JAVASCRIPT - iniciando

Javascript é uma linguagem de programação com foco voltado para aplicações web. Em HTML, o código JavaScript deve ser inserido entre as tags **<script>** e **</ script>**. Exemplos do antigo JavaScript podem usar um atributo do tipo: <script type = "text / javascript">.   
O atributo type não é mais necessário.

**Funções JavaScript e Eventos**

Uma **função** JavaScript é um bloco de código JavaScript, que pode ser executado quando "chamado" para, por exemplo, executar um **evento**, como quando o usuário clica em um botão.

**JavaScript no <head> ou <body>**

Você pode colocar qualquer número de scripts em um documento HTML. Os scripts podem ser colocados no <body>, ou na seção <head> de uma página HTML, ou em ambos. Colocar scripts na parte inferior do elemento <body> melhora a velocidade de exibição, porque a compilação de script retarda o display.

**JavaScript externo**

Scripts também podem ser colocados em arquivos externos: <script src="myScript.js"></script>

Scripts externos são práticos quando o mesmo código é utilizado em muitas páginas da web diferentes. Arquivos JavaScript tem a extensão de arquivo (**.js)** .

Você pode colocar uma referência de script externo em <head> ou <body> como você desejar. O script irá se comportar como se estivesse localizado exatamente onde a tag <script> está localizada. Scripts externos não podem conter tags <script>.

Vantagens JavaScript externo

Colocar scripts em arquivos externos tem algumas vantagens:

* Ele separa o código HTML
* Faz o HTML e o JavaScript mais fáceis de ler e manter
* Arquivos JavaScript em cache podem acelerar o carregamento da página

Para adicionar vários arquivos de script para uma página - use várias tags de script: <script src="myScript1.js"></script>  
<script src="myScript2.js"></script>

**Referências externas**

Scripts externos podem ser referenciado com uma URL completa ou com um caminho relativo para a página da web atual. Este exemplo usa uma URL completa para conectar-se a um script:

<script src="https://www.w3schools.com/js/myScript1.js"></script>

Este exemplo usa um script localizado em uma pasta especificada no web site atual: <script src="/js/myScript1.js"></script>

**JAVASCRIPT OUTPUT**

JavaScript consegue disponibilizar os dados de diferentes maneiras:

* Escrevendo em um elemento HTML, usando **innerHTML** .
* Escrevendo para a saída HTML usando **document.write()** .
* Escrevendo em uma caixa de alerta, usando **window.alert()** .
* Escrevendo para o console do navegador, usando **console.log()** .

**Usando innerHTML**

Para acessar um elemento HTML, JavaScript pode usar o **document.getElementById (id)** método. O **ID de** atributo define o elemento HTML. O **innerHTML** propriedade define o conteúdo HTML: <p id="demo"></p>  
<script>  
document.getElementById("demo").innerHTML = 5 + 6; //saída 11  
</script>

**Usando document.write ()**

Para fins de teste, é conveniente usar **document.write ().** <script>  
document.write(5 + 6);  
</script>

Usando **document.write ()** depois de que um documento HTML é totalmente carregado, apagará todo o HTML existente: <button onclick="document.write(5 + 6)">Try it</button>

**Usando window.alert ()**

Você pode usar uma caixa de alerta para exibir dados: <script>  
window.alert(5 + 6);  
</script>

**Usando console.log ()**

Para fins de depuração, você pode usar o **console.log ()** método para exibir dados: <script>  
console.log(5 + 6);  
</script>

**JAVASCRIPT SINTAXE**

**Sintaxe** JavaScript  é o conjunto de regras, de como programas JavaScript são construídos.

Programas JavaScript

Um **programa de computador** é uma lista de "instruções" para ser "executado" pelo computador. Em uma linguagem de programação, estas instruções de programa são chamados **declarações** . JavaScript é uma **linguagem de programação**. Instruções JavaScript são separados por **ponto e vírgula**(;): var x, y, z;  
x = 5;  
y = 6;  
z = x + y;

Declarações JavaScript

Instruções JavaScript são compostos de: valores, operadores, expressões, palavras-chave e comentários.

**Valores de JavaScript**

A sintaxe JavaScript define dois tipos de valores: valores fixos e valores variáveis.

* Valores fixos são chamados de **literais** .
* Valores de variáveis são chamadas **variáveis** .

Valores literais de JavaScript

As regras mais importantes para escrever valores fixos são:

* **Números** são escritos com ou sem casas decimais: 10.50| 10| 1001
* **Strings** são uma cadeia de textos, escritos entre aspas duplas ou simples: ”João” ‘João’

Valores variáveis de JavaScript

Em uma linguagem de programação, **as variáveis** são usadas para **armazenar** valores de dados.

* JavaScript usa a palavra-chave **var**  para **declarar** variáveis.
* Um **sinal de igual** é usado para **atribuir valores** a variáveis.

Neste exemplo, X é definido como uma variável. Em seguida, ao x é atribuído o valor 6: var x;  
x = 6;

É uma boa prática de programação declarar todas as variáveis ​​no início de um script. Em programas de computador, as variáveis ​​são muitas vezes declaradas sem um valor. O valor pode ser algo que tem de ser calculado, ou algo que será fornecido mais tarde, como a entrada do usuário. Uma variável declarada sem um valor terá o valor **indefinido**: var carName; Se você re-declarar uma variável JavaScript, não vai perder o seu valor.

O carName variável ainda terá o valor "Volvo" após a execução destas declarações: var carName = "Volvo";  
var carName;

**Operadores JavaScript**

JavaScript usa **operadores aritméticos** (+ - \* /) servem para **calcular** valores: (5 + 6) \* 10 JavaScript usa um **operador de atribuição** (=) para **atribuir** valores a variáveis: var x, y;  
x = 5;  
y = 6;

JavaScript Expressões

Uma expressão é uma combinação de valores, variáveis ​​e operadores, que calcula um valor. O cálculo é chamado de resultado. Por exemplo, 5 \* 10 resulta em 50: 5 \* 10 As expressões também pode conter valores de variáveis: x = 5; x \* 10 //50 Os valores podem ser de vários tipos, tais como números e strings: "John" + " " + "Doe"

JavaScript Palavras-chave

Em JavaScript as **palavras-chave** são usadas para identificar ações a serem executadas. A  palavra-chave **var** diz ao navegador para criar variáveis: var x, y;

JavaScript Comentários

Nem todas as instruções JavaScript são "executadas". Código após barras duplas **//** e entre **/\*** e **\*/** é tratado como um **comentário**, os comentários são ignorados, e não serão executados var x = 5;   // Será executado  
// var x = 6;  NAO será executado

Identificadores de JavaScript

Os identificadores são nomes. Em JavaScript, identificadores são usados ​​para nomear variáveis ​​ (e palavras-chave e funções, e labels). As regras para nomes legais são praticamente os mesmos na maioria das linguagens de programação.

Em JavaScript, o primeiro caractere deve ser uma letra ou um sublinhado (\_), ou um cifrão ($). Caracteres subsequentes podem ser letras, números, sublinhados ou cifrões. Os números não são permitidos como o primeiro caractere. Desta forma JavaScript pode facilmente distinguir identificadores de números.

JavaScript é Case Sensitive

JavaScript diferencia idenetificadores com letras **maiúsculas** e **minúsculas** .  As variáveis **lastName** e **lastname** , são duas variáveis diferentes. O JavaScript não interpreta **VAR** ou **Var** como a palavra-chave **var**.

JavaScript é Camel Case

Historicamente, os programadores têm usado diferentes maneiras de juntar várias palavras em um nome de variável:

* **hífens:** primeiro nome, último nome, mestre-card, inter-cidades. Hífens não são permitidos em JavaScript sendo reservados para subtrações.
* **sublinhado:** first\_name, last\_name, master\_card, inter\_city.

**Upper Camel Case (Pascal Case):** Nome, SobreNome, MasterCard, InterCity.

**Minúsculas Camel:** programadores JavaScript tendem a usar case camel que começa com uma letra minúscula:nome, sobreNome, masterCard, interCity.

**ESPECIFICAÇÕES EM JAVASCRIPT**

Em HTML, instruções JavaScript são "instruções" que são executadas pelo navegador da web.

A maioria dos programas de JavaScript podem conter muitas instruções JavaScript. As instruções são executadas, uma por uma, na mesma ordem em que estão escritas. Ponto e vírgula servem para separar instruções JavaScript. Quando separados por ponto e vírgula, várias instruções sobre uma linha são permitidos.

JavaScript ignora múltiplos espaços. Você pode adicionar espaço em branco ao seu script para torná-lo mais legível. Uma boa prática é colocar espaços em torno de operadores (= + - \* /).

Para melhor legibilidade, os programadores muitas vezes gostam de evitar linhas de código com mais de 80 caracteres. Se uma instrução JavaScript não se encaixa em uma linha, o melhor lugar para quebrá-la, é depois de um operador.

Instruções JavaScript podem ser agrupadas em blocos de código, dentro de chaves {...}. A finalidade dos blocos de código é definir declarações a serem executadas em conjunto. Um lugar que você vai encontrar declarações agrupadas em blocos, será em funções JavaScript.

**PALAVRAS RESERVADAS EM JAVASCRIPT**

Algumas da principais e mais usadas palavras-chave utilizadas em Javascript estão na tabela:

|  |  |
| --- | --- |
| **Palavra-chave** | **Descrição** |
| break | Termina a execução de um switch ou loop |
| continue | Sai de um loop e começa no topo |
| debugger | Interrompe a execução de um script, e chama(se possível) a função de depuração |
| do...while | Executa um bloco de instruções, e repete o bloco, enquanto uma condição for verdadeira |
| for | Marca um bloco de instruções a ser executado, desde que uma condição é verdadeira |
| function | Declara uma função |
| if...else | Marca um bloco de instruções a serem executadas, dependendo de uma condição |
| return | Sai de uma função |
| switch | Marca um bloco de instruções a serem executadas, dependendo de diferentes casos |
| try...catch | Implementa o tratamento de erros para um bloco de instruções |
| var | Declara um variável |

Agora todas as palavras reservadas utilizadas em Javascript:

|  |  |  |  |
| --- | --- | --- | --- |
| **abstract** | **arguments** | **await\*** | **boolean** |
| break | byte | case | catch |
| **char** | **class\*** | **const** | **continue** |
| debugger | default | delete | do |
| **double** | **else** | **enum\*** | **eval** |
| export\* | extends\* | false | final |
| **finally** | **float** | **for** | **function** |
| goto | if | implements | import\* |
| **in** | **instanceof** | **int** | **interface** |
| let\* | long | native | new |
| **null** | **package** | **private** | **protected** |
| public | return | short | static |
| **super\*** | **switch** | **synchronized** | **this** |
| throw | throws | transient | true |
| **try** | **typeof** | **var** | **void** |
| volatile | while | with | yield |

Palavras com **\*** são novas no **ECMAScript 5 e 6** (Nome original do Javascript).

JavaScript pode ser usado como a linguagem de programação em muitas aplicações. Você também deve evitar usar o nome dos objetos e propriedades HTML e Window:

|  |  |  |  |
| --- | --- | --- | --- |
| **alert** | **all** | **anchor** | **anchors** |
| area | assign | blur | button |
| **checkbox** | **clearInterval** | **clearTimeout** | **clientInformation** |
| close | closed | confirm | constructor |
| **crypto** | **decodeURI** | **decodeURIComponent** | **defaultStatus** |
| document | element | elements | embed |
| **embeds** | **encodeURI** | **encodeURIComponent** | **escape** |
| event | fileUpload | focus | form |
| **forms** | **frame** | **innerHeight** | **innerWidth** |
| layer | layers | link | location |
| **mimeTypes** | **navigate** | **navigator** | **frames** |
| frameRate | hidden | history | image |
| **images** | **offscreenBuffering** | **open** | **opener** |
| option | outerHeight | outerWidth | packages |
| **pageXOffset** | **pageYOffset** | **parent** | **parseFloat** |
| parseInt | password | pkcs11 | plugin |
| **prompt** | **propertyIsEnum** | **radio** | **reset** |
| screenX | screenY | scroll | secure |
| **select** | **self** | **setInterval** | **setTimeout** |
| status | submit | taint | text |
| **textarea** | **top** | **unescape** | **untaint** |
| window |  |  |  |

Além disso, você deve evitar o uso do nome de todos os manipuladores de eventos HTML.

|  |  |  |  |
| --- | --- | --- | --- |
| onblur | onclick | onfocus | onerror |
| onkeydown | onkeypress | onkeyup | onload |
| onmouseover | onmouseup | onmousedown | onsubmit |

**OPERADORES EM JAVASCRIPT**

Operadores aritméticos: São utilizados para se realizar operações matemáticas em Javascript.

|  |  |
| --- | --- |
| **Operador** | **Descrição** |
| + | Adição |
| - | Subtração |
| \* | Multiplicação |
| / | Divisão |
| % | Resto da divisão |
| ++ | Incremento |
| -- | Decremento |

\***O operador de + também serve para concatenar strings. Ao se concatenar números com strings o Javascript sempre retornará uma string:**

|  |  |
| --- | --- |
| txt1 = "John"; txt2 = "Doe"; txt3 = txt1 + " " + txt2;  //resultado será -> John Doe | x = 5 + 5; // 10 número y = "5" + 5; // 55 string z = "Hello" + 5; // Hello5 string |

Operadores de Atribuição: São utilizados para atribuir valores a variáveis ​​de JavaScript.

|  |  |  |
| --- | --- | --- |
| **Operador** | **Exemplo** | **Igual a** |
| = | x = y | x = y |
| += | x += y | x = x + y |
| -= | x -= y | x = x - y |
| \*= | x \*= y | x = x \* y |
| /= | x /= y | x = x / y |
| %= | x %= y | x = x % y |

Operadores de comparação:

|  |  |
| --- | --- |
| **Operador** | **Descrição** |
| == | igual a |
| === | valor igual e tipo igual |
| != | diferente |
| !== | valor ou tipo diferentes |
| > | maior que |
| < | menor que |
| >= | maior que ou igual a |
| <= | menor que ou igual a |
| ? | operador ternário |

Operadores lógicos:

|  |  |
| --- | --- |
| **Operador** | **Descrição** |
| && | e |
| || | ou |
| ! | não |

Operadores de tipo JavaScript:

|  |  |
| --- | --- |
| **Operador** | **Descrição** |
| typeof | Retorna o tipo da variável |
| instanceof | Retorna true se um object for uma instancia de um tipo de objeto |

**TIPOS DE DADOS EM JAVASCRIPT**

Variáveis JavaScript podem conter muitos **tipos de dados** : números, strings, objetos e mais. JavaScript tem tipos dinâmicos. Isto significa que a mesma variável pode ser usada para armazenar diferentes tipos de dados: var x;               // Agora x está indefinido  
var x = 5;           // Agora x é um número  
var x = "John";      // Agora x é uma string

**\*Para se colocar aspas duplas ou simples em um texto dentro de uma string inverte-se a declaração da variável:** var x = ‘João “comprou” um carro’ **ou** “João ‘comprou’ um carro”

Tipos de dados Booleanos

Booleans só pode ter dois valores: verdadeiro ou falso: var x = true; var y = false; Booleans são muitas vezes utilizados em testes de condicional.

Tipo de dados Arrays

Vetores JavaScript são escritos entre colchetes e o seus itens são separados por vírgulas: var cars = ["Saab", "Volvo", "BMW"]; Índices de vetores são baseados em zero, o que significa que o primeiro item é [0], o segundo é [1], e assim por diante.

Tipo de dados Objects

Objetos JavaScript são escritos com chaves. As propriedades do objeto são escritas com pares de **nome: valor** para as propriedades deste objeto e são separados por vírgulas: var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

Operador typeof

Você pode usar o JavaScript **typeof** operador para encontrar o tipo de uma variável JavaScript. O operador **typeof** retorna o tipo de uma variável ou uma expressão:

|  |  |
| --- | --- |
| typeof ""      // Retorna "string" typeof "John"   // Retorna "string" typeof "John Doe" // Retorna "string" | typeof 0      // Retorna "número" typeof 314    // Retorna "número" typeof 3.14   // Retorna "número" typeof (3 + 4) // Retorna "número" |

Tipo de dados Primitivo

Um valor de dados primitivo é um único valor de dados simples sem nenhuma propriedade e método adicionais. O operador **typeof** pode retornar um desses tipos primitivos:

* string // “João”
* number // 1.15
* boolean // true| false
* null
* undefined

O operador **typeof** pode retornar um dos dois tipos complexos:

* function // myFunc(){}
* object // {nome: “João”, idade:15} ou [1,2,3,4]\*

**\*O operador typeof retorna como um objeto os valores de um vetor por que vetores em Javascript são considerados objetos.**

**\*Em JavaScript, uma variável sem um valor, tem o valor indefinido . O typeof também retornará indefinido :**  var person;

**\*Qualquer variável pode ser esvaziada, definindo o valor para indefinido . O tipo também será indefinido .** varperson = undefined;

Valores vazios

Um valor vazio não tem nada a ver com indefinido. Uma variável de cadeia vazia tem tanto um valor como um tipo: var car = ""; // O valor é "", o typeof é uma "string"

Nulo

Em JavaScript nulo é "nada". É supostamentealgo que não existe. Infelizmente, em JavaScript, o tipo de dados nulo é um objeto. Você pode considerá-lo como um erro no JavaScript que typeof nulo é um objeto. Ele deve ser nulo. Você pode esvaziar um objeto, definindo-o como nulo: var person = null; // O valor é nulo, mas o tipo ainda é um object

Diferenças entre undefined e null

typeof undefined           // undefined  
typeof null                // object  
null === undefined         // false  
null == undefined          // true

**\*null é tratado como um objeto em Javascript e portanto não poderia ser igual ao tipo undefined como está documentado pelos desenvolvedores da linguagem. Isto éconsiderado um erro da linguagem Javascript.**

**FUNÇÕES**

Uma função JavaScript é um bloco de código projetado para executar uma tarefa específica. Uma função JavaScript é executado quando "algo" o invoca (chama). function myFunction(p1, p2) {  
    return p1 \* p2;   // A função retornará o produto de p1 and p2  
}

Sintaxe

Uma função JavaScript é definida com uma palavra-chave **function** , seguido por um **nome** , seguido de parênteses **()** . Nomes de funções podem conter letras, números, sublinhados e cifrões (mesmas regras para as variáveis).

