JAVA**SCRIPT – indo além**

Formulários JavaScript

Validações de formulários HTML podem ser feitos por JavaScript. Se um campo de formulário (fname) está vazia, esta função avisa uma mensagem e retorna false, para impedir que o formulário seja submetido: function validateForm() {  
    var x = document.forms["myForm"]["fname"].value;  
    if (x == "") {  
        alert("Name must be filled out");  
        return false;   
    }  
}

Esta função pode ser chamada quando o formulário for ser enviado: <form name="myForm" action="/action\_page\_post.php" **onsubmit="return validateForm()"** method="post">  
Name: <input type="text" name="fname">  
<input type="submit" value="Submit">  
</form>

**JavaScript pode validar entradas numéricas**

JavaScript é usado frequentemente para validar a entrada de dados numéricos: <p>Please input a number between 1 and 10:</p> <input id="numb"> <button type="button" onclick="myFunction()">Submit</button> <p id="demo"></p>

<script> funcion myFunction() { var x, text; // Pega o valor do campo input com o id="numb" x = document.getElementById("numb").value; // Se x for Not a Number ou menor que 1 ou maior que 10 if (isNaN(x) || x < 1 || x > 10) { text = "Input não válido"; } else { text = "Input OK"; } document.getElementById("demo").innerHTML = text; } </script>

**Validação automática de formulário HTML**

Validações de formulários HTML podem ser realizados automaticamente pelo navegador: Se um campo de formulário (nomearq) está vazio, o atributo **rqeuired**  impede este formulário de ser submetido: <form action="/action\_page\_post.php" method="post">  
  <input type="text" name="fname" **required**>  
  <input type="submit" value="Submit">  
</form> // Não funciona no IE9 ou anteriores

**Validação de dados**

A validação de dados é o processo de assegurar que a entrada do utilizador é limpo, correta, e útil. Tarefas de validação típicas são:

* O usuário preencher todos os campos obrigatórios
* O usuário inserir uma data válida
* O usuário digitar um texto em um campo numérico

Na maioria das vezes, o propósito de validação de dados é garantir a entrada do usuário corretamente. A validação pode ser definida por muitos métodos diferentes, e implantada de muitas maneiras diferentes.

**Validação do lado do servidor** é executada por um servidor web, após a entrada ter sido enviada para o servidor.

**Validação do lado do cliente** é realizada por um navegador web, antes da entrada ser enviado para um servidor web.

HTML restrição de validação

HTML5 introduziu um novo conceito de validação HTML chamado **validação de restrição**. Validação de restrição HTML é baseado em:

* Restrição de validação **HTML em atributos de entrada**
* Restrição de validação **CSS com Pseudo seletores**
* Restrição de validação de **métodos e propriedades DOM**

Restrição de validação HTML em atributos de entrada

|  |  |
| --- | --- |
| **Atributo** | **Descrição** |
| disabled | Especifica que o elemento de entrada deve ser desativado |
| max | Especifica o valor máximo de um elemento de entrada |
| min | Especifica o valor mínimo de um elemento de entrada |
| pattern | Especifica o padrão de valor de um elemento de entrada |
| required | Especifica que o campo de entrada requer um elemento |
| type | Especifica o tipo de um elemento de entrada |
| value | Especifica o valor inicial para um campo de entrada |
| readonly | Especifica que o campo de entrada é apenas de leitura (não pode ser alterado) |
| size | Especifica o tamanho (em caracteres) para o campo de entrada |
| maxlength | Especifica o comprimento máximo permitido para o campo de entrada |
| novalidate | Especifica que os dados do formulário não devem ser validados quando submetidos |
| autofocus | Especifica que o campo de entrada deve receber automaticamente o foco quando a página for carregada |
| autocomplete | Especifica se um campo de formulário ou entrada deve ter autocomplete ligado ou desligado. |
|  |  |

Restrição de validação CSS com Pseudo seletores

|  |  |
| --- | --- |
| **Seletor** | **Descrição** |
| :disabled | Seleciona elementos de entrada com o atributo "disabled" especificado |
| :invalid | Seleciona elementos de entrada com valores inválidos |
| :optional | Seleciona elementos de entrada sem atributo "obrigatório" especificado |
| :required | Seleciona elementos de entrada com o atributo "required" especificado |
| :valid | Seleciona elementos de entrada com valores válidos |

Restrição de validação de métodos DOM

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| checkValidity() | Retorna verdadeiro se um elemento de entrada contiver dados válidos. |
| setCustomValidity() | Define a propriedade validationMessage de um elemento de entrada. |

Restrição de validação de propriedades DOM

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| validity | Contém propriedades booleanas relacionadas à validade de um elemento de entrada |
| validationMessage | Contém a mensagem que um navegador exibirá quando a validade for falsa |
| willValidate | Indica se um elemento de entrada será validado. |

Propriedades de validade

A **propriedade validade** de um elemento de entrada contém um número de propriedades relacionadas com a validade dos dados:

|  |  |
| --- | --- |
| **Propriedade** | **Descrição** |
| customError | Definir como verdadeiro, se uma mensagem de validade personalizada estiver definida. |
| patternMismatch | Define como true, se o valor de um elemento não corresponder ao seu atributo de padrão. |
| rangeOverflow | Define como true, se o valor de um elemento for maior do que seu atributo máximo. |
| rangeUnderflow | Define como true, se o valor de um elemento for menor que o atributo min. |
| stepMismatch | Define como true, se o valor de um elemento for inválido por seu atributo de etapa. |
| tooLong | Definide como true, se o valor de um elemento exceder seu atributo maxLength. |
| typeMismatch | Define como true, se o valor de um elemento for inválido por seu atributo de tipo. |
| valueMissing | Define como true, se um elemento (com um atributo requerido) não tem valor. |
| valid | Define como true, se o valor de um elemento for válido. |

Objetos JavaScript

Em JavaScript, os objetos são reis. Se você entender objetos, você entenderá JavaScript. Um objeto JavaScript é um conjunto de **valores nomeados**. Os objetos JavaScript são recipientes para valores nomeados, chamados propriedades e métodos. Em JavaScript, quase "tudo" é um objeto:

* Booleanos podem ser objetos (se definido com a palavra-chave **new** )
* Os números podem ser objetos (se definido com a palavra-chave **new** )
* Strings podem ser objetos (se definido com a palavra-chave **new** )
* As datas são sempre objetos
* Maths são sempre objetos
* As expressões regulares são sempre objetos
* Arrays são sempre objetos
* Funções são sempre objetos
* Objetos são sempre objetos

Todos os valores JavaScript, exceto primitivos, são objetos.

Valores primitivos JavaScript

Um **valor primitivo** é um valor que não tem propriedades ou métodos. Um **tipo de dado primitivo** são dados que tem um valor primitivo. JavaScript define 5 tipos de tipos de dados primitivos:

* Stri ng
* Number
* Boolean
* Null
* Undefined

Valores primitivos são imutáveis ​​(eles são codificados e, portanto, não podem ser alterados). Se **x = 3,14**, você pode alterar o valor de x. Mas você não pode alterar o valor de 3,14.

|  |  |  |
| --- | --- | --- |
| **Propriedade** | **Tipo** | **Valor** |
| "Hello" | string | "Olá" é sempre "Olá" |
| 3.14 | number | 3,14 é sempre 3,14 |
| true | boolean | verdadeiro é sempre verdadeiro |
| false | boolean | falso é sempre falso |
| null | null (objeto) | nulo é sempre nulo |
| undefined | undefined | indefinido é sempre indefinido |

Objetos são variáveis ​​contendo variáveis

Variáveis ​​JavaScript pode conter valores únicos: var person = "John Doe"; Objetos são variáveis ​​também. Mas os objetos podem conter muitos valores. Os valores são escritos em  pares **nome: valor** (nome e o valor separados por dois pontos). var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

Propriedades de objeto

Os valores nomeados, em objetos JavaScript, são chamados de **propriedades**.

|  |  |
| --- | --- |
| **Propriedade** | **Valor** |
| firtsName | John |
| lastName | Doe |
| age | 50 |
| eyeColor | azul |

As propriedades são a parte mais importante de qualquer objeto JavaScript. As propriedades são os valores associados a um objeto JavaScript. Um objeto JavaScript é um conjunto de propriedades não ordenadas. Propriedades geralmente podem ser mudadas, adicionadas e excluídas, mas algumas são somente de leitura.

