2	0101 >> 1	0010
5 >>> 1	2	0101 >>> 1	0010

## OPERADORES DE ATRIBUIÇÃO

Os operadores de atribuição atribuem valores a uma variável. Eles são utilizados para deixar o código mais “limpo”. São eles:

OPERADOR	EXEMPLO	MESMO QUE
<b>=</b>	$x = y$	$x = y$
<b>+=</b>	$x += y$	$x = x + y$
<b>-=</b>	$x -= y$	$x = x - y$
<b>*=</b>	$x *= y$	$x = x * y$
<b>/=</b>	$x /= y$	$x = x / y$
<b>%=</b>	$x \% = y$	$x = x \% y$
<b>**=</b>	$x ** = y$	$x = x ** y$
<b>&lt;&lt;=</b>	$x << = y$	$x = x << y$
<b>&gt;&gt;=</b>	$x >> = y$	$x = x >> y$
<b>&gt;&gt;&gt;=</b>	$x >>> = y$	$x = x >>> y$
<b>&amp;=</b>	$x \& = y$	$x = x \& y$
<b>^=</b>	$x \wedge = y$	$x = x \wedge y$
<b> =</b>	$x   = y$	$x = x   y$

## typeof

Em JavaScript existem 5 diferentes tipos de dados que podem contar valores:

- string
- number
- boolean
- object
- function

Existem 6 tipos de objetos:

- Object
- Date
- Array
- String
- Number
- Boolean

E 2 tipos de que não podem conter valores:

- null
- undefined

O operador **typeof** pode ser usado para verificar o tipo de dado de uma variável em JavaScript.

```
typeof "John"           // Returns "string"
typeof 3.14             // Returns "number"
typeof NaN              // Returns "number"
typeof false            // Returns "boolean"
typeof [1,2,3,4]        // Returns "object"
typeof {name:'John', age:34} // Returns "object"
typeof new Date()       // Returns "object"
typeof function () {}   // Returns "function"
typeof myCar             // Returns "undefined" *
typeof null             // Returns "object"
```

É possível observar que:

- O tipo de dados de NaN é número
- O tipo de dados de uma array é objeto
- O tipo de dados de uma data é objeto
- O tipo de dados de null é objeto
- O tipo de dados de uma variável indefinida é undefined \*
- O tipo de dados de uma variável que não recebeu um valor também é undefined\*

## CONVERSÃO DE TIPO

Em JavaScript uma variável pode ser convertida em uma nova variável e em outro tipo de dado.

### Converter Em Number

O método global `Number()` pode converter dados em numbers ( método global pois ele pode ser usado para todos os tipos de dados, não somente strings), strings contendo números (como "3.14") podem ser convertidas em números (como 3.14). Strings vazias serão convertidas para 0.

O método `parseInt()` analisa ( tradução do verbo "pares" é "analisar uma palavra") uma string e retorna um número inteiro. Espaços são permitidos. Somente o primeiro valor é convertido.

```
parseInt("-10");      -10
parseInt("-10.33");   -10
parseInt("10");        10
parseInt("10.33");     10
parseInt("10 20 30");  10
parseInt("10 years");  10
parseInt("years 10");  NaN
```

Do mesmo modo, o método `parseFloat()` analisa uma string e retorna um número. Espaços são permitidos, mas somente o primeiro número é convertido.

```
parseFloat("10");      10
parseFloat("10.33");    10.33
parseFloat("10 20 30"); 10
parseFloat("10 years"); 10
parseFloat("years 10"); NaN
```

### Converter Em String

O método global `String()` pode converter números em strings. Ele pode ser usado em qualquer tipo de número, literais, variáveis ou expressões.

```
String(x)      123
String(123)     123
String(100 + 23) 123
```

O método **`toString()`** retorna um número como uma string. Todos os métodos numéricos podem ser usados em qualquer tipo de número (literais, variáveis ou expressões):

```
let x = 123;
x.toString();
(123).toString();      123
(100 + 23).toString(); 123
```



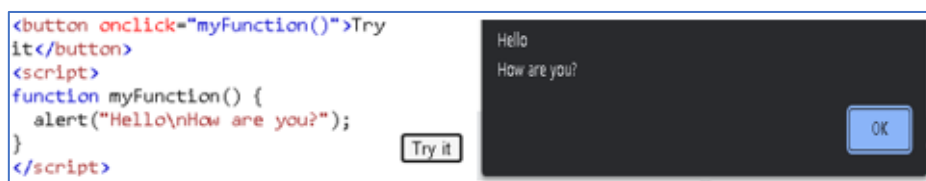
Outros métodos de conversões numéricas:

Method	Description
<b>toExponential()</b>	Returns a string, with a number rounded and written using exponential notation.
<b>toFixed()</b>	Returns a string, with a number rounded and written with a specified number of decimals.
<b>toPrecision()</b>	Returns a string, with a number written with a specified length

## ALERT(), CONFIRM() E PROMPT()

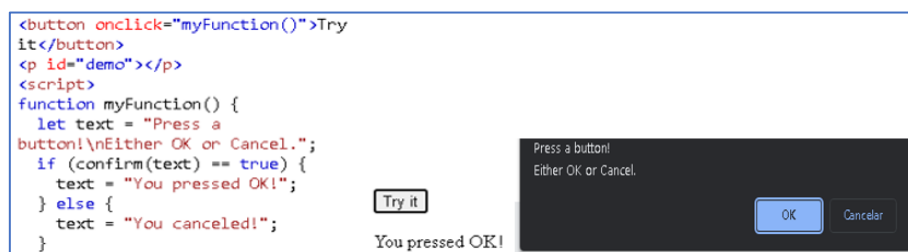
### Window Alert()

O método **alert()** exibe uma caixa de alerta com um botão de “OK”. É usado quando se deseja que certo tipo de informação chegue ao usuário. A caixa de alerta tira o foco da janela atual do navegador e, força o usuário a ler a mensagem. É importante não abusar desse método, pois ele impede que o usuário acesse outras partes da página até que a caixa de alerta seja fechada.



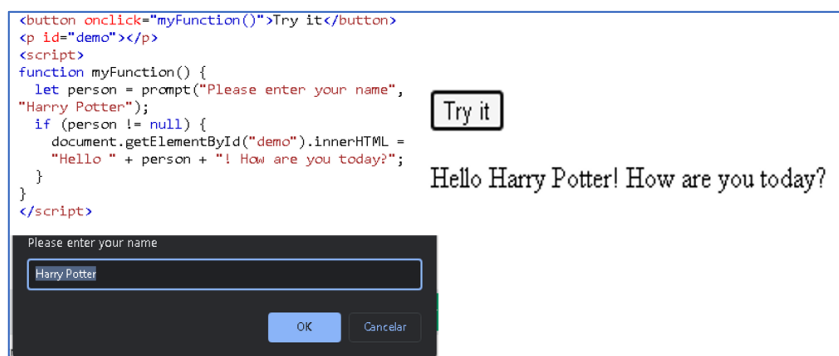
### Window Confirm()

O método **confirm()** exibe uma caixa de diálogo com uma mensagem, um botão de “OK”, e um botão de “Cancel”. Ele pode ser utilizado para receber valores de tipo booleano do usuário, true se o usuário clicou em “OK”, e false caso clique em “Cancel”.



## Window Prompt()

O método **prompt()** exibe uma caixa de diálogo que solicita um input do usuário. O método `prompt()` retorna o valor de entrada se o usuário clicar em "OK", caso contrário, retorna null.



## INSTRUÇÕES DE CONDIÇÕES IF, ELSE E ELSE IF

Instruções condicionais são usadas para realizar diferentes ações baseadas em diferentes condições.

### Instrução Condicional If

A instrução `if` é usada para especificar um bloco de código JavaScript que será executado se uma determinada condição for verdadeira. O exemplo vai gerar a frase de saudação "Good day" **se** as horas forem menores que 18:00.

```
<p id="demo"></p>
<script>
if (new Date().getHours() < 18) {
  document.getElementById("demo").inner
HTML = "Good day!";
}
</script>
```

### Instrução Condicional Else

A instrução `else` é usada para especificar um bloco de código JavaScript que será executado se uma determinada condição for falsa. O exemplo demonstra isso, se as horas forem menores que 18 (condição verdadeira `if`), a saudação "Good day" será atribuída como valor a uma variável, senão (condição falsa `else`), o valor atribuído será "Good Evening", e após a execução desse código a variável será inserida em um determinado parágrafo.

```
<p id="demo"></p>
<script>
const hour = new Date().getHours();
let greeting;
if (hour < 18) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
document.getElementById("demo").innerHTML = greeting;
</script>
```

## Instrução Condicional Else If

É possível observar que as estruturas if e else são frequentemente utilizadas em conjunto, seja a condição for verdadeira e se a condição for falsa, ocasionando em somente duas possibilidades para a execução do bloco de código. Uma das maneiras de contornar isso é utilizando a instrução else if. Essa instrução será usada para especificar uma nova condição se a primeira condição for falsa.

No exemplo, se a hora for menor que 10:00, será criada uma saudação "Good morning", senão, mas se a hora for menor que 20:00, crie uma saudação "Good day", caso contrário, uma saudação "Good Evening":

```
<p id="demo"></p>
<script>
const time = new Date().getHours();
let greeting;
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
document.getElementById("demo").innerHTML = greeting;
</script>
```

## INSTRUÇÃO CONDICIONAL SWITCH

A instrução switch será usada para realizar uma ação diferente baseada em uma diferente condição. Pode ser utilizada para selecionar um dentre muitos códigos de blocos que serão executados.

No exemplo o método getDay() retorna o dia da semana como um número entre 0 e 6. Se hoje for sábado (6) ou domingo (0) uma mensagem será escrita, senão uma mensagem padrão será escrita:

```
switch (new Date().getDay()) {
  case 6:
    text = "Today is Saturday";
    break;
  case 0:
    text = "Today is Sunday";
    break;
  default:
    text = "Looking forward to the Weekend";
}
```

### Sintaxe:

É possível observar que: A expressão switch é avaliada uma vez; O valor da expressão é comparado com os valores de cada um dos casos; Se houver uma correspondência, o bloco de código associado será executado; Se não houver correspondência, o bloco de código padrão (default) será executado.

```
switch(expressão) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

### Keywords Break E Default

Quando o código JavaScript switch atinge uma keyword break, ele sai do bloco switch. Isso vai interromper a execução dentro do bloco switch. Se omitir a instrução break, o próximo case será executado mesmo que a avaliação não corresponda ao case. Não é necessário quebrar o último caso em um bloco de comutação. O bloco quebra (termina) lá de qualquer maneira.

A keyword default (padrão) especifica o código a ser executado se não houver correspondência em nenhum dos cases(casos).

## ESTRUTURAS DE REPETIÇÃO

Repetições (loops) podem executar um bloco de código um determinado número de vezes. O JavaScript suporta diferentes tipos de loops:

- **for** - percorre um bloco de código várias vezes
- **for/in** - percorre as propriedades de um objeto
- **for/of** - percorre os valores de um objeto iterável

- **while** - percorre um bloco de código enquanto uma condição especificada é verdadeira
- **do/while** - também percorre um bloco de código enquanto uma condição especificada for verdadeira

## For Loop

A estrutura de repetição for, é geralmente utilizado quando se sabe os limites da execução das repetições, ao contrário da repetição while. A repetição for tem a seguinte sintaxe:

```
for (statement 1; statement 2; statement 3) {
    // code block to be executed
}
```

Declaração 1 é executado (uma vez) antes da execução do bloco de código. Declaração 2 define a condição para execução do bloco de código. Declaração 3 é executado (todas as vezes) após o bloco de código ter sido executado.

No exemplo, a declaração 1 define a variável antes do loop começar (let i = 0. A declaração 2 define a condição para que o loop seja executado (i deve ser menor do que 5). A declaração 3 acrescenta um valor (i++) cada vez que o bloco de código no loop for executado.

for (let i = 0; i < 5; i++) {	The number is 0
text += "The number is " + i + " ";	The number is 1
}	The number is 2
	The number is 3
	The number is 4

## While Loop

O loop **while** percorre um bloco de código **enquanto** uma condição especificada for verdadeira. É preciso estar atento para utilizar essa repetição pois diferente da repetição for, ela não possui um ponto final para ela ser interrompida, ou seja, ela vai se repetir infinitamente e isso pode causar erros no navegador. A estrutura de repetição while tem a seguinte sintaxe:

```
while (condition) {
    // code block to be executed
}
```

No exemplo a seguir o código dentro do bloco vai se repetir, de novo e de novo, enquanto a variável (i) for menor do que 10:

```
let text = "";
let i = 0;
while (i < 10) {
  text += "<br>The number is " + i;
  i++;
}
```

The number is 0  
The number is 1  
The number is 2  
The number is 3  
The number is 4  
The number is 5  
The number is 6  
The number is 7  
The number is 8  
The number is 9

## Do While Loop

O do while loop é uma variante do loop while. Este loop executará o bloco de código uma vez, antes de verificar se a condição seja verdadeira, então repetirá o loop enquanto a condição for verdadeira. O loop sempre será executado pelo menos uma vez, mesmo que a condição seja falsa, pois o bloco de código é executado antes que a condição seja testada:

```
do {
  // code block to be executed
}
while (condition);
```

## For In Loop

Em JavaScript a estrutura de repetição **for in** é executada entre as propriedades de um objeto. A sua sintaxe deve ser:

```
for (key in object) {
  // code block to be executed
}
```

### (for in, está relacionado ao index do objeto)

No exemplo abaixo, foi criada uma variável nomeada de "person", foram atribuídas propriedades a essa variável como o fname(first name), lname(last name) e age, o loop for in, itera sobre esse objeto, cada iteração retorna uma chave (definida como x em "let x"), cada chave é usada para acessar o endereço chave de cada propriedade do objeto, e no exemplo, atribuir o valor correspondente ao index à variável "txt".