Os parênteses podem incluir nomes de parâmetros separados por vírgulas:   
**( *parâmetro1, parâmetro2, ...* )**

O código a ser executado, pela função, é colocado dentro de chaves: **{}**

function nome(parâmetro1, parâmetro2, parâmetro3) {  
    código a ser executado  
}

* **Parâmetros** são os **nomes** listados na definição da função.
* A**rgumentos** são os verdadeiros **valores** recebidos pela função quando ela é chamada.

Dentro da função, os argumentos (parâmetros) comportam-se como variáveis ​​locais. Uma função é o mesmo que um procedimento ou uma sub-rotina, em outras linguagens.

Invocando uma função

O código dentro da função será executada quando "algo" **invoca** (chamada) a função:

* Quando ocorre um evento (quando um usuário clica em um botão)
* Quando ela é invocada (chamada) a partir do código JavaScript
* Automaticamente (auto invocada)

Retornando uma função

Quando JavaScript atinge uma **instrução de retorno** , a função irá parar a execução. Se a função foi chamad a partir de uma declaração, o script irá "voltar" para executar o código após a instrução de chamada. Funções muitas vezes calculam um **valor de retorno** . O valor de retorno é "devolvida" de volta para o "chamador": var x = myFunction(4, 3);  // Função é chamada, o retorno voltará em x  
  
function myFunction(a, b) {  
    return a \* b;         // Função retornará o produto de a e b  
}

Por que utilizar Funções?

Você pode reutilizar códigos: Definir o código uma vez e usá-lo muitas vezes. Você pode usar o mesmo código muitas vezes com argumentos diferentes, para produzir resultados diferentes.

O operador () invocando uma função

Acessar uma função sem () irá retornar a definição da função, em vez do resultado da função: function toCelsius(fahrenheit) {  
    return (5/9) \* (fahrenheit-32);  
}  
document.getElementById("demo").innerHTML = toCelsius; // retornará a definição da função.

function toCelsius(fahrenheit) {  
    return (5/9) \* (fahrenheit-32);  
}  
document.getElementById("demo").innerHTML = toCelsius(77); //retornará o resultado da função que neste caso é 77

Funções usadas como valores de variáveis

Funções podem ser utilizadas da mesma forma que você usa variáveis, em todos os tipos de fórmulas, atribuições e cálculos.

Em vez de usar uma variável para armazenar o valor de retorno de uma função: var x = toCelsius(77);  
var text = "The temperature is " + x + " Celsius";

Você pode usar a função diretamente, como um valor variável: var text = "The temperature is " + toCelsius(77) + " Celsius";

**OBJETOS EM JAVASCRIPT**

Na vida real, um carro é um **objeto** . Um carro tem **propriedades** como peso e cor, e **métodos** como início e parada:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Objeto**  https://www.w3schools.com/js/objectExplained.gif | **Propriedades** car.name = Fiat   car.model = 500   car.weight = 850 kg   car.color = branco | **Métodos** car.start ()   car.drive ()   car.brake ()   car.stop () |

Todos os carros têm as mesmas **propriedades** , mas os valores de propriedade diferem de carro para carro. Todos os carros têm os mesmos **métodos** , mas os métodos são realizadas em momentos diferentes. Você já aprendeu que as variáveis ​​JavaScript são recipientes para valores de dados. Este código atribui um **valor simples** (Fiat) a uma **variável** carro nomeado: var car = "Fiat";

Objetos são variáveis ​​também. Mas os objetos podem conter muitos valores. Este código atribui **muitos valores** (Fiat, 500, branco) a uma **variável** carro nomeado: var car = {type:"Fiat", model:"500", color:"white"};

Os valores são escritos com pares de **nome: valor** (nome e o valor separados por dois pontos). Objetos JavaScript são recipientes para **valores nomeados**.

Propriedades do objeto

Os pares de nome: valores (em objetos JavaScript) são chamados de **propriedades**.

|  |  |
| --- | --- |
| **Propriedade** | **Valor da propriedade** |
| primeiro nome | John |
| último nome | Doe |
| idade | 50 |
| cor dos olhos | azul |

Métodos de objeto

Os métodos são **ações** que podem ser executadas em objetos. Métodos são armazenados em propriedades como **definições de funções**.

|  |  |
| --- | --- |
| **Propriedade** | **Valor da propriedade** |
| primeiro nome | John |
| último nome | Doe |
| idade | 50 |
| cor dos olhos | azul |
| nome completo | function () {return this.firstName + "" + this.lastName;} |

objetos javascript são recipientes para valores nomeados chamados propriedades ou métodos.

Definição de objetos

Você define (cria) um objeto JavaScript com um objeto literal: var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

Espaços e quebras de linha não são importantes. Uma definição de objeto pode abranger várias linhas: var person = {  
    firstName:"John",  
    lastName:"Doe",  
    age:50,  
    eyeColor:"blue"  
};

Acessando propriedades de objetos

Você pode acessar as propriedades do objeto de duas maneiras: *objectName.propertyName* //person.firstName; ou *objectName["propertyName"]* //person[“firstName”];

Acessando Métodos de objeto

Você pode acessar um método de um objeto com a seguinte sintaxe: *objectName.methodName()* //name = person.fullName(); Se você acessar o **método** fullName , sem (), ele irá retornar a **definição da função**. Um método é realmente uma definição de função armazenado como um valor de propriedade.

**Não declare Strings, números e Booleans como objetos!**

Quando uma variável JavaScript é declarada com a palavra-chave "**new**", a variável é criada como um objeto: var x = new String();        // Declara x como um objeto String   
var y = new Number();        // Declara y como um objeto Number   
var z = new Boolean();       // Declara z como um objeto Boolean

Evite usar objetos String, Number e booleanos. Eles complicam o seu código e desaceleram a velocidade de execução.

**ESCOPO EM JAVASCRIPT**

Escopo é o conjunto de variáveis ​​que você tem acesso. Em JavaScript, objetos e funções também são variáveis. O escopo muda as funções por dentro.

Variáveis ​​ Locais do JavaScript

Variáveis declaradas dentro de uma função JavaScript, tornam-se **LOCAIS** para a função. As variáveis locais têm **escopo local** : Eles só podem ser acessados dentro da função. // aqui o código não pode usar carName  
function myFunction() {  
    var carName = "Volvo";  
    // aqui o código pode usar carName  
}

Como as variáveis ​​locais são reconhecidas apenas dentro de suas funções, variáveis ​​com o mesmo nome podem ser usadas em diferentes funções. As variáveis ​​locais são criadas quando uma função começa, e excluído quando a função é concluída.

Variáveis ​​Globais do JavaScript

Uma variável declarada fora de uma função, torna-se **GLOBAL** . Uma variável global tem **escopo global** : Todos os scripts e funções em uma página web podem acessá-lo.  var carName = " Volvo";  
// aqui o código pode usar carName  
function myFunction() { // aqui o código pode usar carName   
}

**Global automaticamente**

Se você atribuir um valor a uma variável que não tenha sido declarada, ela se tornará automaticamente uma  variável **GLOBAL**.

Este exemplo de código irá declarar uma variável global **carName** , mesmo que o valor seja atribuído dentro de uma função. myFunction();  
// aqui o código pode usar carName -> document.getElementById("demo").innerHTML = "I can display " + carName;  
function myFunction() {  
    carName = "Volvo";  
}

Não crie variáveis ​​globais a menos que você tenha pretensões com ela. Em "modo estrito" variáveis ​​globais automaticamente falharam.

Variáveis ​​globais em HTML

Com JavaScript, o escopo global é o ambiente completo JavaScript. Em HTML, o escopo global é o objeto **window**. Todas as variáveis ​​globais pertencem ao objeto **window**. var carName = "Volvo";  
// aqui o código pode usar window.carName -> document.getElementById("demo").innerHTML = "I can display " + window.carName;

Suas variáveis globais (ou funções) podem substituir variáveis window (ou funções).   
Qualquer função, incluindo o objeto window, pode substituir as variáveis e funções globais.

A vida útil de variáveis ​​JavaScript

O tempo de vida de uma variável JavaScript começa quando ela é declarada. As variáveis ​​locais são apagadas quando a função é concluída.

Em um navegador da web, as variáveis ​​globais são excluídos quando você fechar a janela do navegador (ou guia), mas permanece disponível para novas páginas carregadas na mesma janela. Argumentos de funções (parâmetros) funcionam como variáveis ​​locais dentro de funções.

**EVENTOS EM JAVASCRIPT**

Eventos HTML são **"coisas"** que acontecem a elementos HTML. Quando JavaScript é usado em páginas HTML, JavaScript pode **"reagir"** sobre esses eventos.

Eventos HTML

Um evento HTML pode ser algo que o navegador faz, ou algo que um usuário faz. Aqui estão alguns exemplos de eventos HTML:

* Uma página web HTML terminar de carregar
* Um campo de entrada (input) HTML foi alterado
* Um botão HTML foi clicado

Muitas vezes, quando os eventos acontecem, você pode querer fazer alguma coisa. JavaScript permite executar código quando são detectados eventos.

HTML permite atributos manipuladores de eventos, **com o código JavaScript** , para ser adicionado a elementos HTML.

Com aspas simples: <element event=**'*some JavaScript*'**> ou Com aspas duplas: <element event=**"*some JavaScript*"**>

No exemplo a seguir, um atributo onclick (com código), é adicionada a um elemento de botão: <button onclick="document.getElementById('demo').innerHTML = Date()">The time is?</button> // mostra a data e horário local

Neste exemplo, o código muda o conteúdo de seu próprio elemento (usando **this**.innerHTML): <button onclick="this.innerHTML = Date()">The time is?</button>

Códigos JavaScript possuem muitas vezes várias linhas. É mais comum ver os atributos de eventos chamando funções: <button onclick="displayDate()">The time is?</button>

Eventos HTML DOM permitem ao JavaScript registrar diferentes manipuladores de eventos em elementos de um documento HTML. Eventos são normalmente utilizados em combinação com funções, ea função não será executada antes de ocorrer o evento (por exemplo, quando um usuário clica em um botão). A lista com todos os eventos Javascript está localizada ao final:

**O JavaScript pode fazer?**

Os manipuladores de eventos podem ser usados ​​para lidar e verificar, a entrada do usuário, as ações do usuário, e as ações do navegador:

* Coisas que devem ser feitas cada vez que uma página é carregada
* Coisas que devem ser feitas quando a página é fechada
* Ação que deve ser realizada quando um usuário clica em um botão
* O conteúdo que deverá ser verificado como os dados do usuário por input. E mais ...

Muitos métodos diferentes podem ser usados ​​para deixar o trabalho JavaScript com eventos:

* atributos de eventos HTML podem executar código JavaScript diretamente
* atributos de eventos HTML pode chamar funções de JavaScript
* você pode atribuir suas próprias funções manipuladoras de eventos a elementos HTML
* Você pode impedir que eventos sejam enviados ou trata-los. E mais ...

**STRINGS EM JAVASCRIPT**

Strings JavaScript são usados ​​para armazenar e manipular texto. A solução para evitar este problema, é usar o **caracter escape (\)** .

O caráter de escape (barra invertida) transforma caracteres especiais em caracteres de string:

var x = 'It\'s alright';  
var y = "We are the so-called \"Vikings\" from the north."

O caractere de escape (\) também pode ser usado para inserir outros caracteres especiais em uma string. Estes caracteres especiais são comumente usados ​​e podem ser inseridos em um texto com o sinal de barra invertida:

|  |  |
| --- | --- |
| **Código** | **Saída** |
| \' | asplas simples |
| \" | aspas duplas |
| \\ | barra invertida |

Quebrar linhas de código longas

Para melhor legibilidade, os programadores muitas vezes gostaria de evitar linhas de código mais de 80 caracteres. Se uma instrução JavaScript não se encaixa em uma linha, o melhor lugar para quebrá-lo é depois de um operador. Você também pode acabar com uma linha de código **dentro de uma cadeia de texto** com uma única barra invertida: document.getElementById("demo").innerHTML = "Hello \  
Dolly!";

O \ método não é o método preferido. Pode não ter o apoio universal. Alguns navegadores não permitem espaços atrás do \ caractere. A maneira mais segura para quebrar uma string, é usar além string: document.getElementById("demo").innerHTML = "Hello" +   
"Dolly!";

Você não pode quebrar uma linha de código com uma barra invertida.

Strings podem ser objetos

Normalmente, strings JavaScript são valores primitivos, criados a partir de literais: **var firstName = "John"** // string

Mas strings também podem ser definidos como objetos com a palavra-chave new: **var firstName = new String ( "John")** //objeto

Não crie strings como objetos. Eles retardam a velocidade de execução.   
A palavra-chave **new** complica o código. Isso pode produzir alguns resultados inesperados:

* Ao usar o operador ==, strings iguais são iguais: var x = "John";               
  var y = new String("John");  
  // (x == y) é verdadeiro por que x e y tem valores iguais
* Ao usar o operador ===, strings iguais não são iguais, porque o operador === espera igualdade em ambos os tipo e valor: var x = "John";               
  var y = new String("John");  
  // (x === y) é falso por que x e y tem diferentes tipos (string e objeto)
* Ou ainda pior. Os objetos não podem ser comparados: var x = new String("John");               
  var y = new String("John");  
  // (x == y) é falso por que x e y são objetos diferentes // (x === y) é falso por que x e y são objetos diferentes

Note a diferença entre o (x == y) e (x === y). Comparar dois objetos JavaScript irá **sempre** retornar falso.

Strings Métodos e Propriedades

Valores primitivos, como "John Doe", não pode ter propriedades ou métodos (porque eles não são objetos). Mas com JavaScript, métodos e propriedades também estão disponíveis para valores primitivos, porque o JavaScript trata valores primitivos como objetos ao executar métodos e propriedades.

**Cadeia de comprimento**

A propriedade **lenghth** retorna o comprimento de uma string: var txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";  
var sln = txt.length; // 26

**Encontrar uma String em uma String**

O **indexOf()** retorna o índice (a posição) da **primeira** ocorrência de um texto especificado em uma cadeia de caracteres: var str = "Please locate where 'locate' occurs!";  
var pos = str.indexOf("locate"); // 7

O método **lastIndexOf()** retorna o índice da **última** ocorrência de um texto especificado em uma string: var str = "Please locate where 'locate' occurs!";  
var pos = str.lastIndexOf("locate"); // 21

Tanto os métodos indexOf (), e o lastIndexOf ()retornam -1 se o texto não for encontrado. JavaScript conta posições de zero. 0 representa a primeira posição em uma cadeia, um é o segundo, o terceiro é 2 ... Ambos os métodos aceitam um segundo parâmetro como a posição de partida para a pesquisa: var str = "Please locate where 'locate' occurs!";  
var pos = str.indexOf("locate",15); // 21

**Buscando uma String em uma String**

O método **search()** procura uma string para um valor especificado e retorna a posição da partida: var str = "Please locate where 'locate' occurs!";  
var pos = str.search("locate"); // 7

Reparou que?

Os dois métodos, indexOf() e search(), são iguais. Eles aceitam os mesmos argumentos (parâmetros), e eles retornam o mesmo valor. Os dois métodos são iguais, mas o método search() pode tomar valores de pesquisa muito mais poderosos.

Extraindo partes de strings

Existem 3 métodos para extrair uma parte de uma string:

* slice (início, fim)
* substring (início, fim)
* substr (início, comprimento)

**O Método slice()**

**slice()** extrai uma parte de uma cadeia e retorna a parte extraída em uma nova cadeia. O método toma 2 parâmetros: o índice inicial (localização), e o índice de terminação (posição). Este exemplo fatia para uma porção de uma cadeia de caracteres a partir da posição 7 para a posição 13: var str = "Apple, Banana, Kiwi";  
var res = str.slice(7, 13); // recorta banana e põe em uma nove string

Se um parâmetro é negativo, a posição é contado a partir do fim da cadeia. Este exemplo fatia para uma porção de uma cadeia de caracteres a partir da posição -12 a posicionar -6: var str = "Apple, Banana, Kiwi";  
var res = str.slice(-12, -6); // recorta banana e põe em nova string

Se você omitir o segundo parâmetro, o método irá cortar fora o resto da string: var res = str.slice(7); // retorna banana, kiwi

Ou, contando a partir da extremidade: var res = str.slice(-12); // banana, kiwi

**O Método substring()**

**substring()** é semelhante ao slice(). A diferença é que a substring () não pode aceitar índices negativos. Se você omitir o segundo parâmetro,o substring () irá cortar fora o resto da cadeia.

**O Método substr()**

**substr()** é semelhante ao slice(). A diferença é que o segundo parâmetro especifica o **comprimento** da parte extraída. var str = "Apple, Banana, Kiwi";  
var res = str.substr(7, 6); // retorna banana

Se o primeiro parâmetro for negativo, a posição conta do fim da cadeia. O segundo parâmetro não pode ser negativo, porque define o comprimento. Se você omitir o segundo parâmetro, substr() irá cortar fora o resto da cadeia.

**Substituindo cadeia de conteúdo**

O método **replace()** substitui um valor especificado com outro valor em uma string: str = "Please visit Microsoft!";  
var n = str.replace("Microsoft", "W3Schools"); // retorna Please visit W3Schools

O método replace() não altera a seqüência como é chamada. Ele retorna uma nova string. Por padrão, o replace() substitui **apenas o primeiro** elemento: str = "Please visit Microsoft and Microsoft!";  
var n = str.replace("Microsoft", "W3Schools"); //retorna Please visit W3schools and Microsoft

Para substituir todas os elemetos, use uma **expressão regular** com uma flag **/g**  (match global): str = "Please visit Microsoft and Microsoft!";  
var n = str.replace(/Microsoft/g, "W3Schools"); // retorna Please visit W3Schools and W3Schools

Por padrão, o replace() diferencia maiúsculas de minúsculas. Escrevendo MICROSOFT (com maiúscula) não vai funcionar. Para substituir maiúsculas e minúsculas, use uma **expressão regular** com uma flag **/i** (insensitive): str = "Please visit Microsoft!";  
var n = str.replace(/MICROSOFT/i, "W3Schools"); //retorna Please visit W3Schools

**Convertendo para maiúsculas e minúsculas**

Uma cadeia é convertida em letras maiúsculas com **toUpperCase ()** : var text1 = "Hello World!";       // String  
var text2 = text1.toUpperCase();  // text2 is text1 converted to upper

Uma cadeia é convertido para minúsculas com **toLowerCase ()** : var text1 = "Hello World!";       // String  
var text2 = text1.toLowerCase();  // text2 is text1 converted to lower

**O Método concat()**

**concat ()** junta duas ou mais cadeias. Pode ser usada para substituir o sinal de adição **( + )** var text1 = "Hello";  
var text2 = "World";  
var text3 = text1.concat(" ", text2); //retorna Hello World

Todos os métodos de strings retornam uma nova string. Eles não modificam a string original.   
Formalmente diz-se: Strings são imutáveis: Strings não podem ser alteradas, somente substituídas.

