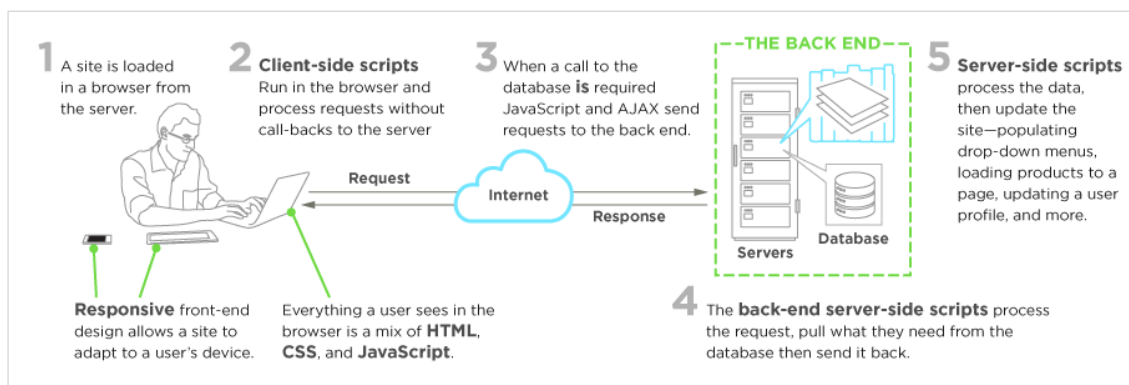


JavaScript

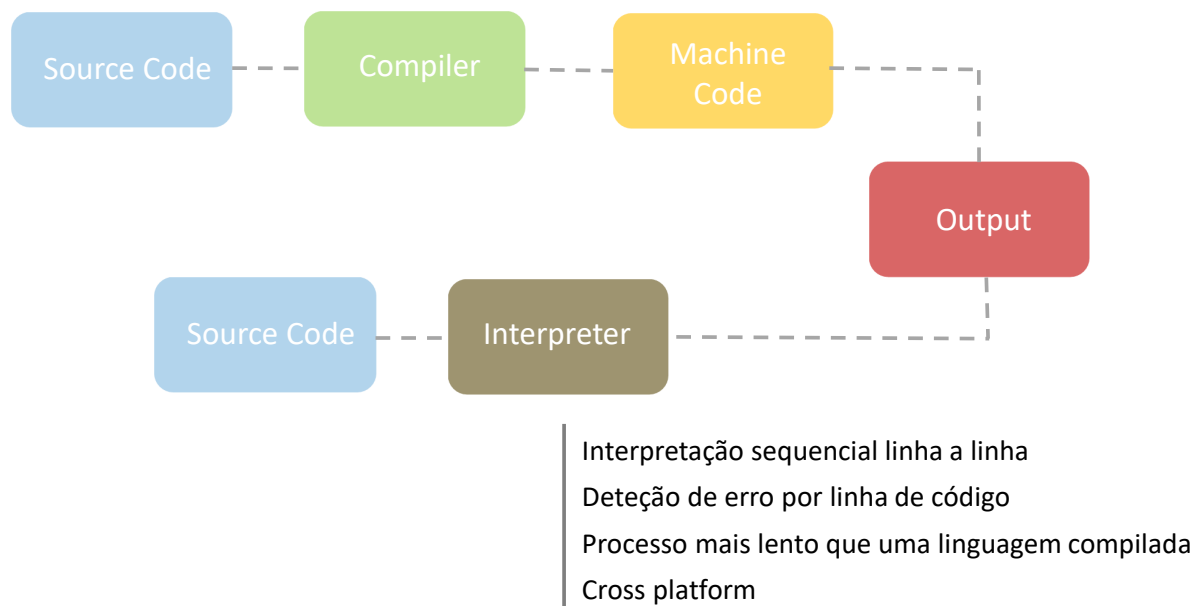
Linguagens de script

- Linguagem de programação integrada em outro programa/código
 - Linguagens interpretadas
 - Não necessitam de compilador
 - Javascript; PHP; ...



<https://www.upwork.com/hiring/development/how-scripting-languages-work/>

■ Linguagens Interpretadas vs. Linguagens Compiladas



JavaScript

■ Scripting language

“JavaScript is **THE** scripting language of the Web.”

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

- Começou por ser exclusivamente uma **client-side scripting language**
 - Interpretada diretamente pelo *browser (on the fly)*, não necessita de ser compilada
- Atualmente também utilizada no lado do servidor (*server-side*)
 - *Node.js*
- Executado
 - Após o download
 - Como resposta a um evento
- Permite:
 - Geração dinâmica de conteúdo / Efeitos
 - Melhorar a experiência do utilizador:
 - Interactividade
 - Resposta a eventos, validação de dados, ...
 - Gerir a comunicação com o servidor
 -



<https://betterdocs.co/top-scripting-languages/>

Inserção de scripts

Embedded Script

- `<script> ... </script>`
 - O *script* pode ser colocado no *head* ou no *body*
 - preferencialmente, para maximizar a performance, o *script* deve ser colocado no final do *body* não influenciando assim o tratamento dos restantes elementos HTML
 - O *script* pode ser executado quando é efetuado o **download** do *.html (sem controlo por um evento)

```
<script>
  init();

  function init(){
    alert('Script executado automaticamente');
  }
</script>
```

This page says
Script executado automaticamente

OK

Embedded Script

(diretamente definido entre as tags `<script>`)

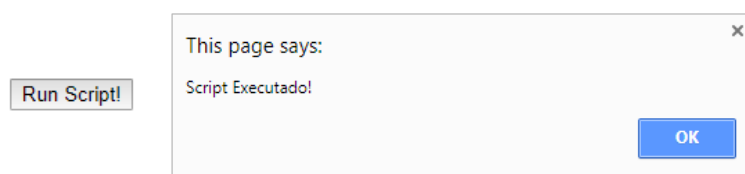
Embedded Script

- **<script> ... </script>**
 - o *script* pode ser executado como resposta a um evento, e.g. *onclick*

```
<button onclick="scriptFunction()">Run Script!</button>

<script>
  function scriptFunction(){
    alert('Script Executado!');
  }
</script>
```

Embedded Script
(diretamente definido entre as tags <script>)



Script Externo (*.js)

- **<script src="*.js"> ... </script>**

```
<body>

  <button onclick="scriptFunction()">Run Script!</button>

  <script src="external.js"></script>

</body>
```

Executa a função `scriptFunction()`
declarada no ficheiro `external.js`

Ligação ao **ficheiro externo** na tag
<script> atributo **src**

```
function scriptFunction(){
  alert('Script Executado!');
}
```

external.js

JavaScript

■ Scripts

Embebido no HTML

Ficheiros Externos
(extensão *.js)

■ Controlo da execução do script

Executado após o
download

Executado só após a
ocorrência de um evento

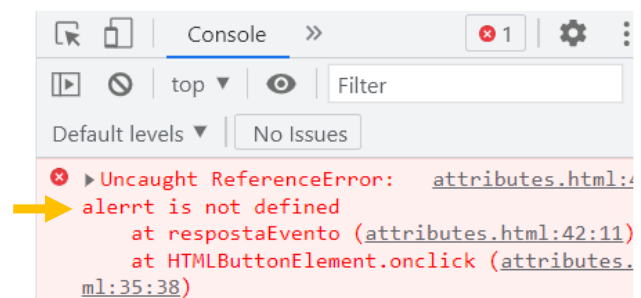
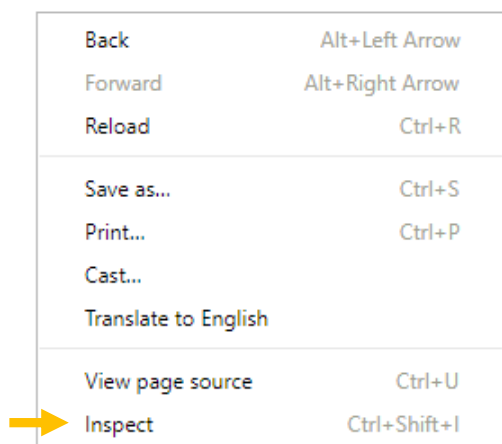
Chamada a uma função, em que a
função pode ser:

- Criada pelo utilizador
- Nativa

Browser (development tools)

■ Browser Console

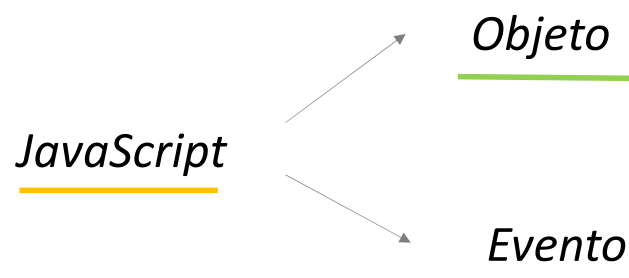
- Permite obter informação sobre o valor das variáveis, erros, *warnings*, *requests*, ...
- Muito importante para efetuar o *debugging*/controlo completo dos scripts
 - Linguagem interpretada (sequencial linha a linha)



Conceitos Chave

JS

Conceitos Chave



Conceitos Chave

Objeto

- armazenar dados, estruturação da aplicação, código mais limpo/modular
 - Identidade
 - Propriedades
 - Métodos

```
<script>
  var hotel = {

    name: 'Coimbra',
    rooms: 20,
    booked: 15,
    gym: true,
    roomTypes: ['single', 'double', 'suite'],

    checkAvailability: function () {
      return this.rooms - this.booked;
    }

  }
</script>
```



Javascript

Evento

- Ação que pode ser detetada pelo *JavaScript* e que provoca uma execução específica:

- Chamada de uma função
- A função **só é executada após a ocorrência** do respetivo evento

- Exemplos:

Evento	É disparado...
click	quando é pressionado e liberado o botão primário do mouse, trackpad, etc.
mousemove	sempre que o cursor do mouse se move.
mouseover	quando o cursor do mouse é movido para sobre algum elemento.
mouseout	quando o cursor do mouse se move para fora dos limites de um elemento.
dblclick	quando acontece um clique duplo com o mouse, trackpad, etc.

Sintaxe

JS

Sintaxe JS

- *case sensitive*.
- `//` símbolo do comentário
 - `/*` comentário para múltiplas linhas `*/`
- Um *script* é composto por um conjunto de *statements/expressions*

```
function Calculo(formulario)
{
    if(confirm("Confirma?"))
        formulario.result.value = eval(formulario.expr.value);
    else
        alert("Novos dados");
}
```

- Os *code blocks* (conjuntos de instruções) são delimitados por `{ ... }`
 - Elementos fundamentais para a estruturação do código

Sintaxe JS

■ *Expression*

- a sua execução origina sempre um valor:

- *numérico*
- *string*
- *boolean*

- podem ser parte de *statements*

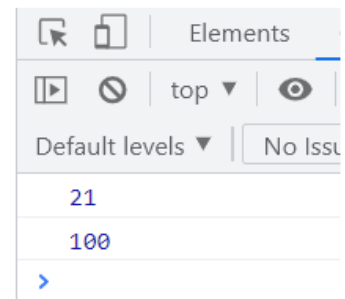
```
var sum;
var a=2;

function Modify(i){
  return i*=10
}

sum = 20;  // assign 20
sum++;    // increment

a=Modify(10); // modifies the value of a

console.log(sum);
console.log(a);
```



Sintaxe JS

■ *Statement*

- a sua execução **produz uma ação mas não gera um valor imediato**

- podem conter *expressions*
- são executados isoladamente pela ordem em que são escritos

```
var sum;  // statement
var a=2   // statement + assignment expression

function Modify(i){    //function declaration statement
  return i*=10
}

sum = 20;
sum++;

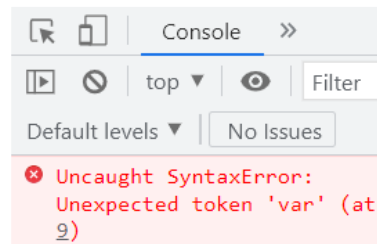
a=Modify(10);

console.log(sum);
console.log(a);
```

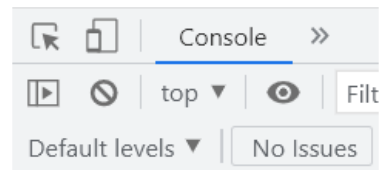
Sintaxe JS

- *Statements* não podem ser utilizados onde é esperada uma *expression*

```
var sum = var a
```



```
var a=2;  
var b = (a=5);  
console.log(b);
```



5

Sintaxe JS

- A utilização do “;” não é consensual, existem 2 perspetivas:

- “Omit Semicolon School” (*Automatic Semicolon Insertion*)

- “Add Semicolon School”

- Código mais estruturado, facilita a leitura:

- Algumas regras:

- usar sempre ; que se tratar de uma expressão *top level*

```
let x=4;
```

- não é necessário ; no final de :

- declaração de uma função *function name (...)* {...}
- if (...) {...} else {...}
- for (...) {...}
- while (...) {...}

- necessário ; quando:

- do{...} while (...);

Variáveis

JS

Variáveis

■ Declaração de variáveis

- armazenamento temporário (uma vez que após o fecho da página o browser não retém o valor atribuído à variável)
- **loosely typed**, não é necessário definir o tipo de variável uma vez que este é automaticamente assumido de acordo com a declaração (atribuição) efetuada

■ **var**

■ **let**

- A variável só é visível no bloco onde foi criada

■ **const**

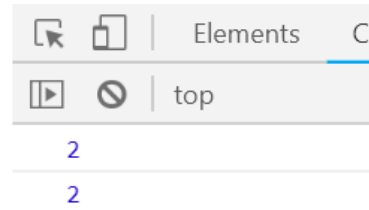
- Não permite alterar a atribuição do valor inicial (declaração).

```
var x=10;      //numeric
var x="ten"    //string
var x;
```

Scope (let)

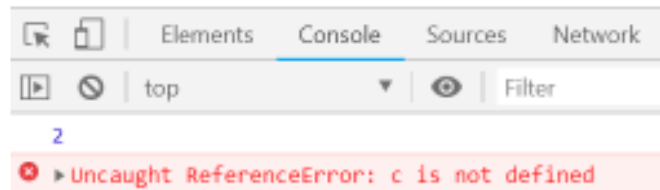
■ var (scope é a função)

```
calculateSum(2);  
  
function calculateSum (a,b = 1){  
  if (b==1)  
  {  
    var c=2;  
    console.log(c);  
  }  
  console.log(c);  
};
```



■ let (scope é o bloco {...})

```
calculateSum(2);  
  
function calculateSum (a,b = 1){  
  if (b==1)  
  {  
    let c=2;  
    console.log(c);  
  }  
  console.log(c);  
};
```

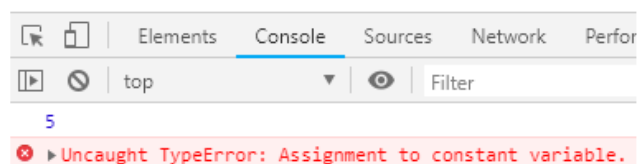


const

■ const

- declarar uma constante
 - sempre que se pretende atribuir a uma variável um valor que não é susceptível de ser alterado
 - frequentemente utilizado para a declaração de objetos (referência constante)

```
calculateSum(2);  
  
function calculateSum (a,b = 1){  
  const c = 5;  
  console.log(c);  
  c=a+b;  
  console.log(c);  
};
```



Variáveis

- Regras para definir o nome das variáveis:

```
var name = 'John';  
var age = 26;  
var isMarried = true;
```

- Significado semântico (ex: *firstName*, ...)
- *camelCase* (convenção)
- Podem começar por uma letra, por “\$” ou por *underscore* “_”.
- Não podem conter espaços nem caracteres especiais (! , / \ + * = ...)
- Não podem conter *keywords* (ex: *var* ,)
- Apesar de ser possível, não se devem diferenciar as variáveis apenas com base nas minúsculas e maiúsculas (ex: *score* e *Score*).