```
const person = {fname:"John", lname:"Doe", age:25};
let txt = "";
for (let x in person) {
  txt += person[x] + " ";
}
```

John Doe 25

## For Of Loop

A instrução JavaScript for of percorre os valores de um objeto iterável. Ela permite que você faça um loop sobre estruturas de dados iteráveis, como Arrays, Strings, Maps, NodeLists e muito mais:

```
for (variable of iterable) {
  // code block to be executed
}
```

**(for of, está relacionado ao valor do objeto)**

```
const cars = ["BMW", "Volvo", "Mini"];
let text = "";
for (let x of cars) {
  text += x + "<br>";
}
```

BMW  
Volvo  
Mini

## ARRAYS

Um array é uma variável especial, que pode conter mais de um valor:

### Sintaxe

Uma array é comumente declarada com a keyword const. Espaços e quebras de linha não são importantes. Uma declaração pode abranger várias linhas

```
const array_name = [item1, item2, ...];
```

```
const cars = ["Saab", "Volvo", "BMW"];
```

É possível criar um array utilizando a keyword "new Array()".

```
const cars = new Array("Saab", "Volvo", "BMW");
```

Os dois exemplos fazem exatamente o mesmo. Não há necessidade de usar new Array(). Para simplicidade, legibilidade e velocidade de execução, use o método literal de matriz.

### Elementos Do Array

Os elementos de uma array são acessados e alterados referindo-se ao número do index dele. O index de uma array começa com 0.

```
const cars = ["Saab", "Volvo", "BMW"];
let car = cars[0];
```

Saab

```
const cars = ["Saab", "Volvo", "BMW"];
cars[0] = "Opel";
```

## Métodos E Propriedades Da Array

METHOD	DESCRIPTION
<b><u>concat()</u></b>	Joins two or more arrays, and returns a copy of the joined arrays
<b><u>copyWithin()</u></b>	Copies array elements within the array, to and from specified positions
<b><u>entries()</u></b>	Returns a key/value pair Array Iteration Object
<b><u>every()</u></b>	Checks if every element in an array pass a test
<b><u>fill()</u></b>	Fill the elements in an array with a static value
<b><u>filter()</u></b>	Creates a new array with every element in an array that pass a test
<b><u>find()</u></b>	Returns the value of the first element in an array that pass a test
<b><u>findIndex()</u></b>	Returns the index of the first element in an array that pass a test
<b><u>forEach()</u></b>	Calls a function for each array element
<b><u>from()</u></b>	Creates an array from an object
<b><u>includes()</u></b>	Check if an array contains the specified element
<b><u>indexOf()</u></b>	Search the array for an element and returns its position
<b><u>isArray()</u></b>	Checks whether an object is an array
<b><u>join()</u></b>	Joins all elements of an array into a string
<b><u>keys()</u></b>	Returns a Array Iteration Object, containing the keys of the original array
<b><u>lastIndexOf()</u></b>	Search the array for an element, starting at the end, and returns its position
<b><u>map()</u></b>	Creates a new array with the result of calling a function for each array element
<b><u>pop()</u></b>	Removes the last element of an array, and returns that element
<b><u>push()</u></b>	Adds new elements to the end of an array, and returns the new length
<b><u>reduce()</u></b>	Reduce the values of an array to a single value (going left-to-right)
<b><u>reduceRight()</u></b>	Reduce the values of an array to a single value (going right-to-left)
<b><u>reverse()</u></b>	Reverses the order of the elements in an array
<b><u>shift()</u></b>	Removes the first element of an array, and returns that element
<b><u>slice()</u></b>	Selects a part of an array, and returns the new array
<b><u>some()</u></b>	Checks if any of the elements in an array pass a test
<b><u>sort()</u></b>	Sorts the elements of an array
<b><u>splice()</u></b>	Adds/Removes elements from an array
<b><u>toString()</u></b>	Converts an array to a string, and returns the result
<b><u>unshift()</u></b>	Adds new elements to the beginning of an array, and returns the new length
<b><u>valueOf()</u></b>	Returns the primitive value of an array

PROPERTY	DESCRIPTION
<a href="#"><u>constructor</u></a>	Returns the function that created the Array object's prototype
<a href="#"><u>length</u></a>	Sets or returns the number of elements in an array
<a href="#"><u>prototype</u></a>	Allows you to add properties and methods to an Array object



## FUNÇÕES

Uma função JavaScript é um bloco de código projetado para executar uma tarefa específica. Uma função é executada quando ela é “invocada” (chamada).

### Sintaxe

Para declarar uma função, primeiro ela deve ser definida com a keyword `function`, seguido pelo seu nome e seguido por parênteses `()`. O nome da função pode conter letras, dígitos underscores e cifrão (mesma regra para nomes de variáveis). Entre os parênteses são incluídos os nomes dos parâmetros, separados por vírgulas. O código que será executado pela função, é colocado dentro de chaves `{ }`.

```
function name(parameter1, parameter2, parameter3) {
    // code to be executed
}
```

Os parâmetros são listados dentro dos parênteses `()` na definição da função. Os argumentos são valores recebidos pela função quando ela for chamada (invocada). Dentro da função os argumentos (parâmetros) se comportam como variáveis.

```
<p id="demo"></p>
<script>
function myFunction(p1, p2) {
    return p1 * p2;
}
document.getElementById("demo").innerHTML = myFunction(4, 3); 12
</script>
```

### Return

Quando o JavaScript atinge uma declaração `return`, a execução da função vai ser parada. As funções geralmente calculam um valor de retorno. O valor de retorno é "retornado" de volta ao "chamador".

```
function toCelsius(fahrenheit) {
    return (5/9) * (fahrenheit-32);
}
document.getElementById("demo").innerHTML = toCelsius(77); 25
```

### Operador ()

Usando o exemplo acima, “`toCelsius`” refere-se ao objeto da função e “`toCelsius()`” refere-se ao resultado da função. Acessar uma função sem `()` retornará o objeto da função em vez do resultado da função.

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}
function toCelsius(f) { return (5/9) * (f-32); }
document.getElementById("demo").innerHTML = toCelsius;
```

## Arrow Functions

As arrow functions foram introduzidas no ES6, elas permitem uma sintaxe curta para escrever expressões de função. Não é preciso da keyword function, da keyword return e das chaves, porém, só é permitido omitir as keywords return e as chaves se a função for uma única instrução.

```
// ES5
var x = function(x, y) {
  return x * y;
}

// ES6
const x = (x, y) => x * y;
```

## OBJECTS

Em JavaScript tudo é um objeto. Booleans podem ser objetos (se forem definidos com a keyword new). Numbers podem ser objetos (se for definido com a keyword new). Strings podem ser objetos (se forem definidas com a keyword new). Dates são sempre objetos. Maths são sempre objetos. Regexp são sempre objetos. Arrays são sempre objetos. Functions são sempre objetos. Objetos são sempre objetos. Todos os valores, exceto os primitivos, são objetos.

Em JavaScript variáveis podem conter um único valor.

```
let person = "John Doe";
```

Objetos são variáveis também, mas objetos podem conter muitos valores, eles são escritos em pares como **name : value** (nomes e valores separados por dois pontos). É uma prática comum declarar objetos com a keyword const.

```
const person = {firstName:"John", lastName:"Doe", age:50,
eyeColor:"blue"};
```

## Criação De Objetos

Com JavaScript, é possível definir e criar objetos de diferentes maneiras:

- Criar um único objeto usando um **object literal**.
- Criar um único objeto com a **keyword new**.
- Definir um **object constructor**, e então criar objetos do tipo construído.
- Criar um objeto usando o método `Object.create()`

### Usando Object Literal

Essa é a maneira mais fácil de criar um objeto em JavaScript. Usando um object literal, é possível definir e criar um objeto em uma única declaração. Um object literal é uma lista de pares de nomes : valores ( como age : 50), dentro de chaves { }.

```
const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

### Usando Keyword New

O seguinte exemplo primeiramente cria um novo objeto vazio usando `new object()`, e após isso adiciona as 4 propriedades:

```
const person = new Object();  
person.firstName = "John";  
person.lastName = "Doe";  
person.age = 50;  
person.eyeColor = "blue";
```

### Objetos São Mutáveis

Objetos são mutáveis, eles são endereçados por referencia não por valores. O seguinte exemplo explica isso. Se uma pessoa é um objeto, a seguinte declaração deveria criar uma cópia de uma pessoa.

```
const x = person;
```

Porém, o objeto x não é uma cópia de person. Ele é o mesmo objeto que person, se alterações forem feitas no object person, as mesmas alterações ocorreram em x pois eles são o mesmo objeto.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50, eyeColor: "blue"
}

const x = person;
x.age = 10; // Will change both x.age and person.age
```

## Object Properties

Propriedades (properties) são a parte mais importante nos objetos. Propriedades são os valores associados com os objetos. Um objeto é uma coleção de propriedades não ordenadas. Geralmente, propriedades podem ser alteradas, adicionadas ou deletadas, mas algumas são read only.

As seguintes sintaxes podem ser usadas para acessar uma propriedade de um objeto.

```
objectName.property // person.age
objectName["property"] // person["age"]
objectName[expression] // x = "age"; person[x]
```

É possível adicionar uma propriedade a um objeto existente, simplesmente nomeando ela e lhe dando um valor, conforme o exemplo.

```
person.nationality = "English";
```

É possível deletar uma propriedade simplesmente usando a keyword delete, conforme o exemplo.

```
delete person.age;
```

Os valores em um objeto podem ser outro objeto.

```
myObj = {
  name: "John",
  age: 30,
  cars: {
    car1: "Ford",
    car2: "BMW",
    car3: "Fiat"
  }
}
```

Os valores em um objeto também podem ser arrays e valores em arrays podem ser objetos.

```
const myObj = {
  name: "John",
  age: 30,
  cars: [
    {name:"Ford", models:["Fiesta", "Focus", "Mustang"]},
    {name:"BMW", models:["320", "X3", "X5"]},
    {name:"Fiat", models:["500", "Panda"]}
  ]
}
```

## Object Methods

Métodos (methods) são ações que podem ser realizadas em objetos. Um método é uma propriedade contendo uma definição de uma função. A seguinte sintaxe é usada para acessar um object methods:

```
objectName.methodName()
```

Se a propriedade for acessada sem parentes, o resultado irá retornar a definição da função.

É possível adicionar um método a um objeto existente, simples mente adicionando uma função como valor de uma propriedade.

```
person.name = function () {
  return this.firstName + " " + this.lastName;
};
```

Existem ainda os **métodos built-in**, são métodos já embutidos no JavaScript que permitem realizar muitas ações diferentes em objetos. O seguinte exemplo usa o método **toUpperCase()** para converter uma objeto string em um texto maiúsculo.

```
let message = "Hello world!";
let x = message.toUpperCase(); HELLO WORLD!
```

## Object Constructors

Criar objetos únicos podem trazer algumas limitações. É possível criar vários objetos do mesmo “tipo” usando uma **object constructor function**. Uma função que cria um “tipo de objeto”. No exemplo a seguir, **function Person()** é uma object constructor function. Objetos do mesmo tipo podem ser criados “chamando” a função com a keyword new. É considerado uma boa prática nomear funções de construtor com a primeira letra maiúscula.

```
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}

const myFather = new Person("John", "Doe", 50, "blue");
const myMother = new Person("Sally", "Rally", 48, "green");
```

## THIS

Em JavaScript, a keyword **this** se refere a um objeto. Ela pode se referir a diferentes objetos dependendo de como ela é usada.