**Extraindo caracteres da cadeia**

Existem 2  métodos **seguros** para extrair caracteres de uma string:

* charAt (posição)
* charCodeAt (posição)

**O Método charAt()**

O **charAt()** retorna o carácter de um índice (posição) especificado numa cadeia: var str = "HELLO WORLD";  
str.charAt(0);            // retorna H

**O Método charCodeAt()**

O **charCodeAt()** retorna o Unicode do caractere em um índice especificado em uma cadeia de caracteres: var str = "HELLO WORLD";  
str.charCodeAt(0);         // returns 72

**Acessar uma string como uma matriz é inseguro**

Você pode ter visto um código como este, o acesso a uma string como um array: var str = "HELLO WORLD";  
str[0];                   // returns H

Isto é **inseguro** e **imprevisível:**

* Ele não funciona em todos os navegadores (não em IE5, IE6, IE7)
* Faz strings parecerem com matrizes (mas não são)
* str [0] = "H" não dá um erro (mas não funciona)

Se você quiser ler uma string como um array, converta-o para uma matriz em primeiro lugar.

Convertendo uma string para um array

Uma string pode ser convertida para uma matriz com o método **split()**: var txt = "a,b,c,d,e";   // String  
txt.split(",");          // Split em vírgulas  
txt.split(" ");          // Split em espaços  
txt.split("|");          // Split em barra vertical

Se o separador for omitido, a matriz retornada irá conter toda a string do índice [0]. Se o separador for " ", o array retornado será um array de caracteres individuais: var txt = "Hello";       // String  
txt.split("");           // Split em caracteres retornará Hello - uma letra em cada linha e uma embaixo da outra

**NÚMEROS EM JAVASCRIPT**

JavaScript tem apenas um tipo de número. Os números podem ser escritos com, ou sem, decimais.: var x = 34.00;    // Um número com decimais  
var y = 34;       // Um número sem decimais

Extra Pequeno números grandes ou extra pode ser escrito com notação científica (expoente): var x = 123e5;    // 12300000  
var y = 123e-5;   // 0.00123

Números JavaScript são sempre 64-bit de ponto flutuante

Ao contrário de muitas outras linguagens de programação, JavaScript não define tipos diferentes de números, como números inteiros, curto, longo, de ponto flutuante etc. Números JavaScript são sempre armazenados como números de ponto flutuante de precisão dupla, seguindo o padrão internacional IEEE 754.   
Este formato armazena os números em 64 bits, onde o número (a fração) é armazenado em bits 0-51, o expoente em pedaços de 52 a 62, e o sinal em bits 63.

|  |  |  |
| --- | --- | --- |
| **Valor (Fração / Mantissa)** | **Expoente** | **sinal** |
| 52 bits (0 - 51) | 11 bits (52 - 62) | 1 bit (63) |

Precisão

Inteiros (números sem um período ou notação expoente) são consideradas precisas até 15 dígitos. var x = 999999999999999;   // x will be 999999999999999  
var y = 9999999999999999;  // y will be 10000000000000000

O número máximo de casas decimais é de 17, mas a aritmética de ponto flutuante não é sempre 100% exata: var x = 0.2 + 0.1; // x will be 0.30000000000000004

Isto ajuda na horaa de multiplicar e dividir: var x = (0.2 \* 10 + 0.1 \* 10) / 10;       // x será 0.3

Hexadecimal

JavaScript interpreta constantes numéricas como hexadecimais se eles são precedidos por 0x: var x = 0xFF;             // x will be 255

Nunca escreva um número com um zero à esquerda (como 07).   
Algumas versões JavaScript interpretam números como octal se eles são escritos com um zero à esquerda.

Por padrão, o JavaScript exibe números como base de 10 casas decimais.

Mas você pode usar o método toString () para números de saída como base 16 (hex), base 8 (octal), ou base 2 (binário): var myNumber = 128;  
myNumber.toString(16);     // retorna 80  
myNumber.toString(8);      // retorna 200  
myNumber.toString(2);      // retorna 10000000

Infinito

Infinito (ou -Infinity) é o valor JavaScript retornado se você calcular um número fora do maior número possível: var myNumber = 2;  
while (myNumber != Infinity) {          // Executa até o infinito  
    myNumber = myNumber \* myNumber;  
}

A divisão por 0 (zero) também gera Infinito: var x =  2 / 0; // x será infinito  
var y = -2 / 0;  // y será –Infinity

O infinito é um número: typeof infinito retorna números: typeof Infinity; // retorna "número"

NaN - Not a Number

NaN é uma palavra reservada JavaScript indicando que um número não é um número legal. Tentando fazer aritmética com uma string não numérica resultará em NaN (não um número): var x = 100 / "Apple";  // x será NaN (Not a Number)

No entanto, se a string conter um valor numérico, o resultado será um número: var x = 100 / "10";     // x will be 10

Você pode usar a função global isNaN JavaScript() para descobrir se um valor é um número: var x = 100 / "Apple";  
isNaN(x);               // retorna true por que x é Not a Number()NaN

Cuidado, se você usar NaN em uma operação matemática, o resultado também será NaN: var x = NaN;  
var y = 5;  
var z = x + y;         // z será NaN

 Ou o resultado pode ser uma concatenação: var x = NaN;  
var y = "5";  
var z = x + y;         // z será NaN5

NaN é um número, e typeof NaN retorna números: typeof NaN; // retorna "número"

Números podem ser objetos

Normalmente números JavaScript são valores primitivos criados a partir de literais: **var x = 123**. Mas os números podem também ser definidos como objetos com a palavra-chave new: **var y = new Número(123).**

var x = 123;  
var y = new Number(123);  
// typeof x retorna número  
// typeof y retorna objeto

Não crie objetos Number. Ele retarda a velocidade de execução.   
A  palavra-chave **new** complica o código. Isso pode produzir alguns resultados inesperados:

Quando se utiliza o operador ==, os números iguais são iguais: var x = 500;               
var y = new Number(500);  
// (x == y) é true por que x e y tem valores iguais

Ao usar o operador ===, números iguais não são iguais, porque o operador === espera igualdade em ambos os tipo e valor: var x = 500;               
var y = new Number(500);  
// (x === y) é falso por que x e y tem tipos diferentes

Ou ainda pior. Os objetos não podem ser comparados: var x = new Number(500);               
var y = new Number(500);  
// (x == y) é falso por que objetos não podem ser comparados

Note a diferença entre o (x == y) e (x === y).   
Comparar dois objetos JavaScript irá sempre retornar falso.

Números Métodos e Propriedades

Os valores primitivos (como 3,14 ou 2,014), não podem ter propriedades e métodos (porque eles não são objetos). Mas com JavaScript, métodos e propriedades também estão disponíveis para valores primitivos, porque o JavaScript trata valores primitivos como objetos ao executar métodos e propriedades.

**O Método toString()**

**toString()** retorna um número como uma string. Todos os métodos numéricos podem ser utilizados em qualquer tipo de números (literais, variáveis, ou expressões): var x = 123;  
x.toString();            // retorna 123 da variável x  
(123).toString();        // retorna 123 do literal 123  
(100 + 23).toString();   // retorna 123 da expressão 100 + 23

**O Método toExponential()**

**toExponential()** retorna uma cadeia de caracteres, com um número arredondado e escrito usando a notação exponencial. Um parâmetro define o número de caracteres por trás do ponto decimal: var x = 9.656;  
x.toExponential(2);     // retorna 9.66e+0  
x.toExponential(4);     // retorna 9.6560e+0  
x.toExponential(6);     // retorna 9.656000e+0 O parâmetro(?) é opcional. Se você não especificá-lo, o script não vai arredondar o número.

**O Método toFixed()**

**toFixed()** retorna uma string, com o número escrito com um número especificado de casas decimais: var x = 9.656;  
x.toFixed(0);           // retorna 10  
x.toFixed(2);           // retorna 9.66  
x.toFixed(4);           // retorna 9.6560  
x.toFixed(6);           // retorna 9.656000 toFixed(2) é perfeito para se trabalhar com valores monetários.

**O Método toPrecision()**

**toPrecision()** retorna uma cadeia de caracteres, com um número de escrita com um comprimento especificado: var x = 9.656;  
x.toPrecision();        // retorna 9.656  
x.toPrecision(2);       // retorna 9.7  
x.toPrecision(4);       // retorna 9.656  
x.toPrecision(6);       // retorna 9.65600

**O Método valueOf()**

**valueOf()** retorna um número como um número: var x = 123;  
x.valueOf();            // retorna 123 da variável x  
(123).valueOf();        // retorna 123 do literal 123  
(100 + 23).valueOf();   // retorna 123 da expressão 100 + 23

Em JavaScript, um número pode ser um valor primitivo (typeof = number) ou um objeto (typeof = object). O método **valueOf()** é usado internamente no JavaScript para converter objectos numéricos para valores primitivos. Não há nenhuma razão para usá-lo em seu código. Todos os tipos de dados de JavaScript têm um método toString() valueOf().

Convertendo Variáveis ​​para Números

Existem 3 métodos JavaScript que podem ser usados ​​para converter as variáveis ​​para os números:

* O método Number()
* O método parselnt()
* O método parseFloat()

Estes métodos não são métodos **númericos** , mas métodos **globais**  JavaScriMétodos globais

Métodos globais de JavaScript pode ser usado em todos os tipos de dados de JavaScript. Estes são os métodos mais relevantes, ao trabalhar com números:

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| Number() | Retorna um número, convertido deum argumento |
| parseFloat() | Analisa um argumento e retorna um número de ponto flutuante |
| parseInt() | Analisa um argumento e retorna um inteiro |

**O Método Number()**

**Number()** pode ser usado para converter variáveis JavaScript para números: x = true;  
Number(x);        // retorna 1

x = false;       
Number(x);        // retorna 0

x = new Date();  
Number(x);        // retorna 1404568027739

x = "10"  
Number(x);        // retorna 10

x = "10 20"  
Number(x);        // retorna NaN

Usado em **Date()**, o método **Number()** retorna o número de milissegundos desde 1.1.1970.

**O Método parselnt()**

**parseInt ()** analisa uma string e retorna um número inteiro. Espaços são permitidos. Apenas o primeiro número é retornado: parseInt("10");         // retorna 10  
parseInt("10.33");      // retorna 10  
parseInt("10 20 30");   // retorna 10  
parseInt("10 years");   // retorna 10  
parseInt("years 10");   // retorna NaN

**O Método parseFloat()**

**parseFloat()** analisa uma string e retorna um número. Espaços são permitidos. Apenas o primeiro número é retornado: parseFloat("10");        // retorna 10  
parseFloat("10.33");     // retorna 10.33  
parseFloat("10 20 30");  // retorna 10  
parseFloat("10 years");  // retorna 10  
parseFloat("years 10");  // retorna NaN

**Propriedades de números**

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| MAX\_VALUE | Retorna o maior número possível em JavaScript (1.7976931348623157e+308) |
| MIN\_VALUE | Retorna o menor número possível em JavaScript  (5e-324) |
| NEGATIVE\_INFINITY | Representa o infinito negativo |
| NaN | representa um valor "Not a Number" |
| POSITIVE\_INFINITY | Representa o infinito |

Propriedades Number pertencem ao invólucro do objeto JavaScript chamado **Number** . Essas propriedades só podem ser acessadas como **Number** .MAX\_VALUE. Usando *myNumber* .MAX\_VALUE, onde *myNumber* é uma variável, expressão, ou valor, irá retornar indefinido: var x = 6;  
var y = x.MAX\_VALUE;    // y será undefined

**OBJETO MATH**

O objeto JavaScript **Math** permite que você execute tarefas matemáticas em números. Math.PI;            // returns 3.141592653589793

**Math.round()**

Math.round (x) retorna o valor de x arredondado para o seu valor inteiro mais próximo: Math.round(4.7);    // retorna 5  
Math.round(4.4);    // retorna 4

**Math.pow()**

Math.pow(x, y) retorna o valor de x para o potência de y: Math.pow(8, 2);//retorna 64

**Math.sqrt()**

Math.sqrt(x) retorna a raiz quadrada de x: Math.sqrt(64); //retorna 8

**Math.abs()**

Math.abs(x) retorna o valor absoluto (positivo) de x: Math.abs(-4.7); //retorna 4.7

**Math.ceil()**

Math.ceil (x) retorna o valor de x arredondado **para cima** para o seu valor inteiro mais próximo: Math.floor(4.7); // retorna 4

**Math.Sin()**

Math.Sin (x) retorna o seno (um valor entre -1 e 1) do ângulo x (dado em radianos). Se você quiser usar graus, em vez de radianos, você tem que converter graus em radianos: ângulo em radianos = ângulo em graus x PI / 180. Math.sin(90 \* Math.PI / 180); //retorna 1 (do seno de 90 graus)

**Math.cos()**

Math.cos (x) retorna o coseno (um valor entre -1 e 1) do ângulo x (dada em radianos). Se você quiser usar graus, em vez de radianos, você tem que converter graus em radianos: ângulo em radianos = ângulo em graus x PI / 180. Math.cos(0 \* Math.PI / 180); // retorna 1 (do coseno de 0 graus)

**Math.min() e Math.max()**

Math.min() e Math.max() pode ser usado para encontrar o valor mais baixo ou mais alto em uma lista de argumentos: Math.max(0, 150, 30, 20, -8, -200);  // retorna 150 Math.min(0, 150, 30, 20, -8, -200);  // retorna -200

**Math.random()**

Math.random() retorna um número aleatório entre 0 (inclusive), e 1 (exclusive): Math.random();     // retorna um número aleatório

Propriedades de matemática (Constantes)

JavaScript fornece 8 constantes matemáticas que podem ser acessados ​​com o objeto Math:

Math.E        // retorna números Euler's   
Math.PI       // retorna PI  
Math.SQRT2    // retorna a raíz quadrada de 2  
Math.SQRT1\_2  // retorna a raíz quadrada de 1/2  
Math.LN2      // retorna o logaritimo natural de 2  
Math.LN10     // retorna o logaritimo natural de 10  
Math.LOG2E    // retorna base 2 do logaritimo E  
Math.LOG10E   // retorna base 10 do logaritimo E

Math Construtor

Ao contrário de outros objetos globais, o objeto Math não tem nenhum construtor. Métodos e propriedades são estáticos. Todos os métodos e as propriedades (constantes) podem ser utilizadas, sem criar um objeto Math primeiro.

**Métodos do Objeto Math**

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| abs(x) | Retorna o valor absoluto de x |
| acos(x) | Retorna a arccosina de x, em radianos |
| asin(x) | Retorna o arcsine de x, em radianos |
| atan(x) | Retorna o arctangente de x como um valor numérico entre -PI / 2 e PI / 2 radianos |
| atan2(y, x) | Retorna o arctangente do quociente de seus argumentos |
| ceil(x) | Retorna o valor de x arredondado para o inteiro mais próximo |
| cos(x) | Retorna o coseno de x (x está em radianos) |
| exp(x) | Retorna o valor de Ex |
| floor(x) | Retorna o valor de x arredondado para o inteiro mais próximo |
| log(x) | Retorna o logaritmo natural (base E) de x |
| max(x, y, z, ..., n) | Retorna o número com o valor mais alto |
| min(x, y, z, ..., n) | Retorna o número com o valor mais baixo |
| pow(x, y) | Retorna o valor de x para a potência de y |
| random() | Retorna um número aleatório entre 0 e 1 |
| round(x) | Retorna o valor de x arredondado para o inteiro mais próximo |
| sin(x) | Retorna o seno de x (x está em radianos) |
| sqrt(x) | Retorna a raiz quadrada de x |
| tan(x) | Retorna a tangente de um ângulo |

Math Random()

Math.random() retorna um número aleatório entre 0 (inclusive), e 1 (exclusive): Math.random();              // retorna um número aleatório

Math.random() retorna sempre um número menor do que 1.

Random Inteiros

Math.random() usado com Math.floor() pode ser usado para retornar inteiros aleatórios. Math.floor(Math.random() \* 10);   // retorna um número entre 0 e 9 Math.floor(Math.random() \* 11);   // retorna um número entre 0 e 10 Math.floor(Math.random() \* 100);  // retorna um número entre 0 e 99 Math.floor(Math.random() \* 101);  // retorna um número entre 0 e 100 Math.floor(Math.random() \* 10) + 1;//retorna um número entre 1 e 10 Math.floor(Math.random() \* 100) + 1;//retorna um número entre 1 e 100

A Função aleatória adequada

Como você pode ver nos exemplos acima, pode ser uma boa idéia criar uma função aleatória própria de se usar em todos os efeitos aleatórios inteiros.Esta função JavaScript sempre retorna um número aleatório entre min (incluído) e máximo (excluído): function getRndInteger(min, max) {  
    return Math.floor(Math.random() \* (max - min) ) + min;  
}

Esta função JavaScript sempre retorna um número aleatório entre os valores mínimo e máximo (ambos incluídos): function getRndInteger(min, max) {  
    return Math.floor(Math.random() \* (max - min + 1) ) + min;  
}

**OBJETO DATE()**

O objeto Date permite trabalhar com datas (anos, meses, dias, horas, minutos, segundos e milissegundos). Objetos Date são estáticos. O tempo do computador está correndo, mas os objetos data, uma vez criado, não estão.

Formatos de datas em JavaScript

As datas em JavaScript podem ser escritas como uma string: **Sáb 17 junho, 2017 20:21:23 GMT-0300 (Hora oficial do Brasil)**

Ou como um número: **1497741683276**

Datas escritas como números, especificam o número de milissegundos desde 1 de janeiro de 1970, 00:00:00.