A sintaxe para acessar a propriedade de um objeto é: *objectName.property*// person.age ou

*objectName*["*property*"]       // person["age"]

ou

*objectName*[*expression*]       // x = "age"; person[x]

A expressão deve ser avaliada como um nome de propriedade.

**Loop for dentro de um objeto**

O loop for pode percorrer as propriedades de um objeto, da seguinte maneira: for (*variável* do *objeto*) {  
*código a ser executado*  
}

O bloco de código dentro do loop for será executado uma vez para cada propriedade:

<p id="demo"></p> <script>

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

document.getElementById("demo").innerHTML = txt; // Retornará John Doe 25

**Adicionando Novas propriedades**

Você pode adicionar novas propriedades a um objeto existente, simplesmente dando-lhe um valor. Suponha que o objeto pessoa já existe - você pode, então, dar-lhe novas propriedades: person.nationality = "English";

**Excluindo Propriedades**

A palavra-chave delete exclui uma propriedade de um objeto: var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};  
delete person.age;   // Retornará age undefined

A palavra-chave delete exclui o valor da propriedade e a própria propriedade. Após a exclusão, a propriedade não pode ser usada antes de ser adicionada de volta.

O operador delete é projetado para ser usado em propriedades de objetos. Não tem nenhum efeito sobre variáveis ​​ou funções.O operador delete não deve ser usado em propriedades de objeto JavaScript predefinidas. Ele pode travar a sua aplicação. Atributos de propriedade

Todas as propriedades têm um nome. Além disso, elas também têm um valor. O valor é um dos atributos da propriedade. Outros atributos são: enumeráveis, configuráveis, e graváveis.

Estes atributos definem como a propriedade pode ser acessada(é legível ?, é gravável?). Em JavaScript, todos os atributos podem ser lidos, mas apenas o atributo de valor pode ser alterado (e somente se a propriedade for gravável).

(ECMAScript 5 tem métodos, tanto para obter e definir todos os atributos de propriedade)

Objetos JavaScript herdam as propriedades de seu protótipo. A palavra-chave delete não exclui propriedades herdadas, mas se você excluir uma propriedade protótipo, isto irá afetar todos os objetos herdados do protótipo.

Métodos de objeto

Os métodos são **ações** que podem ser executadas em objetos. Propriedades de objetos poderão ser os valores primitivos, outros objetos, ou funções. Um **método de objeto** é uma propriedade do objeto contendo uma **definição de função** .

|  |  |
| --- | --- |
| **Propriedade** | **Valor** |
| firtsName | John |
| lastName | Doe |
| age | 50 |
| eyeColor | azul |
| fullName | function() {return this.firstName + "" + this.lastName;} |

**Acessando Métodos de objeto**

Você cria um método de objeto com a seguinte sintaxe:

*methodName : function() { linhas de código }*

Você acessa um método de objeto com a seguinte sintaxe:

*objectName.methodName()*

Você normalmente irá descrever **fullName()** como um método do objeto pessoa, e **fullName** como uma propriedade. A propriedade fullName executará (como uma função) quando ele é chamado com **()**.

Este exemplo acessa o **método** fullName() de um objeto pessoa: name = person.fullName();

Se você acessar o fullName **propriedade** , sem (), ele irá retornar a **definição da função**: name = person.fullName;

**Usando métodos internos**

Este exemplo usa o método **toUpperCase()** do objeto String, para converter um texto em maiúsculas: var message = "Hello world!";  
var x = message.toUpperCase(); // Retornará “HELLO WORLD”

**Adicionando novos métodos**

Adicionar métodos a um objeto é feito dentro da função construtora: function person(firstName, lastName, age, eyeColor) {  
    this.firstName = firstName;    
    this.lastName = lastName;  
    this.age = age;  
    this.eyeColor = eyeColor;  
    this.changeName = function (name) {  
        this.lastName = name;  
    };  
}

A função **changeName()** atribui o valor do nome à propriedade lastName da pessoa.

<p id="demo"></p>

<script>

function person(firstName,lastName,age,eyeColor) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.eyeColor = eyeColor;

this.changeName = function (name) {

this.lastName = name;

}

}

var myMother = new person("Sally","Rally",48,"green");

myMother.changeName("Doe");

document.getElementById("demo").innerHTML =

"My mother's last name is " + myMother.lastName;

</script>

Criando um objeto JavaScript

Com JavaScript, você pode definir e criar seus próprios objetos. Há maneiras diferentes para criar novos objetos:

* Definir e criar um único objeto, utilizando um objeto literal.
* Definir e criar um único objeto, com a palavra chave **new.**
* Definir um construtor de objeto, e, em seguida, criar objetos do tipo construído.

Em ECMAScript 5, um objeto pode também ser criado com a função **Object.create().**

**Usando um objeto Literal**

Esta é a maneira mais fácil de criar um objeto JavaScript. Usando um objeto literal, você define e cria um objeto em uma declaração. Um objeto literal é uma lista de pares nome: valor (como idade: 50) dentro de chaves {}. O exemplo a seguir cria um novo objeto JavaScript com quatro propriedades: var person = {  
    firstName:"John",  
    lastName:"Doe",  
    age:50,  
    eyeColor:"blue"  
};

**Usando a palavra-chave new**

O exemplo que se segue, também cria um novo objeto JavaScript com quatro propriedades: var person = new Object();  
person.firstName = "John";  
person.lastName = "Doe";  
person.age = 50;  
person.eyeColor = "blue";

Os dois exemplos acima fazem exatamente a mesma. Não há necessidade de usar new Object(). Por simplicidade, legibilidade e velocidade de execução, use o primeiro (o método literal de objeto).

**Usando um construtor de objeto**

Os exemplos acima são limitados em muitas situações. Eles só criam um único objeto. Às vezes, gostamos de ter um "tipo de objeto" que possa ser usado para criar muitos objetos de um tipo. A maneira padrão para criar um "tipo de objeto" é a utilização de uma função construtora de objeto: function person(first, last, age, eye) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eye;  
}  
var myFather = new person("John", "Doe", 50, "blue");  
var myMother = new person("Sally", "Rally", 48, "green");

A função acima (person) é um construtor de objeto. Depois de ter um construtor do objeto, você pode criar novos objetos do mesmo tipo: var myFather = new person("John", "Doe", 50, "blue");  
var myMother = new person("Sally", "Rally", 48, "green");

**A palavra-chavethis**

Em JavaScript, a coisa chamada **this** , é o objeto que "possui" o código JavaScript. O valor do **this** , quando usado em uma função, significa o objeto que "possui" a função. O valor do **this** , quando usado em um objeto, é o próprio objeto.

A palavra-chave **this**  em um construtor de objeto não tem um valor. É apenas um substituto para o novo objeto. O valor de **this** se tornará o novo objeto quando o construtor for usado para criar um objeto. Note-se que **this** não é uma variável. É uma palavra-chave. Você não pode alterar o valor do **this** .

**Construtores JavaScript Built-in**

JavaScript tem construtores internos para objetos nativos: var x1 = new Object();    // Um novo objeto Object   
var x2 = new String();    // Um novo objeto String   
var x3 = new Number();    // Um novo objeto Number   
var x4 = new Boolean();   // Um novo objeto Boolean  
var x5 = new Array();     // Um novo objeto Array   
var x6 = new RegExp();    // Um novo objeto RegExp  
var x7 = new Function();  // Um novo objeto Function   
var x8 = new Date();      // Um novo objeto Date

O objeto **Math()** não está na lista. Math é um objeto global. A palavra-chave new não pode ser usada em matemática.

Como você pode ver, JavaScript tem objeto com versões dos tipos de dados primitivos String, número e booleanas. Não há nenhuma razão para criar objetos complexos, valores primitivos comprovadamente executam o código muito mais rápido.

Não há razão para usar new **Array()**. Use literais de vetores em seu lugar: [] Não há razão para usar new **RegExp()**. Use literais padrão em seu lugar: / () / Não há razão para usar a new **function()**. Use expressões de função em seu lugar: function(){} E Não há razão para usar new **Object()**. Use literais de objeto em seu lugar: {}

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

**Objetos JavaScript são mutáveis**

Objetos são mutáveis: Eles são abordados por referência, não por valor. Se pessoa é um objeto, a seguinte declaração não vai criar uma cópia da pessoa: var x = pessoa;  // Isto não criará uma cópia de pessoa

O objeto x é **não uma cópia** de pessoa. Ele **é** a pessoa. Ambos x e pessoa são o mesmo objeto. Quaisquer alterações à x também mudará pessoa, porque x e pessoa são o mesmo objeto.

var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}  
  
var x = person;  
x.age = 10;           // Isto mudará tanto x.age como person.age

Protótipos de objetos

Cada objeto JavaScript tem um protótipo. O protótipo também é um objeto. Todos os objetos JavaScript herdam suas propriedades e métodos do seu protótipo. Objetos criados usando um objeto literal, ou com new Object(), herdam de um protótipo chamado Object.prototype. Objetos criados com new Date() herdam o Date.prototype. O Object.prototype está no topo da cadeia de protótipos.

