

Operadores

Operadores

■ Aritméticos

■ Exemplo: y=5

Operator	Description	Example	Result of x	Result of y
+	Addition	$x=y+2$	7	5
-	Subtraction	$x=y-2$	3	5
*	Multiplication	$x=y*2$	10	5
/	Division	$x=y/2$	2.5	5
%	Modulus (division remainder)	$x=y\%2$	1	5
++	Increment	$x=++y$	6	6
		$x=y++$	5	6
--	Decrement	$x=--y$	4	4
		$x=y--$	5	4

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

Operadores

■ Atribuição

■ Exemplo: $x=10$, $y=5$

Operator	Example	Same As	Result
=	$x=y$		$x=5$
+=	$x+=y$	$x=x+y$	$x=15$
-=	$x-=y$	$x=x-y$	$x=5$
=	$x=y$	$x=x*y$	$x=50$
/=	$x/=y$	$x=x/y$	$x=2$
%=	$x\%=y$	$x=x\%y$	$x=0$

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

Operadores

■ Lógicos

■ Exemplo: $x=6$; $y=3$

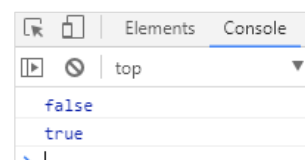
Operator	Description	Example
&&	and	$(x < 10 \ \&\& \ y > 1)$ is true
	or	$(x==5 \ \ y==5)$ is false
!	not	$!(x==y)$ is true

<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

■ Exemplo: $x=5$

Operator	Description	Comparing	Returns
==	is equal to	$x==8$	<i>false</i>
		$x==5$	<i>true</i>
===	is exactly equal to (value and type)	$x==="5"$	<i>false</i>
		$x===5$	<i>true</i>
!=	is not equal	$x!=8$	<i>true</i>
!==	is not equal (neither value nor type)	$x!== "5"$	<i>true</i>
		$x!== 5$	<i>false</i>
>	is greater than	$x>8$	<i>false</i>
<	is less than	$x<8$	<i>true</i>
>=	is greater than or equal to	$x>=8$	<i>false</i>
<=	is less than or equal to	$x<=8$	<i>true</i>

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

Operadores

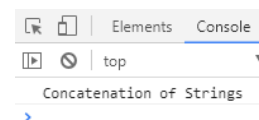
■ Strings

■ Concatenação (+)

■ Operador utilizado muito frequentemente

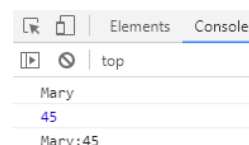
```
<script>
  var strC = "Concatenation " + "of" + " Strings";
  console.log(strC);
</script>
```

Concatenação



■ type coercion (um dos argumentos é uma string):

```
console.log(name);
console.log(age);
console.log(name + ":" + age);
```



Operadores

■ Precedência de Operadores

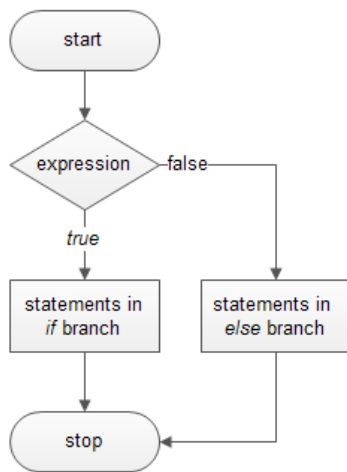
The following table is ordered from highest (20) to lowest (1) precedence.