Exibir datas

Neste tutorial vamos usar um script para exibir datas dentro de um elemento <p> com id = "demo": <p id="demo"></p>  
<script>  
document.getElementById("demo").innerHTML = Date();  
</script> O script acima atribui o valor de Date() com o conteúdo (innerHTML) do elemento com id = "demo". O objeto Date permite-nos trabalhar com datas.

A data é composta de um ano, um mês, um dia, uma hora, um minuto, um segundo e milissegundos. Objetos Date são criados com o construtor **new Date()**. Há **4 maneiras** de iniciar uma data: new Date()  
new Date(milliseconds)  
new Date(dateString)  
new Date(year, month, day, hours, minutes, seconds, milliseconds)

Usando new Date (), cria um novo objeto de data com a **data e hora atuais** : <script>  
var d = new Date();  
document.getElementById("demo").innerHTML = d;  
</script> //Mon Jun 19 2017 11:35:44 GMT-0300 (Hora oficial do Brasil)

Usando new Date ( **string de data** ), cria um novo objeto a partir da **data e hora especificadas** : <script>  
var d = new Date("October 13, 2014 11:13:00");  
document.getElementById("demo").innerHTML = d;  
</script> //Mon Oct 13 2014 11:13:00 GMT-0200 (Horário brasileiro de verão)

Usando new Date ( **número** ), cria um novo objeto data como **tempo zero mais o número** . Tempo zero é 01 de janeiro de 1970 00:00:00 UTC. O número é especificado em milissegundos: <script>  
var d = new Date(86400000);  
document.getElementById("demo").innerHTML = d;  
</script> Datas em JavaScript são calculadas em milissegundos de 01 de janeiro de 1970 00:00:00 Hora Universal (UTC). Um dia contém 86.400.000 milissegundos.

Usando new Date ( **7 números** ), cria um novo objeto de data com a **data e hora especificadas** : Os 7 números especificam o ano, mês, dia, hora, minuto, segundo e milissegundo, nessa ordem: <script>  
var d = new Date(99, 5, 24, 11, 33, 30, 0);  
document.getElementById("demo").innerHTML = d;  
</script> JavaScript conta meses de 0 a 11. janeiro é 0. dezembro é 11.

Métodos Date()

Quando você exibir um objeto de data em HTML, ele é automaticamente convertido para uma cadeia, com o  método **toString ()**. <p id="demo"></p>  
<script>  
d = new Date();  
document.getElementById("demo").innerHTML = d;  
</script>

É o mesmo que: <p id="demo"></p>  
<script>  
d = new Date();  
document.getElementById("demo").innerHTML = d.toString();  
</script>

O método **toUTCString()** converte uma data para uma string UTC (padrão de exibição de data). <script>  
var d = new Date();  
document.getElementById("demo").innerHTML = d.toUTCString();  
</script> //Mon, 19 Jun 2017 14:49:20 GMT

O método **toDateString()** converte uma data para um formato mais legível: <script>  
var d = new Date();  
document.getElementById("demo").innerHTML = d.toDateString();  
</script> //Mon Jun 19 2017

Fusos horários

Ao definir uma data, sem especificar o fuso horário, JavaScript usará o fuso horário do browser. Ao obter uma data, sem especificar o fuso horário, o resultado é convertido para o fuso horário do browser. Em outras palavras: Se a data / hora for criada em GMT (Greenwich Mean Time), a data / hora será convertida para CDT (Central US Daylight Time) se um usuário navegar a partir do centro dos EUA.

JavaScript formatos de data

**JavaScript Date Input**

Em geral, há 4 tipos de formatos de entrada de datas em JavaScript:

|  |  |
| --- | --- |
| **Tipo** | **Exemplo** |
| Data ISO | "2015/03/25" (The International Standard) |
| Data abreviada | "2015/03/25" |
| Data alongada | "Março 25, 2015" ou "Março 25, 2015" |
| Data completa | "Qua 25 março de 2015" |

O formato ISO segue um padrão estrito no JavaScript. Os outros formatos não são tão bem definidos e pode ter específicações do navegador.

**JavaScript Date Output**

Independente do formato de entrada, JavaScript (por padrão) tem a saída de datas em formato de seqüência de texto completo: Tue Mar 24 2015 21:00:00 GMT-0300 (Hora oficial do Brasil)

**Datas JavaScript ISO**

ISO 8601 é o padrão internacional para a representação de datas e horários. A sintaxe ISO 8601 (AAAA-MM-DD) é também o formato de data JavaScript preferido: var d = new Date("2015-03-25"); A data calculada será em relação ao seu fuso horário.   
Dependendo do seu fuso horário, o resultado acima irá variar entre 24 e 25 de Março.

Datas ISO (ano e mês)

Datas ISO podem ser escritas sem especificar o dia (AAAA-MM): var d = new Date("2015-03"); Fusos horários irão variar o resultado acima entre 28 de Fevereiro e 01 de Março.

Datas ISO (somente no ano)

Datas ISO podem ser escrita sem o mês e o dia (AAAA): var d = new Date("2015"); Fusos horários irão variar este resultado de 31 de Dezembro de 2014 e 01 de janeiro de 2015.

Datas ISO (Date-Time)

Datas ISO pode ser escrita com horas adicionadas, minutos e segundos (AAAA-MM-DDThh: mm: SSZ): var d = new Date("2015-03-25T12:00:00Z");

Data e hora é separado com um T. Hora UTC é definido com uma letra maiúscula Z. Se você quiser modificar o tempo em relação ao UTC, retire a Z e adicione + HH: MM ou -HH: MM em vez disso: var d = new Date("2015-03-25T12:00:00-06:30");

UTC (Universal Time Coordinated) é o mesmo que GMT (Greenwich Mean Time). Omitindo T ou Z em uma seqüência de data e hora pode-se obter um resultado diferente em navegadores diferentes.

**JavaScript Short Dates**

Datas curtas são escritas com um "MM DD AAAA: var d = new Date("03/25/2015"); Em alguns navegadores, meses ou dias sem zeros à esquerda podem produzir um erro: var d = new Date("2015-3-25"); O comportamento de "AAAA / MM / DD" é indefinido. Alguns navegadores irá tentar adivinhar o formato. Alguns retornará NaN: var d = new Date("2015/03/25"); O comportamento do "DD-MM-AAAA" também é indefinido.   
Alguns navegadores irá tentar adivinhar o formato. Alguns retornará NaN: var d = new Date("25-03-2015");

**JavaScript Long Dates**

Datas longas são mais frequentemente escritas com uma sintaxe "MMM DD AAAA" como esta: var d = new Date("Mar 25 2015");Mês e dia podm ser em qualquer ordem: var d = new Date("25 Mar 2015"); E, mês podem ser escritos por extenso (Janeiro), ou abreviado (Jan): var d = new Date("January 25 2015"); ou var d = new Date("Jan 25 2015");Vírgulas são ignoradas. Nomes podem ser em maiúsculas ou em minúsculas: var d = new Date("JANUARY, 25, 2015");

**JavaScript Full Date**

JavaScript aceitará textos de data no "formato completo": var d = new Date("Wed Mar 25 2015 09:56:24 GMT+0100 (W. Europe Standard Time)"); JavaScript irá ignorar erros tanto no nome do dia e no tempo entre parênteses: var d = new Date("Fri Mar 25 2015 09:56:24 GMT+0100 (Tokyo Time)");

**Métodos Date**

Métodos de data permite-lhe obter e definir valores de data (anos, meses, dias, horas, minutos, segundos, milissegundos).

Métodos para obter datas(GET)

Métodos **GET** são usados ​​para obter uma parte de uma data. Aqui estão os mais comuns.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| getDate() | Obtenha o dia como um número (1-31) |
| getDay() | Obtenha o dia da semana como um número (0-6) |
| getFullYear() | Obtenha o ano de quatro dígitos (aaaa) |
| getHours() | Obtenha a hora (0-23) |
| getMilliseconds() | Obtenha os milissegundos (0-999) |
| getMinutes() | Obtenha os minutos (0-59) |
| getMonth() | Obtenha o mês (0-11) |
| getSeconds() | Obtenha os segundos (0-59) |
| getTime() | Obtenha o tempo (milissegundos desde 1 de janeiro de 1970) |

**O Método getTime()**

O **getTime()** retorna o número de milissegundos desde 01 de janeiro de 1970: <script>  
var d = new Date();  
document.getElementById("demo").innerHTML = d.getTime();  
</script>

**O Método getFullYear()**

O **getFullYear()** retorna o ano de uma data como um número de quatro dígitos: <script>  
var d = new Date();  
document.getElementById("demo").innerHTML = d.getFullYear();  
</script>

**O Método getDay()**

O **getDay()** retorna a semana como um número (0-6): <script>  
var d = new Date();  
document.getElementById("demo").innerHTML = d.getDay();  
</script> // 1 Em JavaScript, o primeiro dia da semana (0) significa "Sunday - Domingo", mesmo que alguns países do mundo consideram o primeiro dia da semana para ser "Monday - Segunda". Pode-se usar uma vetor de nomes, e getDay() para retornar o dia da semana como um nome: <script>  
var d = new Date();  
var days=["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"];  
document.getElementById("demo").innerHTML = days[d.getDay()];  
</script> //Monday

Métodos para definir datas(SET)

Os métodos set são usados ​​para definir uma parte de uma data. Aqui estão as mais comuns:

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| setDate() | Define o dia como um número (1-31) |
| setFullYear() | Define o ano (opcionalmente mês e dia) |
| setHours() | Define a hora (0-23) |
| setMilliseconds() | Define os milissegundos (0-999) |
| setMinutes() | Define os minutos (0-59) |
| setMonth() | Define o mês (0-11) |
| setSeconds() | Define os segundos (0-59) |
| setTime() | Define o tempo (milissegundos -1 de janeiro de 1970) |

**O Método setFullYear()**

**setFullYear()** define um objeto de data para uma data específica. Neste exemplo, a 14 de Janeiro, 2020: <script>  
var d = new Date();  
d.setFullYear(2020, 0, 14);  
document.getElementById("demo").innerHTML = d;  
</script> // Tue Jan 14 2020 14:23:01 GMT-0200 (Horário brasileiro de verão)

**O Método setDate()**

**setDate()** define o dia do mês (1-31): <script>  
var d = new Date();  
d.setDate(20);  
document.getElementById("demo").innerHTML = d;  
</script> // Thu Jun 15 2017 14:22:53 GMT-0300 (Hora oficial do Brasil) O método setDate () também pode ser usado para **adicionar dias** a uma data: <script>  
var d = new Date();  
d.setDate(d.getDate() + 50);  
document.getElementById("demo").innerHTML = d;  
</script> //Tue Aug 08 2017 14:28:49 GMT-0300 (Hora oficial do Brasil) Se a adição de dias, muda o mês ou ano, as alterações são feitas automaticamente pelo objeto Date.

**Data de Input - Parsing Dates**

Se você tem uma seqüência de data válida, você pode usar o **Date.parse()** método para convertê-lo em milissegundos. **Date.parse()** retorna o número de milissegundos entre a data e 01 de janeiro de 1970: <script>  
var msec = Date.parse("March 21, 2012");  
document.getElementById("demo").innerHTML = msec;  
</script> // 1332298800000 Então você pode usar o número de milissegundos para **convertê-lo para uma** objeto **data**: <script>  
var msec = Date.parse("March 21, 2012");  
var d = new Date(msec);  
document.getElementById("demo").innerHTML = d;  
</script>// Wed Mar 21 2012 00:00:00 GMT-0300 (Hora oficial do Brasil)

Comparar datas

As datas podem ser facilmente comparados. O exemplo a seguir compara a data de hoje com 14 janeiro de 2100: var today, someday, text;  
today = new Date();  
someday = new Date();  
someday.setFullYear(2100, 0, 14);  
  
if (someday > today) {  
    text = "Today is before January 14, 2100.";  
} else {  
    text = "Today is after January 14, 2100.";  
}  
document.getElementById("demo").innerHTML = text; // Today is before January 14, 2100. JavaScript conta meses de 0 a 11, Janeiro é 0 e Dezembro é 11.

Métodos de Data UTC

Os métodos de data UTC são utilizados para datas UTC (Universal Time Zone dates):

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| getUTCDate() | O mesmo que getDate(), mas retorna a data de UTC |
| getUTCDay() | Igual a getDay(), mas retorna o dia UTC |
| getUTCFullYear() | Igual ao getFullYear(), mas retorna o ano UTC |
| getUTCHours() | Igual a getHours(), mas retorna a hora UTC |
| getUTCMilliseconds() | Igual a getMilliseconds(), mas retorna os milissegundos do UTC |
| getUTCMinutes() | Igual a getMinutes(), mas retorna os minutos UTC |
| getUTCMonth() | O mesmo que getMonth(), mas retorna o mês UTC |
| getUTCSeconds() | Igual a getSeconds(), mas retorna os segundos UTC |

**VETORES (ARRAYS)**

Vetores de JavaScript são usados ​​para armazenar vários valores em uma única variável. var cars = ["Saab", "Volvo", "BMW"];

O que é um Array?

Uma vetor é uma variável especial, que pode conter mais de um valor de cada vez. Se você tem uma lista de itens (a lista de nomes de carro, por exemplo), armazenando os carros em variáveis ​​individuais poderia ser assim: var car1 = "Saab"; var car2 = "Volvo";  
var car3 = "BMW";

No entanto, se você quiser percorrer os carros e encontrar um carro específico havendo não apenas 3 carros, mas 300? A solução é um array! Um vetor pode conter muitos valores sob um único nome, e você pode acessar os valores por referência a um número de índice.

Criando um Array

Usar um vetor literal é a maneira mais fácil de criar uma vetor em JavaScript: var array\_name = [item1, item2, ...];   var cars = ["Saab", "Volvo", "BMW"]; Espaços e quebras de linha não são importantes. A declaração pode abranger várias linhas: var cars = [  
    "Saab",  
    "Volvo",  
    "BMW"  
]; //Colocar uma vírgula após o último elemento (como "BMW",) é inconsistente entre os navegadores.

Pose-se usar a palavra-chave **NEW** para criar vetores e atribuir valores a ele: var cars = new Array("Saab", "Volvo", "BMW"); Os dois exemplos acima são iguais, não há necessidade de usar new Array (). Para simplicidade, a legibilidade e velocidade de execução, usar o primeiro (o método vetor literal).

Acesso aos elementos de um array

Você se refere a um elemento de vetor referindo-se ao **número de índice** . Esta declaração acessa o valor do primeiro elemento em carros: var name = cars[0]; Esta declaração modifica o primeiro elemento em carros: cars[0] = "Opel";

var cars = ["Saab", "Volvo", "BMW"];  
document.getElementById("demo").innerHTML = cars[0]; // Saab

[0] é o primeiro elemento de uma matriz. [1] é o segundo. índices de vetor começam com 0.

Acesso completo ao vetor

Com JavaScript, o conjunto completo pode ser acessado por referindo-se ao nome do vetor: var cars = ["Saab", "Volvo", "BMW"];  
document.getElementById("demo").innerHTML = cars;

**Arrays são objetos**

Arrays são um tipo especial de objetos. O **typeof** operador em JavaScript retorna "objeto" para arrays. Mas, vetores de JavaScript são melhor descritos como vetores. Arrays usam **números** para acessar seus "membros". Neste exemplo, **pessoa [0]** retorna John: var person = ["John", "Doe", 46];

Objetos usam **nomes** para acessar seus "membros". Neste exemplo, **person.firstName** retorna John: var person = {firstName:"John", lastName:"Doe", age:46};

**Elementos de vetor podem ser objetos**

Variáveis ​​JavaScript podem ser objetos. Arrays são tipos especiais de objetos. Devido a isso, você pode ter variáveis ​​de tipos diferentes no mesmo array. Você pode ter objetos em um vetor. Você pode ter funções em um vetor. Você pode ter vetores em um vetor: myArray[0] = Date.now;  
myArray[1] = myFunction;  
myArray[2] = myCars;

**Array: Métodos e Propriedades**

Ogrande poder de vetores em JavaScript são o manuseio de propriedades e métodos do vetor: var x = cars.length; //A propriedade length retorna o número de elementos  
var y = cars.sort();   // O método sort() ordena arrays

**A propriedade length**

A propriedade **length** de uma matriz retorna o comprimento de uma matriz. var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.length;       // O comprimento de fruits é 4 A propriedade length é sempre um número a mais do que o maior índice do vetor que começa com 0. lenght é 4 e o índice é 3.

Looping em elementos no vetor

A melhor maneira de criar loops através de um vetor, é usando um loop "for": var fruits, text, fLen, i;  
fruits = ["Banana", "Orange", "Apple", "Mango"];  
fLen = fruits.length;  
text = "<ul>";  
for (i = 0; i < fLen; i++) {  
    text += "<li>" + fruits[i] + "</li>"; //Retorna uma lista com os   
} elementos do vetor

Adicionando elementos no vetor

A maneira mais fácil para adicionar um novo elemento a um vetor é usando o método push: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.push("Lemon"); // Adiciona um novo elemento (Lemon) à fruits

Um novo elemento também pode ser adicionado a um vetor utilizando a propriedade length: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits[fruits.length] = "Lemon"; // Adiciona um novo elemento (Lemon) à fruits

Adicionar elementos com índices maiores pode-se criar "buracos" indefinidos em um vetor: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits[6] = "Lemon"; // Adiciona um novo elemento (Lemon) à fruits Banana  
Orange  
Apple  
Mango  
undefined  
undefined  
Lemon

**Arrays associativos**

Muitas linguagens de programação suportam arrays com índices nomeados. Arrays com índices nomeados são chamados de vetores associativos (ou hashes). O JavaScript **não** suporta vetores com índices nomeados. Em JavaScript, **vetores** sempre utilizam **índices numerados**.  var person = [];  
person[0] = "John";  
person[1] = "Doe";  
person[2] = 46;  
var x = person.length;         // person.length retornará 3  
var y = person[0];             // person[0] retornará "John"

Se você usar índices nomeados, o JavaScript irá redefinir a matriz para um objeto padrão.   
Depois disso, alguns métodos e propriedades do vetor e as irão produzir **resultados incorretos**. var person = [];  
person["firstName"] = "John";  
person["lastName"] = "Doe";  
person["age"] = 46;  
var x = person.length;         // person.length retornará 0  
var y = person[0];             // person[0] retornará undefined

**A diferença entre matrizes e objetos**

Em JavaScript, **vetores** usam **índices numerados**. Em JavaScript, **objetos** usam **índices nomeados**. Os vetores são um tipo especial de objetos, com índices numerados.