Todos os objetos JavaScript (Date, Array, RegExp, função, ....) herdam da Object.prototype.

**Criação de um protótipo**

A forma padrão para criar um protótipo de objeto é usar uma função construtora de objeto: function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
}

Com uma função construtora, você pode usar a palavra-chave **new** para criar novos objetos a partir do mesmo protótipo: var myFather = new Person("John", "Doe", 50, "blue");  
var myMother = new Person("Sally", "Rally", 48, "green");

A função construtora é o protótipo de objetos Person.   
É considerada um boa prática a função construtora ter o nome com a primeira letra maiúscula.

**Adicionando propriedades e métodos para objetos**

Às vezes você quer adicionar novas propriedades (ou métodos) a um objeto existente. Às vezes você quer adicionar novas propriedades (ou métodos) para todos os objetos existentes de um determinado tipo. Às vezes você quer adicionar novas propriedades (ou métodos) para um protótipo de objeto.

Adicionar uma nova propriedade para um objeto existente é fácil: myFather.nationality = "English";

Adicionar um novo método para um objeto existente também é fácil: myFather.name = function () {  
    return this.firstName + " " + this.lastName;  
};

Não é possível adicionar uma nova propriedade para um protótipo da mesma forma que você adiciona uma nova propriedade a um objeto existente, porque o protótipo não é um objeto existente: Person.nationality = "English"; // Error

Para adicionar uma nova propriedade para um protótipo, você deve adicioná-lo à função construtora: function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
    this.nationality = "English";  
}

Propriedades de protótipo podem ter valores Prototype (valores padrão).

**Adicionando métodos a um protótipo**

Sua função construtora também pode definir métodos: function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
    this.name = function() {return this.firstName + " " + this.lastName;};  
}

Usando a propriedade **protótipo**

A propriedade protótipo permite adicionar novas propriedades a um protótipo existente: function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
}  
Person.prototype.nationality = "English";

A propriedade protótipo também permite adicionar novos métodos para um protótipo existente. Apenas modificar seus **próprios** protótipos. Nunca modifique os protótipos de objetos JavaScript padrão: function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
}  
Person.prototype.name = function() {  
    return this.firstName + " " + this.lastName;  
};

Funções JavaScript

Funções em JavaScript são **definidas** com a palavra-chave **function** . Você pode usar uma declaração de função ou uma expressão de função.

**Declarações de função**

Funções declaradas não são executadas imediatamente. Elas são "salvas para uso posterior", e serão executadas mais tarde, quando elas forem invocadas (chamadas). function myFunction(a, b) {  
    return a \* b;  
}

Vírgulas são usadas para separar instruções JavaScript executáveis.   
Uma vez que uma declaração de função não é uma declaração executável, não é comum terminar com um ponto e vírgula.

**Expressões de função**

Uma função JavaScript também pode ser definida utilizando uma **expressão**. Uma expressão de função pode ser armazenada em uma variável:

var x = function (a, b) { return a \* b };

Depois que uma expressão de função for armazenada em uma variável, a variável pode ser usada como uma função: var x = function (a, b) { return a \* b };  
var z = x(4, 3); // Retorna 12

A função acima é na verdade uma **função anônima** (uma função sem um nome). Funções armazenadas em variáveis ​​não precisam de nomes. Elas são sempre invocadas (chamadas) usando o nome da variável.

A função acima termina com um ponto e vírgula, porque é uma parte de uma instrução executável.

**Function() construtora**

Como você viu nos exemplos anteriores, funções JavaScript são definidas com a palavra-chave **function**. As funções também podem ser definidas com um construtor de funções JavaScript embutido chamado **Function().** var myFunction = new Function("a", "b", "return a \* b");  
var x = myFunction(4, 3);

Não é necessário usar o construtor function(). O exemplo acima é o mesmo que escrever: var myFunction = function (a, b) {return a \* b};  
var x = myFunction(4, 3);

**Elevação de função**

Anteriormente neste tutorial, você aprendeu sobre "elevação". Elevação é o comportamento padrão do JavaScript de mover **declarações** ao topo do escopo atual. elevar aplica-se a declarações de variáveis ​​e declarações de função.

Devido a isso, as funções de JavaScript pode ser chamadas antes que eles sejam declaradas: myFunction(5);  
 function myFunction(y) {  
    return y \* y;  
}

Funções definidas que utilizam uma expressão não são elevadas.

**Funções auto-invocadas**

As expressões de função podem se "auto-invocar". A expressão auto-invocada é iniciada automaticamente, sem ser chamada. As expressões de função irão executar automaticamente se a expressão for seguida por (). Você não pode auto-invocar uma declaração de função.

Você tem que adicionar parênteses em torno da função para indicar que é uma expressão de função: (function () {  
    var x = "Hello!!";      // Auto-invacada  
})();

**As funções usadas como valores**

Funções de JavaScript podem ser usadas como valores: function myFunction(a, b) {  
    return a \* b;  
}  
var x = myFunction(4, 3);

Funções de JavaScript podem ser usadas em expressões: function myFunction(a, b) {  
    return a \* b;  
}  
var x = myFunction(4, 3) \* 2;

**Funções são objetos**

O operador **typeof**  em JavaScript retorna "function" para as funções. Mas, funções JavaScript podem ser melhor descritas como objetos. Funções JavaScript têm **propriedades** e **métodos** . A propriedade **arguments.length** retorna o número de argumentos recebidos quando a função for invocada: function myFunction(a, b) {  
    return arguments.length; // Retorna 2  
}

O método **toString()** retorna a função como uma string: function myFunction(a, b) {  
    return a \* b;  
}  
var txt = myFunction.toString(); // Retorna function myFunction(a, b){ return a \* b;}

Uma função definida como uma propriedade de um objeto, é chamada de método para o objeto. Uma função projetada para criar novos objetos, é chamada de construtora do objeto.

Parâmetros e Argumentos de função

**Parâmetros** de Função são os **nomes** listados na definição da função. A**rgumentos** de função  são os verdadeiros **valores** passados para (e recebidas por) a função.

**Regras de parâmetros**

As definições de função JavaScript não especificam tipos de dados para parâmetros. As funções de JavaScript não executam verificação de tipo nos argumentos passados. As funções não conferem o número de argumentos recebidos.

**Padrões de parâmetro**

Se uma função é chamada com **argumentos ausentes** (menos do que os declarados), os valores em falta são definidas como: **undefined**

Às vezes, isso é aceitável, mas às vezes é melhor para atribuir um valor padrão para o parâmetro: function myFunction(x, y) {  
    if (y === undefined) {  
          y = 0;  
    }   
}

Se uma função é chamada com muitos argumentos (mais do que os declarados), esses argumentos podem ser alcançados usando o objeto **arguments**.

**O objeto arguments**

Funções JavaScript tem um objeto built-in chamado objeto de argumentos.

O objeto **arguments** contém uma matriz dos argumentos usados ​​quando a função for chamada. Desta forma, você pode simplesmente usar uma função para encontrar (por exemplo) o maior valor em uma lista de números: x = findMax(1, 123, 500, 115, 44, 88);  
function findMax() {  
    var i;  
    var max = -Infinity;  
    for (i = 0; i < arguments.length; i++) {  
        if (arguments[i] > max) {  
            max = arguments[i];  
        }  
    }  
    return max;  
} document.getElementById("demo").innerHTML = findMax(4, 5, 6); // 6

Ou criar uma função para somar todos os valores de entrada: x = sumAll(1, 123, 500, 115, 44, 88);  
function sumAll() {  
    var i, sum = 0;  
    for (i = 0; i < arguments.length; i++) {  
        sum += arguments[i];  
    }  
    return sum;  
} document.getElementById("demo").innerHTML = sumAll(1, 123, 500, 115, 44, 88); // 871

**Argumentos são passados ​​por valor**

Os parâmetros, em uma chamada de função, são argumentos da função. Argumentos JavaScript são passados por **valor**. A função só começa a conhecer os valores, e não os locais dos argumentos. Se a função alterar o valor de um argumento, isso não mudará o valor original do parâmetro. **Alterações nos argumentos não ficam visíveis (refletidos) fora da função.**

**Objetos são passados ​​por referência**

Em JavaScript, referências de objeto são valores. Por causa disso, os objetos se comportarão como se fossem passados por **referência**. Se uma função altera uma propriedade do objeto, ela altera o valor original. **Alterações nas propriedades do objeto são visíveis fora da função.**

**Invocando uma função JavaScript**

O código dentro de uma função não é executada quando a função é **definida** . O código dentro de uma função é executado quando a função é **invocada** . É comum usar o termo " **chamar uma função** " em vez de " **invocar uma função** ". Também é comum dizer "invocar uma função", "iniciar uma função", ou "executar uma função".