- Em um object methods, **this** refere-se ao objeto.
- Sozinha, **this** refere-se ao global object.
- Em uma função, **this** refere-se ao global object.
- Em uma função, em strict mode, **this** is undefined.
- Em um evento, **this** refere-se ao elemento que está recebendo o evento.
- Métodos como call(), apply() e bind() podem fazer this se referir a qualquer objeto.

### This Em Um Método

Quando usado em um object methods, this refere-se ao objeto. No exemplo a seguir, this refere-se ao objeto “person”, já que, fullName(), é um método do objeto **person**.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  id: 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

Caso seja solicitado a exibição do método fullName, as propriedades em que this faz referencia serão buscadas no objeto “person”.

```
document.getElementById("demo").innerHTML =
person.fullName();           John Doe
```

## Alterando Contexto Do This

Resumidamente, existem funções especiais como `call()`, `apply()` e `bind()`, que são métodos predefinidos no JavaScript que podem mudar o contexto do `this`, logo, alterar como ele está fazendo a referência.

No exemplo a seguir o método `fullName()`, pertence ao objeto `person1`, porém com o uso dos métodos `call()` ou `apply()` esse método pode ser utilizado em outro objeto.

```
const person1 = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}

const person2 = {
  firstName: "John",
  lastName: "Doe",
}

let x = person1.fullName.call(person2);
```

John Doe

A diferença entre os métodos é que `call()` pega os argumentos separadamente, enquanto `apply()` pega os argumentos em um array.

## This Precedência

Para determinar a qual objeto `this` se refere; Use a seguinte ordem de precedência.

Precedence	Object
1	<code>bind()</code>
2	<code>apply()</code> and <code>call()</code>
3	Object method
4	Global scope

E faça as seguintes perguntas:

- Is this in a function being called using `bind()`?
- Is this in a function is being called using `apply()`?
- Is this in a function is being called using `call()`?
- Is this in an object function (method)?
- Is this in a function in the global scope.

## DATAS

Datas são objetos criados com o constructor `new Date()`. Existem 4 maneiras de criar uma data.

```
new Date()
new Date(year, month, day, hours, minutes, seconds, milliseconds)
new Date(milliseconds)
new Date(date string)
```

### New Date()

`new Date()` cria um novo objeto com a exata hora e data atual.

```
const d = new Date();
document.getElementById("demo").innerHTML = d;
```

Sun Jun 05 2022 20:52:46 GMT-0300 (Horário Padrão de Brasília)

### New Date(Year, Month, ...)

Cria um novo objeto data com hora e data específica. Podem ser especificados 7 números: ano, mês, dia, hora minuto, segundo e milissegundo (nessa exata ordem). Importante estar atento que JavaScript conta meses de 0 a 11 ou seja, Janeiro = 0 e Dezembro = 11 .

```
const d = new Date(2018, 11, 24, 10, 33, 30, 0);
document.getElementById("demo").innerHTML = d;
```

Mon Dec 24 2018 10:33:30 GMT-0200 (Horário de Verão de Brasília)

### New Date(Milliseconds)

O JavaScript armazena valores de datas com milissegundos, a partir de **January 01, 1970, 00:00:00 UTC (Universal Time Coordinated)**. `new Date(milliseconds)` Cria uma nova dada a partir do “tempo zero” somado com os milissegundos especificados. Ou seja, agora a data é (consulta da data na criação do texto) **Sun Jun 05 2022 20:44:56 GMT-0300 (Horário Padrão de Brasília)** e **1654472696529** milissegundos passados desde 01 de janeiro de 1970.

### New Date(Datestring)

Cria objetos datas a partir de uma data em formato string.

```
const d = new Date("October 13, 2014 11:13:00");
document.getElementById("demo").innerHTML = d;
```

Mon Oct 13 2014 11:13:00 GMT-0300 (Horário Padrão de Brasília)



## Get Date Methods

Esses são métodos que podem ser usados para obter (getting) informações de um date object:

METHOD	DESCRIPTION
getFullYear()	Get the <b>year</b> as a four digit number (yyyy)
getMonth()	Get the <b>month</b> as a number (0-11)
getDate()	Get the <b>day</b> as a number (1-31)
getHours()	Get the <b>hour</b> (0-23)
getMinutes()	Get the <b>minute</b> (0-59)
getSeconds()	Get the <b>second</b> (0-59)
getMilliseconds()	Get the <b>millisecond</b> (0-999)
getTime()	Get the time (milliseconds since January 1, 1970)
getDay()	Get the weekday as a number (0-6)
Date.now()	Get the time. ECMAScript 5.

```
const d = new Date();
document.getElementById("demo").innerHTML = d.getFullYear(); 2022
```

## Set Date Methods

Os métodos Set Date permitem definir valores de data (anos, meses, dias, horas, minutos, segundos, milissegundos) para um objeto de data. Esses são os métodos Set Date usados para definir uma parte de uma data:

Method	Description
setDate()	Set the day as a number (1-31)
setFullYear()	Set the year (optionally month and day)
setHours()	Set the hour (0-23)
setMilliseconds()	Set the milliseconds (0-999)
setMinutes()	Set the minutes (0-59)
setMonth()	Set the month (0-11)
setSeconds()	Set the seconds (0-59)
setTime()	Set the time (milliseconds since January 1, 1970)

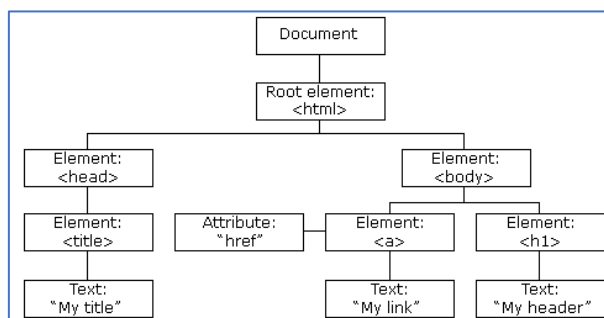
```
const d = new Date();
d.setFullYear(2020);
document.getElementById("demo").innerHTML = d; Fri Jun 05 2020 21:14:43 GMT-0300 (Horário Padrão de Brasília)
```

## WINDOW OBJECT

O objeto **window** é suportado em todos os tipos de navegadores, ele representa a “janela” do navegador. Em JavaScript todos os global objects (objetos globais), funções e variáveis automaticamente fazem parte do window object. Global variables são propriedades do window object, global functions são métodos do window object. Até mesmo o objeto document (do HTML DOM) é uma propriedade do objeto window.

## DOM (DOCUMENT OBJECT MODEL)

Quando uma página web é carregada, o navegador cria um **Document Object Model** da página. O modelo HTML DOM é construído de maneira similar a uma árvore.



Através desse modelo o JavaScript pode ganhar poder para criar um HTML mais dinâmico:

- JavaScript pode **alterar** todos os **elementos** HTML na página
- JavaScript pode **alterar** todos os **atributos** HTML na página
- JavaScript pode **alterar** todos os **estilos** CSS na página
- JavaScript pode **remover elementos e atributos** HTML existentes
- JavaScript pode **adicionar novos elementos** e atributos HTML
- JavaScript pode **reagir a** todos os **eventos** HTML existentes
- JavaScript pode **criar novos eventos** HTML na página

DOM é um padrão W3C (World Wide Web Consortium), ele define uma maneira de acesso a documentos. *"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."*

## Métodos E Propriedades Dom

Métodos HTML DOM são ações que podem ser executados em elementos HTML. Propriedades HTML DOM são valores de elementos HTML que pode ser estabelecidos ou alterados. A partir desse conceito é possível verificar essa diferença no seguinte exemplo.

O `getElementById` é um método, já que é uma ação que será executada, é a maneira mais comum de acessar um elemento HTML. A `innerHTML` é uma propriedade que, de maneira comum, é usada para acessar o conteúdo de um elemento HTML e alterá-lo.

```
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML =
"Hello World!";
</script>                                Hello World!
```

Nesse exemplo foi utilizado o método `getElementById` para “buscar” um elemento pela `id` correspondente, e utilizada a propriedade `innerHTML` para alterar o conteúdo desse elemento.

## Localizando Elementos Html Dom

Muitas vezes, com JavaScript, você deseja manipular elementos HTML. Para fazer isso, você tem que encontrar os elementos primeiro. Existem várias maneiras de fazer isso:

- Localizando elementos HTML por `id`
- Localizando elementos HTML por nome de tag
- Localizando elementos HTML por nome de classe
- Localizando elementos HTML por seletores CSS
- Localizando elementos HTML por coleções de objetos HTML.

A maneira mais fácil de localizar um elemento HTML no DOM é usando o `id` do elemento. O seguinte exemplo localiza o elemento com `id="intro"`

```
const element = document.getElementById("intro");
```

Se o elemento não for encontrado, a variável “`element`” vai receber um valor “`null`”.

O seguinte exemplo localiza todos os elementos com a tag <p>.

```
const element = document.getElementsByTagName("p");
```

O seguinte exemplo encontra todos os elementos HTML com o mesmo nome de classe.

```
const x = document.getElementsByClassName("intro");
```

Para encontrar todos os elementos HTML que correspondem a um seletor CSS específico (id, nomes de classes, tipos, atributos, valores de atributos, etc.), use o método **querySelectorAll()**. Este exemplo retorna uma lista de todos os elementos <p> com class="intro".

```
const x = document.querySelectorAll("p.intro");
```

É possível ainda localizar elementos através da object collections. Este exemplo encontra o elemento de formulário com id="frm1", na coleção de formulários e exibe todos os valores do elemento:

```
const x = document.forms["frm1"];
let text = "";
for (let i = 0; i < x.length; i++) {
    text += x.elements[i].value + "<br>";
}
document.getElementById("demo").innerHTML = text;
```

O primeiro HTML DOM Nível 1 (1998), definiu 11 objetos HTML, coleções de objetos e propriedades. Estes ainda são válidos em HTML5. Mais tarde, no HTML DOM Nível 3, mais objetos, coleções e propriedades foram adicionados.

Property	Description	DOM
document.anchors	Retorna todos os elementos do tipo "âncora" <a>	1
document.applets	Descontinuado	1
document.baseURI	Retorna a base URI absoluta do documento	3
document.body	Retorna o elemento <body>	1
document.cookie	Retorna o cookie do documento	1
document.doctype	Retorna o doctype do documento	3
document.documentElement	Retorna o elemento <html>	3
document.documentMode	Retorna o modo usado pelo navegador	3
document.documentURI	Retorna a URI do documento	3
document.domain	Retorna o nome do domínio do server documento	1

document.domConfig	Obsoleto	3
document.embeds	Retorna todos os elementos <embed>	3
document.forms	Retorna todos os elementos <form>	1
document.head	Retorna o elemento <head>	3
document.images	Retorna todos os elementos <img>	1
document.implementation	Retorna a implementação DOM	3
document.inputEncoding	Retorna a codificação do documento (o conjunto de caracteres)	3
document.lastModified	Retorna a hora e a data de quando o documento foi atualizado	3
document.links	Retorna todos os elementos <a> e <area> que tenham um atributo href	1
document.readyState	Retorna o status (carregamento) do documento	3
document.referrer	Retorna o URI do referenciador (o documento de vinculação)	1
document.scripts	Retorna todos os elementos <script>	3
document.strictErrorChecking	Retorna se a verificação de erros for aplicada	3
document.title	Retorna o elemento <title>	1
document.URL	Retorna a URL completa do documento	1

## Alterando Elementos Html Dom

A maneira mais fácil de modificar um conteúdo de um elemento HTML é usando a propriedade **innerHTML**.

```
<p id="p1">Hello World!</p>
<script>
document.getElementById("p1").innerHTML = "New text!";
</script>
```

New text!

Para mudar o atributo de um elemento é utilizada a seguinte sintaxe:

```
document.getElementById(id).attribute = new value
```

O seguinte exemplo altera o valor do atributo src de um elemento <img>.

```

<script>
document.getElementById("myImage").src = "landscape.jpg";
</script>
```

É possível ainda alterar o conteúdo diretamente no fluxo de saída do HTML com o método **document.write()**. Porém se ele for utilizado após a página ser carregada o documento pode ser sobrescrito.

PROPRIEDADE	DESCRIÇÃO
<i>element.innerHTML = novo conteúdo HTML</i>	Altera a parte interna do HTML de um elemento
<i>element.attribute = novo valor</i>	Altera o valor do atributo de um elemento HTML
<i>element.style.property = novo estilo</i>	Altera o estilo de um elemento HTML
MÉTODO	DESCRIÇÃO
<i>element.setAttribute(atributo, valor)</i>	Altera o valor de um atributo específico

## DOM EVENTS

Um código JavaScript também pode ser executado quando um evento ocorrer, como quando um usuário clica em um elemento HTML ou interage com ele. O seguinte exemplo altera um elemento HTML quando o usuário clicar sobre ele, uma função é chamada através de um manipulador de eventos.

```
<h2>JavaScript HTML Events</h2>
<h2 onclick="changeText(this)">Click on this text!</h2>
<script>
function changeText(id) {
    id.innerHTML = "Oops!";
}
</script>
```

Click on this text!