Quando usar Arrays e quando usar objetos.

* O JavaScript não suportam arrays associativos.
* Você deve usar **objetos** quando você quiser nomear os elementos com **strings**.
* Você deve usar **vetores** quando quiser nomear os elementos com **números** .

**Evite new Array()**

Não há necessidade de usar construtor do JavaScript **new** Array(). **Use [] em vez.** Nestas duas declarações diferentes, ambas criam um novo vetor nomeado points vazio: var points = new Array();         // Errado  
var points = [];                  // Certo

Nestas duas instruções diferentes ambas criam um novo vetor contendo 6 números: var points = new Array(40, 100, 1, 5, 25, 10); // Errado  
var points = [40, 100, 1, 5, 25, 10];          // Certo

A palavra-chave **new** além de complicar o código, também pode produzir alguns resultados inesperados: var points = new Array(40, 100);  // Cria um array com dois elementos (40 and 100)

E se remover um dos elementos: var points = new Array(40); // Cria um array com 40 undefined elementos !!!!!

**Como reconhecer um Array**

Uma pergunta comum é: Como posso saber se a variável é um array? O problema é que o operador JavaScript **typeof** retorna "objeto": var fruits = ["Banana", "Orange", "Apple", "Mango"];  
typeof fruits;             // retorna objeto O operador typeof retorna objeto porque um vetor em JavaScript é um objeto.

Solução 1:

Para resolver este problema ECMAScript 5 define um novo método **Array.isArray()**: Array.isArray(fruits);     // retorna true O problema com esta solução é que ECMAScript 5 é **não é suportado em navegadores antigos**.

Solução 2:

Para resolver este problema, você pode criar sua própria função **IsArray():** function isArray(x) {  
    return x.constructor.toString().indexOf("Array") > -1;  
} // retorna true A função acima sempre retorna true se o argumento é um array. Ou mais precisamente: ele retorna verdadeiro se o protótipo do objeto contém a palavra "**array**".

Solução 3:

O **instanceof** retorna verdadeiro se um objeto é criado por um determinado construtor: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits instanceof Array     // retorna true

Métodos para vetores

A força de vetores em JavaScript reside nos seus métodos.

**Convertendo Arrays para Strings**

O método JavaScript **toString()** converte um conjunto de uma string (separados por vírgulas) de valores de vetor. var fruits = ["Banana", "Orange", "Apple", "Mango"];  
document.getElementById("demo").innerHTML = fruits.toString(); // retorna Banana,Orange,Apple,Mango

O método **join()** também junta todos os elementos da matriz em uma string. Ele se comporta exatamente como toString(), mas, além disso, você pode especificar o separador: var fruits = ["Banana", "Orange","Apple", "Mango"];  
document.getElementById("demo").innerHTML = fruits.join(" \* "); // retorna Banana \* Orange \* Apple \* Mango

**Popping e Pushing**

Quando você trabalha com vetores, é fácil de remover elementos e adicionar novos elementos. Isto é o que popping e pushing fazem: Popping retira itens de um vetor, e pushing adiciona itens em um vetor.

popping

O método **pop()** remove o último elemento em um vetor: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.pop();   // Remove o último elemento ("Mango") de fruits

O método **pop()** retorna o valor que foi "retirado": var fruits = ["Banana", "Orange", "Apple", "Mango"];  
var x = fruits.pop();      // O valor de x é "Mango"

pushing

O método **push()** adiciona um novo elemento em um vetor (no final): var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.push("Kiwi");  //  Adiciona um novo elemento ("Kiwi") à fruits

O método **push()** retorna o novo comprimento do vetor: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
var x = fruits.push("Kiwi");   //  O valor de x é 5

**Deslocando Elementos**

O **shift()** é equivalente ao pop(), porém retira o primeiro elemento em vez do último e "desloca" todos os outros elementos para um índice mais baixo: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.shift();   // Remove o primeiro elemento "Banana" de fruits

O método **shift()** retorna a string que foi "deslocaa para fora": var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.shift();             // Returna "Banana"

O método **unshift()** adiciona um novo elemento a um vetor (no início) e "destrava" elementos mais velhos: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.unshift("Lemon"); // Adiciona um novo elemento "Lemon" à fruits

O método **unshift()** retorna o novo comprimento da matriz: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.unshift("Lemon");    // Retorna 5

**Alterarando Elementos**

Elementos de vetores são acedidos utilizando o seu **número de índice** : Í**ndices** começam com 0, [0] é o primeiro elemento da matriz, [1] representa a segunda, [2] é o terceiro ... var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits[0] = "Kiwi"; // Troca o primeiro elemento de fruits para "Kiwi"

A propriedade de comprimento fornece uma maneira fácil de acrescentar um novo elemento a uma matriz: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits[fruits.length] = "Kiwi";   // Adiciona "Kiwi" à fruit no final

**Excluindo Elementos**

Vetores em JavaScript são objetos, elementos podem ser excluídos usando o operador **delete**: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
delete fruits[0]; //Troca o primeiro elemento em fruits para **undefined**

Usar **delete** pode deixar buracos indefinidos na matriz. Use **pop()** ou **shift()** em vez disso.

**Emendando vetores**

O método **splice()** pode ser usado para adicionar novos itens a uma matriz: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(2, 0, "Lemon", "Kiwi"); // Retorna Banana, Orange, Lemon, Kiwi, Apple, Mango

O primeiro parâmetro (2) define a posição **em que** devem entrar os novos elementos  **adicionados** (spliced em). O segundo parâmetro (0) define **quantos** elementos devem ser **removidos**. O resto dos parâmetros ( "Lemon", "Kiwi") definem os novos elementos a ser **adicionados**.

**Usando splice() para remover elementos**

Com parametrização inteligente, você pode usar splice() para remover elementos sem deixar "buracos" na matriz: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(0, 1);  // Remove o primeiro (Banana) elemento de fruits

O primeiro parâmetro (0) define a posição em que devem ser novos elementos **adicionados** (spliced em).O segundo parâmetro (1) define **quantas** elementos deve ser **removido**. O resto dos parâmetros são omitidos. será adicionado há novos elementos.

**Fundindo vetores(concatenação)**

O método **concat()** cria uma nova matriz, fundindo (concatenando) vetores existentes: var myGirls = ["Cecilie", "Lone"];  
var myBoys = ["Emil", "Tobias","Linus"];  
var myChildren = myGirls.concat(myBoys);     // Concatena (junta) myGirls and myBoys = Cecilie,Lone,Emil,Tobias,Linus

O método **concat()** não muda os vetores existentes. Ele sempre retorna um novo vetor. O método de **concat()** pode pegar qualquer número de um argumento de um vetor: var arr1 = ["Cecilie", "Lone"];  
var arr2 = ["Emil", "Tobias","Linus"];  
var arr3 = ["Robin", "Morgan"];  
var myChildren = arr1.concat(arr2, arr3);     // Concatena arr1 com arr2 e arr3 = Cecilie,Lone,Emil,Tobias,Linus,Robin, Morgan

O método **concat()** também pode ter valores como argumentos: var arr1 = ["Cecilie", "Lone"];  
var myChildren = arr1.concat(["Emil", "Tobias","Linus"]); // Retorna Cecilie,Lone,Emil,Tobias,Linus

**Cortando vetores**

O método **slice()** corta um pedaço de uma vetor para uma novo vetor. Este exemplo corta para fora uma parte de um vetor a partir de um elemento de vetor ( "Orange"): var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(1); // Retorna Orange,Lemon,Apple,Mango

O método **slice()** cria uma novo vetor. Ele não remove todos os elementos do vetor de origem. Este exemplo corta para fora uma parte de um vetor a partir do elemento 3 ( "Apple"): var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(3); // Retorna Apple,Mango

O método **slice()** pode assumir dois argumentos como slice (1, 3). O método, em seguida, selecciona elementos do argumento de início e até (mas não incluindo) o argumento final: var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(1, 3); // Retorna Orange,Lemon

Se o argumento final é omitido, como nos primeiros exemplos, o método **slice()** corta para fora o resto do vetor: var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(2); // Retorna Lemon,Apple,Mango

**toString () automática**

JavaScript converte automaticamente um vetor para uma string separada por vírgulas quando um valor primitivo é esperado. Isto sempre ocorrerá na saída de um array. Todos os objetos JavaScript tem um método toString (). Estes dois exemplos produzem o mesmo resultado: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
document.getElementById("demo").innerHTML = fruits.toString();

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
document.getElementById("demo").innerHTML = fruits; Todos os objetos JavaScript tem um método **toString().**

**Encontrar Max e Min valores de um array**

Não há funções embutidas para encontrar o maior ou menor valor em um vetor JavaScript.

**Ordenando vetores**

O método **sort()** é um dos métodos devetores mais fortes. O método **sort()** classifica um vetor alfabeticamente: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort(); // Ordena alfabéticamente os elementos de fruits = Apple,Banana,Mango,Orange

**Revertendo vetores**

O método **reverse()** inverte os elementos de uma matriz. Você pode usá-lo para ordenar um array em ordem decrescente: var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort();            // Ordena os elementos de fruits   
fruits.reverse();         // Reverte a ordem dos elementos = Orange,Mango,Banana,Apple

**Ordenação numérica**

Por padrão, a função **sort()** classifica valores como **strings.** Isso funciona bem para strings ( "Apple" vem antes de "Banana"). No entanto, se os números forem classificadas como strings, "25" é maior que "100", porque "2" é maior que "1". Devido a isso, o método **sort()** irá produzir resultados incorretos durante a classificação números. Você pode corrigir isso através de uma **função de comparação**: var points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return a - b}); // Retorna 1,5,10,25,40,100

Use o mesmo truque para ordenar um vetor descendente: var points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return b - a}); // Retorna 100,40,25,10,5,1

Função de comparação

O objetivo da função de comparação é definir uma ordem de ordenação alternativa. A função de comparação deve retornar um valor negativo, zero, ou positivo, dependendo dos argumentos: function(a, b){return a-b}

Quando a função **sort()** compara dois valores, ela envia os valores para a função de comparação e ordena os valores de acordo com o valor retornado (negativo, zero, positivo).

Ao comparar 40 e 100, o método **sort()** chama a função compare(40.100). A função calcula 40-100, e retorna -60 (um valor negativo). A função de classificação irá classificar 40 como um valor inferior a 100. Você pode usar este trecho de código para experimentar numericamente e em ordem alfabética classificação: var points = [40, 100, 1, 5, 25, 10];  
document.getElementById("demo").innerHTML = points;  
function myFunction1() {  
    points.sort();  
    document.getElementById("demo").innerHTML = points;  
}  
function myFunction2() {  
    points.sort(function(a, b){return a - b});  
    document.getElementById("demo").innerHTML = points;  
} // sort alfabéticamente = 1,10,100,25,40,5 // sort numéricamente = 1,5,10,25,40,100

**Ordenação de um vetor em ordem aleatória**

var points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return 0.5 - Math.random()});

**Encontrar o mais alto (ou baixo) valor de um vetor**

Não existem funções internas para encontrar o máximo ou valor min em uma matriz. No entanto, depois de ter rordenado um vetor, você pode usar o índice para obter os valores mais altos e menores.

Ordenação ascendente: var points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return a - b});  
// agora points[0] contém o valor mais baixo  
// e points[points.length-1] contém o valor mais alto

Ordenação descendente: var points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return b - a});  
// agora points[0] contém o valor mais alto  
// e points[points.length-1] contém o valor mais baixo

Ordenação de um array inteiro é um método muito ineficiente se você só quer encontrar o menor ou o valor mais alto.

**Usando Math.max() em um vetor**

Você pode usar **Math.max.apply()** para encontrar o número mais alto em um vetor: function myArrayMax(arr) {  
    return Math.max.apply(null, arr);  
} \*Math.max.apply ([1, 2, 3]) é equivalente a Math.max (1, 2, 3).

**Usando Math.min() em um vetor**

Você pode usar **Math.min.apply()** para encontrar o menor número em uma matriz: function myArrayMin(arr) {  
    return Math.min.apply(null, arr);  
} \*Math.min.apply ([1, 2, 3]) é equivalente a Math.min (1, 2, 3).

Métodos caseiros Min / Max

A solução mais rápida é a utilização de um método "caseiro". Esta função laça um vetor de comparação com cada valor mais alto encontrado: function myArrayMax(arr) {  
    var len = arr.length  
    var max = -Infinity;  
    while (len--) {  
        if (arr[len] > max) {  
            max = arr[len];  
        }  
    }  
    return max;  
} // Retorna 100, para se encontrar o min é só fazer a troca do max

Ordenação de objetos de arrays

Vetores em JavaScript muitas vezes contêm objetos: var cars = [  
{type:"Volvo", year:2016},  
{type:"Saab", year:2001},  
{type:"BMW", year:2010}];

Mesmo que os objetos tenham propriedades de diferentes tipos de dados, o método **sort()** pode ser utilizado para classificar a matriz.

A solução é escrever uma função de comparação para comparar os valores de propriedade: cars.sort(function(a, b){return a.year - b.year}); // Retorna Saab 2001 BMW 2010 Volvo 2016

Comparar propriedades de strings é um pouco mais complexo: cars.sort(function(a, b){  
    var x = a.type.toLowerCase();  
    var y = b.type.toLowerCase();  
    if (x < y) {return -1;}  
    if (x > y) {return 1;}  
    return 0;  
}); //Retorna BMW 2010 Saab 2001 Volvo 2016

**BOOLEANOS**

Um valor booleano JavaScript representa um dos dois valores: **verdadeiro** ou **falso** .

**Valores booleanos**

Muitas vezes, na programação, você vai precisar de um tipo de dados que só pode ter um de dois valores, como:

* SIM/NÃO
* LIGADO/DESLIGADO
* VERDADEIRO/FALSO

**A Função Boolean()**

Você pode usar a função **Boolean()** para descobrir se uma expressão (ou uma variável) é verdadeira: Boolean(10 > 9)   ou 10 > 9 // Retorna true Tudo com um valor "real" é verdadeiro: x = 100; // 100 é true  
x = 3.14; //3.14 é true  
x = -15; //-15 é true  
x = "Hello”; //Toda string (não vazia) é true  
x = "false"; //Toda a string 'false' é true  
x = 7 + 1 + 3.14; //Toda expressão (exceto zero) é true

Tudo sem um valor "real" é falso: x = 0; // Retorna falso x = -0; // Retorna falso x = ""; // Toda string vazia retorna falso x; // Indefinido retorna falso x = null; // Nula retorna falso x = false; // retorna falso x = 10 / "H"; // NaN (Not a Number) retorna falso

**Booleanos podem ser objetos**

Normalmente booleanos JavaScript são valores primitivos criados a partir de literais: **var x = falso**. Mas booleanos também podem ser definidos como objetos com a palavra-chave new: **var y = new booleano (false):** var x = false;  
var y = new Boolean(false);  
// typeof x retorna boolean  
// typeof y retorna object

Não crie objetos booleanos. Eles retardam a velocidade de execução.   
A  palavra-chave **new**  complica o código. Isso pode produzir alguns resultados inesperados.

Ao usar o operador ==, booleans iguais são iguais: var x = false;               
var y = new Boolean(false);  
// (x == y) é verdadeiro por que x e y tem valores iguais

Ao usar o operador ===, booleans iguais não são iguais, porque o operador === espera igualdade em ambos os tipo e valor: var x = false;               
var y = new Boolean(false);  
// (x === y) é falso por que x e y tem tipos diferentes de dados

Ou ainda pior. Os objetos não podem ser comparados: var x = new Boolean(false);               
var y = new Boolean(false);  
// (x == y) é falso por que objetos não podem ser comparados

Note-se a diferença entre o (x == y) e (x === y).   
Comparar dois objetos JavaScript sempre retornará false.

**COMPARAÇÃO E OPERADORES LÓGICOS**

**Operadores de comparação**

Os operadores de comparação são usados ​​em declarações lógicas para determinar a igualdade ou diferença entre as variáveis ​​ou valores. Dado que **x = 5** , a tabela abaixo explica os operadores de comparação:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operador** | **Descrição** | **Comparação** | **Retornos** |  |
| == | igual a | x == 8 | false |  |
| x == 5 | true |  |
| x == "5" | true |  |
| === | valor igual e tipo igual | x === 5 | true |  |
| x === "5" | false |  |
| != | diferente | x != 8 | true |  |
| !== | diferente valor ou tipo | x !== 5 | false |  |
| x !== "5" | true |  |
| x !== 8 | true |  |
| > | maior que | x > 8 | false |  |
| < | menor que | x < 8 | true |  |
| >= | maior que ou menor a | x >= 8 | false |  |
| <= | menor que ou maior a | x <= 8 | true |  |

Como eles podem ser usados

Os operadores de comparação podem ser usados ​​em declarações condicionais para comparar valores e agir de acordo com o resultado: if (idade < 18) texto = "Muito jovem";

**Operadores lógicos**

Os operadores lógicos são usados ​​para determinar a lógica entre as variáveis ​​ou valores. Dado que **x = 6** e **y = 3** , a tabela a seguir explica os operadores lógicos:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operador** | **Descrição** | **Exemplo** |  |
| && | and | (x < 10 && y > 1) é true |  |
| || | or | (x == 5 || y == 5) é false |  |
| ! | not | !(x == y) é true |  |

**Condicional Operador (ternário)**

JavaScript também contém um operador condicional que atribui um valor a uma variável com base em alguma condição: nomeVariavel= (condição) ?valor1:valor2  var votar = (idade < 18) ? "Muito jovem":"Idade suficiente";

Comparando tipos diferentes de dados

Comparar dados de tipos diferentes podem dar resultados inesperados. Ao comparar uma string com um número, o JavaScript irá converter a string para um número ao fazer a comparação. Uma string vazia converte-se a 0. A string não numérica converte em NaN que é sempre falsa:

|  |  |  |
| --- | --- | --- |
| **Caso** | **Valor** |  |
| 2 < 12 | true |  |
| 2 < "12" | true |  |
| 2 < "John" | false |  |
| 2 > "John" | false |  |
| 2 == "John" | false |  |
| "2" < "12" | false |  |
| "2" > "12" | true |  |
| "2" == "12" | false |  |

Ao comparar duas strings, "2" será maior do que o "12", por causa (por ordem alfabética) 1 é menor do que dois. Para garantir um bom resultado, variáveis ​​devem ser convertidas para o tipo adequado antes da comparação.