Variáveis

- O *JavaScript* permite a declaração de variáveis tendo por base dois grandes tipos:

Primitive Types

Armazenados como dados simples
Contêm diretamente os valores que
lhes são atribuídos

Reference Types

Armazenados como objetos

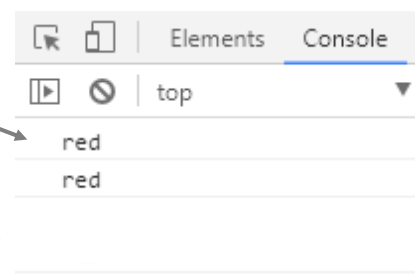
Primitive Types

JS

Primitive Types

- A variável contém diretamente o valor atribuído
 - Se for igualada a outra variável o seu valor é diretamente atribuído a essa variável
 - Apesar de partilharem o mesmo valor, as variáveis são **totalmente independentes**
 - Duas localizações de memória diferentes

```
<script>  
  var color1="red";  
  var color2=color1;  
  
  console.log(color1);  
  console.log(color2);  
  
  color1="blue";  
  
  console.log(color1);  
  console.log(color2);  
</script>
```



Primitive Types

- O JS possui 5 *primitive types*:

- number

```
var ccount=25;
```

- string

```
var name="string exemplo";
```

- boolean

```
var found=true;
```

- null

```
var obj = null;
```

- undefined

- variável sem inicialização definida

```
var data;
```

Primitive Types

- **number**

- Todos os números são representados através de *floats* de 64 bits

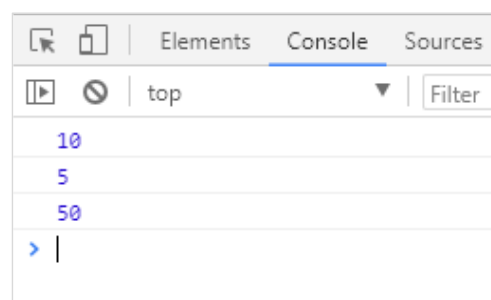
*Unlike many other programming languages, JavaScript **does not define** different types of numbers, like integers, short, long, floating-point etc.*

https://www.w3schools.com/js/js_numbers.asp

- var num1 = 50;
- var num2 = 10.5;
- var num3 = 10 * 10;

```
<script>
  var price=10;
  var quantity=5;
  var total=price*quantity;

  console.log(price);
  console.log(quantity);
  console.log(total);
</script>
```



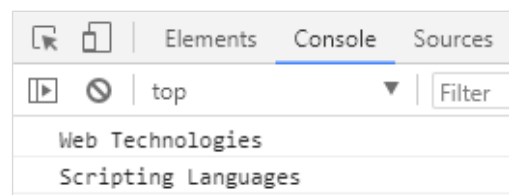
Primitive Types

■ *string*

- Cadeia de caracteres
- Declaração de uma string:
 - pode ser declarada com aspas “ ou com plica ‘, no entanto a declaração tem ser iniciada e finalizada da mesma forma
 - Quando se pretende incorporar “ ou ‘ numa string, deve declarar-se a string com o símbolo que não se pretende representar.
 - Em alternativa pode recorrer-se a uma backslash \ antes da aspa ou da plica que se pretende representar.

```
<script>
  var msg1 = "Web Technologies";
  var msg2 = 'Scripting Languages';

  console.log(msg1);
  console.log(msg2);
</script>
```

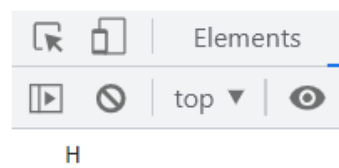


Primitive Types

■ *string*

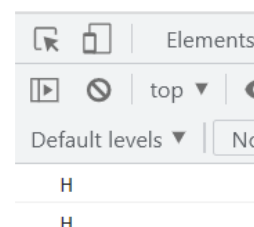
- Uma *string* permite indexação, os **índices iniciam-se em 0**:

```
var s="Hello World!"
console.log(s[0])
```



- Ao contrário de outras linguagens, ex: C, apesar de permitir indexação uma *string* não pode ser diretamente alterada

```
var s="Hello World!"
console.log(s[0])
s[0]='K'
console.log(s[0])
```



Primitive Types

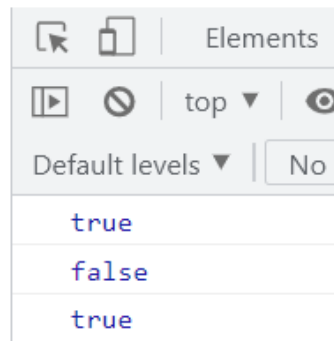
■ **boolean**

- podem assumir apenas dois valores:

- **true**

- **false**

```
var a=this;  
var b=false;  
var c=1;  
  
console.log(Boolean(a));  
console.log(Boolean(b));  
console.log(Boolean(c));
```



Reference Types (objects)

JS

Reference Types

Objeto

- É uma lista não ordenada de **propriedades**, consistindo num nome e num valor.
 - Quando o valor é uma função, cria-se um método.
- Formas diferentes de criar objetos:
 - Forma Literal
 - Operador **new** + constructor **Object()**
 - *constructor* é uma função que permite a criação de um objeto com base no operador **new**
 - Através de uma **class**

Reference Types

- Ao contrário dos *Primitive Types* os **Reference Types** não guardam o objeto diretamente na variável:
 - na realidade a variável contém um **ponteiro (referência)** para a localização em memória onde o objeto existe.

```
var obj1=new Object();
```



- Quando se atribui um objeto a uma variável, na realidade essa variável armazena um ponteiro que referencia o mesmo objeto
 - De facto existe apenas um objeto, o qual está a ser referenciado (apontado) por duas variáveis.

```
var obj1=new Object();  
var obj2=obj1;
```



Reference Types

- O mesmo objeto referenciado por duas variáveis



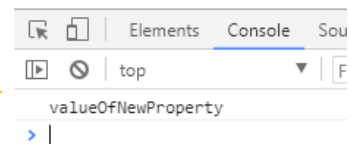
```
<script>
  var obj1=new Object();
  var obj2=obj1;
  obj1.newProperty="valueOfNewProperty";
  console.log(obj2.newProperty);
</script>
```

criado **obj1**, ponteiro para 1 objeto em memória

obj2 contém ponteiro para o mesmo objeto

adicionar nova propriedade ao objeto

uma vez que também é referenciado por obj2



Reference Types

■ Declaração de Objetos

■ Forma Literal

■ Propriedades são formadas por:

- **identificador**
- **:valor**
- múltiplas propriedades são separadas por **virgulas**
- termina com **};**

```
const books={
  title:'Javascript',
  year: 2023
};
```

■ A ordem das propriedades é irrelevante.

■ **new + constructor** Object()

A constructor is useful when you want to **create multiple similar objects** with the same properties and methods.

```
const books= new Object();

books.title= 'Javascript';
books.year= 2023
```

Propriedades / Métodos

(objects)

(.)dot notation

■ Aceder a propriedades

■ *objectName.propertyName*

```
const books={  
  title:'Javascript',  
  pages:456,  
  editor: 'Packt Books'  
}
```

```
alert('Book Title: ' + books.title)
```

Book Title: Javascript

OK

■ Aceder ao editor?

```
alert('Book Editor: ' + books.editor)
```

Book Editor: Packt Books

OK

(.)dot notation

- Aceder a métodos

- `objectName.methodName()`

```
const books={
  title:'Javascript',
  pages:456,
  editor: 'Packt Books',

  showDetails: function(){
    return ('Book title: ' + this.title + '    Book pages:  ' + this.pages)
  }
}

alert(books.showDetails())
```

Book title: Javascript Book pages: 456



Reference Types

- Alterar o valor de propriedades:

```
const books={
  title:'Javascript',
  pages:456,
  editor: 'Packt Books',

  showDetails: function(){
    return ('Book title: ' + this.title + '    Book pages:  ' + this.pages)
  }
}
```

```
books.editor='Willey'
alert('Editor:  ' + books.editor)
```

Editor: Willey



- Os objetos podem ser alterados em qualquer momento:

- Propriedades podem ser alteradas/adicionadas/removidas

Reference Types

- Criar Propriedades/Métodos

```
const books={
  title:'Javascript',
  pages:456,
  editor: 'Packt Books',

  showDetails: function(){
    return ('Book title: ' + this.title + '    Book pages: ' + this.pages)
  }
}
```

```
books.chapters = 14
alert('Book Chapters: ' + books.chapters)
```

Book Chapters: 14



- Os objetos podem ser alterados em qualquer momento:
 - Propriedades podem ser alteradas/adicionadas/removidas

Reference Types

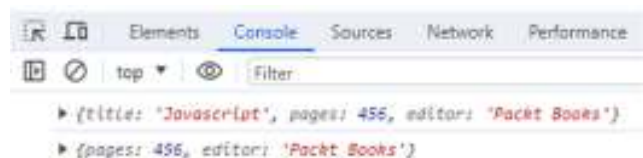
- Apagar propriedades/métodos

```
const books={
  title:'Javascript',
  pages:456,
  editor: 'Packt Books',
}

console.log(books)

delete(books.title)

console.log(books)
```



- Os objetos podem ser alterados em qualquer momento:
 - Propriedades podem ser alteradas/adicionadas/removidas

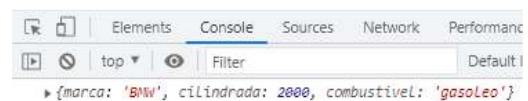
Exercício 1 (properties/methods)

Exercício 1

- Criar um objeto (forma literal) carro, cuja propriedades são:

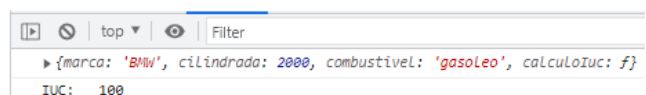
- Marca: BMW
- Cilindrada: 2000
- Combustível: gasoleo

```
const carro={  
  marca: 'BMW',  
  cilindrada:2000,  
  combustivel:'gasoleo'  
}  
console.log(carro)
```



- criar um método que calcule o imposto de circulação (0.05€/cc)

```
const carro={  
  marca: 'BMW',  
  cilindrada:2000,  
  combustivel:'gasoleo',  
  
  calculoIuc: function(){  
    return (this.cilindrada*0.05)  
  }  
}  
console.log(carro)  
console.log('IUC: ' + carro.calculoIuc())
```



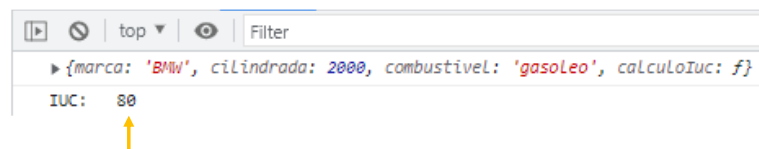
Exercício 1

- garanta que o valor por cc para cálculo do imposto é passado como argumento ao método calculoIuc()

```
var u=0.05
const carro={
  marca: 'BMW',
  cilindrada:2000,
  combustivel:'gasoleo',

  calculoIuc: function(c){
    return (this.cilindrada*c)
  }
}
console.log(carro)
console.log('IUC: ' + carro.calculoIuc(u))
```

- Faça variar o valor do coeficiente/cc
 - ex: 0.04/cc



Exercício 1

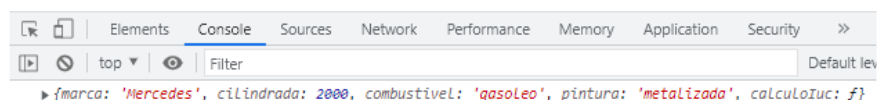
- altere o valor da propriedade marca de 'BMW' para 'Mercedes', sem alterar a declaração original do objeto.

```
carro.marca='Mercedes'
console.log(carro)
```



- Adicione a propriedade pintura com o valor "metalizada"

```
carro.pintura='metalizada'
console.log(carro)
```



Built-in Types (Reference Types)

Built-in Types

■ Built-in types

- Criar *objects* com o constructor (keyword **new**)

- Object `const books = new Object()`

- Array `const items = new Array()`

- Date

- Error

- Function

- ...

- Os *built-in types* podem ter formas literais

- Sintaxe literal permite a criação de objetos sem utilizar o operador `new` e o respetivo constructor

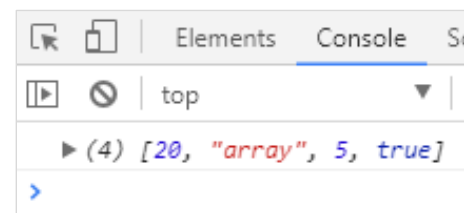
`const items = [];`

Built-in Types

■ Array

- Permite armazenar um conjunto de valores relacionados
- Ao contrário de outras linguagens:
 - O *array* não tem de ser declarado com uma dimensão
 - Inclui **diferentes tipos de dados** no mesmo *array*
- Notação literal
 - Definidos com `[...]` e elementos separados por vírgulas

```
const values=[20,'array',5, true];  
console.log(values);
```



- Baseado num *constructor*:

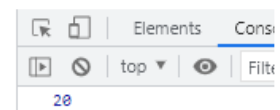
```
const valuesConstructor=new Array(20,'array',5,true)  
console.log(valuesConstructor)
```

Array

■ Indexação

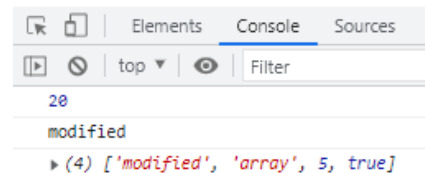
- nomeArray [*posição*]
- Índices iniciam-se em **zero**

```
const values=[20,'array',5, true];  
console.log(values[0]);
```



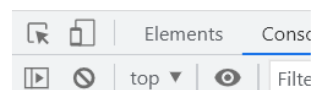
- Um array pode ser diretamente alterado

```
values[0]='modified';  
console.log(values[0]);  
console.log(values);
```



- Propriedade **length** é muito importante (retorna a dimensão do *array*)

```
console.log(values.length);
```



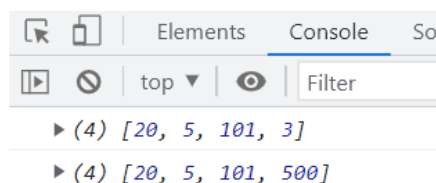
Array

- É usual declarar um array com base em const
 - Quando se define o array como const, na realidade define-se uma referência constante para o array o que não impede de alterar os seus elementos

```
const a=[20,5,101,3]
console.log(a)
a=[5,4]
console.log(a)
```

```
▶ (4) [20, 5, 101, 3]
✖ ▶ Uncaught TypeError: Assignment to constant variable.
```

```
const a=[20,5,101,3]
console.log(a)
a[3]=500
console.log(a)
```



```
Elements Console So
top Filter
▶ (4) [20, 5, 101, 3]
▶ (4) [20, 5, 101, 500]
```

- previne que a referência para o objeto não é alterada **o que não impede** a alteração das suas propriedades

Array



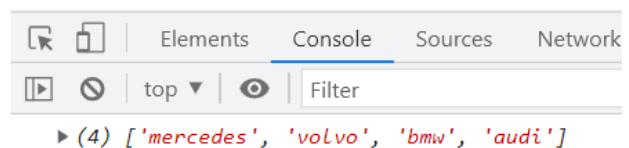
<https://medium.com/@mandeepkaur1/a-list-of-javascript-array-methods-145d09dd19a0>

Exercício 2 (Array object)

Exercício 2

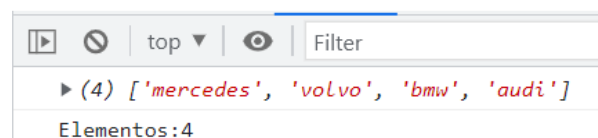
- Declare um array com os valores: mercedes, volvo, bmw, audi

```
const cars=['mercedes','volvo','bmw','audi']  
console.log(cars)
```



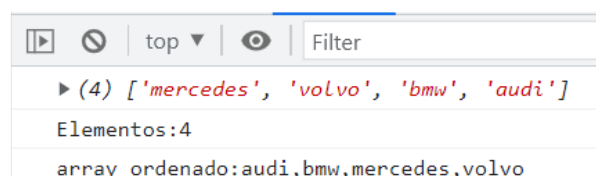
- Determine qual o número de elementos do array (propriedade length)

```
console.log('Elementos:'+cars.length)
```



- Ordene o array (método: sort())

```
console.log('array ordenado:'+cars.sort())
```



■ Functions

- A forma literal é muito mais usada para a declaração de funções do que baseada num constructor, uma vez que é menos sujeita a erros e mais fácil de manter

```
function reflect(value){  
    return value;  
}
```



- Tendo por base um constructor seria:
 - Todo o corpo da função teria de ser encapsulado numa string, o que como é óbvio tem várias desvantagens (dificulta a implementação, dificulta o debug do código, ...)

```
var reflect=new Function("value","return value;");
```



- À excepção do *built-in type Function* (notação literal) não existe uma forma correta ou errada de instanciar *built-in types*.

Primitive Wrapper Types

Primitive Wrapper Types

- Existem três **Primitive Wrapper Types**:

- *String*
- *Number*
- *Boolean*

- Possibilitam o funcionamento da **dot notation** com *Primitive Types* da mesma forma (**dot notation**)

“... any attempt to access a property on a primitive, JavaScript will implicitly create a temporary wrapper object ...” <https://developer.mozilla.org/en-US/docs/Web/JavaScript/>

- Ao contrário dos *reference type*, um *primitive wrapper type* **não permite a adição de propriedades**.