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 Condicionais

Seleção



```
<script type="text/javascript">
  var teste = "verdadeiro";

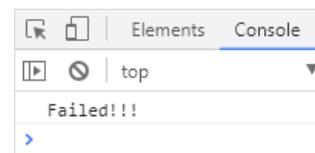
  if (teste == "verdadeiro")
    document.write("Condição Verdadeira!");
  else
    document.write("Condição Falsa!");
</script>
```

Seleção

▪ if ...else

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

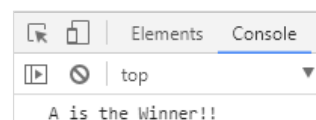
  if (grade>=threshold)
    {console.log("Aproved!!!");}
  else
    {console.log("Failed!!!");}
</script>
```



▪ 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>
```

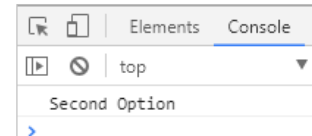


Seleção

■ 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>
```

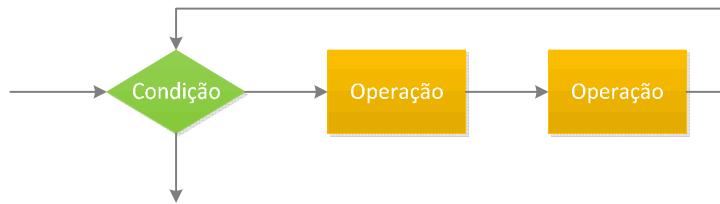


Estruturas de Repetição

Repetição

■ Estruturas de Repetição / Ciclos (Loops)

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



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

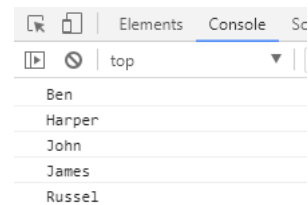


Repetição

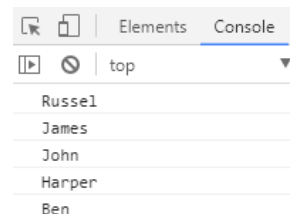
■ Ciclos (Loops)

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

```
names=["Ben", "Harper","John", "James", "Russel"];  
  
for(i=0; i<names.length; i++)  
{  
  console.log(names[i]);  
}
```



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

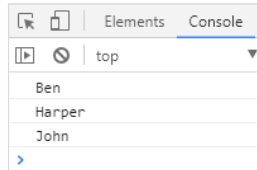


Repetição

■ Ciclos (Loops)

■ *break*

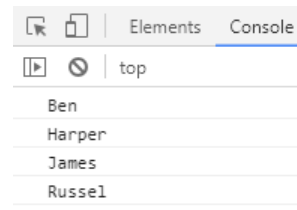
```
for(i=0; i<names.length; i++)  
{  
    console.log(names[i]);  
    if(i===2)  
        {break;}  
}
```



break: interrompe o funcionamento do ciclo

■ *continue*

```
names=["Ben", "Harper","John", "James", "Russel"];  
for(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;	valor booleano false
var highScore = 0;	número 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;	valor booleano true
var highScore = 1;	número ≠ 0
var highScore = 'xxxx';	string com conteúdo
var highScore = 10/5	resultado de um cálculo ≠ 0
var highScore='true'; var highScore='0'; var highScore='false';	definidos como strings

Funções

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);
```

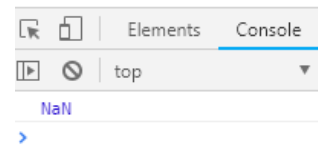
- Argumentos da função definidos através de variáveis

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

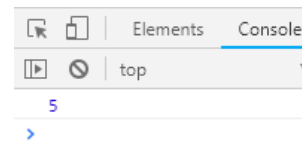
Funções

- 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));
```



