

Programação Web

- ✓ JavaScript
- ✓ Conversões
- ✓ Operadores de comparação e lógicos
- ✓ Estruturas de decisão
- ✓ Estruturas de repetição
- ✓ Funções
- ✓ DOM (Document Object Model)
- ✓ Eventos
- ✓ Acessando os elementos do form em JS
- ✓ Exemplos
- ✓ Exercícios

Estudamos os métodos `parseInt()` e `parseFloat()` na aula anterior, esses métodos convertem respectivamente um valor do tipo `String` para inteiro ou real. No JS, esses tipos são identificados como `number`.

Outras formas de conversão:

```
let x = 23;
```

```
typeof x; //retorna o tipo number
```

```
let c = String(x);
```

```
typeof c; retornar o tipo string
```

```
let y = Number("12.5");
```

```
let z = Number("30");
```

```
typeof y; //retorna o tipo number
```

```
typeof z; //retorna o tipo number
```

Para você verificar a tabela abaixo, declaramos uma variável
let x=10;

Operador	Descrição	Exemplo
==	Igual a	x == 20 (falso)
===	Exatamente igual a (retorna verdadeiro caso os operandos sejam iguais e do mesmo tipo)	x === 10 (verdadeiro) x === "10" (falso) x == "10" (verdadeiro)
!=	Diferente de	x != 20 (verdadeiro)
<	Menor que	x < 20 (verdadeiro)
<=	Menor ou igual a	x <= 10 (verdadeiro)
>	Maior que	x > 20 (falso)
>=	Maior ou igual a	x >= 10 (verdadeiro)

Para você verificar a tabela abaixo, declaramos duas variáveis:

```
let x=10;
```

```
let y=2;
```

Operador	Descrição	Exemplo
&&	e	(x > y && y < 5) (verdadeiro)
	ou	(x > y y > 5) (verdadeiro)
!	Negação	! (x != y) (falso)

Tabela verdade

Expressão 1	Expressão 2	&&	
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Expressão	!
true	false
false	true

true = verdadeiro
false = falso

São blocos de código que serão executados somente se uma dada condição for satisfeita.

No nosso dia-a-dia estamos sempre tomando decisões, por exemplo:

- se estiver chovendo levo o guarda-chuva.
- se for final de semana e estiver sol irei para a praia.

Temos disponível em JavaScript as seguintes estruturas de decisões:

if

if ... else

if ... else if ... else...

switch

Sintaxe:

```
if (condição)
{
    //instruções caso a condição seja verdadeira
}
```

Exemplo:

```
<script>
let data_hora = new Date();
let hora = data_hora.getHours();
```

```
if (hora < 12)
{
    alert("Bom dia!!!");
}
```

```
</script>
```

Na condição da estrutura, utilizamos os operadores de comparação e os operadores lógicos. Se a condição for **verdadeira**, o bloco é executado.

Sintaxe:

```
if (condição)
{
    //instruções caso a condição seja verdadeira
}
else
{
    //instruções caso a condição seja falsa
}
```

Exemplo:

```
<script>
```

```
let data_hora = new Date();
```

```
let hora = data_hora.getHours();
```

```
if (hora < 12)
{
    alert("Bom dia!!!");
}
```

para condição true (verdadeiro)

```
else
{
    alert("Olá, tenha um bom dia!!!");
}
```

para condição false (falso)

Se a condição for **verdadeira**, o bloco do **if** é **executado**, senão, se a condição for **falsa**, o bloco do **else** é que será **executado**.

```
</script>
```


Sintaxe:

```
if (condição 1)
{
    //instruções 1
}
else {
    if (condição 2)
    {
        //instruções 2
    }
    else
    {
        //instruções 3
    }
}
```

Um bloco é um grupo de comandos entre chaves: { }

Se tivermos um único comando não seria obrigatório usar chaves (para uma programação mais clara é recomendado usar as chaves).

A decisão encadeada seria um conjunto de if's. Para cada if devemos ter uma condição e poderemos caso necessário colocar o else (senão).