Lembrando que uma função JavaScript pode ser invocada sem ser chamada.

**Invocando uma função como uma função**

function myFunction(a, b) {  
    return a \* b;  
}  
myFunction(10, 2);           // Retorna 20

A função acima não pertence a qualquer objeto. Mas em JavaScript há sempre um objeto global padrão. Em HTML o objeto global padrão é a página HTML em si, de modo que a função acima "pertence" à página HTML. Em um navegador, o objeto da página é a janela do navegador. A função acima torna-se automaticamente uma função de janela. ***minhaFuncao () e window.myFunction () é a mesma função:***

function myFunction(a, b) {  
    return a \* b;  
}  
window.myFunction(10, 2);    // Retorna 20

Esta é uma forma comum de chamar uma função JavaScript, mas não é uma prática muito boa. Variáveis, métodos ou funções globais podem facilmente criar conflitos e erros de nomes no objeto global.

A palavra-chave  ***this***

Em JavaScript, a coisa chamada **this** , é o objeto que "possui" o código atual. O valor this, quando usado em uma função, é o objeto que "possui" a função. Note que **this** não é uma variável. É uma palavra-chave. Você não pode alterar o valor do **this** .

O objeto global

Quando uma função é chamada sem um objeto proprietário, o valor do **this** se torna o objeto global. Em um navegador web o objeto global é a janela do browser. Este exemplo retorna o objeto de janela como o valor do **this**: function myFunction() {  
    return this;  
}  
myFunction();                // Retorna o objeto janela

Invocando uma função como uma função global, faz com que o valor do **this** vire o objeto global. Usar o objeto de janela como uma variável pode facilmente travar o seu programa.

**Invocar uma função como um método**

Em JavaScript você pode definir a função como métodos de objeto. O exemplo a seguir cria um objeto ( **myObject** ), com duas propriedades ( **firstName** e **lastName** ) e o método ( **fullName** ): var myObject = {  
    firstName:"John",  
    lastName: "Doe",  
    fullName: function () {  
        return this.firstName + " " + this.lastName;  
    }  
}  
myObject.fullName();         // Retorna "John Doe"

O  método **fullName** é uma função. A função pertence ao objeto **myObject** que é o dono da função. O **this** , é o objeto que "possui" o código JavaScript. Neste caso, o valor do **this** é **myObject** . Teste alterar o **fullName** método para retornar o valor do **this**: var myObject = {  
    firstName:"John",  
    lastName: "Doe",  
    fullName: function () {  
        return this;  
    }  
}  
myObject.fullName();  // Retorna [object Object] (o próprio objeto)

Invocar uma função como um método de objeto, faz com que o valor do **this** seja o próprio objeto.

**Chamar uma função com um construtor Function**

Se uma chamada de função é precedida com a palavra-chave **new** , isto está invocacando um construtor. Parece que você cria uma nova função, mas, como as funções JavaScript são objetos, você realmente cria um novo objeto: // Isto é uma função construtora:  
function myFunction(arg1, arg2) {  
    this.firstName = arg1;  
    this.lastName  = arg2;  
}  
// Isto cria um novo objeto  
var x = new myFunction("John", "Doe");  
x.firstName;                             // Retorna "John"

Uma invocação do construtor cria um novo objeto. O novo objeto herda as propriedades e os métodos do seu construtor. A palavra-chave **this**  no construtor não tem um valor. O valor do **this** será o novo objeto criado quando a função for invocada.

**Funções são métodos de objeto**

Em JavaScript funções são métodos de objetos. Se uma função não é um método de um objeto de JavaScript, é uma função do objeto global. O exemplo a seguir cria um objeto com 3 propriedades ( **firstName** , **lastName** , **fullName** ): var person = {  
    firstName:"John",  
    lastName: "Doe",  
    fullName: function () {  
        return this.firstName + " " + this.lastName;  
    }  
}  
person.fullName();         // Retorna "John Doe"

A propriedade fullName é um **método** . O objeto pessoa é o **proprietário** do método. A propriedade fullName é um **método do objeto pessoa** .

**O método call()**

O método **call()** é uma função JavaScript predefinida. Ele pode ser usado para chamar (call) uma função com um objeto proprietário como o primeiro argumento (parâmetro). Com **call()** , você pode usar um método que pertence a outro objeto. Este exemplo chama a função fullName de pessoa, mas é usá-lo em myObject: var person = {  
    firstName:"John",  
    lastName: "Doe",  
    fullName: function() {  
        return this.firstName + " " + this.lastName;  
    }  
}  
var myObject = {  
    firstName:"Mary",  
    lastName: "Doe",  
}  
person.fullName.call(myObject);  // Retorna "Mary Doe"

**O método aplly()**

O método **aplly()** é semelhante ao método de call(): var person = {  
    firstName:"John",  
    lastName: "Doe",  
    fullName: function() {  
        return this.firstName + " " + this.lastName;  
    }  
}  
var myObject = {  
    firstName:"Mary",  
    lastName: "Doe",  
}  
person.fullName.apply(myObject);  // Retorna "Mary Doe"

A diferença entre call() e aplly()

A única diferença é: call() leva quaisquer argumentos de função separadamente, aplly() converte quaisquer argumentos da função como um vetor. O método aplly() é muito útil se você quiser usar um vetor em vez de uma lista de argumentos. Se você quiser obter o maior número numa lista de números você pode usar o método Math.max():Math.max(1,2,3);  // Retorna 3

Uma vez que os arrays de JavaScript não possuem um método **max()**, você pode aplicar **Math.max()** em vez disso: Math.max.apply(null,[1,2,3]); // Também retorna 3

**O valor this**

No modo JavaScript estrito, o primeiro argumento torna-se o valor de **this** na função chamada, mesmo se o argumento não é um objeto. No modo "não-estrito", se o valor do primeiro argumento é nulo ou indefinido, ele é substituído com o objeto global.

**Variáveis ​​globais**

Uma função pode acessar todas as variáveis definidas **dentro** da função, como esta: function myFunction() {  
    var a = 4;  
    return a \* a;  
} // Retorna 16

Mas uma função também pode acessar as variáveis definidas **fora** da função, como esta:

var a = 4;  
function myFunction() {  
    return a \* a;  
} // Retorna 16

No último exemplo, **a** é uma variável **global**. Em uma página web, as variáveis ​​globais pertencem ao objeto de janela. As variáveis ​​globais podem ser utilizadas (e alteradas) por todos os scripts na página (e na janela).

No primeiro exemplo, **a** é uma variável **local** . Uma variável local só pode ser usada dentro da função onde está definido. Ele está escondida de outras funções e outros códigos.

Variáveis ​​globais e locais com o mesmo nome são variáveis ​​diferentes. Modificar uma, não modifica a outra. Variáveis criadas **sem** a palavra-chave **var** , são sempre globais, mesmo se elas são criadas dentro de uma função.

**Duração de variáveis**

As variáveis ​​globais vivem tanto tempo quanto a sua aplicação (sua janela / sua página web) vive. As variáveis ​​locais têm vida curta. Elas são criadas quando a função é chamada, e excluídas quando a função for encerrada.

**O Dilema do contador**

Suponha que você queira usar uma variável para contar algo, e você quer que este contador esteja disponível para todas as funções. Você poderia usar uma variável global, e uma função de aumentar o contador: var counter = 0;  
function add() {  
    counter += 1;  
}  
add();  
add();  
add(); // O contador funciona

O contador só deve ser alterado pela função **add()**. O problema é que qualquer script na página pode alterar o contador, sem chamar **add().** Se eu declarar o contador dentro da função, ninguém será capaz de mudá-lo sem chamar **add()**: function add() {  
    var counter = 0;  
    counter += 1;  
}  
add();  
add();  
add(); // O contador agora só vai a 1, não faz o incremento!