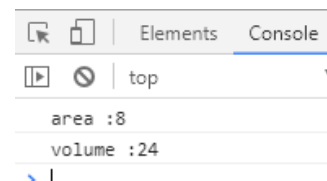
Funções

- 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

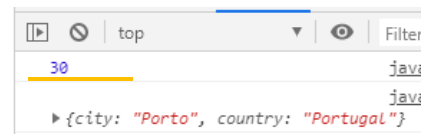
- Os *Primitive Type* e os Objetos são passados à função de forma diferente
 - Nos *Primitive Types* é feita a **passagem do valor** do argumento:
 - todas as alterações efetuadas no parâmetro no interior da função não alteram o valor original
 - Nos *Reference Types* a passagem é **feita por referência**:
 - 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>
```



Funcões

- 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

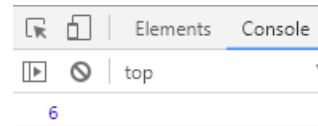
■ Declaração de Função

■ Chamada à função:

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

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

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



6

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)**
 - É tratada como uma expressão, ou seja a função é interpretada só após o processamento da expressão onde está integrada

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

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

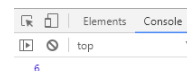
```
<script>
  var area=calculaArea(2,3);

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

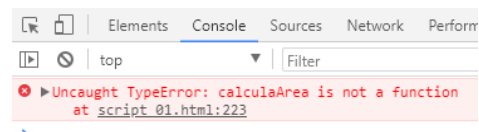
  console.log(area);
</script>
```

6

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



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
 - event handling

```
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**



Funcões

- O JavaScript admite formas diferentes de criar uma função:

Declaração de Função

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

Function Expression

(anonymous function)

```
var calculateSum = function (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**

Arrow Functions

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

Redução do Acoplamento


(keyword *this*)

Funções

■ *anonymous function*

- A definição do método **referencia diretamente** o objeto o que cria um acoplamento direto entre o método e o objeto.

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



- Este **acoplamento** é indesejável uma vez que se a designação do objeto muda também terá de ser alterada a referência ao objeto que é feita no método
- Inviabiliza a aplicação da mesma função a **objetos diferentes**

Funções

- O acoplamento direto entre método e objeto é sempre indesejável:

```
function courseName()  
{  
    document.write(course.name);  
}  
  
var course={  
    name:"Web Technologies <br/>",  
    displayName:courseName  
};  
course.displayName();  
  
var courseAlt={  
    name:"Digital Systems <br/>",  
    displayName:courseName  
};  
courseAlt.displayName();
```



Acoplamento direto com o objeto *course* aqui o método *displayName()* **funciona como desejado**



Com o objeto *courseAlt* o método *displayName()* já **não funciona corretamente**



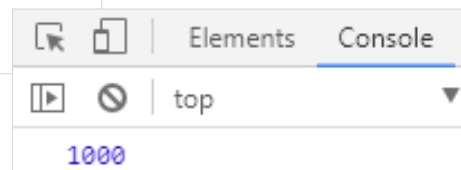
- Como evitar esta situação?
 - **keyword *this***

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 num construtor a utilização do *this* refere-se a cada instância em particular

```
<script>  
    var shape={  
        width:50,  
        height:20,  
        calculateArea:function(){  
            return (this.width * this.height);  
        }  
    }  
  
    console.log(shape.calculateArea());  
</script>
```

width / height do objeto *shape* (container)



this

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

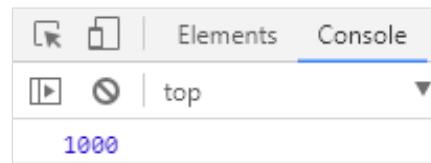
```
<script>
  var width=600;
  var shape={width:50, height:20}

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

  shape.getArea=calculateArea;

  console.log(shape.getArea());
</script>
```

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



this

- Exemplo anterior:

```
<script>

function courseName()
{
  document.write(this.name);
}

var course={
  name:"Web Technologies <br/>",
  displayName:courseName
};
course.displayName();