Oops!

É possível atribuir eventos usando o HTML DOM, assim como atribuir uma propriedade ou um método, como no seguinte exemplo.

```
<script>
document.getElementById("myBtn").onclick = displayDate;
</script>
```

Nesse exemplo foi atribuído o evento "onclick" a um elemento botão, a função **displayDate** é atribuída a um elemento HTML com o **id="myBtn"**, a função então vai ser executada quando o botão for clicado.

```
<h2>JavaScript HTML Events</h2>
<p>Click "Try it" to execute the displayDate() function.</p>

<button id="myBtn">Try it</button>
<p id="demo"></p>
<script>
document.getElementById("myBtn").onclick = displayDate;

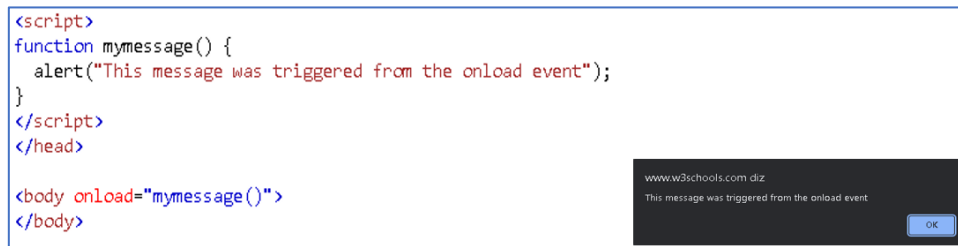
function displayDate() {
    document.getElementById("demo").innerHTML = Date();
}
</script>
```

Try it

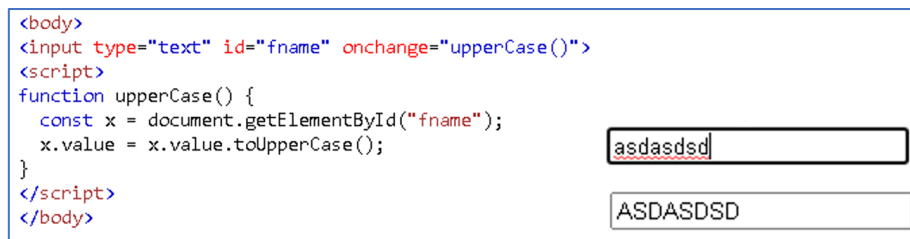
Sat Jun 11 2022 10:28:41 GMT-0300  
(Horário Padrão de Brasília)

Os eventos **onload** e **onunload** são provocados quando um usuário entre ou deixa uma página web, eles podem ser usados para checar se o navegador e a sua versão, e carregar uma página de web adequada com base nessa versão

por exemplo. No seguinte exemplo uma caixa de alerta com uma mensagem vai aparecer assim que a página for completamente carregada.



O evento **onchange** é usado muitas vezes para validação de campos de input. Como no seguinte exemplo, o evento vai chamar uma função "toUpperCase()" quando o usuário alterar "changes" o conteúdo do campo de input, essa função vai converter o valor preenchido no campo de input, em letras maiúsculas.



Os eventos **onmousedown**, **onmouseup** e **onclick** utilizam todo os estágios da ação de click do mouse. Primeiro, quando um botão do mouse é clicado, o evento **onmousedown** é provocado, então, quando o botão é liberado, o evento **onmouseup** é provocado, quando essas duas ações forem completadas o evento **onclick** é provocado.

Quando um evento ocorre em HTML, o evento pertence a um determinado objeto de evento, como um evento de clique do mouse pertence ao objeto MouseEvent. Todos os objetos de evento são baseados no objeto **Event** e herdam todas as suas propriedades e métodos.

### HTML DOM EVENTS:

[https://www.w3schools.com/jsref/dom\\_obj\\_event.asp](https://www.w3schools.com/jsref/dom_obj_event.asp)

## DOM EVENTLISTENER

O método **addEventListener()** anexa um manipulador de eventos a um elemento específico sem substituir manipuladores de eventos existentes. Você pode adicionar muitos manipuladores de eventos a um elemento. Você pode adicionar muitos manipuladores de eventos do mesmo tipo a um elemento, ou seja, dois eventos de "click". Você pode adicionar "event listeners" a qualquer objeto DOM, não apenas a elementos HTML. ou seja, o objeto de janela. É possível remover facilmente um event listeners usando o método **removeEventListener()**.

### Sintaxe

O primeiro parâmetro é o tipo do evento (como "click" ou "mousedown"), o segundo parâmetros é a função que deve ser chamada quando o evento ocorrer, e o terceiro parâmetro é opcional, é um valor booleano especificando quando usar event bubbling ou event capturing.

```
element.addEventListener(event, function, useCapture);
```

O seguinte exemplo exibe uma caixa de alerta na janela do navegador quando o botão com id="myBtn" for clicado (evento "click").

```
<button id="myBtn">Try it</button>
<script>
document.getElementById("myBtn").addEventListener("click",
myFunction);
function myFunction() {
  alert ("Hello World!");
}
</script>
```

Como já foi mencionado, através desse método, se torna possível adicionar muitos manipuladores de eventos, inclusivos os do mesmo tipo em um elemento.

```
<button id="myBtn">Try it</button>
<p id="demo"></p>
<script>
var x = document.getElementById("myBtn");
x.addEventListener("mouseover", myFunction);
x.addEventListener("click", mySecondFunction);
x.addEventListener("mouseout", myThirdFunction);
function myFunction() {
  document.getElementById("demo").innerHTML += "Moused over!<br>";
}

function mySecondFunction() {
  document.getElementById("demo").innerHTML += "Clicked!<br>";
}

function myThirdFunction() {
  document.getElementById("demo").innerHTML += "Moused out!<br>";
}
</script>
```

**Try it**

Moused over!  
Clicked!  
Moused out!



## THIS KEYWORD

Em JavaScript, a keyword **this** refere-se para a um objeto e pode se referir a diferentes objetos dependendo de como ela for usada.

### This Em Um Método

Quando usada em um método de um objeto, **this** refere-se ao objeto. O seguinte exemplo, “fullName” é um método do objeto com nome “person”, esse método executa uma função que utiliza o **this**, logo, o **this** vai referir-se a esse objeto com nome “person”.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  id: 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
document.getElementById("demo").innerHTML = person.fullName(); John Doe
```

### This Sozinho

Quando usado sozinho, **this** refere-se ao objeto global, ou seja, em um navegador, **this** refere-se ao objeto window.

```
let x = this;
document.getElementById("demo").innerHTML = x; [object Window]
```

### This Em Uma Função (Default)

Em uma função, por padrão o objeto global é vinculado ao **this**, ou seja, em um navegador, [object Window].

### This Em Uma Função (Strict)

O Strict Mode (que será abordado em um tópico mais avançado) não permite “vinculações padrões”, logo, **this** em uma função em strict mode será undefined.

### This Em Event Handlers.

Em HTML event handlers, **this** vai se referir ao elemento HTML que vai receber aquele evento.

```
<button
onclick="this.style.display=
'none'">Click to Remove Me!
</button>
```



## Empréstimo De Função Bind()

No seguinte exemplo, o objeto “member” utiliza o método “fullName”, mas esse é um método do objeto “person”, logo, ele vai se referir ao objeto “person”, com o uso do método bind() um objeto pode pegar emprestado o método de outro objeto.

```
<script>
const person = {
  firstName: "John",
  lastName: "Doe",
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}
const member = {
  firstName: "Hege",
  lastName: "Nilsen",
}
let fullName = person.fullName.bind(member);
document.getElementById("demo").innerHTML = fullName(); Hege Nilsen
```

## Ordem De Precedência This

Para identificar a qual objeto this está se referindo, é possível utilizar a seguinte ordem de precedência.

PRECEDENCIA	OBJETO
1	bind()
2	apply() and call()
3	Object method
4	Global scope

- This está em uma função sendo chamada usando os métodos bind(), apply() ou call()?
- This está em uma função de um objeto (método)?
- This está em uma função de escopo global?

## BIBLIOTECA MATH

O objeto Math em JavaScript permite executar cálculos matemáticos com números. Diferente de outros objetos, Math não tem um “constructor”, é um objeto estático.

### Math Properties (Constants)

O JavaScript fornece 8 “constantes matemáticas” que podem ser acessadas com as propriedades Math.

- Math.E // retorna o número de Euler
- Math.PI // retorna PI
- Math.SQRT2 // retorna a raiz quadrada de 2
- Math.SQRT1\_2 // retorna a raiz quadrada de 1/2
- Math.LN2 // retorna o logaritmo natural de 2
- Math.LN10 // retorna o logaritmo natural de 10
- Math.LOG2E // retorna o logaritmo de base 2 de E
- Math.LOG10E // retorna o logaritmo de base 10 de E

### Math Methods

A sintaxe para qualquer método Math é: Math.method(number)

MÉTODO	DESCRIÇÃO
<b>abs(x)</b>	Retorna o valor absoluto de x
<b>acos(x)</b>	Retorna o arco-cosseno de x, em radianos
<b>acosh(x)</b>	Retorna o arco-cosseno hiperbólico de x
<b>asin(x)</b>	Retorna o arco-seno de x, em radianos
<b>asinh(x)</b>	Retorna o arco-seno hiperbólico de x
<b>atan(x)</b>	Retorna o arco-tangente de x como um valor numérico entre -PI/2 e PI/2 radianos
<b>atan2(y, x)</b>	Retorna o arco tangente do quociente de seus argumentos
<b>atanh(x)</b>	Retorna o arcotangente hiperbólico de x
<b>cbrt(x)</b>	Retorna a raiz cúbica de x
<b>ceil(x)</b>	Retorna x, arredondado para cima para o valor inteiro mais próximo
<b>cos(x)</b>	Retorna o cosseno de x ( x deve estar em radianos)
<b>cosh(x)</b>	Retorna o cosseno hiperbólico de x
<b>exp(x)</b>	Retorna o valor de $E^x$
<b>floor(x)</b>	Retorna x, arredondado para baixo para o valor inteiro mais próximo
<b>log(x)</b>	Retorna o logaritmo natural (base E) de X
<b>max(x, y, z, ..., n)</b>	Retorna o número de maior valor
<b>min(x, y, z, ..., n)</b>	Retorna o número de menor valor
<b>pow(x, y)</b>	Retorna o valor de x elevado a y
<b>random()</b>	Retorna um número aleatório entre 0 e 1
<b>round(x)</b>	Arredonda x para o valor inteiro mais próximo
<b>sign(x)</b>	Retorna se x é um valor positivo, negativo ou nulo (-1, 0, 1)
<b>sin(x)</b>	Retorna o seno de x (x deve estar em radianos)

<b>sinh(x)</b>	Retorna o seno hiperbólico de x
<b>sqrt(x)</b>	Retorna a raiz quadrada de x
<b>tan(x)</b>	Retorna a tangente de um ângulo
<b>tanh(x)</b>	Retorn a tangente hiperbólica de um número
<b>trunc(x)</b>	Retorna a parte inteira de x

## Numeros Inteiros

Existem 4 métodos que são utilizados para arredondar números para um número inteiro:

<b>Math.round(x)</b>	Retorna x arredondado para o inteiro mais próximo
<b>Math.ceil(x)</b>	Retorna x arredondado para cima para o inteiro mais próximo
<b>Math.floor(x)</b>	Retorna x arredondado para baixo para o inteiro mais próximo
<b>Math.trunc(x)</b>	Retorna a parte inteira de x

## JSON

JSON significa **J**ava**S**cript **O**bject **N**otation, é um formato de texto para armazenar e transportar dados, é "auto descritivo" e fácil de entender. A sintaxe JSON é derivada da notação de objeto JavaScript, mas o formato JSON é somente texto. O formato JSON é sintaticamente semelhante ao código para criar objetos JavaScript. Por causa disso, um programa JavaScript pode facilmente converter dados JSON em objetos JavaScript. Como o formato é apenas texto, os dados JSON podem ser facilmente enviados entre computadores e usados por qualquer linguagem de programação.

### Jason Parse

O JavaScript tem uma função integrada para converter strings JSON em objetos JavaScript. Um uso comum do JSON é trocar dados de/para um servidor web. Ao receber dados de um servidor web, os dados são sempre uma string. Analise os dados com `JSON.parse()` e os dados se tornam um objeto JavaScript.

Imagine que recebemos este texto de um servidor web:

```
'{"name":"John", "age":30, "city":"New York"}'
```

Use a função JavaScript **JSON.parse()** para converter texto em um objeto JavaScript:

```
const obj = JSON.parse('{"name":"John", "age":30, "city":"New York"}');
```

### Jason Stringify

Ao enviar dados para um servidor web, os dados devem ser uma string. Converta um objeto JavaScript em uma string com **JSON.stringify()**.

Imagine que temos este objeto em JavaScript:

```
const obj = {name: "John", age: 30, city: "New York"};
```

Use a função JavaScript **JSON.stringify()** para convertê-lo em uma string.