Não funcionou! Toda vez que eu chamar a função **add()**, o contador é definido como 1. A função interna JavaScript pode resolver isso.

**Funções JavaScript aninhadas**

Todas as funções têm acesso ao escopo global. De fato, em JavaScript, todas as funções têm acesso ao escopo "acima" deles. JavaScript suporta funções aninhadas, funções aninhadas têm acesso ao escopo "acima" deles. Neste exemplo, a função interior **plus()** tem acesso a variável **contador** na função pai: function add() {  
    var counter = 0;  
    function plus() {counter += 1;}  
    plus();      
    return counter;   
} // Retorna 1

Isso poderia ter resolvido o dilema do contador, se pudéssemos alcançar a função plus() do lado de fora da função. Nós também precisamos encontrar uma maneira de executar o **contador = 0** apenas uma vez. Precisamos de um encerramento: var add = (function () {  
    var counter = 0;  
    return function () {return counter += 1;}  
})();  
add();  
add();  
add(); // O contador funciona

A variável add atribui o valor de retorno de uma função auto-invocada. A função auto-invocada é executada apenas uma vez. Ela define o contador como zero (0) e retorna uma expressão de função. Dessa forma, o add torna-se uma função. A parte "maravilhosa" é que ela pode acessar o contador no escopo dos pais. Isso é chamado de cerramento JavaScript. Isso permite que uma função tenha variáveis "privadas". O contador é protegido pelo escopo da função anônima e só pode ser alterado usando a função add. Um cerramento é uma função que tem acesso ao escopo pai, mesmo após a função principal ter sido encerrada.

JavaScript HTML DOM

Através do DOM HTML, o JavaScript pode acessar e alterar todos os elementos de um documento HTML. Quando uma página da web é carregada, o navegador cria um **D**ocument **O**bject  **M**odel da página. O modelo é construído como uma árvore de **objetos**.



Com o **DOM**, o JavaScript recebe toda a energia de que necessita para criar HTML dinâmico:

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

O que é o DOM?

O DOM é um padrão W3C (World Wide Web Consortium). O DOM define um padrão para aceder aos documentos: *"O modelo de objeto de W3C DOM (Document) é uma interface de plataforma e de linguagem neutra que permite que programas e scripts acessem e atualizem o conteúdo, estrutura e estilo de um documento dinamicamente."*

O padrão W3C DOM é separado em 3 partes diferentes:

* DOM núcleo - modelo padrão para todos os tipos de documentos
* DOM XML - modelo padrão para documentos XML
* HTML DOM - modelo padrão para documentos HTML

O que é o DOM HTML?

O HTML DOM é um modelo de objeto padrão e uma interface de programação para HTML:

* Define os elementos HTML como **objetos**
* Define as **propriedades** de todos os elementos HTML
* Define os **métodos** para acessar todos os elementos HTML
* Define os **eventos** para todos os elementos HTML

Em outras palavras: **O HTML DOM é um padrão para como obter, alterar, adicionar ou excluir elementos HTML.**

**Métodos HTML DOM**

Métodos HTML DOM são **ações** que podem ser executadas (em Elementos HTML). Propriedades HTML DOM são **valores** (de elementos HTML) que você pode definir ou alterar.

**A interface de programação DOM**

O DOM HTML pode ser acessado com JavaScript (e com outras linguagens de programação). No DOM, todos os elementos HTML são definidos como **objetos** . A interface de programação são as propriedades e métodos de cada objeto. Uma **propriedade** é um valor que você pode obter ou definir (como alterar o conteúdo de um elemento HTML). Um **método** é uma ação que você pode fazer (como adicionar ou excluir um elemento HTML). <p id="demo"></p>  
<script>  
document.getElementById("demo").innerHTML = "Hello World!"; </script>

No exemplo acima, getElementById é um **método** , enquanto InnerHTML é uma **propriedade**. A forma mais comum de acessar um elemento HTML é usar o id do elemento. No exemplo acima, o método getElementById usado id = "**demo**" para encontrar o elemento.

A maneira mais fácil de obter o conteúdo de um elemento é usando o **innerHTML**  propriedade. A propriedade innerHTML é útil para obter ou substituir o conteúdo de elementos HTML. A propriedade innerHTML pode ser usado para obter ou alterar qualquer elemento HTML, incluindo <html> e <body>.

**O Objeto de Documento HTML DOM**

O objeto de documento HTML DOM é o proprietário de todos os outros objetos em sua página web. O objeto do documento representa sua página web. Se você quiser acessar qualquer elemento em uma página HTML, você sempre começa a acessar o objeto do documento. Abaixo estão alguns exemplos de como você pode usar o objeto do documento HTML:

Encontrar elementos HTML

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| document.getElementById(*id*) | Encontre um elemento pelo id do elemento |
| document.getElementsByTagName(*name*) | Encontre elementos pelo nome da tag |
| document.getElementsByClassName(*name*) | Encontre elementos pelo nome da classe |

Alterar elementos HTML

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| *element*.innerHTML =  *new html content* | Muda o HTML interno de um elemento |
| *element*.*attribute = new value* | Muda o valor do atributo de um elemento HTML |
| *element*.setAttribute*(attribute, value)* | Muda o valor do atributo de um elemento HTML |
| *element*.style.*property = new style* | Muda o estilo de um elemento HTML |

Adicionar e Excluir elementos HTML

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| document.createElement(*element*) | Cria um elemento HTML |
| document.removeChild(*element*) | Remove um elemento HTML |
| document.appendChild(*element*) | Adiciona um elemento HTML |
| document.replaceChild(*element*) | Substitui um elemento HTML |
| document.write(*text*) | Escreve no fluxo de saída HTML |

Adicionando manipuladores de eventos

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| document.getElementById(*id*).onclick = function(){*code*} | Adding event handler code to an onclick event |

**Encontrando elementos HTML**

Muitas vezes, com JavaScript, você deseja manipular elementos HTML. Para fazer isso, você deve encontrar os elementos primeiro. Há algumas maneiras de fazer isso:

* Encontrar elementos HTML pelo ID
* Encontrar elementos HTML pelo nome da tag
* Encontrar elementos HTML pelo nome de classe
* Encontrar elementos HTML pelo seletores CSS
* Encontrar elementos HTML pelo coleções de objetos HTML

Encontrar o elemento HTML pelo Id

A maneira mais fácil de encontrar um elemento HTML no DOM é usando o ID do elemento. Este exemplo encontra o elemento com **id = "intro":** var myElement = document.getElementById("intro");

Se o elemento é encontrado, o método irá retornar o elemento como um objeto (em myElement). Se o elemento não for encontrado, myElement conterá nulo.

Encontrar elementos HTML pelo nome da tag

Este exemplo encontra todos elementos **<p>:** var x = document.getElementsByTagName("p");

Este exemplo encontra o elemento com **id = "main"**, e, em seguida, encontra todos elementos **<p>** dentro "main": var x = document.getElementById("main");  
var y = x.getElementsByTagName("p");

Encontrar elementos HTML pelo nome da classe

Se você quiser encontrar todos os elementos HTML com o mesmo nome de classe, use **getElementsByClassName().** Este exemplo retorna uma lista de todos os elementos com **class = "intro"**: var x = document.getElementsByClassName("intro");

Encontrar elementos HTML pelos seletores CSS

Se você quiser encontrar todos os elementos HTML que corresponde a um seletor CSS especificado (id, nomes de classes, tipos, atributos, valores de atributos, etc), use o método **querySelectorAll().** Este exemplo retorna uma lista de todos <p> elementos com **class = "intro":** var x = document.querySelectorAll("p.intro");

Encontrar elementos HTML por Coleções de Objetos HTML

Este exemplo encontra o elemento do formulário com **id = "frm1",** na coleção forms, e exibe todos os valores dos elementos: <form id=" frm1r" action="/action\_page.php"> First name: <input type="text" name="fname" value="Donald"><br> Last name: <input type="text" name="lname" value="Duck"><br><br> <input type="submit" value="Submit"> </form>

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

Os seguintes objetos HTML (e coleções de objetos) também são acessíveis:

* document.anchors
* document.body
* document.documentElement
* document.embeds
* document.forms
* document.head
* document.images
* document.links
* document.scripts
* document.title

**JAVASCRIPT HTML DOM - MUDANDO CSS**

O DOM HTML permite JavaScript para alterar o estilo de elementos HTML. Para alterar o estilo de um elemento HTML, use esta sintaxe: document.getElementById(*id*).style.*property*=*new style*

O objeto de estilo pode ser acessado a partir da seção de cabeçalho do documento, ou do elemento HTML específico(tag). Acessando objeto estilo da seção de cabeçalho do documento: <head> <style> myH1 { background-color: yellow; color: red; } </style> </head>

var x = document.getElementsByTagName("STYLE");

Acessar o objeto de estilo de um elemento especificado: <h1 id="myH1" style="color:red">My Header</h1> var x = document.getElementById("myH1").style.color;

Você pode criar um elemento <style> usando o método **document.createElement():** var x = document.createElement("STYLE"); var t = document.createTextNode("body {font: 20px verdana;}"); x.appendChild(t); document.head.appendChild(x);

Você também pode definir as propriedades de estilo de um elemento existente: document.getElementById("myH1").style.color = "red";

**JAVASCRIPT HTML DOM - EVENTLISTENER**

O método **addEventListener()** atribui um processador de eventos para o elemento especificado. O método **addEventListener()** atribui um manipulador de eventos a um elemento sem substituir os manipuladores de eventos existentes.