Também podemos ter um bloco if dentro de um outro bloco if, ou dentro de um bloco else.

```
<script>
  let data_hora = new Date()
  let hora = data_hora.getHours()
  let previsao_tempo = "chuvoso"
  if (hora < 12)
  {
    alert("Bom dia...");
    if (previsao_tempo == "chuvoso")
    {
      alert("Está chovendo, leve o guarda-chuva")
    }
  }
  else {
    if (hora >= 12 && hora < 18)
    {
      alert("Boa tarde...");
    }
    else
    {
      alert("Boa noite...");
    }
  }
</script>
```

Perceba que podemos ter um bloco if dentro de outro, o mesmo pode ocorrer com o bloco else, ou seja, podemos também ter um bloco if dentro do else. Se preferir, utilize chaves para cada sub bloco de if.

Semelhante ao bloco if porém para casos bem simples, aceita somente uma instrução para cada caso, seu uso é mais comum quando precisamos atribuir um valor para uma variável de acordo com uma dada condição.

Sintaxe:

(condição) ? //instrução se true : //instrução se false

Ou

let variável = (condição) ? //instrução se true : //instrução se false

Exemplo

<script>

let x = 10;

let y = 20;

(x > y) ? alert("Sim") : alert("Não"); //evite usar desta forma, é menos legível.

let r = (x < y) ? "Sim" : "Não";

console.log(r);

</script>

Utilizado quando temos várias condições simples

Sintaxe:

```
switch (valor) {  
    case valor1:  
        //instruções 1  
        break;  
    case valor2:  
        //instruções 2  
        break;  
    case valor3:  
        //instruções 3  
        break;  
    default:  
        //instruções padrão  
}  

```

Para cada caso devemos colocar o comando break, este comando irá finalizar o caso e evitar que o caso posterior seja executado.

O default é opcional, ele é executado quando nenhum caso anterior é acionado.

Os valores podem ser String, inteiros, caracteres ou reais.

Não podemos fazer comparações no case, como $x > y$, por exemplo.

Exemplo

```
<script>
  let data_hora = new Date();
  let dia_semana = data_hora.getDay();
  switch (dia_semana)
  {
    case 0:      alert("Domingo de descanso merecido.");
                break;
    case 5:      alert("Obaaa, sexta-feira.");
                break;
    case 6:      alert("Maravilha, sabadão!!");
                break;
    default:     alert("Semana longaaaa.");
  }
</script>
```

São utilizadas quando necessitamos repetir um bloco de instruções.

Podemos executar um laço de repetição com um número específico de vezes ou enquanto uma condição for verdadeira.

Essas estruturas também são conhecidas como laço de repetição ou loop.

Em JavaScript temos:

- for (algumas variações são: for/in, for/of ou forEach, veremos isso em outra aula)

- while

- do while

Utilizada quando sabemos o número de repetições que serão feitas.
Sintaxe:

```
for (valor inicial; condição; incremento/decremento)
{
    //instruções que serão repetidas
}
```

Exemplo:

```
<script>
let cont;
for (cont = 0; cont < 10 ; cont++)
{
    console.log(`Número: ${cont}`);
}
</script>
```

Iremos executar o laço 10 vezes (para cont de 0 a 9). A cada passagem, será impresso o valor da variável cont.

Executa um bloco de instruções enquanto uma certa condição for verdadeira

Sintaxe:

```
while (condição)
{
    //instruções
    //alteração do valor da condição
}
```

A variável cont deve ser inicializada antes do bloco.

Exemplo:

```
<script>
let cont = 0;
while (cont < 10)
{
    console.log("Número: " + cont);
    cont = cont + 1; //ou cont++;
}
</script>
```

Iremos executar o laço 10 vezes. A cada passagem, será impresso o valor da variável cont.

Perceba que dentro do laço é inserido o incremento da variável. Com isso em algum momento a condição se tornará falsa.

Executa um bloco de instruções enquanto uma certa condição for verdadeira, porém na primeira passagem as instruções são executadas visto que o teste da condição é feito somente no final.

Sintaxe:

```
do
{
    //instruções
    //alteração do valor da condição
} while (condição);
```

Exemplo:

```
<script>
let cont=0;
do
{
    console.log("Número: " + cont);
    cont = cont + 1; //ou cont++;
} while (cont < 10);
</script>
```

A variável cont deve ser inicializada antes do bloco.

Iremos executar o laço 10 vezes. A cada passagem, será impresso o valor da variável cont.

Perceba que dentro do laço é inserido o incremento da variável, com isso em algum momento a condição se tornará falsa.

Podemos usar o comando **break** para parar um determinado laço de repetição conforme alguma condição no nosso código.