```
const myJSON = JSON.stringify(obj);
```

## LOCAL STORAGE

A propriedade `localStorage` permite acessar um objeto Storage local. A `localStorage` é similar ao `sessionStorage`. A única diferença é que enquanto os dados armazenados no `localStorage` não expiram, os dados no `sessionStorage` tem os seus dados limpos ao expirar a sessão da página — ou seja, quando a página (aba ou janela) é fechada.

### Sintaxe

Salvar dados no armazenamento local:

```
localStorage.setItem(key, value);
```

Ler dados do armazenamento local:

```
let lastname = localStorage.getItem(key);
```

Remover dados do armazenamento local:

```
localStorage.removeItem(key);
```

Remover tudo (Limpar armazenamento local):

```
localStorage.clear();
```

O seguinte exemplo conta o número de vezes que um usuário clicou em um botão e mantém o valor mesmo se a página for recarregada ou fechada:

```
<p><button onclick="clickCounter()" type="button">Click me!</button></p>
<p>Number of clicks:</p>
<p id="demo"></p>
<script>
clickCounter();

function clickCounter() {
  if (localStorage.clickcount) {
    localStorage.clickcount = Number(localStorage.clickcount)+1;
  } else {
    localStorage.clickcount = 1;
  }
  document.getElementById("demo").innerHTML = localStorage.clickcount;
}
</script>
```

Click me!

Number of clicks: 9

O local storage converte sempre os dados em strings. O seguinte exemplo armazena um objeto no local storage e, através do uso do JSON, converte esse objeto para strings:

```
let a = {nome: "Romulo", n1: 7.3};
localStorage.setItem("aluno", JSON.stringify(a));
```

Após isso, esse objeto é acessado com o uso do método getItem() da propriedade localStorage, porém, os dados estão em formato de string, então é utilizado o JSON para converter em um objeto novamente (o valor é exibido no console para teste).

```
let b = localStorage.getItem("aluno");
console.log(JSON.parse(b))
```

```
{nome: 'Romulo', n1: 7.3}
  n1: 7.3
  nome: "Romulo"
```

## JAVASCRIPT TIMING EVENTS

O objeto window permite a execução de códigos de acordo com intervalos de tempo, esses intervalos de tempo são chamados de timing events.

### Settimeout

O método **setTimeout()** executa uma função, após aguardar um número especificado de milissegundos.

```
window.setTimeout(function, milliseconds);
```

Pode ser escrito sem o prefixo window, o primeiro parâmetro é uma função a ser executada, o segundo parâmetro indica o número de milissegundos antes da execução.

O método **clearTimeout()** interrompe a execução da função especificada em setTimeout(), porém, uma variável deve ser declarada e atribuída ao método setTimeout().

```
window.clearTimeout(timeoutVariable)
```

O seguinte exemplo vai exibir uma mensagem na tela após 3 segundos que o usuário interagir com o elemento botão “Try it”, porém se o usuário clicar no botão “Stop it” antes dos 3 segundos essa mensagem não será exibida.

```
<button onclick="myVar = setTimeout(myFunction,
3000)">Try it</button>
<button onclick="clearTimeout(myVar)">Stop it</button>
<script>
function myFunction() {
  alert("Hello");
}
</script>
```

Try it Stop it

## Setinterval

O método **setInterval()** faz o mesmo que setTimeout(), mas repete a execução da função continuamente.

```
window.setInterval(function, milliseconds);
```

O primeiro parâmetro é a função a ser executada. O segundo parâmetro indica a duração do intervalo de tempo entre cada execução.

O método **clearInterval()** interrompe as execuções da função especificada no método setInterval(). Esse método usa a variável retornada de setInterval().

O seguinte exemplo exibe as horas atuais, atualizando a contagem dos segundos, porém o botão “stop time” interrompe essa contagem quando for pressionado.