Strings

String Properties

Property	Description
<u>constructor</u>	Returns the string's constructor function
<u>length</u>	Returns the length of a string
<u>prototype</u>	Allows you to add properties and methods to an object

String Methods

Method	Description
<u>charAt()</u>	Returns the character at the specified index (position)
<u>charCodeAt()</u>	Returns the Unicode of the character at the specified index
<u>concat()</u>	Joins two or more strings, and returns a new joined strings
<u>endsWith()</u>	Checks whether a string ends with specified string/characters
<u>fromCharCode()</u>	Converts Unicode values to characters
<u>includes()</u>	Checks whether a string contains the specified string/characters
<u>indexOf()</u>	Returns the position of the first found occurrence of a specified value in a string
<u>lastIndexOf()</u>	Returns the position of the last found occurrence of a specified value in a string

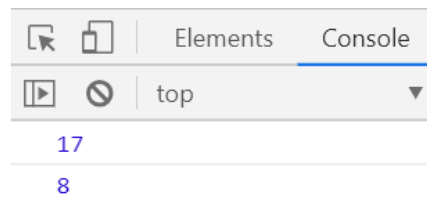
https://www.w3schools.com/jsref/jsref_obj_string.asp

Array

■ Propriedade (exemplo: length)

<script>

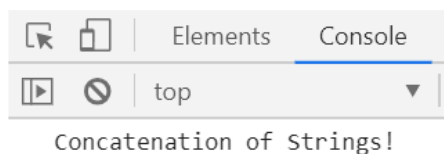
```
let firstStr='Concatenation of ';  
let secondStr='Strings!';  
console.log(firstStr.length);  
console.log(secondStr.length);
```



■ Método (exemplo: concat())

<script>

```
let firstStr='Concatenation of ';  
let secondStr='Strings!';  
console.log(firstStr.concat(secondStr));
```



Variáveis

| <u>Primitive Types</u> | <u>Reference Types (Built-in)</u> | <u>Primitive Wrapper Types</u> |
|------------------------|-----------------------------------|--------------------------------|
| Number | Object | Number |
| String | Array | String |
| boolean | Date | Boolean |
| null | Error | |
| undefined | Function | |
| | RegExp | |
| | Math | |
| | ... | |

Operadores

JS

Operadores

■ Aritméticos

| Operator | Description |
|----------|----------------------------------|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| ** | Exponentiation (<u>ES2016</u>) |
| / | Division |
| % | Modulus (Division Remainder) |
| ++ | Increment |
| -- | Decrement |

Operadores

■ Atribuição

| Operator | Example | Same As |
|----------|---------|------------|
| = | x = y | x = y |
| += | x += y | x = x + y |
| -= | x -= y | x = x - y |
| *= | x *= y | x = x * y |
| /= | x /= y | x = x / y |
| %= | x %= y | x = x % y |
| **= | x **= y | x = x ** y |

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

Operadores

■ Lógicos

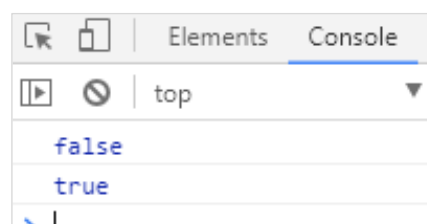
| Operator | Description |
|----------|-------------|
| && | logical and |
| | logical or |
| ! | logical not |

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

```
<script>
  var a=5;
  var b=4;
  var c=6;
  var d=8;

  console.log((a>b)&&(c>d));

  console.log((a>b)||(c>d));
</script>
```



Operadores

■ Comparação

| Operator | Description |
|----------|-----------------------------------|
| == | equal to |
| === | equal value and equal type |
| != | not equal |
| !== | not equal value or not equal type |
| > | greater than |
| < | less than |
| >= | greater than or equal to |
| <= | less than or equal to |
| ? | ternary operator |

https://www.w3schools.com/js/js_comparisons.asp

Operadores

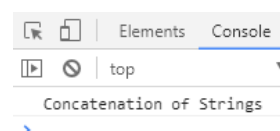
■ Strings

■ Concatenação (+)

- Operador muito frequentemente utilizado

```
<script>
  var strC = "Concatenation " + "of" + " Strings";
  console.log(strC);
</script>
```

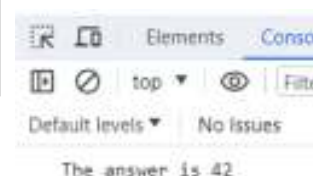
Concatenação



- *type coercion* (um dos argumentos é uma *string*)

- Conversão automática de um tipo de dados em outro tipo de dados

```
var num = 42;
var str = "The answer is " + num; // Coercion: number to string
console.log(str); // Output: "The answer is 42"
```



Operadores

■ Precedência de Operadores

The following table is ordered from highest (20) to lowest (1) precedence.

| Precedence | Operator type | Associativity | Individual operators |
|------------|-----------------------------|---------------|----------------------|
| 20 | Grouping | n/a | (...) |
| 19 | Member Access | left-to-right | |
| | Computed Member Access | left-to-right | [...] |
| | new (with argument list) | n/a | new ... (...) |
| | Function Call | left-to-right | (...) |
| 18 | new (without argument list) | right-to-left | new ... |
| 17 | Postfix Increment | n/a | ... ++ |
| | Postfix Decrement | | ... -- |
| 16 | Logical NOT | right-to-left | ! ... |
| | Bitwise NOT | | ~ ... |
| | Unary Plus | | + ... |
| | Unary Negation | | - ... |
| | Prefix Increment | | ++ ... |
| | Prefix Decrement | | -- ... |
| | typeof | | typeof ... |
| | | | |

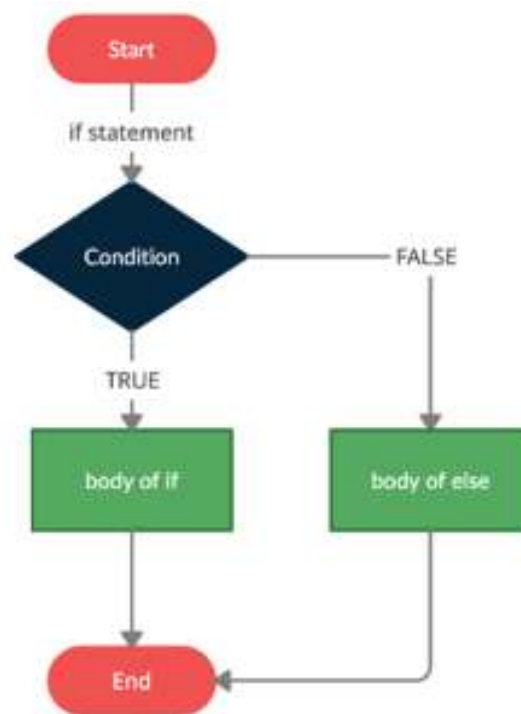
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

Estruturas de Seleção (condicionais)

JS

Seleção

- if (condição) else ...

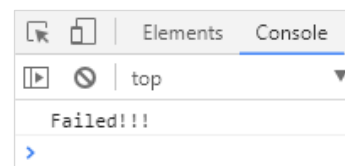


Seleção

- if ...else

```
<script>
  var threshold=50;
  var grade=40;

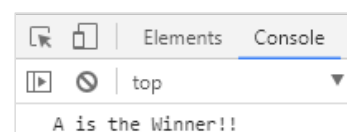
  if (grade>=threshold)
    {console.log("Aproved!!");}
  else
    {console.log("Failed!!");}
</script>
```



- condições encadeadas

```
<script>
  var scoreA=60;
  var scoreB=50;

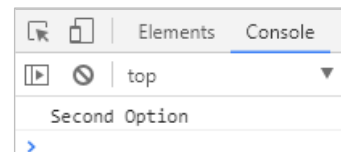
  if (scoreA>scoreB)
    {console.log("A is the Winner!!");}
  else if (scoreA<scoreB)
    {console.log("B is the Winner!!");}
  else
    {console.log("Draw !!")}
</script>
```



- switch (var)

```
{ case value1: statements; break;  
  case value2: statements; break; ...  
  default: statements }
```

```
<script>  
  var msg,a;  
  a=2;  
  
  switch(a)  
  {  
    case (1): msg="First Option"; break;  
    case (2): msg="Second Option"; break;  
    case (3): msg="Third Option"; break;  
    default: msg="No option!"; break;  
  }  
  console.log(msg);  
</script>
```



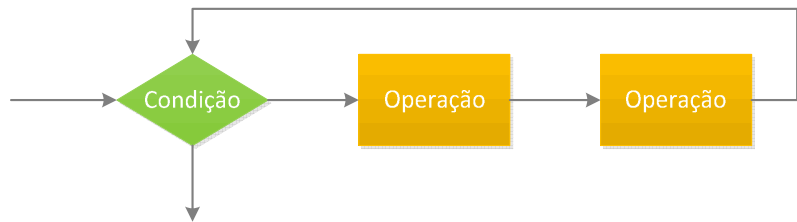
Estruturas de Repetição

JS

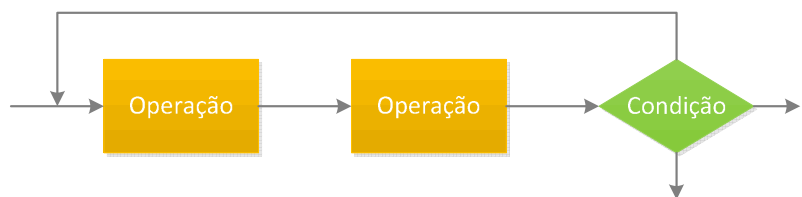
Repetição

■ Estruturas de Repetição / Ciclos (Loops)

■ **while (condição){**
// código bloco
}



■ **do {**
// código bloco
} while (condição);

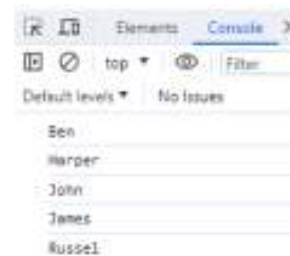


Repetição

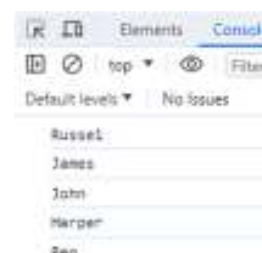
■ Ciclos (Loops)

■ **for (initialization; condition; variable update) { ... }**

```
const names=['Ben', 'Harper','John','James', 'Russel'];  
for (let i=0; i<names.length; i++)  
    console.log(names[i]);
```



```
const names=['Ben', 'Harper','John','James', 'Russel'];  
for (let i=names.length-1; i>=0; i--)  
    console.log(names[i]);
```

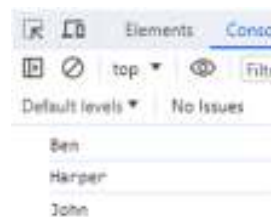


Repetição

■ Ciclos (Loops)

■ **break**

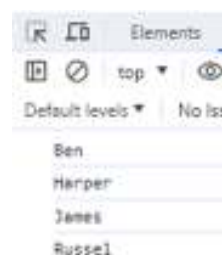
```
const names=['Ben', 'Harper','John','James', 'Russel'];
for (let i=0; i< names.length; i++){
  console.log(names[i]);
  if (i==2)
    break;}
```



break: interrompe o funcionamento do ciclo

■ **continue**

```
const names=['Ben', 'Harper','John','James', 'Russel'];
for (let i=0; i< names.length; i++){
  if (i==2)
    continue;
  console.log(names[i]);
}
```



continue: salta diretamente para o final da iteração e prossegue com o ciclo, neste caso ignora a 3ª iteração

falsy / truthy

■ **falsy**

- valores tratados como **false**

| Value | Description |
|-----------------------------|------------------------------|
| var highScore = false; | false |
| var highScore = 0; | 0 |
| var highScore = ''; | empty value |
| var highScore = 10/'score'; | NaN (not a number) |
| var highScore; | variável sem valor atribuído |

■ **truthy**

- valores tratados como **true**

| Value | Description |
|-------------------------|-----------------------------|
| var highScore = true; | true |
| var highScore = 1; | número ≠ 0 |
| var highScore = 'xxxx'; | string com conteúdo |
| var highScore = 10/5 | resultado de um cálculo ≠ 0 |

Exercício 3 (Repetição)

Exercício 3

- Crie um array com a designação cars, contendo os valores:
 - 'mercedes','volvo','bmw','audi','kia','fiat','renault'
- crie um novo array (newcars) com base no array anterior, o qual contém apenas os valores correspondentes às posições pares
 - deve utilizar o método push() *"The push() method adds new items to the end of an array."*

```
const cars=['mercedes','volvo','bmw','audi','kia','fiat','renault']
const newCars=[]
for (let i=0; i<cars.length;i++ )
{
  if(i%2==0)
    newCars.push(cars[i])
}
console.log(cars)
console.log(newCars)
```

```
▶ (7) ['mercedes', 'volvo', 'bmw', 'audi', 'kia', 'fiat', 'renault']
▶ (4) ['mercedes', 'bmw', 'kia', 'renault']
```


Funções

JS

Funções

- Conjunto de declarações agrupadas para executar uma tarefa específica.
 - Reutilização de código; flexibilidade; ...
 - **Declaração de uma função** (notação literal):

```
function name (param1, param2, ....){  
    código a ser executado;  
}
```

```
function firstFunction(){  
    document.write("hello");  
}
```

- Prefixos uteis para nomes de função:
 - create, show, get, check,

Funções

- Chamada à função:
 - Efetuada através do nome da função seguido de parêntesis
 - Código só é executado após a respetiva chamada

```
firstFunction();
```

- O browser percorre todo o script antes da execução de cada declaração, mas preferencialmente a função deve ser declarada antes da sua chamada.

Funções

- Parâmetros
 - Declaração de uma função com parâmetros:

```
function calculateArea(width,height){  
    return width*height;  
}
```

- Chamada a uma função:
 - Especificação direta dos valores dos argumentos

```
calculateArea (2,4);
```

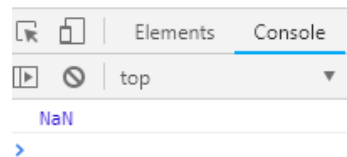
- Argumentos da função definidos através de variáveis

```
rectWidth=2;  
rectHeight=4;  
calculateArea(rectWidth, rectHeight);
```

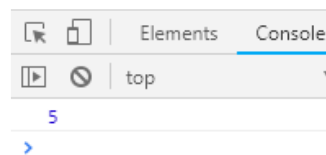
Funções

- Possível definir *default values* para os parâmetros

```
function calculateSum (a,b){  
    return a+b;  
};  
  
console.log(calculateSum(4));
```



```
function calculateSum (a,b = 1){  
    return a+b;  
};  
  
console.log(calculateSum(4));
```



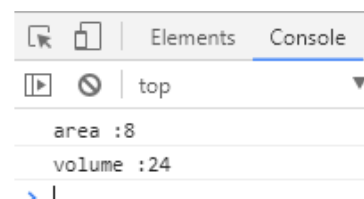
Funções

- Retorno de um valor único:

```
function calculateArea(width,height){  
    return width*height;  
}  
var wallOne=calculateArea(3,5);  
var wallTwo=calculateArea(8,5);
```

- Retorno de vários valores com base em *arrays*:

```
<script>  
  
    function getDimensions(w,h,d)  
    {  
        var area=w*h;  
        var volume=area*d;  
        var values=[area,volume];  
  
        return values;  
    }  
  
    console.log("area : " + getDimensions(2,4,3)[0]);  
  
    console.log("volume : " + getDimensions(2,4,3)[1]);  
  
</script>
```



Passagem de Parâmetros

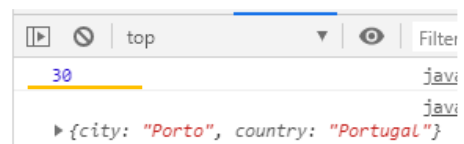
- Os *Primitive Type* e os Objetos são passados à função de forma diferente
 - Nos *Primitive Types* é feita a **passagem do valor** do argumento:
 - todas as alterações efetuadas no parâmetro no interior da função não alteram o valor original
 - Nos *Reference Types* a passagem é **feita por referência**:
 - as alterações feitas no interior da função são na realidade efetuadas no objeto original
 - Objeto é referenciado pelas diversas variáveis

```
<script>
  var age=30;
  var citizen={city:'Coimbra', country:'Portugal'};

  function changeValues(a,b){
    a=50;
    b.city='Porto'
  }

  changeValues(age,citizen);

  console.log(age);
  console.log(citizen);
</script>
```



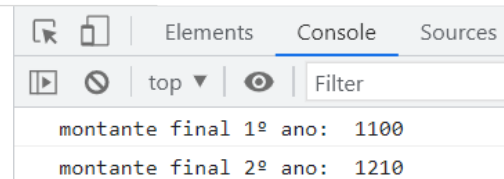
Exercício 4 (funções)

Exercicio 4

- Declare uma função que calcule a valorização de um depósito ao fim do primeiro e do segundo ano.
- A função recebe o montante investido e a taxa de juro (fixa) e deve mostrar o montante ao fim do 1º e do 2º ano. (ex: montante 1000€, juros 10%)

```
function calculaJuros(m,j){  
  const values=[];  
  j/=100;  
  values[0]=m*(1+j);  
  values[1]=values[0]*(1+j);  
  return values  
}
```

```
console.log('montante final 1º ano: ' + calculaJuros(1000,10)[0])  
console.log('montante final 2º ano: ' + calculaJuros(1000,10)[1])
```



| | | | |
|----------|-------|---------|------|
| montante | final | 1º ano: | 1100 |
| montante | final | 2º ano: | 1210 |

Diferentes Tipos de Função

JS

Funções

- O JavaScript admite formas diferentes de criar uma função:

Declaração de Função (statement)

```
function calculateSum (a,b = 1){  
    return a+b;  
}
```

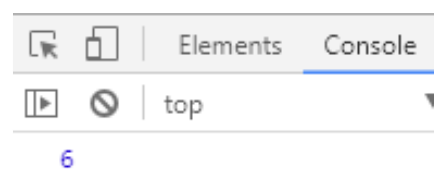
Function Expression (anonymous function)

```
var calculateSum = function (a,b = 1){  
    return a+b;  
}
```

Funções

- Declaração de Função
 - Chamada à função:

```
<script>  
    var area;  
    area=calculaArea(2,3);  
  
    function calculaArea(width,height)  
    {  
        return width*height;  
    }  
  
    console.log(area);  
</script>
```



A declaração normal de uma função permite que a chamada à função seja **executada antes** da declaração da função

Funções

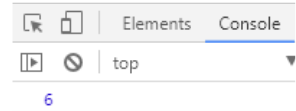
■ *Function Expression (anonymous function)*

- A declaração de uma função pode ser incorporada numa expressão
 - Não é especificado o nome da função depois de **function (anonymous function)**

```
<script>
  var calculaArea=function(width,height){
    return width*height;
  }

  var area=calculaArea(2,3);
  console.log(area);
</script>
```

A declaração de uma **anonymous function** exige que a chamada à função seja efetuada **depois da expressão**