Ex: Vamos ler vários valores do usuário e fazer a soma, quando ele pressionar a tecla enter sem digitar um valor, entraremos na condição IF e vamos parar o laço.

```
<script>
```

```
let soma = 0;
while (true) {
  let num = Number(prompt("Enter a number", ""));
  if (!num)
    break;
  soma += num;
}
console.log(`Soma: ${soma}`);
```

```
</script>
```

A instrução **continue** interrompe a iteração atual e força o laço continuar para a próxima iteração.

Ex: vamos imprimir somente os números ímpares, sempre que for um número par, entramos no IF e executamos o comando continue e pula para a próxima iteração ignorando a impressão do número par

<script>

```
for (let i = 0; i < 10; i++) {  
  if (i % 2 == 0)  
    continue;  
  console.log(i); // imprime somente os ímpares de 0 a 10  
}
```

</script>

Funções são trechos de códigos criados para realizarem tarefas específicas, que podem ser acionados através de uma chamada direta ou através de um evento.

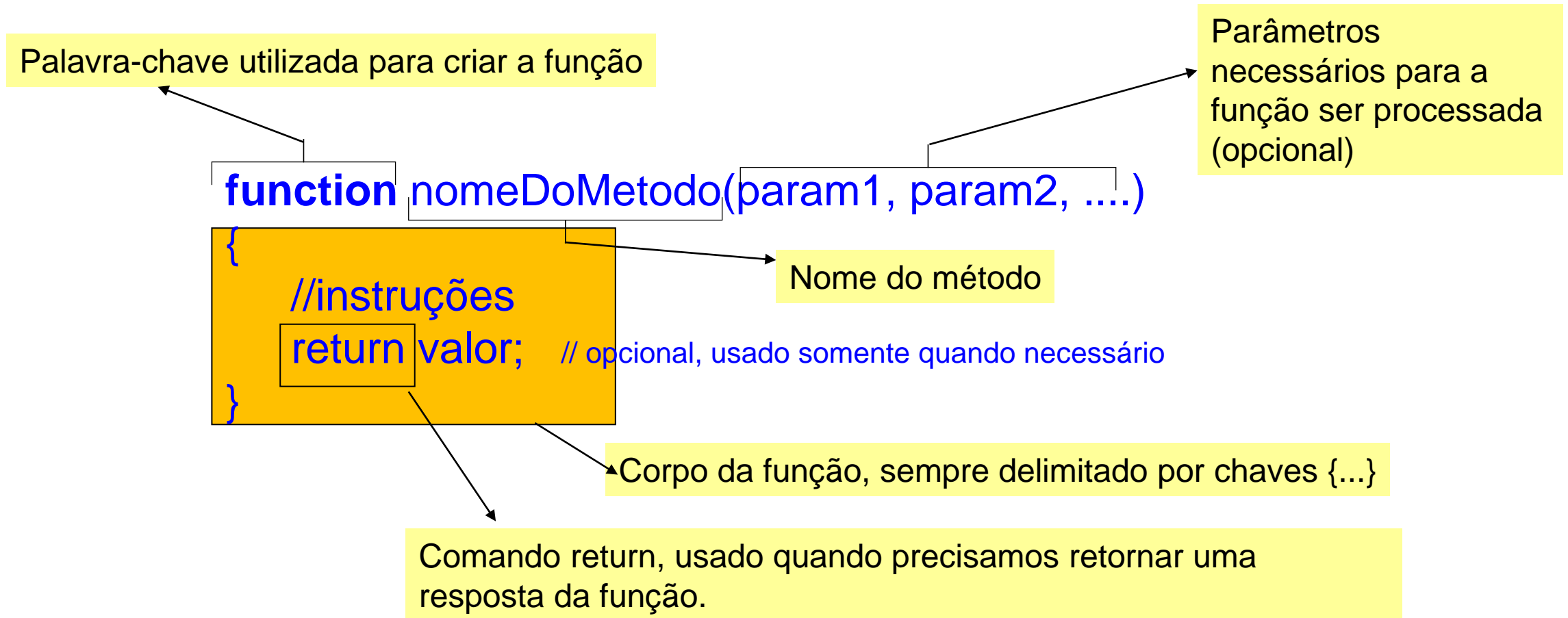
Podemos chamar uma função em qualquer ponto da página e se a mesma estiver em um arquivo .js, podemos chamá-la em qualquer página que faz referência ao arquivo JavaScript.