var courseAlt={
  name:"Digital Systems <br/>",
  displayName:courseName
};
courseAlt.displayName();
```

?
scope

?
scope

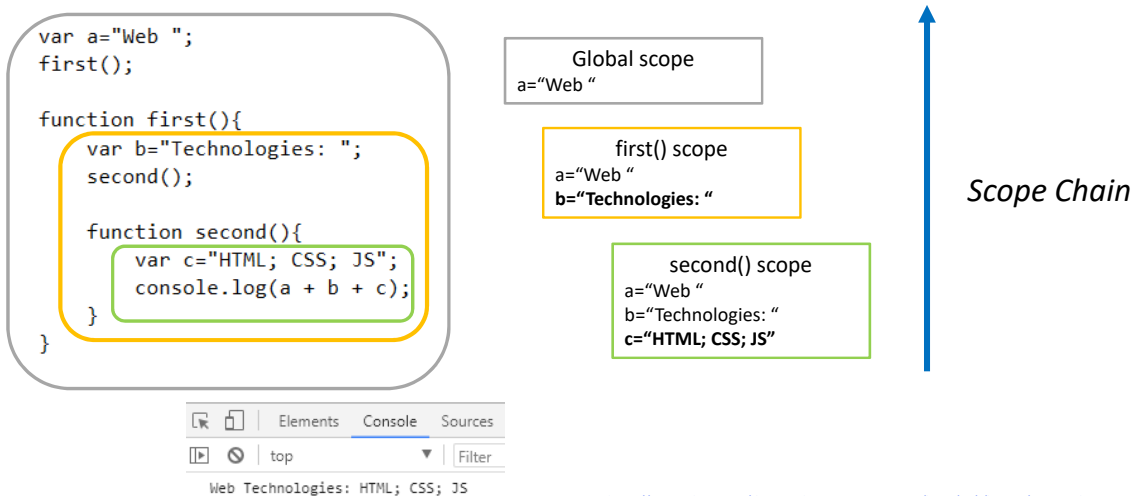


Web Technologies
Digital Systems

Scope das Variáveis

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*)

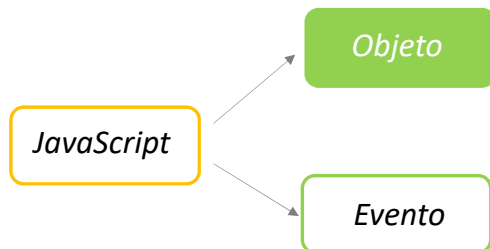


Scope

- 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
- 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

Objetos

Objetos



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.

Objetos

- *Representa uma entidade física ou conceptual. Um objeto tem **estado** (propriedades), comportamento (métodos) e identidade.*

- 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>
```

(.)dot notation

■ Propriedades

- ***objectName.propertyName***

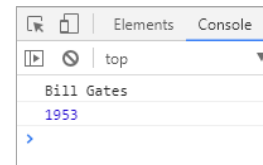
```
console.log(bill.name);
```

```
var bill = {  
  name:"Bill Gates",  
  age:65,  
  height:185,  
  calculateYearBirth: function()  
    {console.log(2018-this.age)}  
}
```

■ Métodos

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

```
bill.calculateYearBirth();
```



Objetos

- Como referido, existem formas diferentes de criar objetos:

1. Forma Literal (literal notation)

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

Nome do objeto

Propriedades em que os valores podem ser:

string
number
boolean
array

Métodos

O scope do **this** é o objeto hotel

Objetos

2. Baseado no constructor **Object()**

- keyword new + Object()
 - var nome = **new Object ()** : cria um objeto ao qual se adicionam propriedades e métodos
 - sintaxe totalmente diferente da situação anterior (forma literal)

```
<script>
    var hotel = new Object();

    hotel.name="Coimbra";
    hotel.rooms=20;
    hotel.booked=15;

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

    console.log(hotel.checkAvailability());
</script>
```

Objetos

3. Baseado na definição de uma class

"A class is a blueprint for objects"

- Por convenção o nome da class inicia-se com maiúscula
- Todas a classes possuem um **constructor** (método) onde se inicializam as propriedades

```
class Player {
    constructor(firstName,lastName){
        this.fname=firstName;
        this.lname=lastName;
    };

    showName(){
        console.log('First name:' + this.fname + ' ; Last name:' + this.lname);
    }
}
```

nome da Classe

constructor (propriedades)