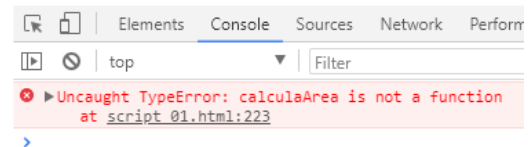
- É tratada como uma expressão, ou seja a função é detetada só após a sua chamada

```
<script>
  var area=calculaArea(2,3);
  console.log(area);

  var calculaArea=function(width,height){
    return width*height;
  }

  console.log(area);
</script>
```

Área não calculada, uma vez que a chamada à função foi feita antes da expressão onde está declarada



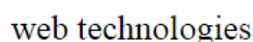
Funções

- A **anonymous function (function expression)** é particularmente importante no Javascript
 - definição de métodos de um objeto

```
var course={
  name:"web technologies",
  displayName:function(){
    document.write(course.name);
  }
}
course.displayName();
```

anonymous function define o método *displayName*

A definição de uma propriedade (nome:valor) é igual à definição de um método (nome:valor), neste último o valor é uma **anonymous function**



- event handling

- O JavaScript admite formas diferentes de criar uma função:

Declaração de Função

```
function calculateSum (a,b = 1){  
    return a+b;  
}
```

- Sintaxe abreviada
- Palavra **function** é eliminada
- A seta => aponta para o corpo da função
- Caso não existam parâmetros são necessários parênteses vazios
- Se a função possuir várias declarações são necessárias chavetas e a key word **return**

Function Expression

(anonymous function)

```
var calculateSum = function (a,b = 1){  
    return a+b;  
}
```

Arrow Functions

```
var calculateSum = (a, b = 1) => a+b;
```

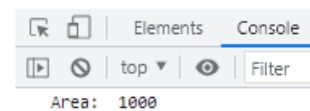
Redução do Acoplamento

Keyword *this*

- A keyword **this** pertence a um determinado *scope*, isto é:
 - Se usado num **método de um objeto** a keyword **this** refere-se a esse objeto
 - Se o objeto é criado com base numa class a utilização do **this** refere-se a cada instância em particular

```
const shape={
  width:50,
  height:20,
  calculateArea:function(){
    return this.width*this.height
  }
}
console.log('Area: ' + shape.calculateArea())
```

width / height do objeto shape



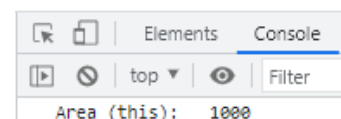
- Se aplicado numa função destinada a definir o método de um objeto, **this** refere-se ao objeto que contém o método

```
var width=100;
const shape={
  width:50,
  height:20
}

var calculateArea=function(){
  return this.width*this.height
}

shape.getArea=calculateArea
console.log('Area (this): ' + shape.getArea())
```

this refere-se ao objeto **shape**, uma vez que a função **calculateArea** é usada para definir o método **getArea** do objeto **shape**



- Se aplicado isolado ou numa função genérica, **this** refere-se ao global object (window object)
- Se aplicado num evento, **this** refere-se ao elemento que recebe o evento

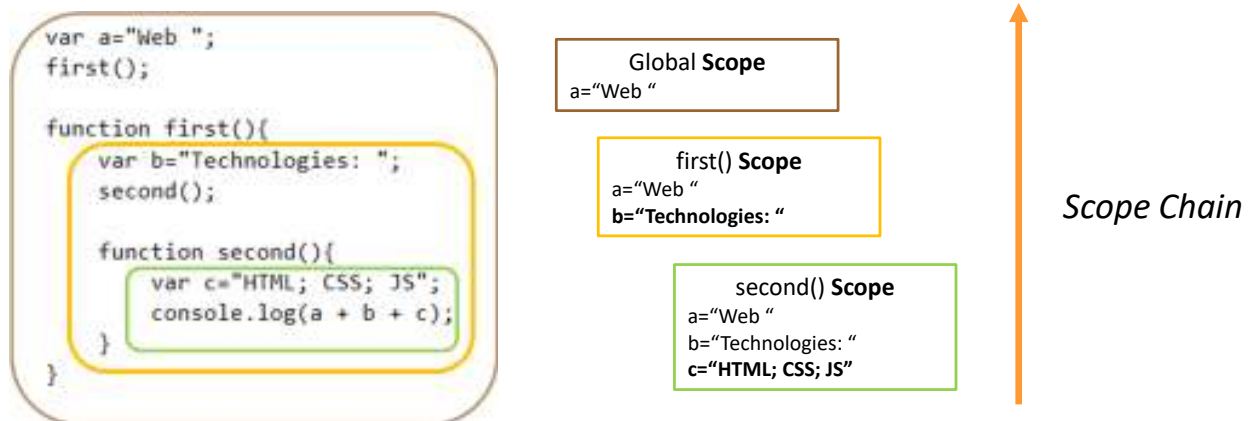
Scope das Variáveis

Scope

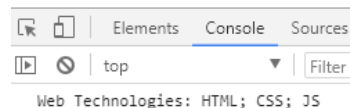
- Variável Global
 - Criada fora de qualquer função
 - É visível em todo o script (contexto global)
 - Armazenamento permanente (a variável existe enquanto a página estiver ativa no browser)
 - requer mais memória que as variáveis locais (eliminadas após a execução a função onde foram declaradas)
 - possíveis conflitos na nomenclatura das variáveis
- Variável Local
 - Criada no interior de uma função
 - Só é visível na função onde foi criada
 - Uma vez terminada a execução da função a variável é eliminada

Scope

- Cada nova função cria um *scope*
 - Um espaço/ambiente onde as respetivas variáveis são acessíveis



- Uma função que é **definida no interior** de uma outra função, tem **acesso ao scope da função que a envolve** (*scope chain*)



<https://www.udemy.com/the-complete-javascript-course/learn/v4/t/lecture/5869134?start=600>

Criação de Objetos JS

JavaScript

Objeto

Evento

JavaScript Objects



In JavaScript, objects are king. If you understand objects, you understand JavaScript.

In JavaScript, almost "everything" is an object.

- Booleans can be objects (or primitive data treated as objects)
- Numbers can be objects (or primitive data treated as objects)
- Strings can be objects (or primitive data treated as objects)
- Dates are always objects
- Maths are always objects
- Regular expressions are always objects
- Arrays are always objects
- Functions are always objects
- Objects are objects

In JavaScript, all values, except primitive values, are objects.

Primitive values are: strings ("John Doe"), numbers (3.14), true, false, null, and undefined.

http://www.w3schools.com/js/js_object_definition.asp

Objetos

- Um objeto representa uma entidade física ou conceptual.
 - Tem estado (**propriedades**), comportamento (**métodos**) e identidade (**nome**).

■ Propriedades

- Valores associados ao objeto

■ Métodos

- Ações associadas ao objeto
- Mesma sintaxe que propriedades
 - O valor é uma função

```
<script>
  var hotel = {
    name: 'Coimbra',
    rooms: 20,
    booked: 15,
    gym: true,
    roomTypes: ['single', 'double', 'suite'],

    checkAvailability: function () {
      return this.rooms - this.booked;
    }
  }
</script>
```

3 formas distintas
de criar objetos

Forma Literal

new + Object()

class

Forma Literal

```
const hotel={  
  
  name:'Coimbra',  
  rooms:20,  
  booked:15,  
  gym:true,  
  roomTypes:['single','double','suite'],  
  
  checkAvailability:function(){  
    return this.rooms - this.booked  
  }  
}
```

Nome do objeto

Propriedades em que os valores
podem ser:

string
number
boolean
array

Métodos

O scope do **this** é o objeto hotel

Objetos

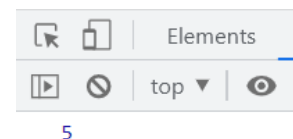
new + Object()

```
const hotel= new Object();

hotel.name='Coimbra';
hotel.rooms=20;
hotel.booked=15;
hotel.gym=true;
hotel.roomTypes=['single','double','suite'];

hotel.checkAvailability=function(){
    return this.rooms - this.booked;
}

console.log(hotel.checkAvailability());
```



- cria um objeto ao qual **se adicionam propriedades e métodos**
- sintaxe totalmente diferente da forma literal.

Objetos

class

- Definição de uma *class*: “A class is a blueprint for objects”
- Todas as classes possuem um **constructor** (método) onde se inicializam as propriedades

```
class Writer{
    constructor(firstName,lastName){
        this.fname=firstName,
        this.lname=lastName
    }
    showName(){
        console.log('Writer name: ' + this.fname + this.lname)
    }
}
```

nome da Classe (Por convenção o nome da class inicia-se com maiúscula)

constructor (propriedades)

método

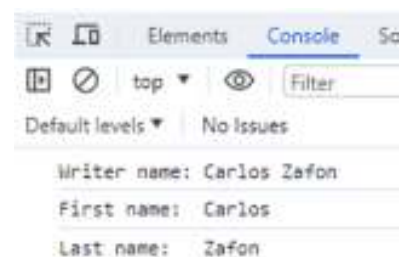
class

- Criar um objeto passa pela criação de uma **instância da class**

```
const writerInstance=new Writer('Carlos',' Zafon')

writerInstance.showName()

console.log('First name: ' + writerInstance.fname)
console.log('Last name: ' + writerInstance.lname)
```



Objetos

- As classes permitem herança
 - uma classe pode fazer o **extend** de outra classe (herdar as suas propriedades e métodos)

```
class Person{
  constructor(a,b){
    this.age=a,
    this.country=b }
  showFeatures(){
    console.log('Age: ' + this.age + ' Country: ' + this.country)
  }
}
```

Person (Classe Pai)

```
class Writer extends Person{
  constructor(firstName,lastName,c,d){
    super(c,d)
    this.fname=firstName,
    this.lname=lastName }
  showName(){
    console.log('Writer name: ' + this.fname + this.lname)
  }
}
```

Writer (Classe Filho)

super() na classe filha é obrigatório, executado para invocar o constructor da classe pai

■ Herança

```
const writerInstance=new Writer('Carlos','Zafon', 50, "Spain")
```

```
writerInstance.showName()
```

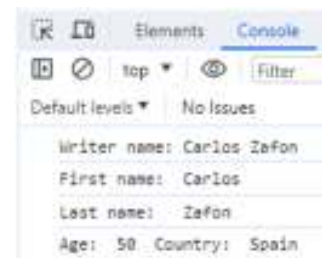
```
console.log('First name: ' + writerInstance.fname)
```

```
console.log('Last name: ' + writerInstance.lname)
```

Writer

```
writerInstance.showFeatures()
```

Person



Exercício 5(Objeto)

- Com base no exemplo anterior, crie uma nova instância da class Writer.
 - Deve mostrar na consola o nome completo do escritor (José Luís Peixoto) assim como a idade (45 anos) e o respetivo país (Portugal).
 - Considere os métodos showName() e show Features()

```
const object2=new Writer('José Luís', 'Peixoto', 45, 'Portugal')  
  
object2.showName()  
object2.showFeatures()
```

```
Writer name: José Luís Peixoto  
Age: 45 Country: Portugal  
>
```

Criação de Objetos em JS

Resumo

Objetos

Forma Literal

- Sintaxe específica

- Pouco eficiente quando se pretendem criar múltiplas instâncias, obriga a definição repetida dos métodos

new + Object()

- Permite a criação de um objeto vazio ao qual se adicionam propriedades e métodos

class

- Definir uma classe e criar múltiplas instâncias dessa classe
- Solução mais eficaz quando se pretende a definição de múltiplos objetos (instâncias)
- Só disponível no ES6

Propriedades / Métodos

Resumo

(.)dot notation

■ Propriedades

- ***objectName.propertyName***

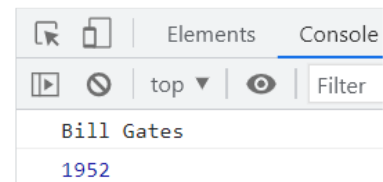
```
console.log(bill.name);
```

```
const bill={  
  name: 'Bill Gates',  
  age:72,  
  height:185,  
  calculateYearBirth: function(a){  
    return a-this.age;  
  }  
}
```

■ Métodos

- ***objectName.methodName()***

```
console.log(bill.calculateYearBirth(2024));
```



Reference Types

■ Adicionar Propriedades

- As propriedades podem ser adicionadas quando o objeto é criado ou então posteriormente em qualquer momento
 - Por defeito, em JavaScript os objetos podem ser sempre modificados

```
const books={  
  title:'Javascript',  
  editor: 'Packt Books',  
}
```

```
books.pages=456  
console.log(books)
```



Objetos

■ Remover propriedades

- Da mesma forma que é possível criar também é possível remover propriedades em qualquer altura através do operador **delete**

```
const books={  
  title:'Javascript',  
  editor: 'Packt Books',  
}
```

```
books.pages=456
```

```
console.log(books)
```

```
delete(books.editor)
```

```
console.log(books)
```



```
▶ {title: 'Javascript', editor: 'Packt Books', pages: 456}  
▶ {title: 'Javascript', pages: 456}
```


- A propriedade foi removida como tal não se encontra definida
- Existem métodos que impedem a possibilidade de alterações nas propriedades de um objeto:
 - *Object.preventExtensions()* ; *Object.seal()*; *Object.freeze()*

Objetos

■ Detetar propriedades

- Existem várias formas de detetar propriedades mas a forma mais fiável passa por utilizar o operador **in**

```
const books={  
  title:'Javascript',  
  editor: 'Packt Books',  
}  
  
books.pages=456  
  
console.log(books)  
  
delete(books.editor)  
  
console.log('editor' in books)
```



```
▶ {title: 'Javascript', editor: 'Packt Books', pages: 456}  
▶ {title: 'Javascript', pages: 456}  
false
```

JavaScript Built-in Objects

JavaScript Built-in Objects

JavaScript Reference

The references describe the properties and methods of all JavaScript objects, along with examples.

| | | | | |
|------------|---------|--------|-----------|--------|
| Array | Boolean | Date | Error | Global |
| JSON | Math | Number | Operators | RegExp |
| Statements | String | | | |

<https://www.w3schools.com/jsref/default.asp>

- **String** (*Primitive Wrapper Types)
- **Number** (*Primitive Wrapper Types)
- **Math**
- **Date**
- **Array**
- **Boolean** (*Primitive Wrapper Types)
- **RegExp**
- ...

*Standard Built-in
Objects*

JavaScript Built-in Objects: String

■ String

| Propriedade | Descrição |
|-------------|--|
| length | retorna o número de caracteres que constituem a string |

| Método | Descrição |
|---------------|--|
| toUpperCase() | Conversão para maiúsculas |
| toLowerCase() | Conversão para minúsculas |
| trim() | Remove espaços em branco do início e do fim da string |
| split() | Permite dividir uma string e guardar cada componente numa string |
| replace() | Considera um valor que deve ser substituído por outro |
| substring() | Retorna os caracteres entre dois índices |
| charAt() | Retorna um carácter numa determinada posição |
| indexOf() | Retorna a posição da primeira ocorrência de um dado valor na string, retorna -1 se o valor nunca é detetado. É case sensitive. |

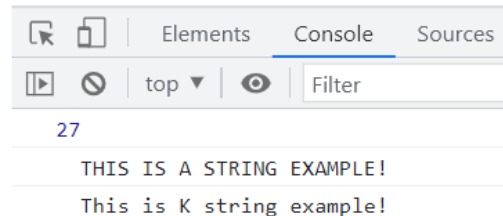
http://www.w3schools.com/jsref/jsref_obj_string.asp

Exercício 6 (propriedades/Métodos String)

JavaScript Built-in Objects

- declare a string "This is a string example!"
- Mostre o número de caracteres
- Converta a string para maiúsculas
- Substitua o primeiro 'a' por 'K'

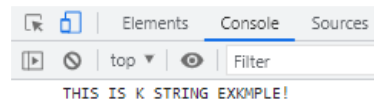
```
var s=" This is a string example!"  
  
console.log(s.length)  
console.log(s.toUpperCase())  
console.log(s.replace('a','K'))
```



Todas as ocorrências:

```
var s=" This is a string example!"  
s=s.toUpperCase().replaceAll('A','K')  
console.log(s)
```

methods chaining



JavaScript Built-in Objects: Number

JavaScript Built-in Objects

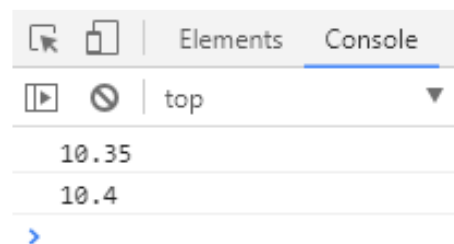
■ Number

| Método | Descrição |
|-----------------|--|
| toExponential() | conversão para notação exponencial |
| toPrecision() | arredonda o número para um dado nº de dígitos |
| toFixed() | arredonda o número para um dado nº de casas decimais |
| toString() | converte para uma <i>string</i> |
| ... | ... |

```
<script>

var x=10.354;

console.log(x.toFixed(2));
console.log(x.toPrecision(3));
</script>
```



JavaScript Built-in Objects: Math

■ Math

- Operações matemáticas, disponibiliza funções matemáticas avançadas (trigonometria, estatística, etc.)
- As propriedades/métodos do Math são chamadas **sem criar de forma explícita um objecto** do tipo Math (exemplo: Math.round(x)) **Exceção!**

| Propriedade | Descrição |
|-------------|---------------------------------------|
| Math.PI | retorna aproximadamente 3.14159265359 |