As funções geralmente são criadas em um arquivo .js ou no cabeçalho (<head>) do documento HTML.

Uma função pode ou não retornar uma resposta para o ponto em que foi chamada. Caso necessite de um retorno (resposta) utilizamos o comando **return**.

Uma função JavaScript que não retorna resultado tem comportamento de *procedimento*. Neste caso não será utilizado o comando **return** (um comando `return;` sem resultado terminará a função e retornará o valor *undefined*). Diferente de Java ou C#, em JavaScript não existe a palavra `void` para funções que não retornam resultados.

Uma função poderá ter uma lista de parâmetros ou argumentos para receber dados. Diferente de Java ou C#, em JavaScript não existem tipos de dados pré-definidos (como `float`, `int`, `char` etc.), apenas escreveremos os nomes dos parâmetros.



```
<script>
```

```
function soma(a , b){  
    console.log(a+b);  
}
```

Função **soma**, recebe dois parâmetros (a , b), efetua a soma entre eles e o escreve no console.

```
//chama a função
```

```
soma(2,2);  
soma(3,4);
```

Chama a função soma duas vezes com valores diferentes para a e b.

```
</script>
```

```
<script>
```

```
function lerNome(){  
  let nome;  
  nome = prompt("Digite seu nome","");  
  return nome;  
}
```

Função **lerNome**, não recebe parâmetros, lê um nome através da janela de prompt e depois retorna esse nome (return) para o ponto em que foi chamada.

```
//chama a função  
let resp;  
resp = lerNome();  
alert(resp);
```

```
</script>
```

Chama a função lerNome, como a função retorna um valor, devemos atribuir este retorno para uma variável, para que posteriormente possamos manipular essa informação.

Formato para criar funções mais otimizado, foi introduzido na versão ES6.

Pode não ser amplamente suportada pelos navegadores.

Uma arrow function é definida por um par de parênteses contendo uma lista de parâmetros (param1, param2, ..., paramN), seguidos por uma seta => e posteriormente um par de chaves {...} que delimitam o corpo da função.

OBS: os parâmetros são opcionais, seu uso vai depender da sua necessidade.

```
let identificador = (param1, param2, ...) => {  
    //instruções  
}
```

Exemplos:

```
let soma = (a, b) => { return a + b };
```

Ou

```
let soma = (a, b) => a + b ;
```

Embora seja possível trabalhar sem as chaves { } quanto temos somente 1 instrução, vamos seguir o padrão de sempre usar para evitar confusão.

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrow Function</h2>
<p>Um exemplo de Arrow Function (função de seta) sem parâmetros.</p>
```

```
<script>
  boasvindas = () => {
    return "Olá, obrigado por usar este app!";
  }

  boasvindas2 = () => "Olá, obrigado por usar este app!"; //mais simples ainda

  alert( boasvindas() ); //chamamos a função boasvindas
  alert( boasvindas2() ); //chamamos a função boasvindas2
</script>
</body>
</html>
```

As funções de seta (arrow functions) permitem uma sintaxe reduzida, mais curta, para funções. Veja mais em: https://www.w3schools.com/js/js_arrow_function.asp

```
<!DOCTYPE html>
```

```
<html>
```

```
<body>
```

```
<h2>JavaScript Arrow Function</h2>
```

```
<p>Um exemplo de Arrow Function (função de seta) com três parâmetros.</p>
```

```
<script>
```

```
  media = (v1, v2, v3) => (v1+v2+v3)/3; //esta é uma arrow function (função de seta)
```

```
  alert( "Média de três valores: " + media(9, 4, 8) ); //chamamos a função 'media'
```

```
</script>
```

```
</body>
```

```
</html>
```

As funções de seta (arrow functions) permitem uma sintaxe reduzida, mais curta, para funções. Veja mais em: https://www.w3schools.com/js/js_arrow_function.asp

```
<script>
//ES5
function somaV1(a,b) {
    return ("Soma:"+(a+b));
}
//ou
let somaV2 = function(a,b) {
    return ("Soma:"+(a+b));
}
```