Você pode adicionar muitos manipuladores de eventos para um elemento. Você pode adicionar muitos manipuladores de eventos do mesmo tipo para um elemento, ou seja, dois eventos "clique". Você pode adicionar **addEventListener** a qualquer objeto DOM não apenas elementos HTML, ou seja, o objeto de janela.

O método **addEventListener()** faz com que seja mais fácil de controlar a forma como o evento reage aos eventos.

Ao usar o método **addEventListener(),** o JavaScript é separado da marcação HTML, para melhor legibilidade e permite que você adicione **addEventListener()**, mesmo quando você não controle a marcação HTML. Você pode facilmente remover um **addEventListener()** usando o método **removeEventListener()**.

Sintaxe

*element*.addEventListener(*evento, função, useCapture*);

O primeiro parâmetro é o tipo de evento (como "click" ou "mousedown"). O segundo parâmetro é a função que deseja chamar quando o evento ocorrer. O terceiro parâmetro é um valor booleano que especifica se deseja usar ações de eventos ou captura de eventos. Este parâmetro é opcional.

Observe que você não usa o prefixo "on" para o evento, use "click" em vez de "onclick" por exemplo: element.addEventListener("click", function(){ alert("Hello World!"); });

Você também pode se referir a uma função externa "nomeada": element.addEventListener("click", minhaFuncao);  
function mimhaFuncao() {  
    alert ("Hello World!");  
}

O método **addEventListener()** permite que você adicione vários eventos ao mesmo elemento, sem substituir eventos existentes: element.addEventListener("click", myFunction);  
element.addEventListener("click", mySecondFunction);

Ocorrerá os eventos um após o outro. Você também pode adicionar eventos de diferentes tipos ao mesmo elemento: element.addEventListener("mouseover", myFunction);  
element.addEventListener("click", mySecondFunction);  
element.addEventListener("mouseout", myThirdFunction);

**Adicionar um manipulador de eventos para o objeto de janela**

O método **addEventListener()** permite que você adicione ouvintes de eventos em qualquer objeto DOM HTML como, elementos HTML, o documento HTML, o objeto de janela ou outros objetos que suportam eventos, como o objeto XMLHttpRequest.

Adiciona um ouvinte de evento que é acionado quando um usuário redimensiona a janela: window.addEventListener("resize", function(){  
    document.getElementById("demo").innerHTML = sometext;  
});

**Passar parâmetros**

Ao passar valores de parâmetro, use uma "função anônima" que chama a função especificada com os parâmetros: element.addEventListener("click", function(){ myFunction(p1, p2); });

**Evento de Ação ou Evento de Captura**

Existem duas formas de propagação de eventos no HTML DOM, ação e captura. A propagação de eventos é uma maneira de definir a ordem do elemento quando ocorre um evento. Se você tem um elemento <p> dentro de um elemento <div> e o usuário clica no elemento <p>, qual evento de "clique" do elemento deve ser tratado primeiro?

**Ação** - o evento do elemento mais interno é tratado primeiro e depois o externo: o evento de clique do elemento <p> é tratado primeiro, depois o evento de clique do elemento <div>.

**Captura -** o evento do elemento mais externo é tratado primeiro e depois o interno: o evento de clique do elemento <div> é tratado primeiro, depois o evento de clique do elemento <p>.

Com o método **addEventListener()** você pode especificar o tipo de propagação usando o parâmetro "useCapture": addEventListener(event, function, useCapture);

O valor padrão é falso, que usará a propagação de ação, quando o valor for definido como verdadeiro, o evento usa a propagação de captura.

**O método removeEventListener()**

O método **removeEventListener()** remove os manipuladores de eventos que foram anexados com o método addEventListener(): element.removeEventListener("mousemove", myFunction);

**JAVASCRIPT HTML DOM - COLLECTIONS**

O método **getElementsByTagName()** retorna um objeto HTMLCollection. Um objeto HTMLCollection é uma lista semelhante a um vetor (coleção) de elementos HTML. O código a seguir seleciona todos os elementos <p> em um documento: var x = document.getElementsByTagName("p");

Os elementos da coleção podem ser acessados por um número de índice. Para acessar o segundo elemento <p> que você pode escrever: y = x[1];

**HTMLCollection Lenght**

A propriedade length define o número de elementos em uma HTMLCollection: var myCollection = document.getElementsByTagName("p");  
document.getElementById("demo").innerHTML = myCollection.length;

Exemplo explicado: Cria uma coleção de todos os elementos <p>. Exibe o comprimento(nímero de parágrafos) da coleção. A propriedade length é útil quando você deseja percorrer os elementos de uma coleção através de um loop: <p> Exemplo </p> <p> Exemplo </p> <p> Exemplo </p> <script> var myCollection = document.getElementsByTagName("p");  
var i;  
for (i = 0; i < myCollection.length; i++) {  
    myCollection[i].style.backgroundColor = "red";  
} // Muda a cor de todos os parágrafos para vermelho </script>

Um HTMLCollection não é um vetor! Um HTMLCollection pode parecer um vetor, mas não é. Você pode percorrer a lista e consultar os elementos com um número (assim como um vetor). No entanto, você não pode usar métodos de vetor como valueOf(), pop(), push() ou join().

**JavaScript HTML DOM Node Lists**

Um objeto NodeList é uma lista (coleção) de nós extraídos de um documento. Um objeto NodeList é quase o mesmo que um objeto HTMLCollection. Alguns navegadores (mais antigos) retornam um objeto NodeList em vez de um HTMLCollection para métodos como getElementsByClassName().

Todos os navegadores retornam um objeto NodeList para o childNodes da propriedade. A maioria dos navegadores retorna um objeto NodeList para o método querySelectorAll ().

O código a seguir seleciona todos os nós <p> em um documento: var myNodeList = document.querySelectorAll("p");

Os elementos no NodeList podem ser acessados por um número de índice. Para acessar o segundo nó <p> que você pode escrever: y = myNodeList[1];

**HTML DOM Node List Length**

A propriedade length define o número de nós em uma NodeList: var myNodelist = document.querySelectorAll("p");  
document.getElementById("demo").innerHTML = myNodelist.length;

Exemplo explicado:

* Crie uma lista de todos os elementos <p>
* Exibir o comprimento da lista

A propriedade length é útil quando você deseja percorrer os nós em uma NodeList: var myNodelist = document.querySelectorAll("p");  
var i;  
for (i = 0; i < myNodelist.length; i++) {  
    myNodelist[i].style.backgroundColor = "red";  
} // Muda a cor de todos os elementos p para a cor vermelha

**Diferenças entre um HTMLCollection e uma NodeList**

* Um HTMLCollection é uma coleção de elementos HTML
* Um NodeList é uma coleção de nós de documento

Em muitos casos, é a mesma coisa. Tanto um objeto HTMLCollection quanto um objeto NodeList são uma lista (coleção) de objetos semelhantes a uma matriz. Ambos têm uma propriedade de comprimento que define o número de itens na lista (coleção). Ambos fornecem um índice (0,1,2,3,4, ...) para acessar cada item como uma matriz.

O objeto HTMLCollection possui um método adicional chamado named item(), para que os itens possam ser acessados por ID ou nome. Somente o objeto NodeList pode conter nós de atributos e nós de texto.

JavaScript HTML BOM

Não há padrões oficiais para o Modelo de Objeto do Browser (BOM). Uma vez que os navegadores modernos implementaram (quase) os mesmos métodos e propriedades para a interatividade JavaScript, sendo muitas vezes referido, como métodos e propriedades da lista técnica.