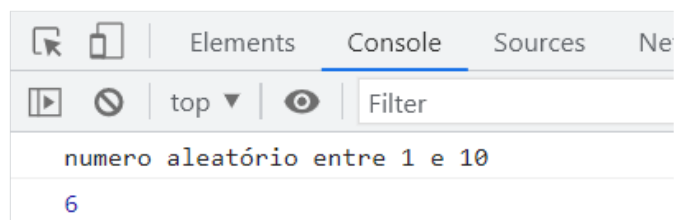
| Método | Descrição |
|---------------|--|
| Math.round() | arredonda o número para o inteiro mais próximo |
| Math.sqrt() | Raiz quadrada de um número positivo |
| Math.ceil() | arredonda o número para o inteiro imediatamente seguinte |
| Math.floor() | arredonda o número para o inteiro imediatamente anterior |
| Math.random() | Gera um número aleatório entre 0 e 1 |

Exercício 7 (propriedades/Métodos Math)

Exercício 7

- Gera um número aleatório entre 1 e 10
 - `Math.random()` (gera um número aleatório no interval [0,1[)
 - `*` (multiplicação)
 - `Math.floor()`
 - `+` (adição)

```
var num = Math.floor(Math.random()*10)+1  
console.log('numero aleatório entre 1 e 10')  
console.log(num)
```



JavaScript Built-in Objects: Date

■ Date

- Disponibiliza métodos e atributos para aceder e manipular horas e datas
- É **necessário instanciar** um objeto deste tipo para aceder aos seus métodos

var data = new Date(); //sem parâmetros o objeto é criado com a data atual

JavaScript Date Methods and Properties

| Name | Description |
|-----------------------------------|---|
| constructor | Returns the function that created the Date object's prototype |
| getDate() | Returns the day of the month (from 1-31) |
| getDay() | Returns the day of the week (from 0-6) |
| getFullYear() | Returns the year |
| getHours() | Returns the hour (from 0-23) |
| getMilliseconds() | Returns the milliseconds (from 0-999) |
| getMinutes() | Returns the minutes (from 0-59) |
| getMonth() | Returns the month (from 0-11) |
| getSeconds() | Returns the seconds (from 0-59) |

http://www.w3schools.com/js/js_obj_date.asp

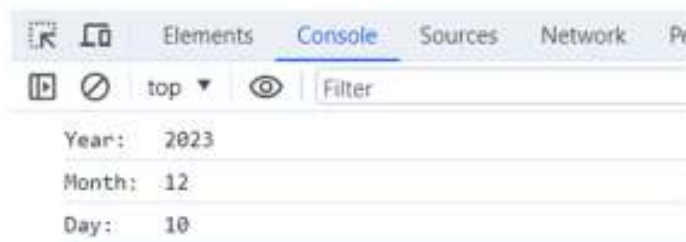
Exercício 8 (propriedades/Métodos Date)

Exercício 8

- Instancie um novo objeto do tipo Date (obtendo a data atual)
- Mostre na consola:

- ano: getFullYear()
- mês: getMonth()
- dia: getDate()

```
const tdDate = new Date()  
console.log('Year: ' + tdDate.getFullYear())  
console.log('Month: ' + (tdDate.getMonth()+1))  
console.log('Day: ' + tdDate.getDate())
```



JavaScript Built-in Objects: Array

JavaScript Built-in Objects

■ Array

- permite armazenar diferentes tipos de elementos

| Propriedade | Descrição |
|-------------|--|
| length | número de elementos de um <i>array</i> |

| Método | Descrição |
|-----------|---|
| indexOf() | pesquisa um elemento e retorna a sua posição |
| pop() | remove o último elemento de um <i>array</i> |
| push() | adiciona um elemento ao <i>array</i> |
| shift() | remove o primeiro elemento de um <i>array</i> |
| sort() | ordena os elementos de um <i>array</i> |
| unshift() | adiciona elementos no início de um <i>array</i> |
| ... | ... |

https://www.w3schools.com/jsref/jsref_obj_array.asp

JavaScript Built-in Objects

■ Arrays

- Os *arrays* são particularmente importantes em *JS* precisamente pela capacidade de armazenar valores relacionados.

- declarados de forma literal ou com base no *constructor Array*
- índices iniciam em zero

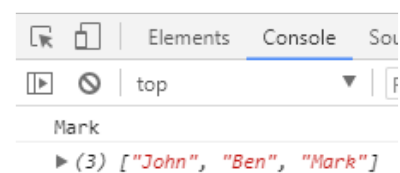
```
<script>

var names = ['John', 'Jane', 'Mark'];
var years = new Array(1990, 1969, 1948);

console.log(names[2]);

names[1] = 'Ben';
console.log(names);

</script>
```



Exercício 9 (propriedades/Métodos Array)

Exercício 9

- Declare um array com os valores
“Audi”, “Mercedes”, “Volvo”, “BMW”

- Adicione ‘Jaguar’ no final: push()
- Apague o primeiro elemento: shift()
- Adicione o primeiro elemento: unshift()
- Apague o ultimo elemento: pop()

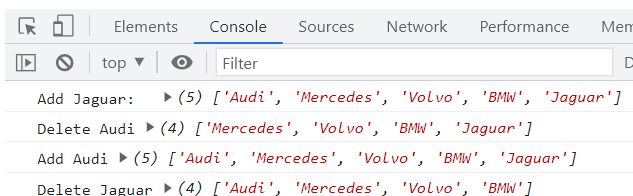
```
const cars=['Audi', 'Mercedes', 'Volvo', 'BMW']
```

```
cars.push('Jaguar')  
console.log('Add Jaguar: ', cars)
```

```
cars.shift()  
console.log('Delete Audi', cars)
```

```
cars.unshift('Audi')  
console.log('Add Audi',cars)
```

```
cars.pop()  
console.log('Delete Jaguar',cars)
```



JavaScript Built-in Objects

■ *map()*

- Executa uma função para cada um dos elementos do array

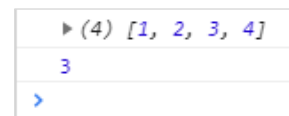
```
const numbers=[1,2,3,4]
const newnumbers= numbers.map((num) => num**2)
console.log(numbers)
console.log(newnumbers)
```



■ *find()*

- Retorna o valor do primeiro elemento que verifica a condição

```
const numbers=[1,2,3,4]
const newnumbers= numbers.find((num) => num>2)
console.log(numbers)
console.log(newnumbers)
```



JavaScript Built-in Objects

■ *findIndex()*

- Retorna o índice do primeiro elemento que verifica a condição. Caso não exista nenhum elemento que verifique a condição retorna -1.

```
const numbers=[1,2,3,4]
const newnumbers= numbers.findIndex((num) => num>2)
console.log(numbers)
console.log(newnumbers)
```



■ *filter()*

- Cria um novo array com todos os elementos que verificam a condição.

```
const numbers=[1,2,3,4]
const newnumbers= numbers.filter((num) => num>2)
console.log(numbers)
console.log(newnumbers)
```



JavaScript Built-in Objects

■ `reduce()`

- Executa uma função definida pelo utilizador em cada elemento do array e cujo resultado é um valor único (ex: a soma de todos os elementos).

```
const numbers=[1,2,3,4]
const newnumbers= numbers.reduce((acc,acv)=>acc*acv)
console.log(numbers)
console.log(newnumbers)
```

*acc – previous value
acv – current value*



■ `concat()`

- faz a concatenação de dois arrays. Os arrays originais não são alterados.

```
const numbers=[1,2,3,4]
const numbers2=[5,6,7]
console.log(numbers.concat(numbers2))
```



JavaScript Built-in Objects

■ `slice()`

- retorna uma cópia parcial desde uma posição inicial até ao final (a posição final não é incluída).

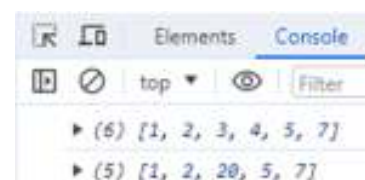
```
const numbers=[1,2,3,4,5,7]
const newnumbers= numbers.slice(1,4)
console.log(numbers)
console.log(newnumbers)
```



■ `splice()`

- altera o conteúdo de um array adicionando ou substituindo elementos

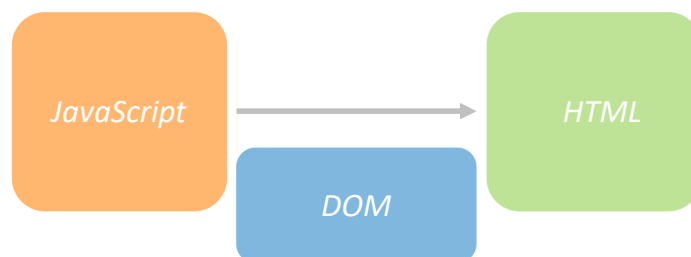
```
const numbers=[1,2,3,4,5,7]
console.log(numbers)
numbers.splice(2,2,20)
console.log(numbers)
```



Document Object Model - DOM

Document Object Model (DOM)

- Um dos principais objetivos da utilização do *JavaScript* é a capacidade de manipular/alterar/controlar elementos HTML
 - DOM disponibiliza:
 - Métodos / Propriedades de forma a aceder a todo o documento HTML permitindo dessa forma a geração/alteração dinâmica de conteúdo HTML



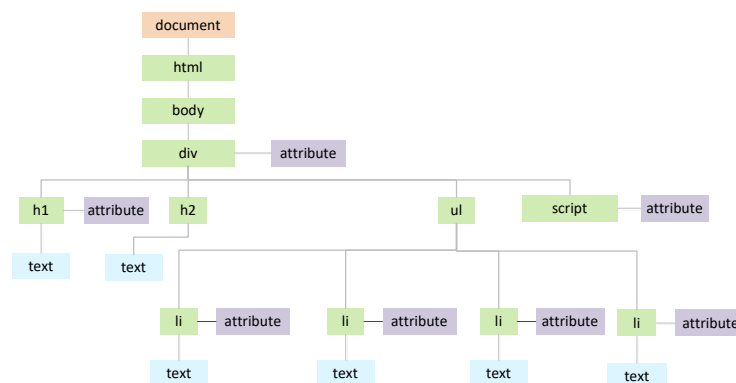
Document Object Model

*The **HTML DOM** is a standard for how to get, change, add, or delete HTML elements.*

http://www.w3schools.com/js/js_htmlDOM.asp

Document Object Model (DOM)

- Quando é feito o download de um documento HTML, este torna-se num **document object**
 - É o *root node* do document HTML e o "owner" de todos os outros nós:
 - element nodes; text nodes; attribute nodes; comment nodes
- O *document object* disponibiliza propriedades e métodos para aceder a todos os nós com base em JavaScript.



DOM tree

Document Object Model (DOM)

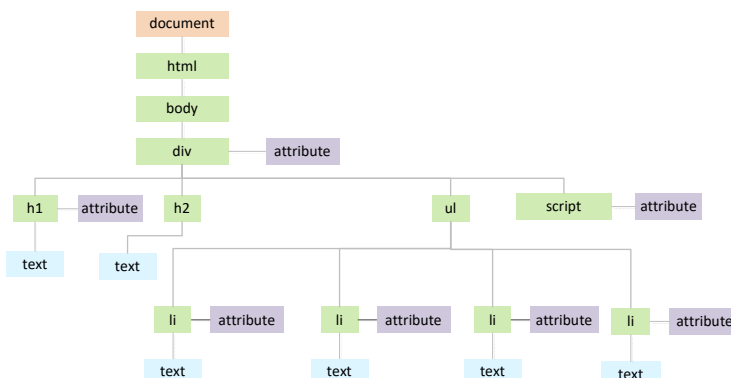
- HTML DOM considera tudo como nó (DOM Tree)
 - O documento (**root element**) é um **document node**
 - Qualquer elemento HTML é um **HTML node**
 - O texto contido nos elementos HTML é um **text node**
 - Qualquer atributo HTML é um **attribute node**

document

Element HTML

Text

Attribute



DOM tree

HTML DOM Reference

The references describe the properties and methods of each object, along with examples.

| | | | | |
|---------------|-------------|----------|----------------|----------|
| Attributes | Console | Document | Element | Events |
| Event Objects | Geolocation | History | HTMLCollection | Location |
| Navigator | Screen | Style | Window | Storage |

Document Object

DOM

HTML DOM Reference

The references describe the properties and methods of each object, along with examples.

| | | | | |
|---------------|-------------|-----------------|----------------|----------|
| Attributes | Console | <u>Document</u> | Element | Events |
| Event Objects | Geolocation | History | HTMLCollection | Location |
| Navigator | Screen | Style | Window | Storage |

DOM: Document Object

DOM Methods

`document.getElementById()`

Retorna o *element object* que tem o *id* com o valor especificado

`document.querySelector()`

Retorna o primeiro *elemento object* referenciado pelo seletor CSS

`document.querySelectorAll()`

Retorna o conjunto de elementos referenciados pelo seletor CSS

`document.getElementsByTagName()`

Retorna o conjunto de elementos (HTML Collection) cuja designação da *tag* corresponde ao nome especificado

`document.getElementsByClassName()`

Retorna um conjunto de elementos (HTML Collection) que tem o atributo *class* com o nome especificado

DOM: Document Object

DOM Methods

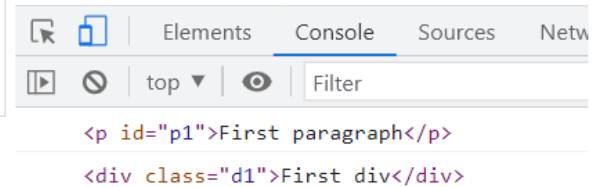
- Sempre que necessário utilizar o mesmo elemento mais do que uma vez, o *element object* deve ser referenciado por uma variável:

```
var identificaElemento = document.getElementById('one');
```

```
<p id="p1">First paragraph</p>  
<div class="d1">First div</div>
```

```
<script>  
  var a=document.getElementById('p1');  
  var b=document.querySelector('.d1')  
  
  console.log(a)  
  console.log(b)  
</script>
```

Métodos DOM que referenciam elementos HTML



Element Object

DOM

DOM: Document Object

■ Element Object

- Representa um elemento HTML (ex: <div>, <form>, <a>, ...)
- Tem métodos e propriedades associadas
 - Algumas Propriedades **muito importantes**:

<code>innerHTML</code>	Acesso/alteração do conteúdo HTML de um elemento
<code>textContent</code>	Acesso/alteração de texto
<code>className</code>	Retorna/altera o valor do atributo class de um elemento *: substitui todos os valores anteriormente definidos para o atributo class
<code>classList</code>	Retorna todos os valores do atributo class de um elemento
<code>id</code>	Acesso/alteração do atributo id de um elemento

DOM: Document Object

■ Element Object

- `textContent`:

```
<p id="p1">First paragraph</p>
```

```
<script>  
var a=document.getElementById('p1');  
a.textContent='CHANGED !'
```

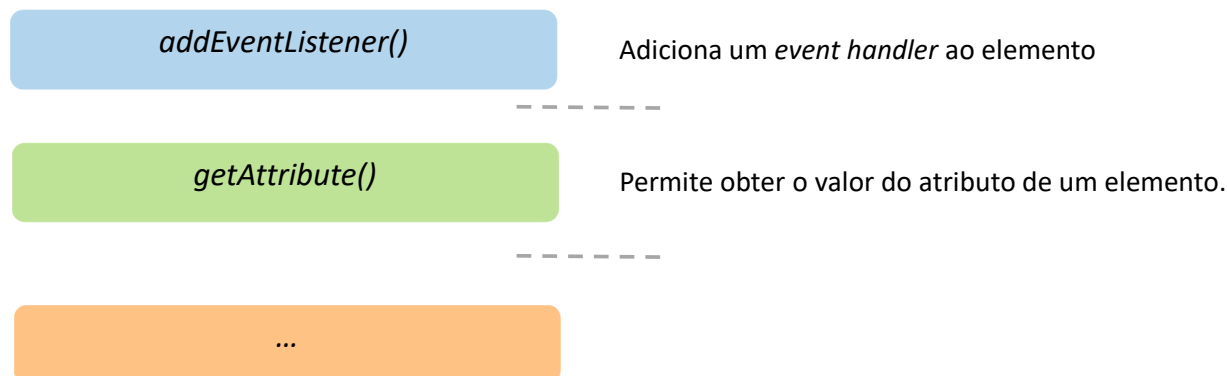
Propriedade **textContent** permite alterar de forma dinâmica o conteúdo do element object referenciado pela variável **a**

CHANGED !