```
//ES6
const somaV3 = (a, b) => { return ("Soma:"+(a+b)); }
</script>
```

```
<script>
    console.log(somaV1(5,5)    );
    console.log(somaV2(10,20)  );
    console.log(somaV3(130,230));
</script>
```

<http://www.w3.org/DOM/DOMTR>

<https://dom.spec.whatwg.org/>

DOM Level 1 provided a complete model for an entire HTML or XML document, including the means to change any portion of the document.

DOM Level 2 was published in late 2000. It introduced the `getElementById` function as well as an event model and support for XML namespaces and CSS.

DOM Level 3, published in April 2004, added support for XPath and keyboard event handling, as well as an interface for serializing documents as XML.

DOM Level 4 was published in 2015. It is a snapshot of the WHATWG living standard.

http://en.wikipedia.org/wiki/Document_Object_Model

Document Object Model ou Modelo de Objetos de Documentos.

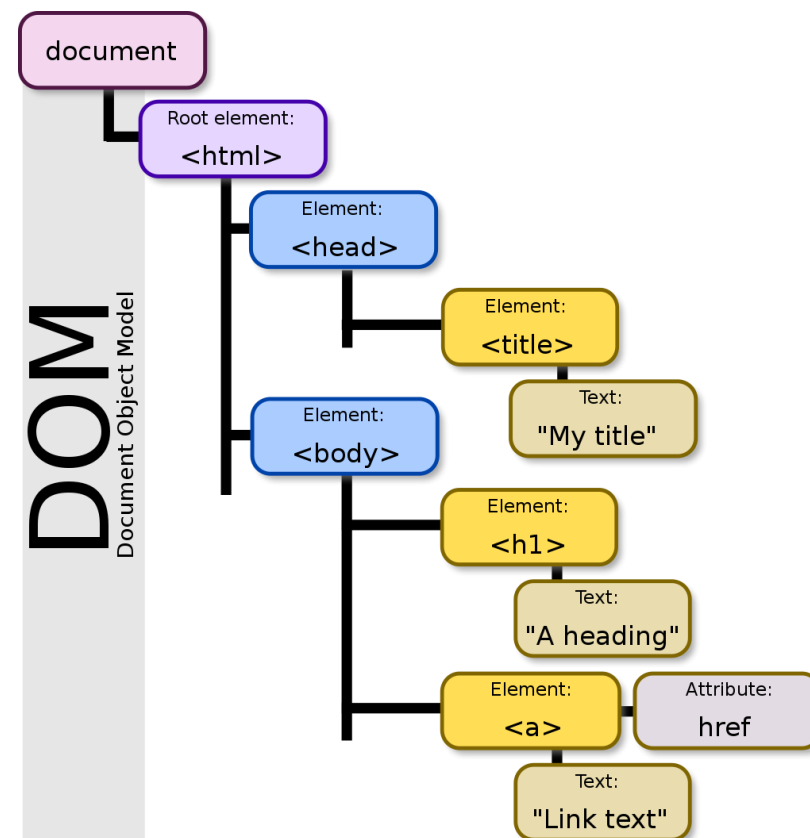
É um modelo que permite acessar os elementos do HTML, possibilitando alterar e editar a sua estrutura, conteúdo e estilo.

Fornece diversos objetos, funções e propriedades que representam todo o conteúdo HTML. Sendo assim, fazendo modificações no DOM, automaticamente estamos realizando modificações na estrutura HTML e aparência da página.

Podemos acessar e modificar qualquer elemento que possua um ID, Class, etc.

Para manipular o DOM, é necessário que o documento esteja totalmente carregado. Sendo assim, podemos dizer que nossos scripts precisam conhecer o DOM para conseguir manipular ele.

Os browsers têm que estar preparados para processar JavaScript + DOM na máquina do cliente.



Alguns métodos/propriedades fornecidos pelo DOM (existem a muito tempo no JS):

getElementById(id) – acessa o elemento através do atributo **id**, retorna o elemento como objeto, lembre-se que o **id** deve ser único no HTML

elemento.style.propriedade = "valor" – permite definir regras de estilo

getElementsByName(name) – acessa o elemento através do atributo **name**, retorna o(s) elemento(s) como uma coleção de valores ou, em outras palavras, como vetor.

getElementsByTagName(nome) – acessa todos os elementos através do **nome** da tag, muito utilizado para manipular documentos XML, retorna sempre um vetor com os elementos, cada elemento em uma posição.

getElementsByClassName(nome) – acessa todos os elementos através do **nome** da classe, retorna o(s) elemento(s) como uma coleção de valores ou, em outras palavras, como vetor.

Alguns métodos/propriedades fornecidos pelo DOM (mais novos):

- **querySelector(seletor css)** – acessa o elemento através do seletor. O seletor pode ser por exemplo um id, uma tag, uma classe, ou qualquer outro tipo de seletor CSS, retorna o elemento como objeto.
- **querySelectorAll(seletor)** – retorna uma lista de elementos

O querySelector é um recurso mais recente que o getElementById, contudo, o getElementById é melhor suportado pelos navegadores e está presente em sistemas mais antigos.

O querySelector permite encontrar elementos com regras que não podem ser expressas com getElementById.

Vamos dar prioridade para o uso do querySelector e querySelectorAll.

Sempre que ocorre uma interação com o documento ou página um evento é disparado, um evento pode ser qualquer interatividade do usuário com um elemento HTML, alguns eventos também podem ser disparados pelo navegador.

Alguns termos:

- Manipulador de evento (event handler)
é uma função a ser executada quando o evento é acionado
- Disparador de evento
é o elemento HTML onde o manipulador de evento foi adicionado

Eventos acionados com mouse

- click
botão esquerdo do mouse ou tecla enter
- mousedown
quando pressionamos qualquer um dos botões do mouse
- mouseup
quando liberamos o botão pressionado anteriormente
- mouseover
quando colocamos o mouse sobre um elemento
- mouseout
quando tiramos o mouse de um elemento
- mousemove
quando movemos o ponteiro do mouse

Eventos de teclado

- keydown
quando pressionarmos uma tecla
- keypress
quando pressionarmos uma tecla que resulte em um caractere
- keyup
quando soltarmos a tecla pressionada

Eventos HTML, são eventos que não estão associados diretamente com o usuário, os principais são:

- load
carregamento completo do conteúdo
- unload
fechamento de um documento
- focus
ocorre quando a janela ou algum elemento HTML recebe o foco
- change
ocorre quando um elemento de formulário perde o foco após ter sido alterado seu conteúdo
- select
ocorre quando selecionamos um texto em elementos de formulário
- submit
ocorre quando clicamos em um botão submit

Existem diversas formas para adicionar um evento a um elemento HTML

No HTML