**DECLARAÇÕES IF...ELSE**

Muitas vezes, quando você escrever código, você quer executar ações diferentes para diferentes decisões. Você pode usar declarações condicionais em seu código para fazer isso.

Em JavaScript temos as seguintes declarações condicionais:

* Use **if** para especificar um bloco de código a ser executado, se uma condição especificada for verdadeira
* Use **else** para especificar um bloco de código a ser executado, se a mesma condição é falsa
* Use **else if** para especificar uma nova condição para testar, se a primeira condição é falsa
* Use **switch** para especificar muitos blocos alternativos de código a ser executado

**A instrução if**

Use a instrução **if** para especificar um bloco de código JavaScript a ser executado se a condição for verdadeira. if (*condition*) {  
    block of code to be executed if the condition is true}

Note que **if** está em letras minúsculas. Letras maiúsculas (If ou IF) irá gerar um erro.

Faça uma saudação de "Bom Dia" se a hora for inferior a 18:00: if (hora < 18) {  
    saudacao = "Bom dia”; //Retorna Bom dia  
}

**A instrução else**

Use a instrução else para especificar um bloco de código a ser executado se a condição for falsa. if (*condition*) {  
    block of code to be executed if the condition is true} else {   
    block of code to be executed if the condition is false}

Se a hora for inferior ás 18:00 hrs, crie uma saudação "Bom dia", caso contrário, "Boa noite": if (hora < 18) {  
    saudacao = "Good day";  
} else {  
    saudacao = "Good evening";  
}

**A instrução else if**

Use a instrução **else if** para especificar uma nova condição, se a primeira condição for falsa. if (*condition1*) {  
    block of code to be executed if condition1 is true} else if (*condition2*) {  
    block of code to be executed if the condition1 is false and condition2 is true  
} else {  
    block of code to be executed if the condition1 is false and condition2 is false}

Se o tempo é inferior a 10:00 hrs, crie uma saudação "Bom dia", se não, porém o tempo é inferior a 20:00, crie uma saudação "Boa tarde", caso contrário, um "Boa noite": if (tempo < 10) {  
    saudacao = "Bom dia";  
} else if (time < 20) {  
    saudacao = "Boa tarde";  
} else {  
    saudacao = "Boa noite";  
}

**SWITCH**

A instrução switch é usada para executar ações diferentes com base em diferentes condições, selecionando um dos muitos blocos de código para ser executado.

switch(expressão) {  
    case n:  
        bloco de código        break;  
    case n:  
        bloco de código        break;  
    default:  
        bloco de código  
}

Funciona assim:

* A expressão switch é avaliada uma vez.
* O valor da expressão é comparado com os valores de cada caso.
* Se houver uma correspondência, o bloco associado de código é executado.

O método **getDay()** retorna a semana como um número entre 0 e 6. (Domingo = 0, Segunda = 1, Terça-feira = 2 ..). Este exemplo usa o número da semana para calcular o dia da semana: switch (new Date().getDay()) {  
    case 0:  
        day = "Sunday";  
        break;  
    case 1:  
        day = "Monday";  
        break;  
    case 2:  
        day = "Tuesday";  
        break;  
    case 3:  
        day = "Wednesday";  
        break;  
    case 4:  
        day = "Thursday";  
        break;  
    case 5:  
        day = "Friday";  
        break;  
    case 6:  
        day = "Saturday";  
} // Retorna Wednesday

**A palavra-chave Break**

Quando JavaScript atinge a palavra-chave **break**, ele salta para fora do bloco switch. Isto irá parar a execução de mais casos nos testes do código dentro do bloco.

Quando for encontrada uma correspondência, e o trabalho é feito, é hora de dar um **break**. Não há necessidade de mais testes.

Um **break** pode salvar um monte de tempo de execução porque "ignora" a execução de todo o resto do código no bloco switch. Não é necessário **break** no último caso em um bloco switch. Os intervalos de bloco (extremidades) já fazem o **break** de qualquer maneira.

A instrução break também pode ser usado para saltar para fora de um loop. A **instrução break** quebra o loop e continua a executar o código após o loop (se houver): for (i = 0; i < 10; i++) {  
    if (i === 3) { break; }  
    text += "The number is " + i + "<br>";  
} // Retorna 0 , 1, 2

**A palavra-chave Continue**

A **instrução continue** quebra uma iteração (no loop), se uma condição especificada ocorrer, e continua com a iteração seguinte do loop. Este exemplo ignora o valor de 3: for (i = 0; i < 10; i++) {  
    if (i === 3) { continue; }  
    text += "The number is " + i + "<br>";  
} // Retorna 0, 1, 2, 4, 5, 6, 7, 8, 9

**A palavra-chave Default**

A palavra-chave **default**  especifica o código a ser executado se não houver correspondência. O método **getDay()** retorna a semana como um número entre 0 e 6. Se hoje não é nem sábado (6), nem domingo (0), escrever uma mensagem padrão: 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";  
} // Retorna Looking forward to the Weekend

O **default** não tem que ser o último caso em um bloco switch: switch (new Date().getDay()) {  
    default:   
        text = "Looking forward to the Weekend";  
        break;  
    case 6:  
        text = "Today is Saturday";  
        break;   
    case 0:  
        text = "Today is Sunday";  
} // Retorna Looking forward to the Weekend Se o **default** não for o último caso no bloco switch, lembre-se de encerra-locom um **break**.

**Blocos de código comum**

Às vezes, você vai querer diferentes casos switch para usar o mesmo código. Neste caso, quatro e cinco partes do mesmo bloco de código, e 0 e 6 partes outro bloco de código exemplo: switch (new Date().getDay()) {  
    case 4:  
    case 5:  
        text = "Soon it is Weekend";  
        break;   
    case 0:  
    case 6:  
        text = "It is Weekend";  
        break;  
    default:   
        text = "Looking forward to the Weekend";  
} // Retorna Looking forward to the Weekend

**Labels em JavaScript**

Para criar rótulos em JavaScript os rótulos antecedem as declarações com um nome e dois pontos: label: statements

As declarações **break** e **continue** são as únicas instruções JavaScript que podem "saltar para fora de" um bloco de código. break labelname;   
continue labelname;

A declaração **continue** (com ou sem um rótulo de referência) só pode ser usado para **ignorar uma iteração** . A instrução **break**, sem uma referência de rótulo, só pode ser utilizado para **saltar de um loop ou um switch**. Com uma referência de rótulo, a instrução **break** pode ser usado para **saltar para fora de qualquer bloco de código**: var cars = ["BMW", "Volvo", "Saab", "Ford"];  
list: {  
    text += cars[0] + "<br>";   
    text += cars[1] + "<br>";   
    text += cars[2] + "<br>";   
    break list;  
    text += cars[3] + "<br>";   
    text += cars[4] + "<br>";   
    text += cars[5] + "<br>";   
} // Retorna BMW, Volvo, Saab O bloco de código está localizado entre os sinais **{** e **}**.

**LOOP FOR**

Loops são úteis, se você deseja executar o mesmo código várias vezes, cada vez com um valor diferente.

Muitas vezes, esse é o caso quando se trabalha com vetores:

text += cars[0] + "<br>"; // BMW  
text += cars[1] + "<br>"; // Volvo  
text += cars[2] + "<br>"; // Saab  
text += cars[3] + "<br>"; // Ford  
text += cars[4] + "<br>"; // Fiat  
text += cars[5] + "<br>"; // Audi

Ao invés disto você pode fazer assim: for (i = 0; i < cars.length; i++) {   
    text += cars[i] + "<br>";  
} // Retorna BMW, Volvo, Saab, Ford, Fiat, Audi

Diferentes tipos de Loops

JavaScript suporta diferentes tipos de loops:

* **for** - loops através de um bloco de código com um número de vezes especificada
* **for / in**- percorre as propriedades de um objeto
* **while** - loops através de um bloco de código enquanto uma condição especificada seja verdadeira
* **do/ while** - também loops através de um bloco de código enquanto uma condição especificada seja verdadeira

**O Loop For**

O laço for é muitas vezes a ferramenta que você vai usar quando você quer criar um loop. O loop tem a seguinte sintaxe: for (*declaração 1*;*declaração 2*;*declaração 3*) {  
    bloco de código a ser executado  
}

**Declaração 1** é executado antes do loop (o bloco de código) começar. **Declaração 2** define a condição para o funcionamento do loop (o bloco de código). **Declaração 3** é executada cada vez após que o loop (o bloco de código) foi executado. for (i = 0; i < 5; i++) {  
    text += "The number is " + i + "<br>";  
} // Retorna 0, 1, 2, 3, 4

A partir do exemplo acima, você pode ler: **Declaração 1** define uma variável antes do laço começar (var i = 0). **Declaração 2** define a condição para o loop para executar (i deve ser inferior a 5). **Declaração 3** aumenta um valor (i ++) cada vez que o bloco de código do loop for executado.

Declaração 1

Normalmente você vai usar a declaração 1 para inicializar a variável usada no loop (i = 0). Isso nem sempre é o caso, JavaScript não se importa. A declaração 1 é opcional. Você pode iniciar muitos valores na declaração 1 (separados por vírgula): for (**i = 0, len = cars.length, text = ""**; i < len; i++) {   
    text += cars[i] + "<br>";  
} // Retorna BMW, Volvo, Saab, Ford, Fiat, Audi

E você pode omitir declaração 1, como quando seus valores são definidos antes do loop iniciar: var i = 2; // Define que o loop começará a partir da posição 2  
var len = cars.length;  
var text = "";  
for (; i < len; i++) {   
    text += cars[i] + "<br>";  
} // Retorna Saab, Ford, Fiat, Audi

Declaração 2

Muitas vezes, declaração 2 é usado para avaliar a condição da variável inicial. Isso nem sempre é o caso, JavaScript não se importa. A declaração 2 também é opcional. Se a declaração 2 retornar true, o loop vai começar tudo de novo, se ele retornar false, o loop vai terminar.

Se você omitir declaração 2, você deve fornecer um **break** dentro do loop. Caso contrário, o loop nunca irá terminar. Isto irá travar o seu browser.

Declaração 3

Muitas vezes afirmação 3 incrementa o valor da variável inicial. Isso nem sempre é o caso, JavaScript não se importa, também é opcional. A declaração 3 pode fazer qualquer coisa como incremento negativo (i--), incremento positivo (i = i + 15), ou qualquer outra coisa. Declaração 3 também pode ser omitida (como quando você incrementa seus valores dentro do loop): var i = 0;  
var len = cars.length;  
for (; i < len; ) {   
    text += cars[i] + "<br>";  
    i++;  
} // Retorna BMW, Volvo, Saab, Ford, Fiat, Audi

**O loop For / In**

A declaração JavaScript **for / in** percorre as propriedades de um objeto: var person = {fname:"John", lname:"Doe", age:25};   
var text = "";  
var x;  
for (x in person) {  
    text += person[x];  
} // Retorna Jhon Doe 25

**LOOP WHILE**

O loop **while** circula através de um bloco de código enquanto uma condição especificada seja verdadeira. while (condição) {  
*bloco de código a ser executado*  
} No exemplo a seguir, o código do loop será executado, várias vezes, desde que uma variável (i) seja inferior a 10: while (i < 10) {  
    text += "The number is " + i;  
    i++;  
} // Retorna 0, 1, 2, 4, 5, 6, 7, 8, 9

Caso você esqueça de aumentar a variável usada na condição (i++), o loop nunca vai acabar. Isto irá travar o seu browser.

**O Loop Do / While**

A / while loop é uma variante do circuito de tempo. Este loop será executado o bloco de código uma vez, antes de verificar se a condição for verdadeira, então ele vai repetir o ciclo enquanto a condição é verdadeira. do {  
*bloco de código a ser executado*}  
while (condição);

O exemplo a seguir utiliza um loop do / while. O loop será sempre executado pelo menos uma vez, mesmo se a condição for falsa, porque o bloco de código é executado antes da condição ser testada: var text = "" var i = 0; do {  
    text += "The number is " + i;  
    i++;  
} // Retorna 0, 1, 2, 4, 5, 6, 7, 8, 9  
while (i < 10);

Não se esqueça de aumentar a variável usada na condição, caso contrário, o loop nunca vai acabar!

Comparando for e while

Um loop **while** é o mesmo que um loop **for**, com a declaração 1 e declaração 3 omitidos. Neste exemplo usa-se um loop **for** para recolher os nomes de carro a partir do vetor carros: var cars = ["BMW", "Volvo", "Saab", "Ford"];  
var i = 0;  
var text = "";  
for (;cars[i];) {  
    text += cars[i] + "<br>";  
    i++;  
} // Retorna BMW, Volvo, Saab, Ford

Nesteexemplo usa-se um loop **while** para recolher os nomes de carro a partir da matriz carros: var cars = ["BMW", "Volvo", "Saab", "Ford"];  
var i = 0;  
var text = "";  
while (cars[i]) {  
    text += cars[i] + "<br>";  
    i++;  
} // Retorna BMW, Volvo, Saab, Ford

**JAVASCRIPT CONVERSÃO DE TIPO**

**Number()** converte para um número, **String()** converte a uma string, **Boolean()** converte para um valor booleano.

Tipos de dados JavaScript

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

* corda
* número
* boleano
* objeto
* função

Existem 3 tipos de objetos:

* Objeto
* Encontro
* ordem

E 2 tipos de dados que não podem conter valores:

* nulo
* Indefinido

A propriedade typeof

Você pode usar o **typeof** operador para encontrar o tipo de dado de uma variável JavaScript: typeof "John"                 // Retorna "string"   
typeof 3.14                   // Retorna "number"  
typeof NaN                    // Retorna "number"  
typeof false                  // Retorna "boolean"  
typeof [1,2,3,4]              // Retorna "object"  
typeof {name:'John', age:34}  // Retorna "object"  
typeof new Date()             // Retorna "object"  
typeof function () {}         // Retorna "function"  
typeof myCar                  // Retorna "undefined" \*  
typeof null                   // Retorna "object"

Favor observar:

* O tipo de dados de NaN é número
* O tipo de dado de um vetor é um objeto
* O tipo de dado de uma data é um objeto
* O tipo de dado de nulo é um objeto
* O tipo de dado de uma variável indefinida é **undefined** \*
* O tipo de dado de uma variável que não tenha sido atribuído um valor também é **undefined** \*

Você não pode usar **typeof** para determinar se um objeto JavaScript é um array (ou uma data). O operador **typeof** não é uma variável. É um operador. Operadores (+ - \* /) não têm qualquer tipo de dados. Mas, o operador typeof sempre **retorna uma string** (contendo o tipo do operando).

A propriedade construtora

A propriedade **construtora**  retorna a função de construtor de todas as variáveis de JavaScript. "John".constructor                // Retorna function String()   
(3.14).constructor                // Retorna function Number()   
false.constructor                 // Retorna function Boolean()  
[1,2,3,4].constructor             // Retorna function Array()    
{name:'John',age:34}.constructor  // Retorna function Object()   
new Date().constructor            // Retorna function Date()     
function () {}.constructor        // Retorna function

Você pode verificar a propriedade construtora para descobrir se um objeto é uma Array (contém a palavra "Array"): var fruits = ["Banana", "Orange", "Apple", "Mango"]; function isArray(myArray) {  
    return myArray.constructor.toString().indexOf("Array") > -1;  
} // Retorna true

Ou ainda mais simples, você pode verificar se o objeto é uma função de vetor: var fruits = ["Banana", "Orange", "Apple", "Mango"]; function isArray(myArray) {  
    return myArray.constructor === Array;  
} // Retorna true

Você pode verificar a propriedade construtor para descobrir se um objeto é um Data (contém a palavra "Date"): var myDate = new Date(); function isDate(myDate) {  
    return myDate.constructor.toString().indexOf("Date") > -1;  
} // Retorna true

Ou ainda mais simples, você pode verificar se o objeto é uma função Data: var myDate = new Date(); function isDate(myDate) {  
    return myDate.constructor === Date;  
}

Conversão de tipo

JavaScript variáveis ​​pode ser convertido para uma nova variável e outro tipo de dados:

* Com o uso de uma função JavaScript
* **Automaticamente** , por si só

**Convertendo números em strings**

O método global **String()** pode converter números para strings. Ele pode ser usado em qualquer tipo de números, literais, variáveis ​​ou expressões: String(x)    // Retorna uma string a partir de uma variável numérica x  
String(123)    // Retorna uma string a partir de um número literal 123  
String(100 + 23) // Retorna uma string com um número de uma expressão

O método **toString()** faz o mesmo: x.toString() // Retorna uma string a partir de uma variável numérica x  
(123).toString() // Retorna uma string a partir de um número literal 123  
(100 + 23).toString()// Retorna uma string com um número de uma expressão

**Convertendo booleans para strings**

O método global **String()** pode converter booleans para strings: String(false)        // retorna "false"  
String(true)         // retorna "true"

O método booleano **toString()** faz o mesmo: false.toString()     // retorna "false"  
true.toString()      // retorna "true"

**Converter datas para strings**

O método global **String()** pode converter datas para strings: String(Date())       // Retorna "Thu Jul 17 2014 15:38:19 GMT+0200

O método Data **toString ()** faz o mesmo: Date().toString()    // Retorna "Thu Jul 17 2014 15:38:19 GMT+0200

**Convertendo strings em números**

O método global **Number()** pode converter strings para números. Strings contendo números (como "3,14") convertem-se em números (como 3,14). Strings vazias converter-m-se a 0. Qualquer outra coisa se converte-rá em NaN (não é um número). Number("3.14")    // retorna 3.14  
Number(" ")       // retorna 0   
Number("")        // retorna 0  
Number("99 88")   // retorna NaN

**O operador unário +**

O **operador unário**  **+** pode ser utilizado para converter uma variável em um número: var y = "5";      // y é uma string  
var x = + y;      // x é um number

Se a variável não poder ser convertida, ainda vai tornar-se um número, mas com o valor **NaN**: var y = "John";   // y é uma string  
var x = + y;      // x é um number (NaN)

**Convertendo booleanos para números**

O método global **Number()** também pode converter booleans a números: Number(false)     // retorna 0  
Number(true)      // retorna 1

**Converter datas para números**

O método global **Number()** pode ser usado para converter datas para números: d = new Date();  
Number(d)          // retorna 1404568027739

O método de data **getTime()** faz o mesmo: d = new Date();  
d.getTime()        // retorna 1404568027739

Conversão automática de tipo

Quando JavaScript tenta operar em um tipo de dado "errado", ele tentará converter o valor para um tipo "certo". O resultado nem sempre é o que você espera: 5 + null    // retorna 5         porque null é convertido a 0  
"5" + null  // retorna "5null"   porque null é convertido a "null"  
"5" + 2     // retorna 52      porque 2 é convertido a "2"  
"5" - 2     // retorna 3      porque "5" é convertido a 5  
"5"\*"2" // retorna 10      porque "5" e "2" são convertidos a 5 e 2

Conversão automática de strings

JavaScript automaticamente chama o método **toString()** quando você tenta um "output" um objeto ou uma variável: document.getElementById("demo").innerHTML = myVar;  
// se myVar = {name:"Fjohn"}  // toString converte para "[object Object]"  
// se myVar = [1,2,3,4]       // toString converte para "1,2,3,4"  
// se myVar = new Date()      // toString converte para "Fri Jul 18 2014 09:08:55 GMT+0200"

Números e booleans também são convertidos, mas isso não é muito visível: //se myVar = 123             // toString converte para "123"  
//se myVar = true            // toString converte para "true"  
//se myVar = false           // toString converte para "false"

**EXPRESSÕES REGULARES**

Uma expressão regular é uma seqüência de caracteres que formam um padrão de pesquisa. O padrão de pesquisa pode ser usado para operações de pesquisa de texto e substituição de texto.