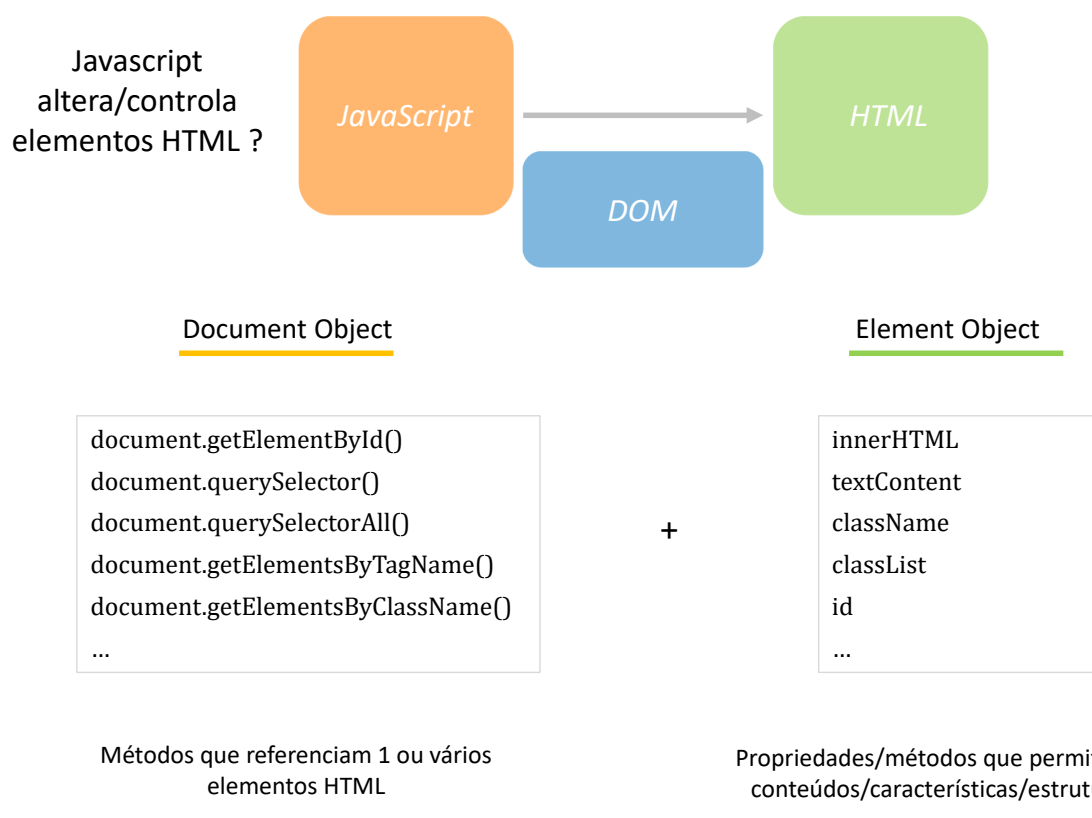
DOM: Document Object

■ Element Object

- Métodos importantes:



DOM



HTML Collection Object

DOM

HTML DOM Reference

The references describe the properties and methods of each object, along with examples.

Attributes	Console	Document	Element	Events
Event Objects	Geolocation	History	<u>HTMLCollection</u>	Location
Navigator	Screen	Style	Window	Storage

HTMLCollection Object

- Representa uma lista de elementos HTML
- Permite indexação como se de um array se tratasse (**não é um array!**)
 - **Não permite** a utilização de métodos do objeto array.
 - Possui a propriedade length (muito útil!)

■ Método *item()*

```
var elements=document.getElementsByClassName('vegetables');  
  
if(elements.length>=1){  
    var firstItem=elements.item(0);}
```

■ Array syntax

```
var elements=document.getElementsByClassName('vegetables');  
  
if(elements.length>=1){  
    var firstItem=elements[0];}
```

DOM: HTML Collection

```
<p id="p1" class="par">First paragraph</p>  
<p id="p2" class="par">Second paragraph</p>  
  
<script>  
  
var a=document.getElementsByClassName('par')  
a[1].textContent='CHANGED !'  
console.log(a)
```

a referencia um conjunto de elementos HTML (**HTML collection object**)
através da sua class

Requer sempre indexação, índices iniciam em 0

First paragraph
CHANGED !

HTMLCollection(2)
▶ 0: p#p1.par
▶ 1: p#p2.par

Document Object

Seleção de Elementos (DOM nodes)

Seleção de Elementos

■ `document.getElementById()`

- a forma mais rápida e mais eficaz de aceder a um único elemento
- retorna o nó cujo elemento tem o id especificado na *DOM Query*

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
</ul>
```

```
const veg = document.getElementById('two')
veg.className='promo'
```

```
li{list-style-type: none;}
.promo{
  background-color: blueviolet;
  font-size: 1.5em;
  color:white}
```

1. A variável `veg` referencia o elemento HTML com `id='two'`
2. É atribuído o valor `'promo'` ao atributo `class` da variável `veg`
3. É aplicada a formatação definida no código CSS
(substitui o valor anterior, neste caso `'vegetables'`)



Seleção de Elementos

- `document.querySelector()`
 - Baseado em seletores CSS
 - Retorna o **primeiro elemento** que verifica o seletor CSS definido como argumento

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
</ul>
```

```
li{list-style-type: none;}
.promo{
  background-color: blueviolet;
  font-size: 1.5em;
  color:white}
```

```
const veg = document.querySelector('#three')
veg.className='promo'
```

1. A variável `veg` referencia o elemento HTML com `id='three'`
(`querySelector()` utiliza seletores CSS)
2. É atribuído o valor `'promo'` ao atributo `class` da variável `element`
3. É aplicada a formatação definida no código CSS
(substitui o valor anterior, neste caso `'vegetables'`)



DOM: Document Object

- Métodos que retornam uma *HTML Collection*

`document.querySelectorAll()`

Retorna o conjunto de elementos referenciados pelo seletor CSS

`document.getElementsByTagName()`

Retorna o conjunto de elementos (HTML Collection) cuja designação da *tag* corresponde ao nome especificado

`document.getElementsByClassName()`

Retorna um conjunto de elementos (HTML Collection) que tem o atributo `class` com o nome especificado

Seleção de Elementos

▪ `document.querySelectorAll()`

- permite aceder aos *DOM nodes* com base num seletor CSS
- Requer indexação mesmo que exista apenas um element ao qual se aplique o seletor CSS

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
  <li id="four" class="fish">Tuna</li>
  <li id="five" class="fish">Cod</li>
</ul>
```

```
li{list-style-type: none;}
.promo{
  background-color: blueviolet;
  font-size: 1.5em;
  color:white}
```

```
var elements=document.querySelectorAll(".fish")
```

```
for (var i=0; i<elements.length; i++)
  elements[i].className='promo'
```

1. A variável `elements` referencia todos os elementos HTML com class “fish”
2. Percorre `elements` e atribui a cada um dos elementos a class ‘promo’
3. É aplicada a formatação definida no código CSS

Grocery List

Onions
Garlic
Cabage
Tuna
Cod

Seleção de Elementos

▪ `document.getElementsByTagName()`

- retorna uma *HTML Collection* com todos os elementos que verificam o nome da *tag*

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
  <li id="four" class="fish">Tuna</li>
  <li id="five" class="fish">Cod</li>
</ul>
```

```
li{list-style-type: none;}
.promo{
  background-color: blueviolet;
  font-size: 1.5em;
  color:white}
```

```
var elements=document.getElementsByTagName("li")
```

```
if (elements.length>3)
  elements[2].className='promo'
```

1. A variável `elements` referencia **todos** os elementos HTML com a tag ‘li’
2. Como a condição é verificada
3. É aplicada a formatação definida no código CSS ao elemento de índice [2]

Grocery List

Onions
Garlic
Cabage
Tuna
Cod

Seleção de Elementos

- `document.getElementsByClassName()`
 - À semelhança do `getElementsByTagName()` retorna uma node list com os elementos com o valor do atributo `class` definido como argumento
 - Requer indexação

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
  <li id="four" class="fish">Tuna</li>
  <li id="five" class="fish">Cod</li>
</ul>
```

```
li{list-style-type: none;}
.promo{
  background-color: blueviolet;
  font-size: 1.5em;
  color:white}
```

```
var elements=document.getElementsByClassName("vegetables")
elements[0].className='promo'
```

Grocery List

Onions
Garlic
Cabage
Tuna
Cod

Seleção de Elementos

- Para todos os métodos que retornam um **HTML Collection Object**
 - Os elementos tem que ser indexados, mesmo que apenas exista **uma ocorrência** da tag name (posição [0]) **Importante!**
- Utilização de métodos de seleção de elementos diretamente numa condição
 - Caso o elemento exista é retornado o valor *true*
 - A presença de um *element object* é um **truthy value**

```
if (document.getElementById('idValue'))
{
    /* processamento se o elemento existe */
}
else
{
    /* processamento se o elemento não foi encontrado*/
}
```

Seleção de Elementos

■ Form Elements Collection

- Forma alternativa de aceder aos elementos de um **formulário**
- A *elements collection* retorna o conjunto de todos os elementos definidos num *form*
 - Os elementos estão ordenados de acordo com a ordem pela qual foram definidos no *form*
 - Indexados pela posição (início em 0) ou pelo valor do atributo id ou name.

```
function mostraElementos(){
    var el=document.getElementById('myForm')
    var txt=""

    for (var i=0; i<el.length;i++)
        txt += el[i].value + '<br>'

    document.getElementById('elementos').innerHTML=txt
}
```

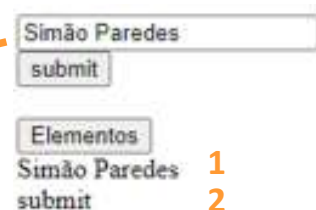

Seleção de Elementos

```
<form id="myForm">
    <input type="text" name="nome" placeholder="Nome completo" id="d1"><br> 1
    <input type="submit" value="submit"><br> 2
</form><br>
```

```
<button onclick="mostraElementos()">Elementos</button>
<div id="elementos"></div>
```

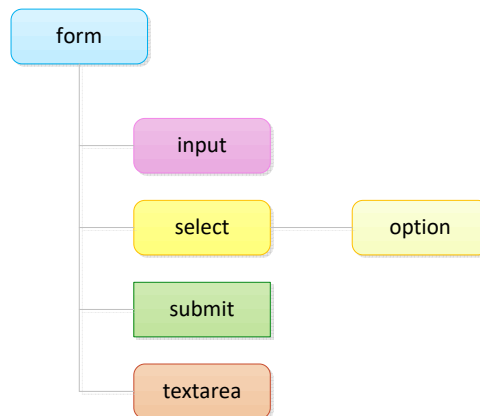
```
<script>
function mostraElementos(){
    var el=document.getElementById('myForm')
    var txt=""
    for (var i=0; i<el.length;i++)
        txt += el [i].value + '<br>'

    document.getElementById('elementos').innerHTML=txt
}
```



Document Object Model (DOM)

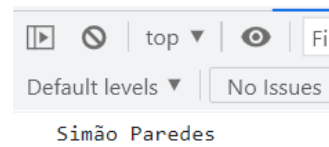
- Sintaxe alternativa para acesso aos elementos de um formulário



- Aceder a um campo do formulário:

`document.getElementById('idForm').idElement`

```
function mostraElementos(){  
  var el=document.getElementById('myForm')  
  console.log(el.d1.value)  
  ...  
}
```

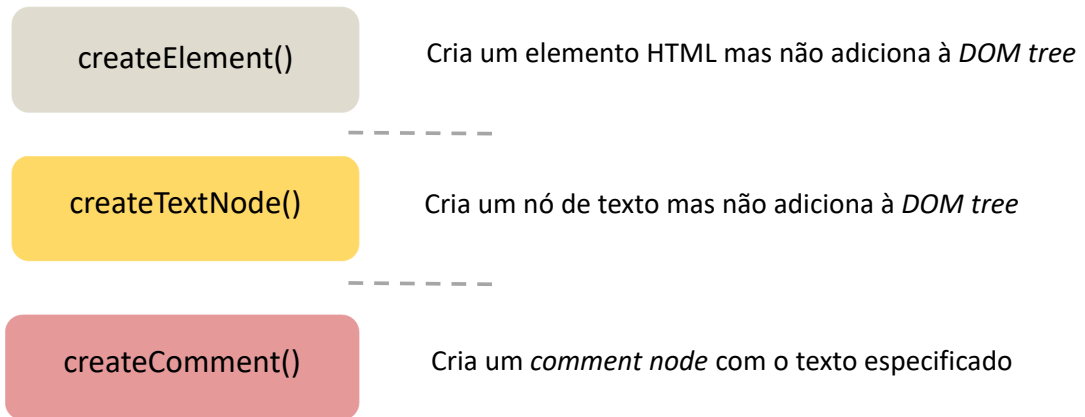


Document Object

Criação de Elementos (DOM nodes)

DOM: Document Object

■ Métodos Específicos para Alteração da Estrutura



DOM: Document Object

■ Criação de Elementos

- *createElement()*
- *createTextNode()*

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
</ul>
```

Criação de elementos

Element node

```
var newElement = document.createElement('li')
var newText = document.createTextNode('Cod')
```

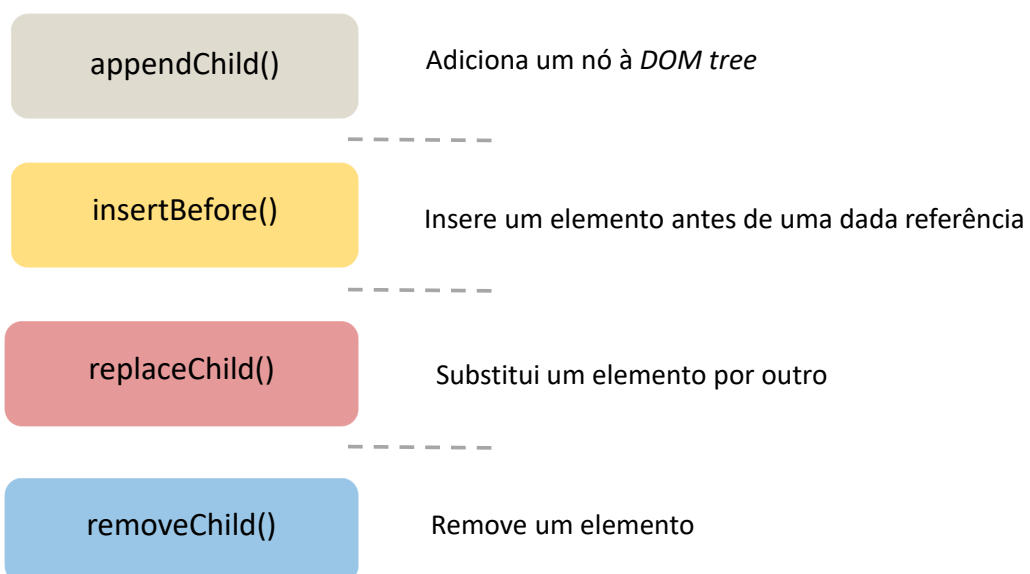
Text Node

Element Object

Alteração de Estrutura

DOM: Document Object

■ Métodos Específicos para Alteração da Estrutura



appendChild()

Adicionar à DOM Tree!

- Permite a integração/inclusão de um novo nó (elemento HTML / texto) no documento HTML que se pretende alterar
- Aceita como **único argumento** o nó que se **pretende adicionar**
- **O método deve ser chamado** no elemento que se pretende que seja o **pai** (estrutura HTML) do nó a inserir.

Alteração de Estrutura

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
</ul>
```

appendChild()

```
li{list-style-type: none;}
.promo{
  background-color: blueviolet;
  font-size: 1.5em;
  color:white}
```

```
var newElement=document.createElement('li')
var newText=document.createTextNode('Cod')
```

```
var positionReference=document.getElementsByTagName('ul')[0]
```

Referência para posicionamento
(elemento pai)

```
newElement.appendChild(newText)
positionReference.appendChild(newElement)
```

Adição dos elementos à DOM tree
(elemento pai)

```
newElement.className='promo'
```

Grocery List

```
Onions
Garlic
Cabage
Cod
```

Alteração de Estrutura

■ `insertBefore()`

- Permite a inserção de um elemento antes de um dado elemento
 - Dois argumentos: nó a ser inserido; nó que determina o posicionamento

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
</ul>
```

```
li{list-style-type: none;}
.promo{
  background-color: blueviolet;
  font-size: 1.5em;
  color:white}
```

```
var newElement=document.createElement('li')
var newText=document.createTextNode('Cod')
var positionReference=document.getElementsByTagName('ul')[0]
  newElement.appendChild(newText)
  positionReference.appendChild(newElement)
  newElement.className='promo'

  positionReference.insertBefore(newElement,document.getElementById('two'))
```

Grocery List

Onions
Cod
Garlic
Cabage

Nó a ser inserido

Referência para posicionamento

Alteração de Estrutura

■ `replaceChild()`

- Permite a substituição de um elemento por outro
- Aceita dois argumentos: o primeiro é o nó a ser inserido e o segundo o nó (id) a ser substituído

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
</ul>
```

```
li{list-style-type: none;}
.promo{
  background-color: blueviolet;
  font-size: 1.5em;
  color:white}
```

```
var newElement=document.createElement('li')
var newText=document.createTextNode('Cod')
var positionReference=document.getElementsByTagName('ul')[0]

newElement.appendChild(newText)
positionReference.appendChild(newElement)
newElement.className='promo'

positionReference.replaceChild(newElement,document.getElementById('two'))
```

Grocery List

Onions
Cod
Cabage

Nó a ser inserido

Nó a ser substituído

■ `removeChild()`

- apaga um nó que aceita como argumento
- à semelhança do **`appendChild()`** o método **`removeChild()`** tem que ser **invocado no nó pai**

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
</ul>
```

Elemento a ser removido

```
var positionReference=document.getElementsByTagName('ul')[0]
var deletedElement=document.getElementById('two')

positionReference.removeChild(deletedElement)
```

Grocery List

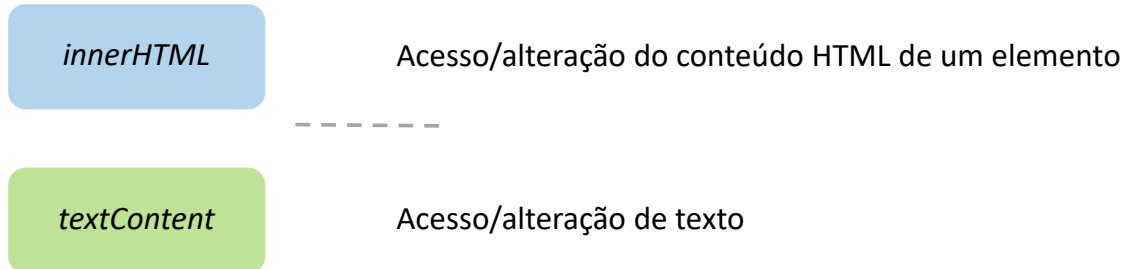
Onions
Cabage

Element Object

Alteração de Conteúdo (propriedades)

Alteração de Conteúdos

- Propriedades muito importantes para efetuar alteração de conteúdo



Alteração de Conteúdo

- `innerHTML`
 - a propriedade mais versátil uma vez que permite alterar/definir o conteúdo (texto/**markup**) de um dado elemento
 - alterar o *markup* de um elemento (ex: adicionar um link)

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
</ul>
```

```
li{list-style-type: none;}
.promo{
  background-color: blueviolet;
  font-size: 1.5em;
  color:white}
a{color:white}
```

```
var element = document.getElementById('three')
element.innerHTML='<a href="https://peixinhodalota.pt/"> Sardines</a>'

element.className='promo'
```

Grocery List

Onions
Garlic
Sardines

Redefinição do conteúdo
do element

(Permite markup HTML)

Alteração de Conteúdo

■ *textContent*

- obtém/altera o conteúdo textual de um elemento e ignora todo o *markup* HTML contido por esse elemento

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
</ul>
```

```
li{list-style-type: none;}
.promo{
  background-color: blueviolet;
  font-size: 1.5em;
  color:white}
a{color:white}
```

```
var element = document.getElementById('three')
element.textContent='<a href="https://peixinhodalota.pt/"> Sardines</a>'

element.className='promo'
```

Grocery List

Onions
Garlic

[Sardines](https://peixinhodalota.pt/)

Redefinição do conteúdo textual do element

(Não permite markup HTML)

Alteração de Conteúdo

document.write()

Vantagens:

Rápido e simples de mostrar o resultado da adição de conteúdo

Desvantagens:

Funciona corretamente quando é feito o download da página

Se utilizado depois efetua o **overwrite** do conteúdo anterior



element.innerHTML

Vantagens:

Permite a inserção de markup com menos código do que os métodos DOM

É mais rápido do que os métodos DOM

É a forma mais simples de remover todo o conteúdo de um elemento (string vazia).