**O OBJETO WINDOW**

O objeto **window** é suportado por todos os navegadores. Representa a janela do navegador. Todos os objetos, funções e variáveis JavaScript globais se tornam automaticamente membros do objeto da janela. As variáveis globais são propriedades do objeto da janela. As funções globais são métodos do objeto de janela. Mesmo o objeto do documento do HTML (DOM) também é uma propriedade do objeto da janela: window.document.getElementById("header");

é o mesmo que: document.getElementById("header");

Tamanho da janela

Duas propriedades podem ser usadas para determinar o tamanho da janela do navegador. Ambas as propriedades retornam os tamanhos em pixels:

* **window.innerHeight** - a altura interna da janela do navegador (em pixels)
* **window.innerWidth** - a largura interna da janela do navegador (em pixels)

A janela do navegador NÃO inclui barras de ferramentas e barras de rolagem. Para o Internet Explorer 8, 7, 6, 5:

* document.documentElement.clientHeight
* document.documentElement.clientWidth

ou

* document.body.clientHeight
* document.body.clientWidth

Uma solução prática de JavaScript (cobrindo todos os navegadores): var w = window.innerWidth  
|| document.documentElement.clientWidth  
|| document.body.clientWidth;  
  
var h = window.innerHeight  
|| document.documentElement.clientHeight  
|| document.body.clientHeight;

O exemplo exibe a altura e a largura da janela do navegador: (NÃO incluindo barras de ferramentas / barras de rolagem)

**Outros métodos de janela**

* **window.open()** - abre uma nova janela
* **window.close()** - fecha a janela atual
* **window.moveTo()** - move a janela atual
* **window.resizeTo()** - redimensiona a janela atual

**Tela da janela**

O **objeto window.screen** contém informações sobre a tela do usuário. O objeto **window.screen** pode ser escrito sem o prefixo window e tem as seguintes proprieades:

* screen.**width**
* screen.**height**
* screen.**availWidth**
* screen.**availHeight**
* screen.**colorDepth**
* screen.**pixelDepth**

**Largura da tela da janela**

A propriedade **screen.width** retorna a largura da tela do visitante em pixels: document.getElementById("demo").innerHTML =  
" Retorna: " + screen.width; // Retorna 1280

**Altura da tela da janela**

A propriedade **screen.height** retorna a altura da tela do visitante em pixels: document.getElementById("demo").innerHTML =  
" Retorna: " + screen.height; // Retorna 1024

**Largura disponível na tela da janela**

A propriedade **screen.availWidth** retorna a largura da tela do visitante, em pixels, menos recursos de interface, como a Barra de Tarefas do Windows: document.getElementById("demo").innerHTML =  
" Retorna: " + screen.availWidth; // Retorna 1280

**Altura da tela da janela Altura**

A propriedade **screen.availHeight** retorna a altura da tela do visitante, em pixels, recursos de interface menos, como a Barra de Tarefas do Windows: document.getElementById("demo").innerHTML =  
"Retorna: " + screen.availHeight; // Retorna 984

**Profundidade de cor da tela da janela**

A propriedade **screen.colorDepth** retorna o número de bits usados para exibir uma cor. Todos os computadores modernos usam hardware de 24 bits ou 32 bits para resolução de cores:

* 24 bits = 16,777,216 "cores verdadeiras"
* 32 bits = 4,294,967,296 "cores profundas"

Computadores mais antigos usavam 16 bits: 65.536 "cores altas" de resolução. Computadores muito antigos e telefones celulares usados usaram 8 bits: 256 "cores VGA" .

document.getElementById("demo").innerHTML =   
"Retorna: " + screen.colorDepth; // Retorna 24

**Profundidade do pixel da tela da janela**

A propriedade **screen.pixelDepth** retorna a profundidade de pixels da tela: document.getElementById("demo").innerHTML =  
"Retorna: " + screen.pixelDepth; // Retorna 24

Para computadores modernos, a profundidade de cor e a profundidade de pixels são iguais. Os valores #rrggbb (rgb) usados em HTML representam "cores verdadeiras" (16.777.216).

**JavaScript Window Location**

O objeto **window.location** pode ser usado para obter o endereço da página atual (URL) e para redirecionar o navegador para uma nova página. O objeto **window.location** pode ser escrito sem o prefixo window. Alguns exemplos:

* window.**location.href** - retorna o href (URL) da página atual
* window.**location.hostname** - retorna o nome de domínio do host
* window.**location.pathname** - retorna o caminho e o nome do arquivo (página atual)
* window.**location.protocol** - retorna o protocolo web usado (http: ou https :)
* **window.location.port** - retorna o número da porta do host da internet (página atual)
* window.**location.assign** - carrega um novo documento (nova página)

**JavaScript Window History**

O objeto **window.history** contém o histórico do navegador. O objeto **window.history** pode ser escrito sem o prefixo da janela. Para proteger a privacidade dos usuários, existem limitações de como o JavaScript pode acessar esse objeto. Alguns métodos:

* **history.back()** - o mesmo que clicar voltar no navegador
* **history.forward()** - o mesmo que clicar avançar no navegador

<script>  
function goBack() {  
    window.history.back()  
}  
</script> <input type="button" value="Forward" onclick="goBack()">

**JavaScript Window Navigator**

O objeto **window.navigator** contém informações sobre o navegador do visitante. O objeto **window.navigator** pode ser escrito sem o prefixo window. Alguns exemplos:

* navigator.cockieEnabled - retorna verdadeiro se os cookies estiverem ativados
* navigator.appName - retorna o nome do aplicativo do navegador:
* navigator.appCodeName - retorna o nome do código do aplicativo do navegador
* navigator.product - retorna o nome do produto do mecanismo do navegador
* navigator.appVersion - retorna informações sobre a versão do navegador
* navigator.platform - retorna a plataforma do navegador (sistema operacional)
* navigator.language - retorna o idioma do navegador
* navigator.onLine - retorna verdadeiro se o navegador estiver online
* navigator.javaEnabled() - Retorna verdadeiro se Java estiver habilitado
* navigator.userAgent - retorna o cabeçalho do user-agent enviado pelo navegador para o servidor

Atenção !!!

As informações do objeto **window** geralmente podem ser enganosas e não devem ser usadas para detectar versões do navegador porque:

* Navegadores diferentes podem usar o mesmo nome
* Os dados do navegador podem ser alterados pelo proprietário do navegador
* Alguns navegadores se identificam erroneamente para ignorar os testes do site
* Os navegadores não podem denunciar novos sistemas operacionais, lançados depois do navegador

**CAIXAS DE POPUP**

O JavaScript tem três tipos de caixas de pop-up: caixa de alerta, caixa de confirmação e caixa de solicitação.

Caixa de alerta

Uma caixa de alerta é frequentemente usada se você quiser garantir que a informação chegue ao usuário. Quando uma caixa de alerta aparecer, o usuário terá que clicar em "OK" para prosseguir. Sintaxe: window.alert(“Exemplo”); ou alert(“Exemplo”);

Caixa de confirmação

Uma caixa de confirmação é frequentemente usada se desejar que o usuário verifique ou aceite algo. Quando uma caixa de confirmação aparecer, o usuário terá que clicar em "OK" ou "Cancelar" para prosseguir. Se o usuário clicar em "OK", a caixa retornará verdadeira. Se o usuário clicar em "Cancelar", a caixa retornará falsa. Sintaxe: window.confirm(“Exemplo”);

if (confirm("Pressione um botão!") == true) {  
    txt = "Você pressionou OK!";  
} else {  
    txt = "Você pressionou Cancel!";  
}

Caixa de solicitação

Uma caixa de prompt geralmente é usada se desejar que o usuário insira um valor antes de inserir uma página. Quando uma caixa de prompt aparecer, o usuário terá que clicar em "OK" ou "Cancelar" para continuar depois de inserir um valor de entrada. Se o usuário clicar em "OK", a caixa retornará o valor de entrada. Se o usuário clicar em "Cancelar", a caixa retornará null. Sintaxe: window.prompt("*Seu nome:*","John");

<button onclick="myFunction()">Aperte</button> <p id="demo"></p> <script> var person = prompt("Entreo com seu nome", "John");  
if (person == null || person == "") {  
    txt = "Você cancelou o prompt.";  
} else {  
    txt = "Alô " + person + "! Como vai?";  
} document.getElementById("demo").innerHTML = txt; </script>

Quebras de linha

Para exibir quebras de linha dentro de uma caixa pop-up, use uma barra invertida seguida do caractere n. Sintaxe: alert("Alô\nComo vai?");

**EVENTOS DE TEMPORIZAÇÃO**

O objeto window permite a execução do código em intervalos de tempo especificados. Esses intervalos de tempo são chamados eventos de temporização. Os dois métodos principais para usar com JavaScript são:

* **SetTimeout** (função, milissegundos) - Executa uma função, depois de esperar um número especificado de milissegundos.
* **SetInterval** (função, milissegundos) - O mesmo que setTimeout(), mas repete a execução da função de forma contínua.