O que é uma expressão regular?

Quando você pesquisar dados em um texto, você pode usar este padrão de busca para descrever o que você está procurando. Uma expressão regular pode ser um único caractere, ou um padrão mais complexo. As expressões regulares podem ser usados para realizar todos os tipos de **pesquisa de texto** e **substituição de texto**. A sua sintaxe é a seguinte: /padrão/modificador;

var patt = /w3schools/i; // Expressão regular

**/ w3schools / i**   é uma expressão regular. **w3schools**  é um padrão (para ser usado em uma pesquisa). **i**   é um modificador (modifica a pesquisa a ser case-insensitive).

**Usando métodos de string**

Em JavaScript, expressões regulares são muitas vezes utilizadas com os dois **métodos de string** : search() e substituir().**O método search()** usa uma expressão para procurar uma correspondência e retorna a posição encontrada. **O método de substituir()** retorna uma string modificada em que o padrão foi substituído.

Usar uma expressão regular para fazer uma busca case-insensitive para "w3schools": var str = "Visit W3Schools";  
var n = str.search(/w3schools/i); // Retorna 6

**Usando a pesquisa string() com uma string**

O método de pesquisa também vai aceitar uma string como argumento de pesquisa. O argumento de string será convertida para uma expressão regular.

Usar uma string para fazer uma busca por "w3schools" em uma string: var str = "Visit W3Schools!";  
var n = str.search("W3Schools"); // Retorna 6

**Usar string replace() com uma expressão regular**

Use uma expressão regular case-insensitive para substituir Microsoft com W3Schools em uma string: var str = "Visit Microsoft!";  
var res = str.replace(/microsoft/i, "W3Schools"); // Retorna Visit W3Schools!

**Usando replace() string com uma string**

O método replace() também aceita uma string como argumento de pesquisa: <button onclick="myFunction()"> Tente </button>

<p id="demo">Please visit Microsoft!</p>

<script> function myFunction() {

var str = document.getElementById("demo").innerHTML;

var txt = str.replace("Microsoft","W3Schools");

document.getElementById("demo").innerHTML = txt;

}

<script>

**Os modificadores expressão regular**

**Modificadores** podem ser usados para realizar pesquisas case-insensitive mais globais:

|  |  |
| --- | --- |
| **Modificador** | **Descrição** |
| i | Executa uma combinação insensível a maiúsculas e minúsculas |
| g | Executa uma busca global (encontra todas as correspondências em vez de parar depois da primeira encontrada) |
| m | Executa a combinação de várias linhas |

**Padrões de expressões regulares**

**Colchetes** são usados para encontrar um intervalo de caracteres:

|  |  |
| --- | --- |
| **Expressão** | **Descrição** |
| [abc] | Encontra qualquer um dos caracteres entre os colchetes |
| [0-9] | Encontra qualquer um dos dígitos entre os colchetes |
| (x|y) | Encontra qualquer uma das alternativas separadas com | |

**Metacaracteres** são caracteres com um significado especial:

|  |  |
| --- | --- |
| **Metacaratere** | **Descrição** |
| \d | Encontra um dígito |
| \s | Encontra um caractere de espaço em branco |
| \b | Encontre um início ou fim de uma palavra |
| \uxxxx | Encontre o caractere Unicode especificado pelo número hexadecimal xxxx |

**Quantificadores** definem quantidades:

|  |  |
| --- | --- |
| **Quantificador** | **Descrição** |
| n+ | Corresponde a qualquer string que contenha pelo menos um n |
| n\* | Corresponde a qualquer string que contenha zero ou mais ocorrências de n |
| n? | Corresponde a qualquer string que contenha zero ou uma ocorrência de n |

Usando o objeto RegExp

O objeto **RegExp** é um expressão regular com propriedades e métodos predefinidos.

**Usando test()**

O método de **test()** é um método de expressão RegExp. Ele procura uma string para um padrão, e retorna verdadeiro ou falso, dependendo do resultado. O exemplo a seguir procura uma string para o caractere "e": var patt = /e/;  
patt.test("The best things in life are free!"); // Uma vez que exista um "e" na string, a saída do código acima será: true

Você não tem que colocar a expressão regular em uma variável em primeiro lugar. As duas linhas de cima pode ser encurtadas para uma: /e/.test("The best things in life are free!");

**Usando exec()**

O método **exec()** é um método de expressão RegExp. Ele procura uma string para um padrão especificado, e retorna o texto encontrado. Se nenhuma correspondência for encontrada, ele retorna *null.*O exemplo a seguir procura uma string para o caractere "e": /e/.exec("The best things in life are free!"); // Uma vez que exista um "e" na string, a saída do código acima será: e

**ERROS EM JAVASCRIPT**

A  declaração **try**  permite testar um bloco de código para erros. A **catch** declaração permite manipular o erro. O **throw** permite criar erros personalizados. O **finally** permite executar código, depois de tentativa e captura, independentemente do resultado.

Erros vão acontecer!

Ao executar um código JavaScript, diferentes erros podem ocorrer. Os erros podem ser de codificação feitos pelo programador, erros devido à entrada errada de dados, e outras coisas imprevisíveis.

**Declaração try e catch**

A declaração **try**  permite que você defina um bloco de código para ser testado para erros enquanto ele está sendo executado. A declaração **catch** permite que você defina um bloco de código a ser executado, se ocorrer um erro no bloco try. As instruções JavaScript **try** e **catch** vêm em pares: try {  
    Bloco de código do try}  
catch(err) {  
    Bloco de código para manipular erros}

**Throw (lançar) erros**

Quando ocorre um erro em JavaScript, normalmente para e gera uma mensagem de erro. O termo técnico para isso é: O JavaScript irá **lançar uma exceção (lançar um erro)** . JavaScript vai realmente criar um **objeto de erro** com duas propriedades: **nome** e **mensagem** .

A instrução throw

A declaração **throw** permite-lhe criar um erro personalizado. Técnicamente, você pode **lançar uma exceção (lançar um erro)**. A exceção pode ser uma string JavaScript, um número, um booleano ou um objeto: throw "Too big";    // lança umn texto  
throw 500;          // lança um número

Se você usar **throw** juntamente com **try** e **catch** , você pode controlar o fluxo do programa e gerar mensagens de erro personalizadas.

Exemplo Validação de Entrada

Este exemplo examina uma entrada. Se o valor está errado, uma exceção (err) é lançada. A exceção (err) é apanhada pela instrução catch e uma mensagem de erro personalizada é exibida: <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="message"></p>  
  
<script>  
function myFunction() {  
    var message, x;  
    message = document.getElementById("message");  
    message.innerHTML = "";  
    x = document.getElementById("demo").value;  
    try {   
        if(x == "") throw "empty"; // Retorna vazio  
        if(isNaN(x)) throw "not a number"; // Retorna não é um número  
        x = Number(x);  
        if(x < 5) throw "too low"; // Retorna muito baixo  
        if(x > 10) throw "too high"; // Retorna muito alto  
    }  
    catch(err) {  
        message.innerHTML = "Input is " + err;  
    }  
}  
</script>

**Validação HTML**

O código acima é apenas um exemplo. Navegadores modernos, muitas vezes, usam uma combinação de JavaScript e built-in de validação HTML, usando regras de validação predefinidas definidas nos atributos HTML: <input id="demo" type="number" min="5" max="10" step="1"

**A Declaração finally**

A declaração **finally** permite executar o código, depois do try e catch, independentemente do resultado: try {  
    Bloco de código do try}  
catch(err) {  
    Bloco de código para manipular erros}   
finally {  
    Bloco de código para executar independentemente do resultado try / catch }

function myFunction() {  
    var message, x;  
    message = document.getElementById("message");  
    message.innerHTML = "";  
    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 = "Error: " + err + ".";  
    }  
    finally {  
        document.getElementById("demo").value = "";  
    }  
}

**O objeto Error**

JavaScript foi construído em um objeto de erro que fornece informações de erro quando eles ocorrem. O objeto de erro fornece duas propriedades úteis: nome e mensagem.

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| nome | Define ou retorna um nome de erro |
| mensagem | Define ou retorna uma mensagem de erro (a string) |

**Nomes para os erros**

Seis diferentes valores podem ser devolvidos pela propriedade nome de erro:

|  |  |
| --- | --- |
| **Nome de erro** | **Descrição** |
| EvalError | Ocorreu um erro na função Eval () |
| RangeError | Um número "fora de alcance" ocorreu |
| ReferenceError | Uma referência ilegal ocorreu |
| SyntaxError | Um erro de sintaxe ocorreu |
| TypeError | Um tipo de erro que ocorreu |
| URIError | Um erro na encodeURI () ocorreu |

Erro

Um **EvalError** indica um erro na função Eval(). Versões mais recentes do JavaScript não lançam mais EvalError. Em vez de EvalError use SyntaxError.

Erro Range

Um **RangeError** é lançado se você usar um número que está fora da faixa de valores legais.Por exemplo: Você não pode definir o número significativo de dígitos de um número para 500. var num = 1;  
try {  
    num.toPrecision(500);   // Um número não pode ter 500 dígitos  
}  
catch(err) {  
    document.getElementById("demo").innerHTML = err.name;  
}

Erro de referência

Um **ReferenceError** é lançado se você usa (referencia) uma variável que não tenha sido declarada: var x;  
try {  
    x = y + 1;   // y não pode ser referenciado (usado)  
}  
catch(err) {  
    document.getElementById("demo").innerHTML = err.name;  
}

Erro de sintaxe

Um **SyntaxError** é lançado se você tentar avaliar o código com um erro de sintaxe. try {  
    eval("alert('Hello)");   // () produzirá um erro de sintaxe  
}  
catch(err) {  
    document.getElementById("demo").innerHTML = err.name;  
}

Erro de tipo

Um **TypeError** é lançado se você usar um valor que está fora da gama de tipos esperados: var num = 1;  
try {  
    num.toUpperCase();   // Você não pode converter um número para upper case  
}  
catch(err) {  
    document.getElementById("demo").innerHTML = err.name;  
}

Erro URI

Um **URIError** é lançado se você usar caracteres ilegais em uma função URI: try {  
    decodeURI("%%%");   // Não é possível que URI decodifique esses sinais de porcentagem  
}  
catch(err) {  
    document.getElementById("demo").innerHTML = err.name;  
}

Mozilla e Microsoft(IE) definem algumas propriedades do objeto de erro não-padrão:

* fileName (Mozilla)
* lineNumber (Mozilla)
* columnNumber (Mozilla)
* stack (Mozilla)
* description (Microsoft)
* number (Microsoft)

Não use essas propriedades em sites públicos. Eles não irá funcionar em todos os navegadores.

**DEBUGGING**

**Depuração de código**

Código de programação podem conter erros de sintaxe ou erros lógicos. Muitos destes erros são difíceis de se diagnosticar. Muitas vezes, quando o código de programação conter erros, nada vai acontecer. Não há mensagens de erro, e você naõ vai ter nenhuma indicação de onde procurar erros. Procurar (corrigir) erros no código de programação é chamado de código de depuração.

**Depuradores de JavaScript**

A depuração não é fácil. Mas, felizmente, todos os navegadores modernos têm um built-in depurador JavaScript. Depuradores built-in pode ser ligados e desligados, forçando os erros serem relatados para o usuário.

Com um depurador, você também pode definir pontos de interrupção **(breakpoints)** que são lugares onde a execução do código pode ser parado, e examinar variáveis ​​enquanto o código está sendo executado.

Normalmente, você ativa a depuração no seu navegador com a tecla F12 e então selecione "Console" no menu depurador.

O Método console.log()

Se o seu navegador suporta a depuração, você pode usar **console.log()** para exibir valores de JavaScript na janela do depurador: <!DOCTYPE html>  
<html>  
<body>  
  
<h1>My First Web Page</h1>  
<script>  
a = 5;  
b = 6;  
c = a + b;  
console.log(c);  
</script>  
</body>  
</html>

Definir pontos de interrupção

Na janela do depurador, você pode definir pontos de interrupção no código JavaScript. Em cada ponto de interrupção, JavaScript irá parar a execução, e deixá-lo examinar os valores de JavaScript. Depois de examinar os valores, você pode retomar a execução de código (tipicamente com um botão de reprodução).

A palavra-chave debugging

A palvara-chave **debugging**  pára a execução de JavaScript, e chamadas (se disponível) de funções de depuração. Isto tem a mesma função que definir um ponto de interrupção no depurador. Se nenhum debugging estiver disponível, a declaração debugging não tem efeito. Com o depurador ligado, este código irá interromper a execução antes de executar a terceira linha. var x = 15 \* 5;  
debugger;  
document.getElementById("demo").innerHTML = x;

**Google Chrome**

* Abra o navegador.
* A partir do menu, selecione Ferramentas.
* A partir de ferramentas, escolha ferramentas para desenvolvedores.
* Por fim, selecione Console.

**Firefox Firebug**

* Abra o navegador.
* Vá para a página web:  http://www.getfirebug.com
* Siga as instruções como:  instalar Firebug

**Internet Explorer**

* Abra o navegador.
* A partir do menu, selecione Ferramentas.
* A partir de ferramentas, escolha ferramentas para desenvolvedores.
* Por fim, selecione Console.

**Ópera**

* Abra o navegador.
* Ir para o site: http://dev.opera.com
* Siga as instruções como: adicione um botão Console de desenvolvedor para sua barra de ferramentas.

**Safari Firebug**

* Abra o navegador.
* Ir para o site:   
  http://safari-extensions.apple.com
* Siga as instruções como: instalar Firebug Lite.

**Safari Develop menu**

* Ir para Safari, Preferências, Avançado no menu principal.
* Marque a opção "Ativar menu Mostrar na barra de menu Desenvolver".
* Quando a nova opção "Desenvolver" aparecer no menu:  Escolha "Show Error Console".

**HOISTING**

Hoisting(Elevação) é o comportamento padrão do JavaScript de mover declarações ao topo. As declarações de JavaScript são içadas. Em JavaScript, uma variável pode ser declarada depois de ter sido utilizado. Em outras palavras uma variável pode ser usada antes de ter sido declarada. O **Exemplo 1** retornará o mesmo resultado que o **Exemplo 2**.

**Exemplo 1**: Retorna 5 x = 5; // Atribui 5 ao x  
elem = document.getElementById("demo"); // Encontra um elemento   
elem.innerHTML = x;                     // Exibe x no elemento  
var x; // Declara x

**Exemplo 2** : Retorna 5 var x; // Declara x  
x = 5; // Atribui 5 ao x  
elem = document.getElementById("demo"); // Encontra um elemento  
elem.innerHTML = x;                     // Exibe x no elemento

Para entender isso, você tem que entender o termo "elevação". Elevação é o comportamento padrão do JavaScript de mover todas as declarações ao topo do escopo atual (para o topo do script atual ou a função atual).

**JavaScript Não inicializa apenas eleva**

JavaScript apenas eleva declarações, não as inicializa. O **Exemplo 1** que **não** retornará o mesmo resultado que o **Exemplo 2**.

**Exemplo 1**: Retorna 5 7 var x = 5; // Inicializa x  
var y = 7; // Inicializa y  
elem = document.getElementById("demo"); // Encontra um elemento  
elem.innerHTML = x + " " + y;           // Exibe x no elemento

**Exemplo 2:** Retorna 5 e undefinedvar x = 5; // Inicializa x  
elem = document.getElementById("demo"); // Encontra um elemento  
elem.innerHTML = x + " " + y;           // Exibe x no elemento  
var y = 7; // Inicializa y

Faz sentido que y é indefinido no último exemplo? Isto se deu porque apenas a declaração (var y)foi elevada para o topo e não a sua inicialização (= 7). Por causa da elevação, y foi declarada antes de ser usado, como inicializações não são elevadas, o valor de y ficou indefinido.

No Exemplo 2 mesmo com a declaração da variável elevada e sua inicialização feita após sua chamada o seu valor também será como indefinido: **Exemplo 2:** Retorna 5 e undefined var x = 5; // Inicializa x  
var y;     // Declara y  
elem = document.getElementById("demo"); // Encontra um elemento  
elem.innerHTML = x + " " + y;           // Exibe x no elemento  
y = 7;    // Atribui 7 ao y

**Declarar as variáveis ​​no topo!**

Elevação é (para muitos desenvolvedores) um comportamento desconhecido ou esquecido de JavaScript. Se um desenvolvedor não entende elevação, seus programas podem conter bugs (erros). Para evitar erros, sempre declarar todas as variáveis ​​no início de cada escopo. Uma vez que esta é a forma como JavaScript interpreta o código, é sempre uma boa regra. JavaScript no modo estrito não permite variáveis a serem utilizadas se não forem declarados.