Desvantagens:

Potenciais problemas com *event handlers*, uma vez que origina a eliminação de todos os *child elements* os quais podem estar a ser utilizados para disparar um evento

Métodos DOM

Vantagens:

Permite aceder de forma precisa a todos os elementos da *DOM Tree*.

Não afeta os *event handlers*

Permite a adição incremental de elementos.

Desvantagens:

Caso se pretenda efetuar muitas alterações ao conteúdo é mais lento que *innerHTML*

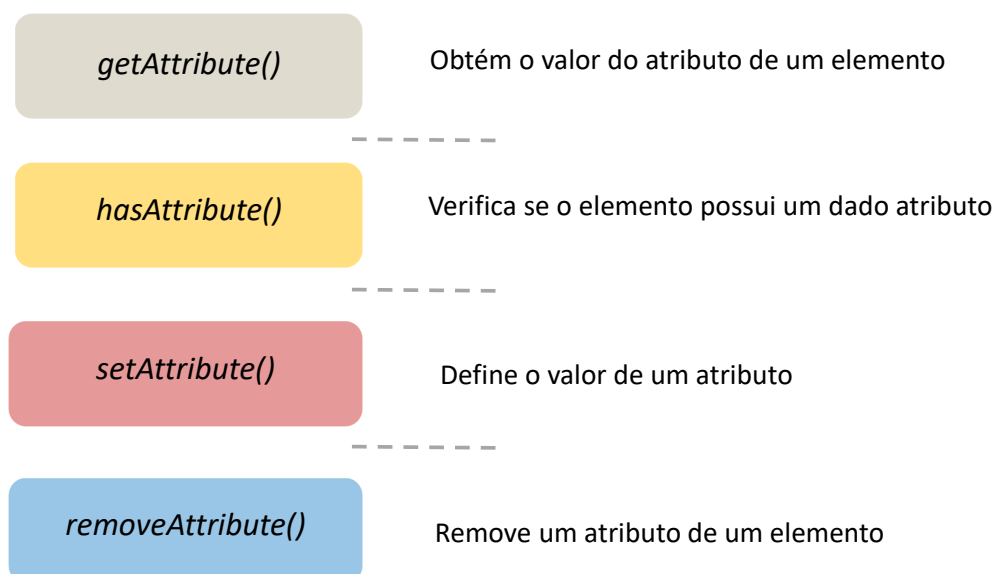
Para atingir um determinado objetivo, necessita de mais código quando comparado com o *innerHTML*

Element Object

Alteração de Atributos

Alteração de Atributos

- Métodos do *Element Object* para alteração de atributos



Alteração de Atributos

■ `getAttribute()`

- Obter o valor atual de qualquer atributo de um elemento

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
</ul>
<p></p>
```

```
var msg=""
var element = document.getElementById('three')

msg = 'Current Class Value: ' + element.getAttribute('class') + '<br>'

document.getElementsByTagName('p')[0].innerHTML=msg
```

Grocery List

Onions
Garlic
Cabage

Current Class Value: vegetables

← Acede ao valor
do atributo class

Alteração de Atributos

■ `setAttribute()`

- Definir o valor de qualquer atributo de um elemento
 - Alterar imagem (src); destino (href), ...

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
</ul>
<p></p>
```

```
li{list-style-type: none;}
.promo{
  background-color: blueviolet;
  font-size: 1.5em;
  color:white}
a{color:white}
```

```
var msg=""
var element = document.getElementById('three')

element.setAttribute('class', 'promo') + '<br>'
msg = 'Current Class Value: ' + element.getAttribute('class') + '<br>'

document.getElementsByTagName('p')[0].innerHTML=msg
```

Define o valor do
atributo class

Grocery List

Onions
Garlic
Cabage

Current Class Value: promo

Alteração de Conteúdos

- Propriedades muito importantes para efetuar alteração de conteúdo

style	Permite definir/obter o valor de propriedades CSS
classList	Permite obter o valor do atributo class de um element Aplicada em conjunto com os métodos add() e remove(), toggle() para adicionar ou eliminar uma class)
className	Permite obter/estabelecer o valor do atributo class
id	Permite obter/estabelecer o valor do atributo id

Alteração de Atributos

- Propriedade **style**
 - mais eficaz e expedita que `setAttribute()`
 - retorna um `CSSStyleDeclaration` object o qual representa o atributo `style` de um elemento e que possui um conjunto alargado de propriedades CSS

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
</ul>
```

Grocery List

Onions
Garlic
Cabage

```
var element = document.getElementById('three')

element.style.backgroundColor='blueviolet'
element.style.color='white'
element.style.fontSize='1.5em'
```

Propriedades do
`CSSStyleDeclaration` object:

backgroundColor
color
fontSize

Alteração de Atributos

■ Propriedade **style**

- O nome das propriedades difere ligeiramente/pontualmente das propriedades CSS originais
 - Propriedades CSS com 2 palavras separadas por hífen são definidas em camelCase

Style Object Properties

Property	Description
alignContent	Sets or returns the alignment between the lines inside a flexible container when the items do not use all available space
alignItems	Sets or returns the alignment for items inside a flexible container
alignSelf	Sets or returns the alignment for selected items inside a flexible container
animation	A shorthand property for all the animation properties below, except the animationPlayState property
animationDelay	Sets or returns when the animation will start
animationDirection	Sets or returns whether or not the animation should play in reverse on alternate cycles
animationDuration	Sets or returns how many seconds or milliseconds an animation takes to complete one cycle

https://www.w3schools.com/jsref/dom_obj_style.asp

Alteração de Atributos

■ Propriedade **classList**

- Propriedade muito importante para controlar a formatação de um elemento através de classes CSS previamente definidas.
- Esta propriedade retorna um *DOMTokenList object* (objecto que contém uma lista de strings)

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="value1 value2 value3 value4">Cabage</li>
</ul>
```

```
var element = document.getElementById('three')
console.log(element.classList)
```

```
DOMTokenList(4)
value4'] 1
0: "value1"
1: "value2"
2: "value3"
3: "value4"
```

Alteração de Atributos

- Propriedade ***classList***

```
DOMTokenList(4)  
value4']  
0: "value1"  
1: "value2"  
2: "value3"  
3: "value4"
```

- Ao DOMTokenList object podem ser aplicados métodos muito úteis para controlar quais as classes aplicadas a um element
 - Controlar de forma dinâmica a formatação de elementos
 - add()
 - remove()
 - contains()

Alteração de Atributos

- Exemplo de controlo dinâmico da aplicação de seletores de class CSS
 - *contains(); add(); remove()*

```
<h1> Grocery List</h1>  
<ul>  
  <li id="one" class="vegetables">Onions</li>  
  <li id="two" class="vegetables">Garlic</li>  
  <li id="three" class="value1 value2 value3 value4">Cabage</li>  
</ul>
```

```
var element = document.getElementById('three')  
  
if (element.classList.contains('promo')) ←  
  element.classList.remove('promo') ←  
else  
  element.classList.add('promo') ←
```

Grocery List

Onions
Garlic

Cabage

Alteração de Atributos

■ Propriedade `className`

- Permite definir o valor da class
 - Substituí sempre o valor anterior

```
li{list-style-type: none;}  
.promo{  
  background-color: blueviolet;  
  font-size: 1.5em;  
  color:white }  
.promo2{ background-color: orangered;  
  font-size: 1.5em;  
  color:white}
```

```
<h1> Grocery List</h1>  
<ul>  
  <li id="one" class="vegetables">Onions</li>  
  <li id="two" class="vegetables">Garlic</li>  
  <li id="three" class="promo">Cabage</li>  
</ul>
```

```
var element = document.getElementById('three')  
  
element.className='promo2'
```

Grocery List

Onions
Garlic
Cabage

Grocery List

Onions
Garlic
Cabage

Alteração de Atributos

■ `className`

- apesar de menos frequente é possível utilizar a propriedade `className` para adicionar valores ao atributo class sem substituir o anterior
 - A solução passa por utilizar uma atribuição composta com o novo valor (string) iniciar por espaço

```
var element = document.getElementById('three')  
  
element.className += ' promo2'  
console.log(element.classList)
```

Grocery List

Onions
Garlic
Cabage

```
▼ DOMTokenList(2)  
0: "promo"  
1: "promo2"  
length: 2
```

Element Object

Seleção de Elementos (traversing properties)

Seleção de Elementos

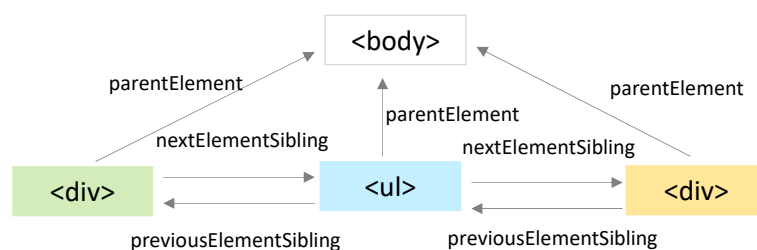
Traversing Properties

- Têm por base a relação hierárquica (DOM Tree) dos elementos

- `previousElementSibling`
- `nextElementSibling`
- `firstElementChild`
- `lastElementChild`
- `parentElement`
- ...

```
<!DOCTYPE html>
<html>

<head>
  <title>My page</title>
</head>
<body>
  <div>Header</div>
  <ul>
    <li>List</li>
  </ul>
  <div>Footer</div>
</body>
</html>
```



Traversing Properties

```
<h1> Grocery List</h1>
<ul>
  <li id="one" class="vegetables">Onions</li>
  <li id="two" class="vegetables">Garlic</li>
  <li id="three" class="vegetables">Cabage</li>
</ul>
<p></p>
```

```
li{list-style-type: none;}
.promo{
  background-color: blueviolet;
  font-size: 1.5em;
  color:white
}
.promo2{ background-color: orangered;
  font-size: 1.5em;
  color:white}
```

↓

```
var el = document.getElementById('two')
```

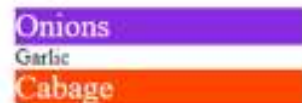
```
var preElement=el.previousElementSibling
```

```
var nextElement=el.nextElementSibling
```

```
preElement.className='promo'
```

```
nextElement.className='promo2'
```

Grocery List



Window Object

HTML DOM Reference

The references describe the properties and methods of each object, along with examples.

Attributes	Console	Document	Element	Events
Event Objects	Geolocation	History	HTMLCollection	Location
Navigator	Screen	Style	<u>Window</u>	Storage

Window Object

- Métodos do Window Object destinados a controlar a página ativa no browser

`window.open()`

Abre uma nova janela

`window.close()`

Fecha a janela atual

`window.moveTo()`

Movimenta a janela atual

`window.resizeTo()`

Redimensiona a janela atual

`window.print()`

Imprime a página atual

Window Object

- Métodos adicionais do window object (sem referência explícita ao objeto)

`alert()`

Gera uma caixa de alerta

`confirm()`

Gera uma caixa de confirmação

`prompt()`

Permite a inserção de valores por parte do utilizador

- Funções de Temporização

`setInterval()`

Chama repetidamente uma função com base num intervalo constante (ms)

`setTimeout()`

Chama uma função após um determinado tempo (ms)

Window Object

- Propriedades

`window.innerHeight`

Altura da janela do browser em px

`window.innerWidth`

Largura da janela do browser em px

window Object

- O window object contém outros objetos:

<u>window</u>	current browser window (top level object)
<u>history</u>	pages in browser history
<u>location</u>	URL of current page
<u>navigator</u>	information about the browser
<u>screen</u>	device's display information

Properties of window.location

Let's take a look at the properties of the `window.location` object and what information they provide:

1. `window.location.href`: The full URL of the current document, including the protocol, domain, port, path, and query string. For example, if the current URL is `https://codedamn.com/blog/javascript-window-location`, the value of `window.location.href` would be the same.
2. `window.location.protocol`: The protocol used by the current URL, such as `http:` or `https:`.
3. `window.location.host`: The full domain and port of the current URL, such as

<https://codedamn.com/news/javascript/window-location-explained>

window Object

- Funções de Temporização
 - Particularmente interessantes :
 - Permitem controlar o tempo de chamada de uma outra função
 - Muito úteis por exemplo na implementação de *slideshows*, ...
 - `setTimeout(nomeFunção, tempo(ms));`
 - chama a função após um período de tempo definido em ms

```
var randomNumber;  
function alteraValor() {  
    randomNumber = Math.random();  
    document.getElementById('global').innerHTML = randomNumber;  
}  
  
setTimeout(alteraValor, 2000);  
</script>
```



2 seg

0.7736800073180348

- `setInterval (nomeFunção, tempo(ms));`
 - chama uma função de forma repetida a intervalos regulares definidos em ms

```
<span id="global"></span> <br />

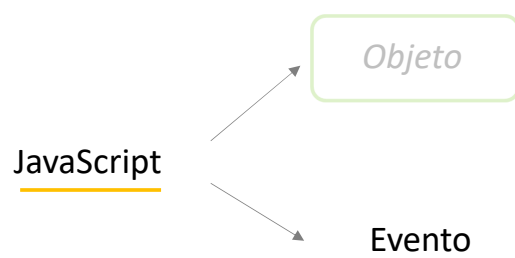
<script>
  var randomNumber;
  function alteraValor() {
    randomNumber = Math.random();
    document.getElementById('global').innerHTML = randomNumber;
  }
  setInterval(alteraValor, 2000);
</script>
```

0.06333285407163203  2 seg 0.9507259053643793  2 seg 0.3472738463897258

- `clearInterval()`
 - Elimina um *timer* que tenha sido estabelecido com `setInterval()`.
 - O *timer* deverá estar guardado numa variável que será o parâmetro do `clearInterval()`

```
var myVar = setInterval(function(){ myTimer() }, 1000);
...
function myStopFunction() {
  clearInterval(myVar);
}
```

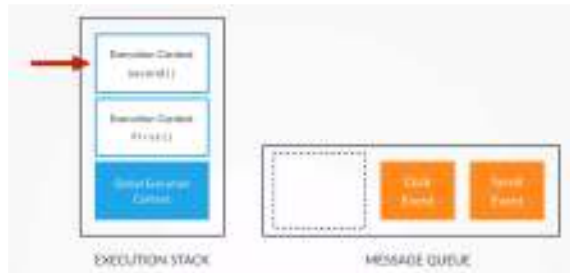
Eventos



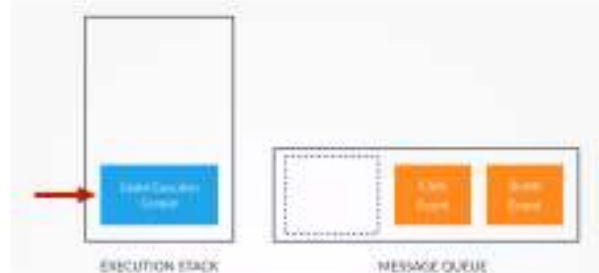
Evento	É disparado...
click	quando é pressionado e liberado o botão primário do mouse, trackpad, etc.
mousemove	sempre que o cursor do mouse se move.
mouseover	quando o cursor do mouse é movido para sobre algum elemento.
mouseout	quando o cursor do mouse se move para fora dos limites de um elemento.
dblclick	quando acontece um clique duplo com o mouse, trackpad, etc.