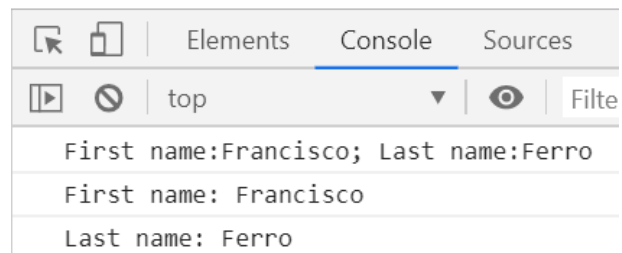
método

Objetos

3. Baseado na definição de uma class

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

```
let pInstance = new Player('Francisco','Ferro');  
  
pInstance.showName();  
  
console.log('First name: ' + pInstance.fname);  
console.log('Last name: ' + pInstance.lname);
```



Objetos

- As classes permitem herança:
 - uma classe pode fazer o *extend* de outra classe (herdar as suas propriedades e métodos.
 - `super()` na classe filha é obrigatório e executado para invocar o construtor da classe pai

```
class Person{  
  constructor(age, country){  
    this.age=age;  
    this.country=country;  
  }  
  showFeatures(){  
    console.log('Age: ' + this.age + ' Country: ' + this.country);  
  }  
};  
  
class Player extends Person{  
  constructor(firstName, lastName){  
    super(22, 'Portugal');  
    this.fname=firstName;  
    this.lname=lastName;  
  };  
  
  showName(){  
    console.log('First name:' + this.fname + '; Last name:' + this.lname);  
  }  
}
```

Person

Player

Objetos

■ Herança

```
let pInstance = new Player('Francisco', 'Ferro');
```

```
pInstance.showName();
```

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

Player

```
pInstance.showFeatures();
```

Person

First name:Francisco; Last name:Ferro

First name: Francisco

Last name: Ferro

Age: 22 Country: Portugal

Criação de Objetos em JS

Resumo

Objetos

- Criar um objeto **vazio**, ao qual se **adicionam** propriedades e métodos

```
var hotel = {};  
hotel.name = 'Coimbra';  
hotel.rooms = 20;  
hotel.booked = 15;  
  
hotel.checkAvailability = function () {  
    return this.rooms - this.booked;  
}
```

Literal notation

```
var hotel = new Object();  
hotel.name = 'Coimbra';  
hotel.rooms = 20;  
hotel.booked = 15;  
  
hotel.checkAvailability = function () {  
    return this.rooms - this.booked;  
}
```

*Object Constructor
notation*

Objetos

- Criar um objeto de forma literal **com** propriedades e métodos
 - Muito pouco eficiente quando se pretendem criar múltiplas instâncias, obriga a definição repetida dos métodos
 - Sintaxe diferente

```
var hotel = {  
    name: 'Coimbra',  
    rooms: 20,  
    booked: 15,  
    gym: true,  
    checkAvailability: function () {  
        return this.rooms - this.booked;  
    }  
}
```

Literal notation

Objetos

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

```
class Player {  
  constructor(firstName,lastName){  
    this.fname=firstName;  
    this.lname=lastName;  
  };  
  
  showName(){  
    console.log('First name:' + this.fname + '; Last name:' + this.lname);  
  }  
}  
  
let pInstance = new Player('Francisco','Ferro');
```

*Adicionar/Remover/Deletar
propriedades*

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

```
book={
  title:"Javascrit",
  year:2018,
  editor:"O'Reilly",

  showDetails:function(){
    return ("Book title: " + this.title + " ;    Book editor: " + this.editor);
  }
}
```

```
...
}

book.pages=250;

alert("Number of pages: " + book.pages);
```

Number of pages: 250

OK

Objetos

■ Remover propriedades

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

```
book={
  title:"Javascrit",
  year:2018,
  editor:"O'Reilly"
}

console.log('Title: ' + book.title);

delete book.title;

console.log('Title: ' + book.title);
```