```

<p id="demo"></p>
<button onclick="clearInterval(myVar)">Stop time</button>
<script>
let myVar = setInterval(myTimer ,1000);
function myTimer() {
  const d = new Date();
  document.getElementById("demo").innerHTML =
d.toLocaleTimeString();
}
</script>

```

18:29:55

Stop time

## HTML DOM EVENTS REFERENCE

Os eventos HTML DOM permitem que o JavaScript registre diferentes event handlers (manipuladores eventos) em elementos de um documento HTML. Eventos são normalmente usados com uma combinação de funções e essas funções não vão ser executadas antes desses eventos ocorrerem.

EVENTOS	DESCRIÇÃO	PERTENCE A
<b><u>abort</u></b>	O evento ocorre quando o carregamento de uma mídia é interrompido	UiEvent, Event
<b><u>afterprint</u></b>	O evento ocorre quando uma página está sendo impressa, ou se a caixa de diálogo de impressão for fechada	<u>Event</u>
<b><u>animationend</u></b>	O evento ocorre quando uma animação CSS for completada	<u>AnimationEvent</u>
<b><u>animationiteration</u></b>	O evento ocorre quando uma animação CSS é repetida	<u>AnimationEvent</u>
<b><u>animationstart</u></b>	O evento ocorre quando uma animação CSS é começada	<u>AnimationEvent</u>
<b><u>beforeprint</u></b>	O evento ocorre uma página está prestes a ser impressa	<u>Event</u>
<b><u>beforeunload</u></b>	O evento ocorre antes do documento ser carregado	UiEvent, Event
<b><u>blur</u></b>	O evento ocorre quando um elemento perde o foco	<u>FocusEvent</u>
<b><u>canplay</u></b>	O evento ocorre quando o navegador pode começar a reproduzir a mídia (quando tiver buffer suficiente para começar)	<u>Event</u>
<b><u>canplaythrough</u></b>	O evento ocorre quando o navegador pode reproduzir a mídia sem parar para armazenamento em buffer	<u>Event</u>
<b><u>change</u></b>	O evento ocorre quando o conteúdo de um elemento de formulário, seleção ou verificação de estado for alterado (para <input>, <select> e <textarea>)	<u>Event</u>
<b><u>click</u></b>	O evento ocorre quando o usuário clicar em um elemento	<u>MouseEvent</u>
<b><u>contextmenu</u></b>	O evento ocorre quando o usuário clica com o botão direito em um elemento para abrir um context menu	<u>MouseEvent</u>



<b><u>copy</u></b>	O evento ocorre quando o usuário copia o conteúdo de um elemento	<a href="#"><u>ClipboardEvent</u></a>
<b><u>cut</u></b>	O evento ocorre quando o usuário recorta o conteúdo de um elemento	<a href="#"><u>ClipboardEvent</u></a>
<b><u>dblclick</u></b>	O evento ocorre quando o usuário clica duas vezes em um elemento	<a href="#"><u>MouseEvent</u></a>
<b><u>drag</u></b>	O evento ocorre quando um elemento está sendo arrastado	<a href="#"><u>DragEvent</u></a>
<b><u>dragend</u></b>	O evento ocorre quando o usuário para de arrastar um elemento	<a href="#"><u>DragEvent</u></a>
<b><u>dragenter</u></b>	O evento ocorre quando o elemento arrastado entra em um lugar onde ele pode ser "solto"	<a href="#"><u>DragEvent</u></a>
<b><u>dragleave</u></b>	O evento ocorre quando o elemento deixa o lugar onde ele pode ser solto	<a href="#"><u>DragEvent</u></a>
<b><u>dragover</u></b>	O evento ocorre quando o elemento arrastado está sobre o lugar onde ele pode ser solto	<a href="#"><u>DragEvent</u></a>
<b><u>dragstart</u></b>	O evento ocorre quando o usuário começa a arrastar um elemento	<a href="#"><u>DragEvent</u></a>
<b><u>drop</u></b>	O evento ocorre quando o usuário solta um elemento que estava sendo arrastado	<a href="#"><u>DragEvent</u></a>
<b><u>durationchange</u></b>	O evento ocorre quando a duração da mídia é alterada	<a href="#"><u>Event</u></a>
<b><u>ended</u></b>	O evento ocorre quando a mídia está chegando ao fim (útil para "obrigado por assistir")	<a href="#"><u>Event</u></a>
<b><u>error</u></b>	O evento ocorre quando há um erro durante o carregamento de arquivos externos	<a href="#"><u>ProgressEvent</u></a> , <a href="#"><u>UiEvent</u></a> , <a href="#"><u>Event</u></a>
<b><u>focus</u></b>	O evento ocorre quando um elemento está sendo focado	<a href="#"><u>FocusEvent</u></a>
<b><u>focusin</u></b>	O evento ocorre quando um elemento está prestes a ser focado	<a href="#"><u>FocusEvent</u></a>
<b><u>focusout</u></b>	O evento ocorre quando o elemento está prestes a perder o foco	<a href="#"><u>FocusEvent</u></a>
<b><u>fullscreenchange</u></b>	O evento ocorre quando um elemento é exibido em modo de tela-cheia	<a href="#"><u>Event</u></a>
<b><u>fullscreenerror</u></b>	O evento ocorre quando um elemento não pode ser exibido em modo de tela-cheia	<a href="#"><u>Event</u></a>
<b><u>hashchange</u></b>	O evento ocorre quando houver alterações na URL na "âncora"	<a href="#"><u>HashChangeEvent</u></a>
<b><u>input</u></b>	O evento ocorre quando um elemento recebe um input do usuário	<a href="#"><u>InputEvent</u></a> , <a href="#"><u>Event</u></a>
<b><u>invalid</u></b>	O evento ocorre quando um elemento é inválido	<a href="#"><u>Event</u></a>
<b><u>keydown</u></b>	O evento ocorre quando o usuário aperta uma tecla	<a href="#"><u>KeyboardEvent</u></a>
<b><u>keypress</u></b>	O evento ocorre quando o usuário pressiona uma tecla	<a href="#"><u>KeyboardEvent</u></a>
<b><u>keyup</u></b>	O evento ocorre quando o usuário solta uma tecla	<a href="#"><u>KeyboardEvent</u></a>
<b><u>load</u></b>	O evento ocorre quando um objeto é carregado	<a href="#"><u>UiEvent</u></a> , <a href="#"><u>Event</u></a>

<b><u>loadeddata</u></b>	O evento ocorre quando os dados de uma mídia forem carregados	<a href="#"><u>Event</u></a>
<b><u>loadedmetadata</u></b>	O evento ocorre quando os metadados são carregados	<a href="#"><u>Event</u></a>
<b><u>loadstart</u></b>	O evento ocorre quando o navegador começar a procurar por uma mídia específica	<a href="#"><u>ProgressEvent</u></a>
<b><u>message</u></b>	O evento ocorre quando uma mensagem for recebida através da fonte do evento	<a href="#"><u>Event</u></a>
<b><u>mousedown</u></b>	O evento ocorre quando o usuário pressiona um botão do mouse sobre um elemento	<a href="#"><u>MouseEvent</u></a>
<b><u>mouseenter</u></b>	O evento ocorre quando o ponteiro do mouse está sobre um elemento	<a href="#"><u>MouseEvent</u></a>
<b><u>mouseleave</u></b>	O evento ocorre quando o ponteiro do mouse deixa um elemento	<a href="#"><u>MouseEvent</u></a>
<b><u>mousemove</u></b>	O evento ocorre quando o ponteiro do mouse está se movendo sobre um elemento	<a href="#"><u>MouseEvent</u></a>
<b><u>mouseover</u></b>	O evento ocorre quando o ponteiro do mouse está sobre um elemento ou sobre seus "filhos"	<a href="#"><u>MouseEvent</u></a>
<b><u>mouseout</u></b>	O evento ocorre quando o usuário move o ponteiro do mouse para fora de um elemento ou de seus "filhos"	<a href="#"><u>MouseEvent</u></a>
<b><u>mouseup</u></b>	O evento ocorre quando o usuário solta o botão do mouse	<a href="#"><u>MouseEvent</u></a>
<b><u>mousewheel</u></b>	<u>Deprecated. Use the wheel event instead</u>	<a href="#"><u>WheelEvent</u></a>
<b><u>offline</u></b>	O evento ocorre quando o navegador começa a trabalhar de maneira offline.	<a href="#"><u>Event</u></a>
<b><u>online</u></b>	O evento ocorre quando o navegador começa a trabalhar de maneira online.	<a href="#"><u>Event</u></a>
<b><u>open</u></b>	O evento ocorre quando uma conexão com um evento de servidor é aberta.	<a href="#"><u>Event</u></a>
<b><u>pagehide</u></b>	O evento ocorre quando o usuário navega para fora de uma página web	<a href="#"><u>PageTransitionEvent</u></a>
<b><u>pageshow</u></b>	O evento ocorre quando o usuário navega para uma página web.	<a href="#"><u>PageTransitionEvent</u></a>
<b><u>paste</u></b>	O evento ocorre quando o usuário cola algum conteúdo em um elemento	<a href="#"><u>ClipboardEvent</u></a>
<b><u>pause</u></b>	O evento ocorre quando uma mídia é pausada	<a href="#"><u>Event</u></a>
<b><u>play</u></b>	O evento ocorre quando uma mídia começa a ser reproduzida	<a href="#"><u>Event</u></a>
<b><u>playing</u></b>	O evento ocorre quando a mídia está sendo reproduzida após ter sido pausada ou interrompida para armazenamento em buffer	<a href="#"><u>Event</u></a>
<b><u>popstate</u></b>	O evento ocorre quando o histórico é alterado	<a href="#"><u>PopStateEvent</u></a>
<b><u>progress</u></b>	O evento ocorre quando o navegador está em processo de obtenção dos dados de mídia (baixando a mídia)	<a href="#"><u>Event</u></a>
<b><u>ratechange</u></b>	O evento ocorre quando a velocidade de reprodução de uma mídia é alterada	<a href="#"><u>Event</u></a>
<b><u>resize</u></b>	O evento ocorre quando a visualização de um documento for redimensionada	<a href="#"><u>UiEvent</u></a> , <a href="#"><u>Event</u></a>

<b><u>reset</u></b>	O evento ocorre quando um formulário é resetado	<a href="#"><u>Event</u></a>
<b><u>scroll</u></b>	O evento ocorre quando a barra de rolagem de um elemento está sendo rolada	<a href="#"><u>UiEvent</u></a> , <a href="#"><u>Event</u></a>
<b><u>search</u></b>	O evento ocorre quando o usuário escreve algo em um campo de pesquisa (para <code>&lt;input="search"&gt;</code> )	<a href="#"><u>Event</u></a>
<b><u>seeked</u></b>	O evento ocorre quando o usuário para de mover ou pular para uma nova posição da mídia.	<a href="#"><u>Event</u></a>
<b><u>seeking</u></b>	O evento ocorre quando o usuário começa a mover ou pular para uma nova posição da mídia.	<a href="#"><u>Event</u></a>
<b><u>select</u></b>	O evento ocorre após o usuário selecionar algum texto (para <code>&lt;input&gt;</code> e <code>&lt;textarea&gt;</code> )	<a href="#"><u>UiEvent</u></a> , <a href="#"><u>Event</u></a>
<b><u>show</u></b>	O evento ocorre quando um elemento <code>&lt;menu&gt;</code> é exibido como um context menu	<a href="#"><u>Event</u></a>
<b><u>stalled</u></b>	O evento ocorre quando o navegador está tentando receber dados de mídia, mas os dados não são disponíveis	<a href="#"><u>Event</u></a>
<b><u>storage</u></b>	O evento ocorre quando uma área de Web Storage é atualizada	<a href="#"><u>StorageEvent</u></a>
<b><u>submit</u></b>	O evento ocorre quando um formulário é enviado	<a href="#"><u>Event</u></a>
<b><u>suspend</u></b>	O evento ocorre quando o navegador não está recebendo dados de mídia de maneira intencional	<a href="#"><u>Event</u></a>
<b><u>timeupdate</u></b>	O evento ocorre quando a posição de reprodução é alterada.	<a href="#"><u>Event</u></a>
<b><u>toggle</u></b>	O evento ocorre quando o usuário fecha ou abre o elemento <code>&lt;details&gt;</code>	<a href="#"><u>Event</u></a>
<b><u>touchcancel</u></b>	O evento ocorre quando o usuário interrompe o touch	<a href="#"><u>TouchEvent</u></a>
<b><u>touchend</u></b>	O evento ocorre quando o dedo é removido de uma tela touch screen	<a href="#"><u>TouchEvent</u></a>
<b><u>touchmove</u></b>	O evento ocorre quando o dedo é arrastado na tela	<a href="#"><u>TouchEvent</u></a>
<b><u>touchstart</u></b>	O evento ocorre quando um dedo é pressionado em uma tela touch screen	<a href="#"><u>TouchEvent</u></a>
<b><u>transitionend</u></b>	O evento ocorre quando uma transição CSS é completada	<a href="#"><u>TransitionEvent</u></a>
<b><u>unload</u></b>	O evento ocorre quando uma página estiver completamente carregada (para <code>&lt;body&gt;</code> )	<a href="#"><u>UiEvent</u></a> , <a href="#"><u>Event</u></a>
<b><u>volumechange</u></b>	O evento ocorre quando o volume de uma mídia é alterado	<a href="#"><u>Event</u></a>
<b><u>waiting</u></b>	O evento ocorre quando uma mídia é pausada, mas está esperando para ser reproduzida	<a href="#"><u>Event</u></a>
<b><u>wheel</u></b>	O evento ocorre quando a roda do mouse rola para cima ou para baixo sobre um elemento	<a href="#"><u>WheelEvent</u></a>

## OBJETOS DE EVENTO HTML DOM

Quando um evento ocorre em HTML, o evento pertence a um determinado objeto de evento, como um evento de clique do mouse pertence ao objeto MouseEvent. Todos os objetos de evento são baseados no objeto Event e herdam todas as suas propriedades e métodos

Event Object	Descrição
<u>Event</u>	Parent de todos os objetos de evento

Objeto de Evento	Descrição
<u>AnimationEvent</u>	Para animações CSS
<u>ClipboardEvent</u>	Para modificação da área de transferência
<u>DragEvent</u>	Para interações de arrastar e soltar
<u>FocusEvent</u>	Para eventos de foco
<u>HashChangeEvent</u>	Para alterações na parte âncora do URL
<u>InputEvent</u>	Para input do usuário
<u>KeyboardEvent</u>	Para interação com o teclado
<u>MouseEvent</u>	Para interação com o mouse
<u>PageTransitionEvent</u>	Para navegar e sair de páginas web
<u>PopStateEvent</u>	Para alterações no histórico de entrada
<u>ProgressEvent</u>	Para progresso de carregamento de recursos extenos
<u>StorageEvent</u>	Para alterações na área de armazenamento da janela.
<u>TouchEvent</u>	Para interações touch
<u>TransitionEvent</u>	Para transições CSS
<u>UiEvent</u>	Para interações de interface de usuário UI
<u>WheelEvent</u>	Para interações com a roda do mouse

## PROPRIEDADES E MÉTODOS DO EVENTO HTML DOM

PROPRIEDADES / MÉTODOS	DESCRIÇÃO
<u>altKey</u>	Retorna se a tecla "ALT" for pressionada quando um evento de mouse for acionado
<u>altKey</u>	Retorna se a tecla "ALT" for pressionada quando um evento de tecla for acionado
<u>animationName</u>	Retorna o nome da animação
<u>bubbles</u>	Retorna se um evento específico é um "bubbling event"
<u>button</u>	Retorna qual botão do mouse foi pressionado quando o "mouse event" foi acionado
<u>buttons</u>	Retorna quais botões do mouse foram pressionados quando o "mouse event" foi acionado
<u>cancelable</u>	Retorna se um evento pode ou não ter sua ação padrão impedida

<b><u>charCode</u></b>	Retorna o código de caractere UNICODE da tecla que acionou o evento onkeypress
<b>changeTouches</b>	Retorna uma lista de todos os objetos touch que tiveram seus estados alterados
<b><u>clientX</u></b>	Retorna a coordenada horizontal do ponteiro do mouse relativa a janela atual quando um "mouse event" for acionado
<b><u>clientY</u></b>	Retorna a coordenada vertical do ponteiro do mouse relativa a janela atual quando um "mouse event" for acionado
<b>clipboardData</b>	Retorna um objeto contendo os dados afetados pela execução da área de transferência
<b><u>code</u></b>	Retorna o código da tecla que acionou o evento
<b>composed</b>	Retorna se um evento é composto
<b><u>ctrlKey</u></b>	Retorna se a tecla "CTRL" for pressionada quando um evento de mouse for acionado
<b><u>ctrlKey</u></b>	Retorna se a tecla "CTRL" for pressionada quando um evento de tecla for acionado
<b><u>currentTarget</u></b>	Retorna o elemento em que foi acionado um evento pelos "event listeners"
<b><u>data</u></b>	Retorna os caracteres inseridos
<b>dataTransfer</b>	Retorna o elemento em que foi acionado um evento pelos "event listeners"
<b><u>defaultPrevented</u></b>	Retorna se o método preventDefault() foi ou não chamado para o evento
<b><u>deltaX</u></b>	Retorna a quantidade de rolagem horizontal da roda do mouse
<b><u>deltaY</u></b>	Retorna a quantidade de rolagem vertical da roda do mouse
<b><u>deltaZ</u></b>	Retorna a quantidade de rolagem de uma roda do mouse para o eixo z
<b><u>deltaMode</u></b>	Retorna a unidade de medida dos valores delta (pixels, linhas ou páginas)
<b><u>detail</u></b>	Retorna o número que indica quantas vezes o mouse foi clicado
<b><u>elapsedTime</u></b>	Retorna o número de segundos de duração de uma animação
<b><u>elapsedTime</u></b>	Retorna o número de segundos de duração de uma transição
<b><u>eventPhase</u></b>	Retorna qual fase de fluxo de eventos está sendo avaliada no momento
<b>getTargetRanges()</b>	Retorna um array contendo intervalos de destino que podem ser inseridos ou deletados.
<b><u>getModifierState()</u></b>	Retorna um array contendo intervalos de destino que podem ser inseridos ou deletados.
<b><u>inputType</u></b>	Retorna o tipo de alteração ("inserir" ou "excluir")
<b>isComposing</b>	Retorna se o estado de um evento é composto ou não
<b><u>isTrusted</u></b>	Retorna se um evento é verdadeiro
<b><u>key</u></b>	Retorna o valor da tecla representada por um evento
<b>key</b>	Retorna a chave do item de armazenamento alterado

<b><u>keyCode</u></b>	Retorna o código de caractere Unicode da tecla que acionou o evento onkeypress, onkeydown ou onkeyup.
<b><u>location</u></b>	Retorna a localização da tecla em um teclado ou dispositivo
<b>lengthComputable</b>	Retorna se a comprimento de uma progressão pode ser computável
<b>loaded</b>	Retorna quanto foi carregado
<b><u>metaKey</u></b>	Retorna se a tecla "META" foi pressionada quando o evento de tecla foi acionado
<b><u>metaKey</u></b>	Retorna se a tecla "meta" foi pressionada quando o evento de tecla foi acionado
<b>MovementX</b>	Retorna a coordenada horizontal do ponteiro do mouse relativa à posição do último mouse event
<b>MovementY</b>	Retorna a coordenada vertical do ponteiro do mouse relativa à posição do último mouse event
<b>newValue</b>	Retorna o novo valor do item de armazenamento alterado
<b><u>newURL</u></b>	Retorna a URL do documento, após o hash ter sido alterado
<b>offsetX</b>	Retorna a coordenada horizontal do ponteiro do mouse em relação à posição da borda do elemento alvo
<b>offsetY</b>	Retorna a coordenada vertical do ponteiro do mouse em relação à posição da borda do elemento alvo
<b>oldValue</b>	Retorna o valor antigo do item de armazenamento alterado
<b><u>oldURL</u></b>	Retorna a URL do documento, antes da alteração do hash
<b>onemptied</b>	O evento ocorre quando algo ruim acontece e o arquivo de mídia fica repentinamente indisponível (como desconexões inesperadas)
<b><u>pageX</u></b>	Retorna a coordenada horizontal do ponteiro do mouse, relativa ao documento, quando o evento do mouse foi acionado
<b><u>pageY</u></b>	Retorna a coordenada vertical do ponteiro do mouse, relativa ao documento, quando o evento do mouse foi acionado
<b><u>persisted</u></b>	Retorna se a página da web foi armazenada em cache pelo navegador
<b><u>preventDefault()</u></b>	Cancela o evento se for cancelável, o que significa que a ação padrão que pertence ao evento não ocorrerá
<b><u>propertyName</u></b>	Retorna o nome da propriedade CSS associada à animação ou transição
<b>pseudoElement</b>	Retorna o nome do pseudoelemento da animação ou transição
<b>region</b>	
<b><u>relatedTarget</u></b>	Retorna o elemento relacionado ao elemento que acionou o evento do mouse
<b><u>relatedTarget</u></b>	Retorna o elemento relacionado ao elemento que acionou o evento


<b>repeat</b>	Retorna se uma tecla está sendo pressionada repetidamente ou não
<b><u>screenX</u></b>	Retorna a coordenada horizontal do ponteiro do mouse, relativa à tela, quando um evento foi acionado
<b><u>screenY</u></b>	Retorna a coordenada horizontal do ponteiro do mouse, relativa à tela, quando um evento foi acionado
<b><u>shiftKey</u></b>	Retorna se a tecla "SHIFT" foi pressionada quando um evento foi acionado
<b><u>shiftKey</u></b>	Retorna se a tecla "SHIFT" foi pressionada quando um evento de tecla foi acionado
<b>state</b>	Retorna um objeto contendo uma cópia das entradas do histórico
<b><u>stopImmediatePropagation()</u></b>	Impede que outros "event listeners" do mesmo evento sejam chamados
<b><u>stopPropagation()</u></b>	Impede a propagação de um evento durante o fluxo de eventos
<b>storageArea</b>	Retorna um objeto que representa o objeto de armazenamento afetado
<b><u>target</u></b>	Retorna o elemento que acionou o evento
<b><u>targetTouches</u></b>	Retorna uma lista de todos os objetos de toque que estão em contato com a superfície e onde o evento touchstart ocorreu no mesmo elemento de destino que o elemento de destino atual
<b><u>timeStamp</u></b>	Retorna a hora (em milissegundos em relação à época) em que o evento foi criado
<b>total</b>	Retorna a quantidade total de trabalho que será carregado
<b><u>touches</u></b>	Retorna uma lista de todos os objetos de toque que estão atualmente em contato com a superfície
<b><u>transitionend</u></b>	O evento ocorre quando uma transição CSS é concluída
<b><u>type</u></b>	Retorna o nome do evento
<b>url</b>	Retorna a URL do documento do item alterado
<b><u>which</u></b>	Retorna qual botão do mouse foi pressionado quando o evento do mouse foi acionado
<b><u>which</u></b>	Retorna o código de caractere Unicode da tecla que acionou o evento onkeypress, onkeydown ou onkeyup
<b><u>view</u></b>	Retorna uma referência ao objeto Window onde ocorreu o evento



## JS STRICT MODE

O propósito da expressão **"use strict"**; é definir que o JavaScript deve ser executado em "strict mode". Com o strict mode não é possível utilizar variáveis não declaradas, tornando o código mais limpo. A expressão deve ser declarada no começo do script ou da função, declarar no começo para escopo global e no começo de uma função para escopo local.