- É necessário um identificador (*event handler*) que permita iniciar (*trigger*) a execução do *script* quando esse evento ocorre
 - Existem diferentes tipos de eventos:
 - User Interface Events
 - HTML elements Events
 - Mouse events
 - Keyboard events
 - Mutation events
 - ...

Processamento de Eventos



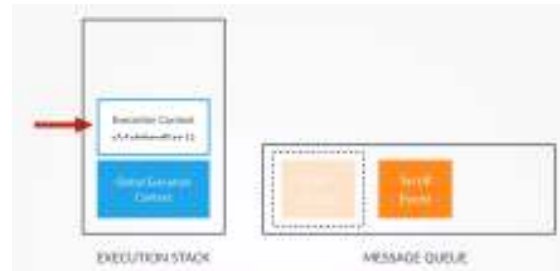
1. Enquanto funções estiverem a ser executadas, os eventos não podem ser considerados;



2. Uma vez libertados os contextos de execução (terminada a execução das funções), os eventos podem ser considerados



3. Quando um determinado evento ocorre é processado na *message queue*



4. É então criado o seu próprio contexto de execução (processamento/função associado à deteção do evento)

<https://www.udemy.com/the-complete-javascript-course/learn/v4/t/lecture/5869158?start=0>

Eventos

■ Event handling

- Seleção do elemento HTML onde é definido o evento
 - Os eventos relacionados com o *Window Object* não se referem à DOM tree.
- Definir o evento que faz o trigger da ação pretendida
 - Alguns eventos podem ser aplicados à maioria dos elementos
 - Exemplo: *mouseover*, ...
 - Outros eventos apenas podem ser definidos em elementos específicos
 - Exemplo: *submit*, ...
- Definir o processamento
 - Código necessário para executar uma determinada ação

Eventos

■ Três formas distintas de declarar um evento:

■ *HTML Event Handlers*

- Usa diretamente atributos dos elementos HTML para responder a eventos no elemento onde foram declarados.
- Considerada **má prática** uma vez que associa de forma direta o HTML e o JavaScript
 - Deve existir uma separação clara na aplicação das duas tecnologias

```
<a onclick="hide()"> ... </a>
```

■ *DOM Event Handlers*

- Considerados melhores que a opção anterior uma vez que permitem uma separação clara entre Javascript e HTML
- A limitação principal é que permite associar apenas uma função por evento e não permite a passagem de valores à função

```
element.ontevent=functionName;
```

■ *DOM Event Listeners*

- sintaxe diferente
- Permitem associar múltiplas funções ao mesmo evento e passagem de argumentos

```
element.addEventListener('event',functionName,[Boolean]);
```

Eventos

■ *HTML Event Handler*

- **Não permite** uma separação clara entre o HTML e o Javascript

```
<form action="#">  
  <input type="text" placeholder="Insert Name" onblur="checkUser()" id="user"><br><br>  
  <input type="password" placeholder="Insert Password">  
  <input type="submit">  
</form>
```

Quando o utilizador
seleciona outro campo é
disparado o evento

```
function checkUser(){  
  var elUser=document.getElementById('user')  
  if (elUser.value.length<5)  
    alert('valor incorreto - minimo 5 carateres')  
}
```



Eventos

■ DOM Event handler

- Permite uma separação clara entre o HTML e o Javascript
 - O evento é definido no <script>

```
<form action="#">
  <input type="text" placeholder="Insert Name" id="user"><br><br>
  <input type="password" placeholder="Insert Password">
  <input type="submit">
</form>
```

```
var element=document.getElementById('user')

function checkUser(){
  var elUser=document.getElementById('user')
  if (elUser.value.length<5)
    alert('valor incorreto - minimo 5 carateres')
}
```

element.onblur=checkUser



Evento definido no JS
(não permite a passagem de parâmetros, apenas o nome da função)

Eventos

■ Event Listener

- Podem invocar mais do que uma função mas não são suportados pelos browsers mais antigos
 - O evento não é definido no HTML

`element.addEventListener('event',functionName,[Boolean])`

```
<form action="#">
  <input type="text" placeholder="Insert Name" id="user"><br><br>
  <input type="password" placeholder="Insert Password">
  <input type="submit">
</form>
```

```
var element=document.getElementById('user')

function checkUser(){
  var elUser=document.getElementById('user')
  if (elUser.value.length<5)
    alert('valor incorreto - minimo 5 carateres')
}

element.addEventListener('blur', checkUser, false)
```

* Ao contrário das duas situações anteriores o evento não é precedido de 'on'

** apenas o nome da função, os parêntesis são omitidos.

*** Quando existem eventos encadeados define a ordem dos eventos. Valor por defeito *false* (*bubling*)

■ Passagem de Valores a uma Função

■ *Event handlers & Listeners*

- Solução passa por definir uma **anonymous function** tal que permita receber argumentos
- A *anonymous function* pode conter várias chamadas a funções diferentes.

```
var element=document.getElementById('user')

function checkUser(minlength){
    var elUser=document.getElementById('user')
    if (elUser.value.length<minlength)
        alert('valor incorreto - minimo 5 carateres')
}

element.addEventListener('blur',function(){checkUser(10)},false)
```



Com base numa **anonymous function** é
passado o valor 10 à função *checkUsername*

Event Object

HTML DOM Reference

The references describe the properties and methods of each object, along with examples.

Attributes	Console	Document	Element	<u>Events</u>
Event Objects	Geolocation	History	HTMLCollection	Location
Navigator	Screen	Style	Window	Storage

Event Object

- Quando ocorre um evento, é criado automaticamente um **event object**
 - Guarda informação sobre o evento (tipo, elemento onde ocorreu o evento, ...)
 - É passado automaticamente a qualquer função que seja **event handler ou listener**
 - A função é definida geralmente com o parâmetro `e (event)`

Property	Description
<u>bubbles</u>	Returns whether or not a specific event is a bubbling event
<u>cancelable</u>	Returns whether or not an event can have its default action prevented
<u>target</u>	Returns the element that triggered the event
<u>timestamp</u>	Returns the time (in milliseconds relative to the epoch) at which the event was created
<u>type</u>	Returns the name of the event
<u>view</u>	Returns a reference to the Window object where the event occurred

Method	Description
<u>preventDefault()</u>	Cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur
<u>stopImmediatePropagation()</u>	Prevents other listeners of the same event from being called
<u>stopPropagation()</u>	Prevents further propagation of an event during event flow

Event Object

- O *event object* é passado de forma automática à função que é disparada pelo evento
 - por convenção identifica-se por (e) (não é obrigatório!)
 - permite utilizar as propriedades e métodos do *event object* no interior da função

```
<form action="#">
  <input type="text" placeholder="Insert Name" id="user"><br><br>
  <input type="password" placeholder="Insert Password">
  <input type="submit">
</form>
```



Event Object

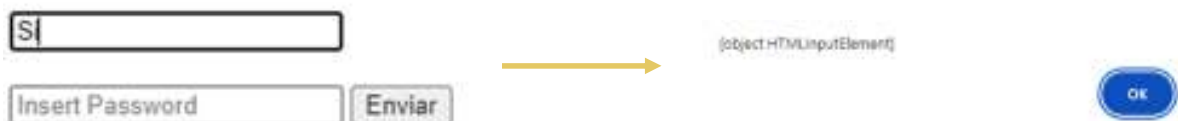
- Função executada **não possui** parâmetros de entrada

```
var element=document.getElementById('user')

function checkUser(e){
  var blurInput=e.currentTarget
  var elUser=document.getElementById('user')
  if (elUser.value.length<5)
    alert(blurInput)
}
element.addEventListener('blur',checkUser,false)
```

Evento disparado, o event object é passado à função

Propriedades e métodos disponíveis no event (ex: currentTarget)



Event Object

- Função executada **possui** parâmetros de entrada

```
var element=document.getElementById('user')

function checkUser(e, minlength){
    var blurInput=e.currentTarget
    var elUser=document.getElementById('user')
    if (elUser.value.length<minlength)
        alert(blurInput + " minlength :" + minlength)
    }

element.addEventListener('blur',function(eObj){checkUser(eObj,10)},false)
```



Tipos de Eventos

Tipos de Eventos

■ Mouse Events

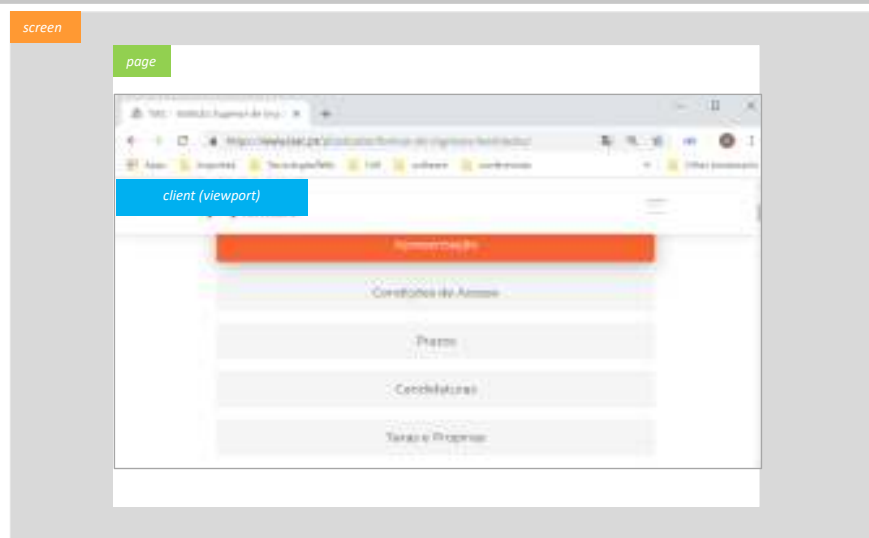
- Dispara quando o:

event	trigger
<i>click</i>	utilizador clica no botão esquerdo do rato. Também dispara quando o utilizador pressiona a tecla <i>Enter</i> num elemento com <i>focus</i>
<i>dblclick</i>	utilizador faz um <i>double click</i> no botão esquerdo do rato
<i>mousedown</i>	utilizador pressiona qualquer botão do rato
<i>mouseup</i>	utilizador liberta qualquer botão do rato (não pode ser iniciado por teclado)
<i>mouseover</i>	cursor do rato é sobreposto a um determinado elemento
<i>mouseout</i>	cursor do rato está sobre um elemento e se move para outro elemento (externo ao elemento inicial / não é filho do elemento inicial)
<i>mousemove</i>	cursor é movido em torno de um elemento (este evento é disparado de forma repetida)

Mouse Events

Exemplo

Propriedades específicas mouse events



screen

screenX e **screenY** indicam a posição relativamente à totalidade do ecrã (top left corner)

page

pageX e **pageY** indicam a posição do cursor relativamente à página. O topo da página pode estar fora do viewport, ou seja mesmo com o cursor na mesma posição as coordenadas podem ser diferentes

client

clientX e **clientY** indicam a posição do cursor relativamente ao viewport. O scroll da página não afeta estas coordenadas

Mouse Events (propriedades específicas)

```
<ul id="posicao">
  <li id="sx">Screen X</li>
  <li id="sy">Screen Y</li>
  <li id="px">Page X</li>
  <li id="py">Page Y</li>
  <li id="cx">Client X</li>
  <li id="cy">Client Y</li>
</ul>
```

Posição do cursor do rato

Screen X
Screen Y
Page X
Page Y
Client X
Client Y

```
var eventPos=document.getElementById('posicao')
```

```
function mostraCoord (){
```

```
  document.getElementById('sx').innerHTML = event.screenX
```

```
  document.getElementById('sy').innerHTML = event.screenY
```

```
  document.getElementById('px').innerHTML = event.pageX
```

```
  document.getElementById('py').innerHTML = event.pageY
```

```
  document.getElementById('cx').innerHTML = event.clientX
```

```
  document.getElementById('cy').innerHTML = event.clientY }
```

```
eventPos.addEventListener('click',mostraCoord,false)
```

55
197
55
76
55
76

screen

page

client

Tipos de Eventos

■ User Interface Events

■ Dispara:

event	trigger
load	quando é finalizado o download da página. Também pode ser aplicado a outros elementos em que é necessário o download, como <i>images, scripts, objects</i>
error	quando o browser encontra um erro JavaScript
resize	quando a janela do browser foi redimensionada
scroll	quando o user faz um scroll (down/up) da página

■ Exemplo: load event

```
<form id="myForm">
  <input type="text" name="nome" placeholder="Nome completo" id="inp1"><br>
  <input type="submit" value="submit"><br>
</form><br>
```

```
function loadEvent(){
  document.getElementById('inp1').value="Simão Paredes"
}

window.addEventListener('load',loadEvent,false)
```



Evento **load** não é associado a nenhum elemento HTML.

Evento associado ao Window object

Tipos de Eventos

■ Keyboard Events

- Dispara quando o:

event	trigger
input	conteúdo de um elemento input ou textarea é alterado
keydown	utilizador pressiona qualquer tecla do teclado. Se o utilizador mantiver a tecla pressionada o evento dispara de forma repetida.
keypress	utilizador pressiona qualquer tecla que resultaria num carácter visível no ecrã.
keyup	utilizador liberta uma tecla do teclado. Este evento, ao contrário de keydown e keypress, é disparado após a representação do carácter.

KeyboardEvent Object

Property	Description
altKey	Returns whether the "ALT" key was pressed when the key event was triggered
ctrlKey	Returns whether the "CTRL" key was pressed when the key event was triggered
charCode	Returns the Unicode character code of the key that triggered the onkeypress event
key	Returns the key value of the key represented by the event
keyCode	Returns the Unicode character code of the key that triggered the onkeypress event, or the Unicode key code of the key that triggered the onkeydown or onkeyup event
location	Returns the location of a key on the keyboard or device
metaKey	Returns whether the "meta" key was pressed when the key event was triggered
shiftKey	Returns whether the "SHIFT" key was pressed when the key event was triggered
which	Returns the Unicode character code of the key that triggered the onkeypress event, or the Unicode key code of the key that triggered the onkeydown or onkeyup event

Tipos de Eventos

```
<form action="#">
  <textarea id="t1" rows="5" cols="30"></textarea>
  <input type="submit" value="submit"><br>
</form><br>
<p id="charleft"></p>
```



168

```
var textArea = document.getElementById('t1')

function contaCarateres(){
  var textEnter = document.getElementById('t1').value
  var charDisplay=document.getElementById('charleft')
  var counter=180 - (textEnter.length)

  console.log(counter)
  charDisplay.textContent=counter
}

textArea.addEventListener('keypress',contaCarateres,false)
```

2) atualiza o contador

1) disparado repetidamente

Tipos de Eventos

■ Form Events

<i>event</i>	Trigger
<i>submit</i>	dispara no nó que representa o elemento <i>form</i> , geralmente utilizado para verificar os valores introduzidos antes de enviar ao servidor
<i>change</i>	dispara quando uma alteração é efetuada em alguns elementos do <i>form</i> (ex: seleção num menu <i>drop-down</i> , seleção de um <i>radio button</i> , seleção de uma <i>checkbox</i> , ...)
<i>input</i>	dispara quando são introduzidos valores nos elementos <i>input</i> e/ou <i>textarea</i>
<i>blur</i>	dispara quando se abandona um determinado elemento. Não é exclusivo dos <i>forms</i> (pode por exemplo ser utilizado com <i>links</i>).
<i>focus</i>	dispara quando se seleciona um determinado elemento. Não é exclusivo dos <i>forms</i> (pode por exemplo ser utilizado com <i>links</i>).

Tipos de Eventos

■ Mutation Events

- Sempre que a estrutura da *DOM tree* é alterada (elementos adicionados ou removidos) é disparado um *mutation event*

<i>event</i>	Trigger
<i>DOMNodeInserted</i>	dispara quando um nó é inserido na DOM Tree
<i>DOMNodeRemoved</i>	dispara quando um nó é removido da DOM Tree
<i>DOMSubtreeModified</i>	dispara quando a estrutura da DOM Tree é alterada.
<i>DOMNodeInsertedIntoDocument</i>	dispara quando um nó é inserido na DOM Tree como descendente de outro nó já existente
<i>DOMNodeRemovedFromDocument</i>	dispara quando um nó é removido da DOM Tree como descendente de outro nó já existente

Mutation Events

```
<h2>Lista de Compras <span id="counter">1</span></h2>
<ul id="list">
  <li>Morangos</li>
</ul>
<div><a href="#">Novo Item</a></div>
```

```
var listRef=document.getElementById('list')
var linkElement=document.getElementsByTagName('a')[0]
var counterElement=document.getElementById('counter')

function adicionaElemento(){
  var newText = document.createTextNode('Bananas')
  var newElement=document.createElement('li')
  newElement.appendChild(newText)
  listRef.appendChild(newElement)}

function atualizaContador(){
  counterElement.innerHTML=document.getElementsByTagName('li').length}

linkElement.addEventListener('click',adicionaElemento,false)
listRef.addEventListener('DOMNodeInserted',atualizaContador,false)
```

Lista de Compras1

Morangos

[Novo Item](#)

Lista de Compras2

Morangos
Bananas

[Novo Item](#)

É disparado a cada alteração

Tipos de Eventos

■ Other HTML Events

Event	Trigger
<i>DOMContentLoaded</i>	Dispara no window object quando a DOM Tree está formada ou seja é efetuado download dos elementos HTML. O script é executado antes de ter sido efetuado o download de todos os outros elementos (css, imagens, ...).
<i>beforeunload</i>	Dispara no window object antes de se efetuar o unload da página. Constitui uma ajuda ao utilizador, por exemplo informar o utilizador de que as alterações a um formulário não foram gravadas.

FIM!