	Elements	Console
▶	top	
	Title: Javascrit	
	Title: undefined	

- A propriedade foi removida como tal não se encontra definida (*undefined*)
- 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*

```
book={  
  title:"Javascrit",  
  year:2018,  
  editor:"O'Reilly"  
}
```

```
delete book.title;
```

```
console.log('title' in book);
```



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/js>

JavaScript Built-in Objects

- **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

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

JavaScript Built-in Objects

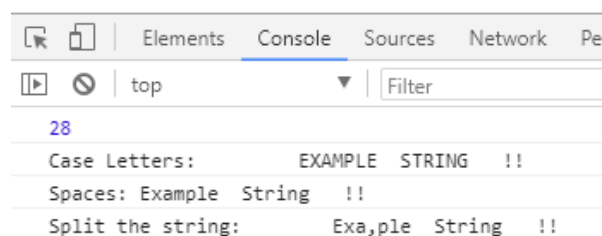
■ String

```
<script>

var exStr="      Example String  !!"

console.log(exStr.length);

console.log("Case Letters: " + exStr.toUpperCase());
console.log("Spaces: " + exStr.trim());
console.log("Split the string:" + exStr.split('m'));
</script>
```



JavaScript Built-in Objects

■ Number

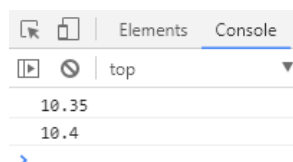
Propriedade	Descrição
MAX_VALUE	retorna o maior número que pode ser representado em JS
MIN_VALUE	retorna o menor número que pode ser representado em JS

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 string
isNaN()	Verifica se o valor não é um número

```
<script>

var x=10.354;

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



JavaScript Built-in Objects

■ 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))

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

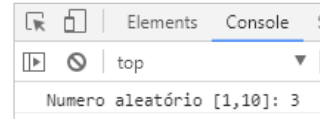
JavaScript Built-in Objects

■ Math

- Gera um número aleatório entre 1 e 10
 - é gerado um número aleatório entre 0 e 1
 - multiplicado por 10 e arredondado para o número inteiro imediatamente anterior
 - número entre 0 e 9 ao qual se adiciona 1

```
<script>
  var x = Math.floor(Math.random()*10)+1;

  console.log("Numero aleatório [1,10]: " + x);
</script>
```



JavaScript Built-in Objects

■ 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
```

Date Get Methods	
Get methods are used for getting a part of a date. Here are the most common (alphabetically):	
Method	Description
<code>getDate()</code>	Get the day as a number (1-31)
<code>getDay()</code>	Get the weekday as a number (0-6)
<code>getFullYear()</code>	Get the four digit year (yyyy)
<code>getHours()</code>	Get the hour (0-23)
<code>getMilliseconds()</code>	Get the milliseconds (0-999)
<code>getMinutes()</code>	Get the minutes (0-59)
<code>getMonth()</code>	Get the month (0-11)
<code>getSeconds()</code>	Get the seconds (0-59)
<code>getTime()</code>	Get the time (milliseconds since January 1, 1970)

Date Set Methods	
Set methods are used for setting a part of a date. Here are the most common (alphabetically):	
Method	Description
<code>setDate()</code>	Set the day as a number (1-31)
<code>setFullYear()</code>	Set the year (optionally month and day yyyy.mm.dd)
<code>setHours()</code>	Set the hour (0-23)
<code>setMilliseconds()</code>	Set the milliseconds (0-999)
<code>setMinutes()</code>	Set the minutes (0-59)
<code>setMonth()</code>	Set the month (0-11)
<code>setSeconds()</code>	Set the seconds (0-59)
<code>setTime()</code>	Set the time (milliseconds since January 1, 1970)

JavaScript Built-in Objects

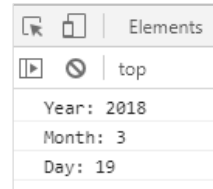
■ Date

```
<script>

var tdyDate = new Date();

console.log("Year: " + tdyDate.getFullYear());
console.log("Month: " + (tdyDate.getMonth()+1));
console.log("Day: " + tdyDate.getDate());

</script>
```