```
<script>  
"use strict";  
x = 3.14;  
</script>
```

A screenshot of a browser console showing a red error message: "Uncaught ReferenceError: x is not defined". The message is preceded by a red 'x' icon.

### Por Que Usar Strict Mode

O modo estrito torna mais fácil escrever um código JavaScript "seguro". No JavaScript normal, digitar incorretamente um nome de variável cria uma nova variável global. No modo estrito, isso gerará um erro, impossibilitando a criação acidental de uma variável global. No JavaScript normal, um desenvolvedor não receberá nenhum feedback de erro atribuindo valores a propriedades non-writable. No modo estrito, qualquer atribuição a uma propriedade non-writable, uma propriedade getter-only, uma propriedade non-existing, uma variável non-existing ou um objeto non-existing gerará um erro.

### Não Permitido No Strict Mode

- Usar uma variável, sem declará-la, não é permitido
- Usar um objeto, sem declará-lo, não é permitido
- A exclusão de uma variável (ou objeto) não é permitida
- A exclusão de uma função não é permitida
- Não é permitido duplicar um nome de parâmetro
- Literais numéricos octais não são permitidos
- Caracteres de escape octais não são permitidos
- Não é permitido gravar em uma propriedade somente leitura
- Não é permitido gravar em uma propriedade get-only
- A exclusão de uma propriedade que não pode ser excluída não é permitida
- A palavra eval não pode ser usada como variável
- A palavra arguments não pode ser usada como variável
- A declaração with não é permitida
- Por questões de segurança, eval() não é permitido criar variáveis no escopo do qual foi chamado
- A palavra-chave this em funções se comporta de maneira diferente no modo estrito. A palavra-chave this refere-se ao objeto que chamou a função.
- Palavras-chave reservadas para futuras versões do JavaScript NÃO podem ser usadas como nomes de variáveis no modo estrito.



## ARROW FUNCTION

As arrow functions foram introduzidas no ES6 elas permitem escrever uma sintaxe de função mais curta.

```
let myFunction = (a, b) => a * b;
```

Se a função tiver apenas uma instrução e a instrução for retornar um valor, você poderá remover os colchetes e a palavra-chave return:

```
hello = function() {  
  return "Hello World!";  
}  
hello = () => "Hello World!";
```

### Arrow Function This

O manuseio do this é diferente nas arrow functions em comparação com as funções normais pois não há vinculação de this. Em funções regulares, a palavra-chave this representa o objeto que chama a função, que poderia ser a janela, o documento, um botão ou qualquer outra coisa. Com arrow functions, a palavra-chave this sempre representa o objeto que definiu a função de seta. No seguinte exemplo com uma função regular, this representa o objeto que chama a função:

```
<body>  
<button id="btn">Click Me!</button>  
<p id="demo"></p>  
<script>  
var hello;  
hello = function() {  
  document.getElementById("demo").innerHTML += this;  
}  
window.addEventListener("load", hello);  
document.getElementById("btn").addEventListener("click", hello);  
</script>  
</body>
```

[object Window][object HTMLButtonElement]

Com uma arrow function, this representa o proprietário da função:

```
<body>  
<button id="btn">Click Me!</button>  
<p id="demo"></p>  
<script>  
var hello;  
hello = () => {  
  document.getElementById("demo").innerHTML += this;  
}  
window.addEventListener("load", hello);  
document.getElementById("btn").addEventListener("click", hello);  
</script>  
</body>
```

[object Window][object Window]

## JS ASYNC

### Callback

Um callback é quando uma função passa um argumento para outra função, uma função pode ser executada apenas quando outra função for finalizada. Em JavaScript uma função é executada na sequência em que ela é chamada, não na sequência em que ela é definida.

No seguinte exemplo, é necessário estabelecer um controle melhor na execução de uma função. No exemplo é desejado realizar um cálculo simples e exibir o seu resultado.

No primeiro exemplo a função de cálculo é chamada (myCalculator), atribuído o resultado a uma variável, e então chamando novamente outra função (myDisplayer) para exibir o resultado.

```
<p id="demo"></p>
<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2) {
  let sum = num1 + num2;
  return sum;
}
let result = myCalculator(5, 5);
myDisplayer(result);
</script>
```

Do a calculation and then display the result.

10

No segundo exemplo a função é possível chamar somente uma função (myCalculator), e então essa função vai chamar a outra função (myDisplayer).

```
<p id="demo"></p>
<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2) {
  let sum = num1 + num2;
  myDisplayer(sum);
}
myCalculator(5, 5);
</script>
```

Do a calculation and then display the result.

10

O problema no primeiro exemplo é que vai ser necessário chamar duas funções para solucionar o resultado e exibi-lo. O problema no segundo exemplo é que, não é possível prevenir a função “myCalculator” de exibir o resultado, mesmo esse não sendo o objetivo dessa função que era somente “calcular”.

É possível chamar uma função (myCalculator) com um callback (outra função como um argumento), e então a função vai realizar o callback depois de terminar o cálculo. Como no seguinte exemplo:

```
<p id="demo"></p>
<script>
function myDisplayer(something) {
  document.getElementById("demo").innerHTML = something;
}
function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}
myCalculator(5, 5, myDisplayer);
</script>
```

Do a calculation and then display the result  
10

Nesse exemplo, “myDisplayer” é o nome de uma função ela é passada como argumento para outra função “myCalculator()”. É possível observar que quando uma função é utilizada como argumento, não é utilizado os parenteses.

## Asynchronous Javascript

O exemplo anterior foi um exemplo bem simples que tinha apenas o propósito de demonstrar a sintaxe e comportamento das “funções callback”. Geralmente as funções callback são utilizadas em funções assíncronas. Funções assíncronas são funções que estão sendo executadas em paralelo a outros funções.

Um exemplo típico é o uso de callback no método **setTimeout()**. Quando uma função setTimeout() é utilizada, é possível especificar uma função callback para ser executada quando terminar o tempo limite especificado. No seguinte exemplo acima “myFunction” é uma função callback que é passada como um argumento para a função “setTimeout()”, somente após 3000 milissegundos “myFunction()” é chamada.

```
<h1 id="demo"></h1>
<script>
setTimeout(myFunction, 3000);
function myFunction() {
  document.getElementById("demo").innerHTML = "Hello World!";
}
</script>
```

Wait 3 seconds (3000 milliseconds) for this page to change.  
**Hello World!**

No seguinte exemplo, “function(){ myFunction("Hello World!"); }” é usada como callback, é uma função completa que é passada para setTimeout() como um argumento.

```
<h1 id="demo"></h1>
<script>
setTimeout(function() { myFunction("Hello World!"); }, 3000);
function myFunction(value) {
  document.getElementById("demo").innerHTML = value;
}
</script>
```

Wait 3 seconds (3000 milliseconds) for this page to change.  
**Hello World!**

No próximo exemplo é utilizada a função **setInterval()** especificando uma função callback para ser executada a cada intervalo de tempo.

```
<h1 id="demo"></h1>
<script>
setInterval(myFunction, 1000);
function myFunction() {
  let d = new Date();
  document.getElementById("demo").innerHTML=
    d.getHours() + ":" +
    d.getMinutes() + ":" +
    d.getSeconds();
}
</script>
```

Using setInterval() to display the time every second (1000 milliseconds).

**14:14:47**

No exemplo anterior “myFunction” é uma função usada como callback, ou seja, um argumento para função “setInterval()”. A cada intervalo de 1000 milissegundos a função “myFunction()” é chamada.

É possível criar uma função que carrega um recurso externo (como um script ou arquivo) e não pode ser executada antes do conteúdo desse recurso ser completamente carregado.

No seguinte exemplo é carregado um arquivo HTML (mycar.html) e então exibido em uma web page somente após ele ser completamente carregado. A função “myDisplayer” é usada como callback pois é passada para “getFile()” como argumento.

```
mycar.html



<p>A car is a wheeled, self-powered motor vehicle used for transportation.
Most definitions of the term specify that cars are designed to run primarily on roads, to have seating
for one to eight people, to typically have four wheels.</p>

<p>(Wikipedia)</p>
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Callbacks</h2>
<p id="demo"></p>
<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function getFile(myCallback) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.html");
  req.onload = function() {
    if (req.status == 200) {
      myCallback(this.responseText);
    } else {
      myCallback("Error: " + req.status);
    }
  }
  req.send();
}
getFile(myDisplayer);
</script>
</body>
</html>
```

## JavaScript Callbacks



A car is a wheeled, self-powered motor vehicle used for transportation. Most definitions of the term specify that cars are designed to run primarily on roads, to have seating for one to eight people, to typically have four wheels.

(Wikipedia)

## Promises

Uma promises é um objeto JavaScript que vincula um código produtor e o código consumidor (“utilizar no lugar do callback”). O "código produtor" é um código que pode levar algum tempo, já o "código consumidor" é o código que aguarda um resultado. Um objeto Promise contém ambos.

### Sintaxe:

```
let myPromise = new Promise(function(myResolve, myReject) {
  // "Producing Code" (May take some time)

  myResolve(); // when successful
  myReject(); // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

O exemplo a seguir demonstra a utilização de do método Promise() no lugar de callback. O objetivo do exemplo é inserir em um elemento HTML específico um texto após o timeout de 3 segundos. No seguinte exemplo um arquivo HTML é carregado e depois ele é exibido, é possível fazer isso como callback ou com Promises.

<pre>&lt;p id="demo"&gt;&lt;/p&gt; &lt;script&gt; function myDisplayer(some) {   document.getElementById("demo").innerHTML = some; }  function getFile(myCallback) {   let req = new XMLHttpRequest();   req.open('GET', "mycar.html");   req.onload = function() {     if (req.status == 200) {       myCallback(this.responseText);     } else {       myCallback("Error: " + req.status);     }   }   req.send(); } getFile(myDisplayer); &lt;/script&gt;</pre>	<pre>&lt;p id="demo"&gt;&lt;/p&gt; &lt;script&gt; function myDisplayer(some) {   document.getElementById("demo").innerHTML = some; }  let myPromise = new Promise(function(myResolve, myReject) {   let req = new XMLHttpRequest();   req.open('GET', "mycar.html");   req.onload = function() {     if (req.status == 200) {       myResolve(req.response);     } else {       myReject("File not Found");     }   };   req.send(); });  myPromise.then(   function(value) {myDisplayer(value);},   function(error) {myDisplayer(error);} ); &lt;/script&gt;</pre>
--	---

## Async/Await

Async faz uma função retornar uma promessa. Await faz uma função esperar por uma promessa. As palavras-chave async e await, implementadas a partir do ES2017, são uma sintaxe que simplifica a programação assíncrona, facilitando o fluxo de escrita e leitura do código; assim é possível escrever código que funciona de forma assíncrona, porém é lido e estruturado de forma síncrona.