**MODO ESTRITO**

**"use strict**", define que o código JavaScript deve ser executado em "modo estrito". A diretiva "use stric" é nova em JavaScript 1.8.5 (ECMAScript versão 5). Não é uma declaração, mas uma expressão literal, ignorada por versões anteriores do JavaScript. Com o modo estrito, não é possível, por exemplo, usar variáveis ​​não declaradas.

Declarando modo estrito

Modo estrito é declarado pela adição de "use strict"; para o início de um script ou uma função. Declarando no início de um script, ele tem escopo global (todo o código no script será executado no modo estrito): "use strict";  
x = 3.14;       // Isto causará um erro porque x não foi declarado

Exemplo: "use strict";  
myFunction();  
function myFunction() {  
y = 3.14;    // Isto causará um erro porque y não foi declarado  
}

Declarando dentro de uma função, que tem escopo local (somente o código no interior da função é no modo estrito): x = 3.14;       // Isto não causará um erro  
myFunction();  
  
function myFunction() {  
   "use strict";  
    y = 3.14;   // Isto causará um erro  
}

A Sintaxe "use strict"

A sintaxe, para declarar modo estrito, foi projetada para ser compatível com versões mais antigas do JavaScript. Compilando um literal numérico (4 + 5;) ou uma string literal ( "John Doe";) em um programa JavaScript não tem efeitos colaterais. Ele simplesmente compila para uma variável não existente e morre. Assim, "use strict" só importa para novos compiladores que "entender" o significado dela.

**Por que utilizar modo estrito?**

Modo estrito torna mais fácil para escrever JavaScript "seguro". Como exemplo, em JavaScript normal, mistyping é um nome de variável que cria uma nova variável global. No modo estrito, isto irá lançar um erro, o que torna impossível criar acidentalmente uma variável global.

Em JavaScript normal, um desenvolvedor não vai receber qualquer feedback de erro atribuição de valores a propriedades não-graváveis.

No modo estrito, qualquer atribuição a uma propriedade não-gravável, uma propriedade só de getter, uma propriedade não-existente, uma variável não-existente, ou um objeto não-existente, irá lançar um erro.

**Não é permitido no modo estrito**

Usar uma variável sem a declarar, isto não é permitido:

"use strict";  
x = 3.14;                // Isto causará um erro

Usar um objeto sem o declarar, isto não é permitido: "use strict";  
x = {p1:10, p2:20};      // Isto causará um erro

Excluir uma variável (ou objetos) não é permitido: "use strict";  
var x = 3.14;  
delete x;                // Isto causará um erro

Excluir uma função não é permitido: "use strict";  
function x(p1, p2) {};   
delete x;                // Isto causará um erro

Duplicar um nome de parâmetro não é permitido: "use strict";  
var x = 010;             // Isto causará um erro

Caracteres de escape não são permitidos: "use strict";  
var x = \010;            // Isto causará um erro

Escrever em uma propriedade somente de leitura não é permitido: "use strict";  
var obj = {};  
Object.defineProperty(obj, "x", {value:0, writable:false});  
obj.x = 3.14;            // Isto causará um erro

Escrever para uma propriedade somente get não é permitido: "use strict";  
var obj = {get x() {return 0} };  
obj.x = 3.14;            // Isto causará um erro

Deletar uma propriedade undeletable não é permitido: "use strict";  
delete Object.prototype; // Isto causará um erro

A palavra "eval" não pode ser usada como uma variável: "use strict";  
var eval = 3.14;         // Isto causará um erro

A palavra "arguments" não pode ser usada como uma variável: "use strict";  
var arguments = 3.14;    // Isto causará um erro

A declaração "with" não é permitida: "use strict";  
with (Math){x = cos(2)}; // Isto causará um erro

Por razões de segurança, eval() não é permitido para criar variáveis ​​no âmbito da qual foi chamado: "use strict";  
eval ("var x = 2");  
alert (x);               // Isto causará um erro

Cuidado!

A diretiva "use strict" só é reconhecida no início de um script ou uma função.

Modo estrito é suportado em:  IE a partir da versão 10. Firefox a partir da versão 4.   
Chrome a partir da versão 13. Safari a partir da versão 5.1. Opera a partir da versão 12.

Em chamadas de função como f(), este valor era o objeto global. No modo estrito, agora é indefinido. Algumas palavras-chave reservadas não são permitidas no modo estrito, que são:

* implements
* interface
* let
* package
* private
* protected
* public
* static
* yield

"use strict";  
var public = 1500;      // Isto causará um erro

**JAVASCRIPT - GUIA DE ESTILO E CONVEÇÕES**

Convenções de codificação são **diretrizes de estilo para a programação** . Eles geralmente cobrem:

* regras de nomeação e de declaração para variáveis ​​e funções.
* Regras para o uso do espaço em branco, recuo, e comentários.
* Práticas e princípios de programação

Convenções de codificação **de qualidade e segurança** :

* Melhora a legibilidade do código
* Torna a manutenção do código mais fácil

Convenções de codificação podem ser documentadas em regras para as equipes seguirem, ou ser apenas a sua prática de codificação individual.

Nomes de variáveis

Na W3Schools usamos **camelCase** para nomes de identificadores (variáveis e funções). Todos os nomes começam com uma **letra** .

Espaços entre os operadores

Sempre coloque espaços entre operadores (= + - \* /), e depois de vírgulas.

Indentação de código

Sempre usar 4 espaços para o recuo dos blocos de código. Não use abas (Tabulações) para recuo. Diferentes editores interpretam tabs de maneiras diferentes.

Regras de declaração

Regras gerais para declarações simples:

* Sempre terminar uma declaração simples com um ponto e vírgula.

var values = ["Volvo", "Saab", "Fiat"];  
  
var person = {  
    firstName: "John",  
    lastName: "Doe",  
    age: 50,  
    eyeColor: "blue"  
};

Regras gerais de declarações (compostas) complexas:

* Colocar o colchete de abertura na extremidade da primeira linha
* Usar um espaço antes do colchete de abertura
* Coloque o colchete de fechamento em uma nova linha, sem espaços à esquerda
* Não termine uma declaração complexa com um ponto e vírgula

FUNÇÕES: function toCelsius(fahrenheit) {  
    return (5 / 9) \* (fahrenheit - 32);  
}

LOOPS: for (i = 0; i < 5; i++) {  
    x += i;  
}

CONDICIONAIS: if (time < 20) {  
    greeting = "Good day";  
} else {  
    greeting = "Good evening";  
}

Regras de objetos

Regras gerais para definições de objetos:

* Colocar o colchete de abertura na mesma linha que o nome do objeto
* Use : mais um espaço entre cada propriedade e seu valor
* Usar aspas em torno de valores de strings, não em torno de valores numéricos
* Não adicione uma vírgula após o último par da propriedade de valor
* Coloque o colchete de fechamento em uma nova linha, sem espaços à esquerda
* Sempre terminar uma definição de objeto com um ponto e vírgula

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

objetos curtos podem ser escritos comprimidos em uma linha, usando espaços únicos entre as propriedades, assim: var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

Comprimento de linhas

Para facilitar a leitura, evite linhas com mais de 80 caracteres. Se uma instrução JavaScript não se encaixa em uma linha, o melhor lugar para quebrá-lo, é depois de um operador ou uma vírgula: document.getElementById("demo").innerHTML =  
    "Hello Dolly.";

Convenções de nomenclatura

Usar sempre a mesma convenção de nomenclatura para todo o seu código. Por exemplo:

* Nomes de variáveis e funções escritas como **camelCase**
* As variáveis globais escritas em **MAIÚSCULAS**
* Constantes (como PI) escritas em **MAIÚSCULAS**

**Hífens em HTML e CSS:**

Atributos em HTML5 podem conter hífens. CSS usa hífens em nomes de propriedades(font-size). Hífens podem ser confundidos como tentativas de subtração. Hífens não são permitidos em nomes de variáveis em JavaScript. Muitos programadores preferem usar underline(first\_name). Não comece nomes de variáveis com um sinal $. Ele vai colocá-lo em conflito com muitos nomes de biblioteca JavaScript.

Acessando Elementos HTML

A consequência do uso "desordenado" de estilos HTML, pode resultar em erros de JavaScript. Estas duas instruções JavaScript irão produzir resultados diferentes: var obj = getElementById("Demo")  
var obj = getElementById("demo")

Extensões de arquivo

Arquivos HTML devem ter uma extensão**.html** (não **.htm** ).Arquivos CSS deve ter uma extensão**.css** . Arquivos JavaScript deve ter uma extensão .**js** .

Use minúsculas em nomes de arquivos

A maioria dos servidores web (Apache, Unix) são case sensitive sobre nomes dos arquivos:

london.jpg não pode ser acessado como London.jpg.

Outros servidores web (Microsoft, IIS) não se importam com uso de maiúsculas ou minúsculas:

london.jpg pode ser acessado como London.jpg ou london.jpg.

Se você usar uma mistura de letras maiúsculas e minúsculas, você tem que ser extremamente consistente. Se você migra de um servidor case sensitive, para um servidor case insensitive, pequenos erros podem quebrar o seu web site.

Para evitar esses problemas, use sempre nomes de arquivos em minúsculos (se possível).

Performance

Convenções de codificação não são usadas ​​por computadores. A maioria das regras têm pouco impacto sobre a execução de programas. Indentação e espaços extras não são significativos em pequenas scripts. Para um código em desenvolvimento, a legibilidade deve ser preferida. Scripts de produção maiores devem ser minificados.

**MELHORES PRÁTICAS**

**Evite variáveis ​​globais**

Minimize o uso de variáveis ​​globais. Isso inclui tipos todos os dados, objetos e funções. Variáveis ​​e funções globais podem ser substituídos por outros scripts. Use variáveis locais em vez disso, e aprenda a usar encerramentos.

**Sempre declarar variáveis ​​locais**

Todas as variáveis usadas em uma função devem ser declaradas como variáveis **locais** . As variáveis locais **devem** ser declaradas com a palavra-chave **var** , caso contrário, elas vão se tornar variáveis globais. No modo estrito não é permitido o uso de variáveis ​​não declaradas.

**Declarações no topo**

É uma boa prática de codificação colocar todas as declarações no topo de cada script ou função. Isso vai:

* Deixar o código mais limpo
* Fornecer um único lugar para as variáveis ​​locais
* Tornará mais fácil evitar (implícitamente) variáveis ​​globais indesejadas
* Reduzir a possibilidade de re-declarações indesejadas

**Inicializar variáveis**

É uma boa prática de codificação inicializar variáveis ​​ao declará-las. Isso vai:

* Deixar o código mais limpo
* Fornecer um único local para inicializar variáveis
* Evitar valores indefinidos

// Declare e inicialize no topo  
var firstName = "",  
    lastName = "",  
    price = 0,  
    discount = 0,  
    fullPrice = 0,  
    myArray = [],  
    myObject = {};

**Nunca declarar Number, String, Booleanos como objetos**

Sempre tratar números, strings, ou booleanos como valores primitivos. Não como objetos. Declarar estes tipos como objetos, retarda a velocidade de execução, e produz efeitos colaterais desagradáveis: var  x = "John";               
var y = new String("John");  
(x === y) // Falso por que x é uma string e y é um objeto. OU var x = new String("John");               
var y = new String("John");  
(x == y) // Falso por que não podemos comparar objetos.

**Não usar new Object()**

* Use {} em vez de new Object()
* Use "" em vez de new String()
* Use 0 em vez de nova Number()
* Use falsa em vez de nova booleano()
* Use [] em vez de new Array()
* Use / () / em vez de new RegExp()
* Use function () {} em vez de nova função()

var x1 = {};           // novo objeto  
var x2 = "";           // nova string primitiva   
var x3 = 0;            // novo número primitivo  
var x4 = false;        // novo boolean primitivo  
var x5 = [];           // novo objeto array   
var x6 = /()/;         // novo objeto regexp   
var x7 = function(){}; // novo objeto função

**Cuidado com conversões automáticas de tipo**

Tenha atenção onde os números podem acidentalmente serem convertidos em strings ou NaN (não um número). JavaScript é fracamente tipado. Uma variável pode conter diferentes tipos de dados, e um variável pode modificar o seu tipo de dados: var x = "Hello";     // O typeof de x é uma string  
x = 5;               // Modifica o typeof x para um número

var x = 5 + 7;       // x.valueOf() é 12,  typeof de x é um número  
var x = 5 + "7";     // x.valueOf() é 57,  typeof de x é uma string  
var x = "5" + 7;     // x.valueOf() é 57,  typeof de x é uma string  
var x = 5 - 7;       // x.valueOf() é -2,  typeof de x é um número  
var x = 5 - "7";     // x.valueOf() é -2,  typeof de x é um número  
var x = "5" - 7;     // x.valueOf() é -2,  typeof de x é um número  
var x = 5 - "x";     // x.valueOf() é NaN, typeof de x é um número

Subtrair uma string de uma string, não gera um erro, mas retorna NaN (Not a Number): "Hello" - "Dolly"    // retorna NaN

**Comparações**

O operador de comparação == sempre converte (a tipos correspondentes). O operador === força a comparação de valores e tipo: 0 == "";        // true  
1 == "1";       // true  
1 == true;      // true  
  
0 === "";       // falso  
1 === "1";      // falso  
1 === true;     // falso

**Usar padrões de parâmetro**

Se uma função é chamada com um argumento ausente, o valor do argumento ausente é definida como **indefinido** . Valores indefinidos podem quebrar seu código. É um bom hábito atribuir valores padrões para argumentos.

**Terminar o seu Switches com Defaults**

Sempre terminar suas declarações switch com um default. Mesmo se você acha que não há necessidade para isso.

**Evitar usar eval()**

A função eval() é usada para executar texto como código. Em quase todos os casos, não deve ser necessário usá-lo. Porque permite que um código arbitrário seja executado, também representa um problema de segurança.

**ERROS**

**Quebrar uma String JavaScript**

JavaScript permite que você quebre uma declaração em duas linhas: var x =  
"Hello World!";

Mas, quebrar uma instrução no meio de uma string não irá funcionar: var x = "Hello  
World!"

Você deve usar uma "barra invertida" se você quer quebrar a declaração em uma string: var x = "Hello \  
World!";

**Colocar vírgulas em lugares errados**

Por causa de uma vírgula mal colocada, este bloco de código irá executar independentemente do valor de x: if (x == 19);  
{  
    // bloco de código    
}

**Terminando Definições com uma vírgula**

As vírgulas na definição de objetos e de vetores são legais no ECMAScript 5: person = {firstName:"John", lastName:"Doe", age:46,} // Objeto points = [40, 100, 1, 5, 25, 10,]; // Vetor

**O tipo indefinido não é nulo**

Em JavaScript, **nulo** é para objetos, **indefinido** é para variáveis, propriedades e métodos. Para ser nulo, um objeto tem de ser definido, caso contrário irá ser indefinido. Se você quiser testar se existe um objeto, isto irá lançar um erro se o objeto for indefinido: if (myObj !== null && typeof myObj !== "undefined") // Incorreto if (typeof myObj !== "undefined" && myObj !== null) // Correto

**Novos blocos a nível de escopo**

O JavaScript não cria um novo escopo para cada bloco de código. É verdade em muitas linguagens de programação, mas não é verdade em JavaScript. É um erro comum, entre os novos desenvolvedores de JavaScript, acreditar que esse código retorna indefinido: for (var i = 0; i < 10; i++) {  
    // algum código  
}  
return i; // Retorna 10

**DESEMPENHO**

**Reduzir a atividade em Loops**

Loops são frequentemente utilizados em programação. Cada declaração em um loop, incluindo a instrução for, é executada para cada iteração do loop. Declarações ou atribuições que podem ser colocadas fora do loop vão fazer o loop funcionar mais rápido: var i;  
for (i = 0; i < arr.length; i++) { // Maneira ruim O código ruim acessa a propriedade length de uma matriz cada vez que o ciclo é reiterado.

var i; var l = arr.length;  
for (i = 0; i < l; i++) { // Maneira Certa O código certo acessa a propriedade length fora do loop e faz o loop correr mais rápido.

**Reduzir o acesso ao DOM**

Acessar o DOM HTML é muito lento, em comparação com outras instruções JavaScript. Se você espera acessar um elemento DOM várias vezes, acesse-o uma vez e use-o como uma variável local: var obj;  
obj = document.getElementById("demo");  
obj.innerHTML = "Hello";

**Reduzir tamanho do DOM**

Manter o número de elementos no DOM HTML pequeno, irá sempre melhorar o carregamento da página, e acelerar a renderização (exibição da página), especialmente em dispositivos menores. Toda tentativa de procurar o DOM (como getElementsByTagName) se beneficiarão de um DOM menor.

**Evite variáveis ​​desnecessárias**

Não crie novas variáveis ​​se você não planeja salvar valores. Muitas vezes, você pode substituir um código como este: var fullName = firstName + " " + lastName;  
document.getElementById("demo").innerHTML = fullName; POR: document.getElementById("demo").innerHTML = firstName + " " + lastName

**Delay de carregamento**

Colocando seus scripts na parte inferior do corpo da página permitirá que o navegador carregue a página em primeiro lugar. Enquanto um script está baixando, o navegador não irá iniciar quaisquer outros downloads. Além disso todas as atividades de análise e processamento podem ser bloqueados. A especificação HTTP define que os navegadores não devem baixar mais de dois componentes em paralelo. Uma alternativa é usar **defer = "true"** na tag script. O atributo defer especifica que o script deve ser executado após a página ter terminado a análise, mas ele só funciona para scripts externos.

Se possível, você pode adicionar seu script para a página de código, após a página ter sido carregada: <script>  
window.onload = function() {  
    var element = document.createElement("script");  
    element.src = "myScript.js";  
    document.body.appendChild(element);  
};  
</script>

**Evite usar with**

Evite usar o palavra-chave **with**. Ela tem um efeito negativo sobre a velocidade. Ela também atravanca escopos JavaScript. A palavra-chave with **não é autorizada** no modo estrito.