```
<input type="button" onclick="abrirAlgo()" value="Clique aqui" />
```

OBS: esse modo deve ser evitado

No DOM (primeira forma)

```
<input type="button" id="botao" value="Clique aqui">
```

e um script como mostrado a seguir:

```
<script>
  window.onload = function(){
    let btn = document.getElementById("botao");
    btn.onclick = function (){
      alert("Oi!");
    }
  }
</script>
```

No Dom (segunda forma)

- `addEventListener(evento, função)`
 - Onde :
evento = tipo do evento
função = função que será executada para atender o evento

Exemplo 1:

```
<script>
  window.addEventListener("load", function(){
    let btn = document.getElementById("botao");
    btn.addEventListener("click", function(){
      alert("Oi!");
    });
  });
</script>
```

Exemplo 2:

```
<script>
window.addEventListener("load", function(){
    let btn = document.getElementById("botao");
    let minhafunc = function() {
        alert("Minha função");
    }
    btn.addEventListener("click", minhafunc);
});
</script>
```

elemento.addEventListener(evento, função, useCapture)

useCapture true => elemento externo, interno

useCapture false => elemento interno, externo

Exemplo 3:

```
<script>

window.addEventListener("load", function(){
    let btn = document.getElementById("botao");
    btn.addEventListener("click", minhafunc, false);
});

function minhafunc() {
    alert("Minha função");
}

</script>
```

Quando criamos um formulário, podemos inserir diversos elementos, como por exemplo:

Caixa de texto

```
<input type="text" name="nome" id="nome" />
```

Campo de texto (várias linhas)

```
<textarea name="mensagem" id="mensagem"></textarea>
```

Botões

```
<input type="submit" value="Enviar" />
```

Caixa de seleção/lista

```
<select name="cidade" id="cidade">  
    <option value=""></option>  
    <option value="SP">SP</option>  
    <option value="RJ">RJ</option>  
</select>
```

Botões de opções