O `async/await` trabalha com o código baseado em Promises, porém esconde as promessas para que a leitura seja mais fluída e simples de entender. Definindo uma função como `async`, podemos utilizar a palavra-chave `await` antes de qualquer expressão que retorne uma promessa. Dessa forma, a execução da função externa (a função `async`) será pausada até que a Promise seja resolvida. A palavra-chave `await` recebe uma Promise e a transforma em um valor de retorno (ou lança uma exceção em caso de erro). Quando utilizamos `await`, o JavaScript vai aguardar até que a Promise finalize. Se for finalizada com sucesso (o termo utilizado é `fulfilled`), o valor obtido é retornado. Se a Promise for rejeitada (o termo utilizado é `rejected`), é retornado o erro lançado pela exceção.

```
<h1 id="demo"></h1>
<script>
const myPromise = new Promise(function(myResolve, myReject) {
  setTimeout(function(){ myResolve("I love You !!"); }, 3000);
});
myPromise.then(function(value) {
  document.getElementById("demo").innerHTML = value;
});
</script>
```

```
<h1 id="demo"></h1>
<script>
async function myDisplay() {
  let myPromise = new Promise(function(resolve) {
    setTimeout(function() {resolve("I love You !!");}, 3000);
  });
  document.getElementById("demo").innerHTML = await myPromise;
}
myDisplay();
</script>
```

OBS: Só é possível usar `await` em funções declaradas com a palavra-chave `async`. Os dois argumentos (`resolve` e `reject`) são predefinidos pelo JavaScript, mas muitas vezes não é necessário utilizar o `reject`.

## ARRAY ITERATION

### Filter()

O método `filter()` cria uma nova array com os elementos array que passam em um teste determinado por uma função callback. No seguinte exemplo temos uma array contendo idade de pessoas e objetivo é identificar quais são adultos (maior de 18 anos) então é utilizado o método `filter()` e passando uma função callback que vai ser o teste que será realizado em cada elemento da array.

```
const ages = [32, 33, 16, 40];
const result = ages.filter(checkAdult);

function checkAdult(age) {
  return age >= 18;
}
```

32,33,40

## Map()

O método `map()` cria uma nova array utilizando elementos de outra array realizando uma função em cada um. Esse método não altera array original. No seguinte exemplo uma array possui alguns valores numéricos e outra array é criada contendo esses valores multiplicados por 2.

```
const numbers1 = [45, 4, 9, 16, 25];
const numbers2 = numbers1.map(myFunction);
function myFunction(value) {
  return value * 2;
}
```

numbers2 90,8,18,32,50

## Reduce()

O método `reduce()` executa uma função em cada um dos elementos de uma array produzindo (reduzindo para) um único valor. Esse método não altera a array original os valores são reduzidos em uma nova array como no seguinte exemplo.

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce(myFunction);
function myFunction(total, value, index, array) {
  return total + value;
}
```

99

Também é possível adicionar um valor inicial.

```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce(myFunction, 100);
function myFunction(total, value) {
  return total + value;
}
```

199

## ARRAY REFERÊNCIA METHODS AND PROPERTIES

Nome	Descrição
<b><u>concat()</u></b>	Combina arrays e retorna uma array com as arrays combinadas
<b><u>constructor</u></b>	Retorna a função que criou o protótipo objeto array
<b><u>copyWithin()</u></b>	Copia um elemento array em uma posição específica dentro da array
<b><u>entries()</u></b>	Retorna um Object Iteration Array (chave/valor)
<b><u>every()</u></b>	Checa se todos os elementos em uma array passaram em um teste
<b><u>fill()</u></b>	Preenche todos os elementos em uma array com um valor estático
<b><u>filter()</u></b>	Cria uma nova array com cada elemento que passou em um teste
<b><u>find()</u></b>	Retorna o valor do primeiro elemento que passou em um teste
<b><u>findIndex()</u></b>	Retorna o index do primeiro elemento que passou em um teste
<b><u>forEach()</u></b>	Chama uma função para cada elemento em uma array

<b><u>from()</u></b>	Cria uma array a partir de um objeto
<b><u>includes()</u></b>	Verifica se uma array contém um elemento específico
<b><u>indexOf()</u></b>	Procura na array por um elemento e retornando a sua posição.
<b><u>isArray()</u></b>	Verifica se um objeto é um array
<b><u>join()</u></b>	Combina todos os elementos de um array em uma string
<b><u>keys()</u></b>	Retorna um Array Iteration Object contendo as chaves da array original
<b><u>lastIndexOf()</u></b>	Procura na array por um elemento, começando do final, e retornando a sua posição.
<b><u>length</u></b>	Especifica e retorna o numero de elemento de uma array
<b><u>map()</u></b>	Cria uma nova array contendo o resultado de uma função chamada em cada elemento de uma array
<b><u>pop()</u></b>	Remove o ultimo elemento de uma array e retorna esse elemento
<b><u>prototype</u></b>	Permite adicionar propriedades e métodos a um objeto Array
<b><u>push()</u></b>	Adiciona novos elementos no final de um array e retorna uma nova largura
<b><u>reduce()</u></b>	Reduz os valores de uma array em um único valor (left-to-right)
<b><u>reduceRight()</u></b>	Reduz os valores de uma array em um único valor (right-to-left)
<b><u>reverse()</u></b>	Inverte a ordem dos elementos de uma array
<b><u>shift()</u></b>	Remove o primeiro elemento de uma array, e retorna esse elemento
<b><u>slice()</u></b>	Seleciona uma parte de uma array e retorna uma nova array
<b><u>some()</u></b>	Verifica se algum elemento de uma array passou em um teste
<b><u>sort()</u></b>	Organiza os elementos de uma array
<b><u>splice()</u></b>	Adiciona/remove elementos de uma array
<b><u>toString()</u></b>	Converte uma array em uma string, e retorna o resultado
<b><u>unshift()</u></b>	Adiciona novos elementos para começarem em uma array, e retorna uma nova largura
<b><u>valueOf()</u></b>	Retorna o valor primitivo de uma array

## SPREAD OPERATOR

O operador spread (...) permite rapidamente criar uma copia de um objeto (ou array) ou de parte dele. No seguinte exemplo existem dois arrays com três valores cada, uma terceira array é criada contendo todos os valores dessas arrays combinados.

```
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const numbersCombined = [...numbersOne, ...numbersTwo];
document.write(numbersCombined);
```

1,2,3,4,5,6

No seguinte exemplo o spread operator é utilizado com objetos, criando um terceiro objeto contendo todas as propriedades e valores de outros dois. É possível observar que os valores de algumas propriedades podem ser alterados, e esses valores não vão ser alterados no objeto de referência (as propriedades que não corresponderam foram combinadas, mas a propriedade que correspondeu, color, foi substituída pelo último objeto que foi passado).



<pre>const myVehicle = {   brand: 'Ford',   model: 'Mustang',   color: 'red' } const updateMyVehicle = {   type: 'car',   year: 2021,   color: 'yellow' } const myUpdatedVehicle = {...myVehicle, ...updateMyVehicle}</pre>	<pre>myUpdatedVehicle = {   brand: 'Ford',   model: 'Mustang',   color: 'yellow',   type: 'car',   year: 2021, }</pre>
---	--

É possível ainda adicionar propriedades que não contém em nenhum outro objeto, somente adicionando-a após o sinal de vírgula.

<pre>const myUpdatedVehicle = {...myVehicle, fuel: 'gasoline', ...updateMyVehicle }</pre>	<pre>myUpdatedVehicle = {   brand: 'Ford',   model: 'Mustang',   color: 'yellow',   fuel: 'gasoline',   type: 'car',   year: 2021, }</pre>
---	--

## JAVASCRIPT ERRORS

Ao executar o código JavaScript, diferentes erros podem ocorrer. Erros podem ser erros de codificação feitos pelo programador, erros devido a entrada errada e outras coisas imprevisíveis.

### Try Catch

As instruções JavaScript try e catch vêm em pares. A instrução **try** permite que você defina um bloco de código para testar erros enquanto está sendo executado. A instrução **catch** permite definir um bloco de código a ser executado, caso ocorra um erro no bloco try.

```
try {
  Block of code to try
}
catch(err) {
  Block of code to handle errors
}
```

No seguinte exemplo, o método "alert" está digitado incorretamente como "adddlernt" para produzir deliberadamente um erro:

<pre>&lt;p id="demo"&gt;&lt;/p&gt; &lt;script&gt; try {   adddlernt("Welcome guest!"); } catch(err) {   document.getElementById("demo").innerHTML = err.message; } &lt;/script&gt;</pre>	<p>How to use <b>catch</b> to display an error.</p> <p>adddlernt is not defined</p>
--	---

### Throw

A instrução **throw** permite que criar um erro personalizado. Tecnicamente, você pode **throw(lançar) uma exceção (lançar um erro)**. A exceção pode ser um JavaScript String, um Number, um Boolean ou um Object.

No seguinte exemplo, é examinado um input, se o valor for errado, uma exceção (err) é lançada (throw an exception). A exceção é capturada pelo catch statement e uma mensagem de erro é exibida.

```
<body>
<p>Please input a number between 5 and 10:</p>
<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="p01"></p>
<script>
function myFunction() {
  const message = document.getElementById("p01");
  message.innerHTML = "";
  let x = document.getElementById("demo").value;
  try {
    if(x == "") throw "empty";
    if(isNaN(x)) throw "not a number";
    x = Number(x);
    if(x < 5) throw "too low";
    if(x > 10) throw "too high";
  }
  catch(err) {
    message.innerHTML = "Input is " + err;
  }
}
</script>
```

Please input a number between 5 and 10:

Input is too low

## Finally

A instrução **finally** executa o código após try e catch independente do resultado.

```
try {
  Block of code to try
}
catch(err) {
  Block of code to handle errors
}
finally {
  Block of code to be executed regardless of the try / catch result
}
```

No seguinte exemplo finally executa o código de limpar os valores dentro do input após o tratamento de erros pelo try e catch.

```
<p>Please input a number between 5 and 10:</p>
<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="p01"></p>
<script>
function myFunction() {
  const message = document.getElementById("p01");
  message.innerHTML = "";
  let x = document.getElementById("demo").value;
  try {
    if(x == "") throw "is empty";
    if(isNaN(x)) throw "is not a number";
    x = Number(x);
    if(x > 10) throw "is too high";
    if(x < 5) throw "is too low";
  }
  catch(err) {
    message.innerHTML = "Input " + err;
  }
  finally {
    document.getElementById("demo").value = "";
  }
}
</script>
```

## FETCH

Quando são utilizados arquivos ou dados externos de um servidor ou fornecidos por uma API, isso deve ser feito através de callback ou promises já que serão solicitações assíncronas. É possível observar nos exemplos de callback o uso da função XMLHttpRequest(). Antes de o JSON dominar o mundo, o formato principal de troca de dados era o XML. A XMLHttpRequest() é uma função do JavaScript que tornou possível obter dados das APIs que retornavam dados em XML. **Fetch API é uma versão mais simples** e mais fácil de usar da XMLHttpRequest para consumir recursos de modo assíncrono. Fetch permite que você trabalhe com as APIs REST com opções adicionais como cache de dados, leitura de respostas em streaming e mais.

No seguinte exemplo é obtido um arquivo txt e o texto desse arquivo é introduzido em um elemento <p>.

```
<p id="demo">Fetch a file to change this text.</p>
<script>
let file = "fetch_info.txt"
fetch (file)
.then(x => x.text())
.then(y => document.getElementById("demo").innerHTML = y);
</script>
</body>
</html>
```

**Fetch API**  
The Fetch API interface allows web browser to make HTTP requests to web servers.  
If you use the XMLHttpRequest Object, Fetch can do the same in a simpler way.

É possível ainda utilizar async/await para facilitar e melhorar o código.

```
<p id="demo">Fetch a file to change this text.</p>
<script>
getText("fetch_info.txt");
async function getText(file) {
  let x = await fetch(file);
  let y = await x.text();
  document.getElementById("demo").innerHTML = y;
}
</script>
```