O **setTimeout()** e **setInterval()** são ambos os métodos do objeto **window** do HTML DOM.

O método setTimeout()

O método window.setTimeout() pode ser escrito sem o prefixo window. O primeiro parâmetro é uma função a ser executada. O segundo parâmetro indica o número de milissegundos antes da execução. Sintaxe: window.setTimeout(*function*,*milliseconds*);

<button onclick="setTimeout(myFunction, 3000)">Aperte</button>  
<script>  
function myFunction() {  
    alert('Hello');  
} // Mostra um alert 3 segundos depois do aperto do botão  
</script>

O método clearTimeOut()

O método **clearTimeout()** interrompe a execução da função especificada no setTimeout(). Sintaxe: window.clearTimeout(timeoutVariable)

O método **clearTimeout()** usa a variável retornada de setTimeout():

myVar = setTimeout(*function*,*milliseconds*);  
clearTimeout(myVar);

Se a função ainda não foi executada, você pode parar a execução chamando o método **clearTimeout():** <button onclick="myVar = setTimeout(myFunction, 3000)">Aperte</button>  
<button onclick="clearTimeout(myVar)">Pare</button><script>  
function myFunction() {  
    alert('Hello');  
} // Mostra um alert 3 segundos depois do aperto do botão mas não mostra o alert se for apertado o botão Pare antes dos 3 segundos  
</script>

O método setInterval()

O método **setInterval()** repete uma determinada função em cada intervalo de tempo. Sintaxe: window.setInterval(*function*,*milliseconds*);

O primeiro parâmetro é a função a ser executada. O segundo parâmetro indica o comprimento do intervalo de tempo entre cada execução. Este exemplo executa uma função chamada "myTimer" uma vez por segundo (como um relógio digital). <p id="demo"></p> var myVar = setInterval(myTimer, 1000);  
function myTimer() {  
    var d = new Date();  
    document.getElementById("demo").innerHTML = d.toLocaleTimeString();  
}

O método clearInterval()

O método **clearInterval()** interrompe as execuções da função especificada no método setInterval(). Sintaxe: window.clearInterval(timerVariable)

O método **clearInterval()** usa a variável retornada de setInterval(): myVar = setInterval(*function*,*milliseconds*);  
clearInterval(myVar);

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

**COOKIES**

Os cookies permitem que você armazene informações do usuário em páginas da web. Os cookies são dados, armazenados em pequenos arquivos de texto, no seu computador. Quando um servidor web enviou uma página da Web para um navegador, a conexão é desligada e o servidor esquece tudo sobre o usuário.

Cookies foram inventados para resolver o problema "como lembrar informações sobre o usuário":

* Quando um usuário visita uma página da Web, seu nome pode ser armazenado em um cookie.
* Da próxima vez que o usuário visitar a página, o cookie "lembra" seu nome.

Os cookies são salvos em pares nome-valor como: username = John Doe

Quando um navegador solicita uma página da Web de um servidor, os cookies pertencentes à página são adicionados ao pedido. Desta forma, o servidor obtém os dados necessários para "lembrar" as informações sobre os usuários.

**Criar um cookie com JavaScript**

O JavaScript pode criar, ler e excluir cookies com a propriedade **document.cookie**. Com JavaScript, um cookie pode ser criado assim: document.cookie = "username=John Doe";

Você também pode adicionar uma data de validade (em horário UTC). Por padrão, o cookie é excluído quando o navegador é fechado: document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC";

Com um parâmetro de **path**, você pode informar ao navegador a qual path o cookie pertence. Por padrão, o cookie pertence à página atual. document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";

**Ler um Cookie com JavaScript**

Com o JavaScript, os cookies podem ser lidos assim: var x = document.cookie; document.cookie retornará todos os cookies em uma string como: cookie1 = valor; cookie2 = valor; cookie3 = valor;

**Alterar um cookie com JavaScript**

Com o JavaScript, você pode alterar um cookie da mesma maneira que você o cria: document.cookie = "username=John Smith; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/"; // O velho cookie é substituído.

**Excluir um cookie com JavaScript**

Eliminar um cookie é muito simples. Você não precisa especificar um valor de cookie quando você excluir um cookie. Basta configurar o parâmetro **expires** para uma data passada: document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";

Você deve definir o caminho do cookie para garantir que você exclua o cookie certo. Alguns navegadores não permitem que você exclua um cookie se você não especificar o caminho.

**Cookie String**

A propriedade **document.cookie** parece uma string de texto normal. Mas não é. Mesmo se você escrever uma string de cookie inteira para **document.cookie**, quando você lê o cookie novamente, você só verá o par nome-valor dele.

Se você definir um novo cookie, os cookies mais antigos não são substituídos. O novo cookie é adicionado ao **document.cookie**, então se você ler o document.cookie novamente, você receberá algo como: Cookie1 = valor; Cookie2 = valor;

Se você quiser encontrar o valor de um cookie específico, você deve escrever uma função JavaScript que pesquise o valor do cookie na sequência dos cookies.

**Exemplo de Cookie com JavaScript**

No exemplo a seguir, criaremos um cookie que armazena o nome de um visitante. A primeira vez que um visitante chegar à página web, ele será convidado a preencher seu nome. O nome é então armazenado em um cookie. Na próxima vez que o visitante chegar na mesma página, ele receberá uma mensagem de boas-vindas.

Para o exemplo, criaremos 3 funções de JavaScript:

* Uma função para definir um valor do cookie
* Uma função para obter o valor do cookie
* Uma função para verificar o valor do cookie

Uma função para definir um cookie

Primeiro, criamos uma função que armazena o nome do visitante em uma variável de cookie: function setCookie(cname, cvalue, exdays) {  
    var d = new Date();  
    d.setTime(d.getTime() + (exdays\*24\*60\*60\*1000));  
    var expires = "expires="+ d.toUTCString();  
    document.cookie = cname + "=" + cvalue + ";" + expires + ";path=/";  
}

**Exemplo explicado:** Os parâmetros da função acima são o nome do cookie (cname), o valor do cookie (cvalue) e o número de dias até que o cookie expire (exdays). A função define um cookie juntando o cookiename, o valor do cookie e a string expires.

Uma função para obter o cookie

Em seguida, criamos uma função que retorna o valor do cookie especificado: function getCookie(cname) {  
    var name = cname + "=";  
    var decodedCookie = decodeURIComponent(document.cookie);  
    var ca = decodedCookie.split(';');  
    for(var i = 0; i <ca.length; i++) {  
        var c = ca[i];  
        while (c.charAt(0) == ' ') {  
            c = c.substring(1);  
        }  
        if (c.indexOf(name) == 0) {  
            return c.substring(name.length, c.length);  
        }  
    }  
    return "";  
}

**Função explicada:** Pega o parâmetro cookiename como (cname).Cria uma variável (nome) com o texto para procurar (cname + "=").Decodifica a string de cookies, para lidar com cookies com caracteres especiais, p. Ex. '$'**.** Divide document.cookie com ponto e vírgula em um vetor chamado ca (ca = decodedCookie.split (';')).Loop através do vetor ca (i = 0; i <ca.length; i ++) e lê cada valor c = ca [i]).Se o cookie for encontrado (c.indexOf (name) == 0), devolve o valor do cookie (c.substring (name.length, c.length).Se o cookie não for encontrado, return " ".

Uma função para verificar o cookie

Por último, criamos a função que verifica se um cookie está configurado. Se o cookie estiver configurado, ele exibirá uma saudação. Se o cookie não estiver configurado, ele exibirá uma caixa de solicitação, solicitando o nome do usuário e armazenando o cookie de nome de usuário por 365 dias, chamando a função setCookie: function checkCookie() {  
    var username = getCookie("username");  
    if (username != "") {  
        alert("Welcome again " + username);  
    } else {  
        username = prompt("Please enter your name:", "");  
        if (username != "" && username != null) {  
            setCookie("username", username, 365);  
        }  
    }  
}