```
M <input type="radio" name="sexo" id="sexo_m" />
```

```
F <input type="radio" name="sexo" id="sexo_f" />
```

Listas de checkbox

```
Op1 <input type="checkbox" name="lista" id="op1" />
```

```
Op2 <input type="checkbox" name="lista" id="op2" />
```

```
Op3 <input type="checkbox" name="lista" id="op3" />
```

No exemplo anterior foram demonstrados diversos elementos de um formulário. O que existe de comum a todos?

Perceba que todos os elementos possuem um "name" e um "id". Mas, por que esses dois atributos juntos?

O **id**, como visto em CSS, serve para aplicar um estilo a uma tag específica do HTML e principalmente serve para identificar esse tag (DOM). Com isso podemos manipulá-la posteriormente em JavaScript. Lembre-se: o id deve ser único em uma página, não pode se repetir.

Já o atributo **name** é o que enviamos para o servidor para recuperar os dados do formulário; cada campo possui a sua variável para armazenar o valor especificado pelo usuário e, ao clicar no botão "submit", enviaremos essas variáveis para o servidor.

Antes de aprendermos a validar um formulário, devemos aprender como acessar seus elementos.

Podemos acessar os elementos de diversas formas, por exemplo:

- Através do seu **id** (cada elemento possui o seu, mesmo as listas de checkbox e radio possuem id únicos)
- Através do seu **name** (cada elemento possui o seu, e as listas de checkbox e radio possuem um único name)

Para acessar os elementos pelo ID, utilizamos o método fornecido pelo DOM:

Sintaxe:

```
document.getElementById("id_do_elemento").propriedade
```

Ou

```
document.querySelector("seletor").propriedade (Ex: seletor pode ser #id_do_elemento)
```

Para acessar os elementos pelo NAME, utilizamos o comando fornecido pelo DOM :

Sintaxe:

```
document.getElementsByName("name_do_elemento")[indice].propriedade
```

ou

```
document.querySelectorAll("seletor_css")[indice].propriedade
```

- O `getElementById` ou `querySelector` retorna um elemento propriamente dito.
- O `getElementsByName` ou o `querySelectorAll` retorna uma coleção (vetor que estudaremos mais adiante), onde cada posição corresponde a um elemento. Geralmente utilizamos esse comando para acessar listas do tipo radio e checkbox.

value

retorna o valor inserido ou selecionado no elemento, aplicável praticamente em todos os elementos do formulário

checked

retorna ou seta um elemento do tipo checkbox ou radio, aceita e retorna valores booleanos, true ou false

options[]

acessa os elementos de uma lista do tipo select

disabled

habilita ou desabilita um elemento, aceita e retorna os valores true ou false

focus()

função que move o foco para o elemento que está associado

```
<html>
<head>
<meta charset="utf-8">
<title>Exemplo 15</title>
<script>
function mostraDados(){
    let msg = "";
    msg += "Login: " + document.getElementById("login").value+" | ";
    msg += "Senha: " + document.getElementById("senha").value;
    alert(msg);
    console.log(msg)
}

window.onload = function(){
    document.getElementById("btn").onclick = function(){
        mostraDados();
    }
}
</script>
```



```
</head>
<body>
<form name="form1" id="form1" method="post" action="" >
  Login:
  <input name="login" type="text" id="login" size="15"><br>
  Senha:
  <input name="senha" type="password" id="senha" size="10"><br>
  <input type="button" name="button" id="btn" value="OK">
</form>
</body>
</html>
```

Analisemos alguns exemplos adicionais de processamento de dados de formulários

- exemplo16.html
- exemplo16-1.html
- exemplo17.html

Mauricio SAMY Silva. **HTML 5 - A Linguagem de Marcação que revolucionou a WEB**. São Paulo: Novatec, 2011

MAURICIO SAMY SILVA. **Construindo Sites Com Css e (X)Html**. São Paulo: Novatec, 2007.

ERIC FREEMAN; ELISABETH FREEMAN. **Use a Cabeça! Html Com Css e Xhtml**. São Paulo: Alta Books, 2008.

DANNY GOODMAN. **Javascript e Dhtml Guia Pratico**. São Paulo: Alta Books, 2008.

MICHAEL MORRISON. **Use a Cabeça Javascript**. São Paulo: Alta Books, 2008.

Maurício Samy Silva. **JavaScript (Guia do Programador)**. São Paulo: Novatec, 2010

<http://www.w3schools.com/html/default.asp>

<http://www.w3schools.com/css/default.asp>

<http://www.w3schools.com/js/default.asp>

<https://codeburst.io/es5-vs-es6-with-example-code-9901fa0136fc>

<http://kangax.github.io/compat-table/es2016plus/>