Elements
top
Year: 2018
Month: 3
Day: 19

Date Get Methods

Get methods are used for getting a part of a date. Here are the most common (alphabetically):

Method	Description
<code>getDate()</code>	Get the day as a number (1-31)
<code>getDay()</code>	Get the weekday as a number (0-6)
<code>getFullYear()</code>	Get the four digit year (yyyy)
<code>getHours()</code>	Get the hour (0-23)
<code>getMilliseconds()</code>	Get the milliseconds (0-999)
<code>getMinutes()</code>	Get the minutes (0-59)
<code>getMonth()</code>	Get the month (0-11)
<code>getSeconds()</code>	Get the seconds (0-59)
<code>getTime()</code>	Get the time (milliseconds since January 1, 1970)

JavaScript Built-in Objects

■ Array

- permite diferentes tipos de elementos

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

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

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>
```



JavaScript Built-in Objects

■ Arrays

- métodos (exemplo):

```
<script>

var john = ['John', 'Smith', 1990, 'designer', false];

john.push('blue');
john.unshift('Mr. ');

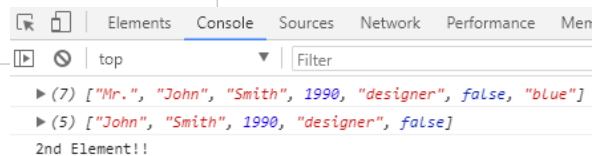
console.log(john);

john.pop();
john.shift();

console.log(john);

if (john.indexOf('Smith') === 1) {
  console.log('2nd Element!!');
}

</script>
```



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) => {return num * 2;});  
  
console.log(numbers);  
console.log(newNumbers);
```

```
▶ (4) [1, 2, 3, 4]  
▶ (4) [2, 4, 6, 8]  
>
```

▪ *find()*

- Retorna o valor do primeiro elemento que verifica a condição

```
const numbers = [1, 2, 3, 4];  
  
const newNumbers = numbers.find((num) => {return num > 2;});  
  
console.log(numbers);  
console.log(newNumbers);
```

```
▶ (4) [1, 2, 3, 4]  
3  
>
```

JavaScript Built-in Objects

▪ *findIndex()*

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

```
const numbers = [1, 2, 3, 4];  
  
const newNumbers = numbers.findIndex((num) => {return num === 2;});  
  
console.log(numbers);  
console.log(newNumbers);
```

```
▶ (4) [1, 2, 3, 4]  
1  
>
```

▪ *filter()*

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

```
const numbers = [1, 2, 3, 4];  
  
const newNumbers = numbers.filter((num) => {return num > 2;});  
  
console.log(numbers);  
console.log(newNumbers);
```

```
▶ (4) [1, 2, 3, 4]  
▶ (2) [3, 4]  
>
```

JavaScript Built-in Objects

▪ `reduce()`

- Executa uma função definida pelo utilizador executada 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)=>{return acc + acv;})

console.log(numbers);
console.log(newNumbers);
```

```
▶ (4) [1, 2, 3, 4]
10
>
```

▪ `concat()`

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

```
const numbers = [1,2,3,4];

const addNumbers = [5,6];

console.log(numbers.concat(addNumbers));
```

```
▶ (6) [1, 2, 3, 4, 5, 6]
>
```

JavaScript Built-in Objects

▪ `slice()`

- retorna uma cópia parcial desde uma posição inicial até ao final (a posição final não é especificada). O array original não é alterado.

```
const numbers = [1,2,3,4];

console.log(numbers.slice(2));
```

```
▶ (2) [3, 4]
>
```

▪ `splice()`

- altera o conteúdo de um array adicionando ou substituindo elementos (exemplo: remove 0 elementos a partir da posição 1, e adiciona o número 5)

```
const numbers = [1,2,3,4];
numbers.splice(1,0,5);

console.log(numbers);
```

```
▶ (5) [1, 5, 2, 3, 4]
```