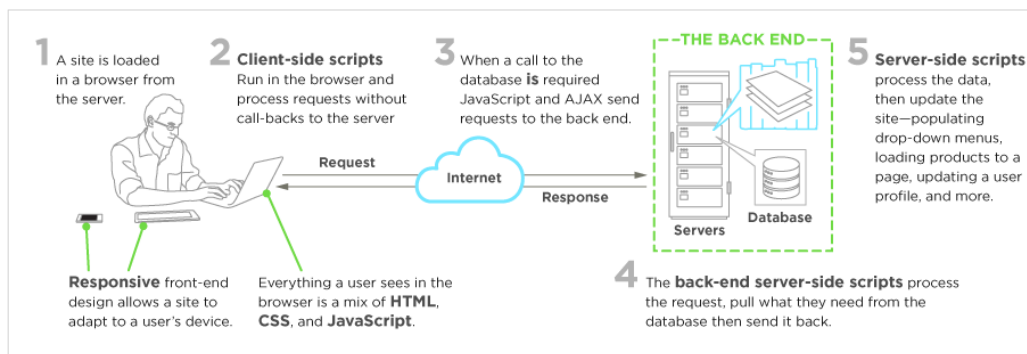


JavaScript

Tecnologias Web 2019/2020
Simão Paredes sparedes@lsec.pt

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/>

■ 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
- Mais recentemente, também utilizada no lado do servidor (*server-side*)
 - *Node.js*
- Executado
 - Quando a página é recebida
 - Como resposta a um evento resultado de uma ação do utilizador
- Permite:
 - Geração dinâmica de conteúdo / Efeitos
 - Melhorar a experiência do utilizador (interactividade)
 - Resposta a eventos, validação de dados, ...
 - Gerir a comunicação com o servidor (AJAX)
 -



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 eventos)

```
<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>`)

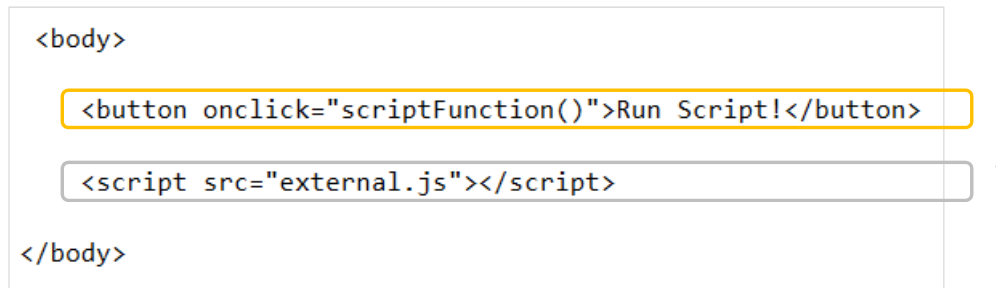
Run Script!

This page says:
Script Executado!

OK

Script Externo (*.js)

▪ `<script src="*.js"> ... </script>`



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

1. Embebido no HTML
2. Ficheiros externos (extensão .js)

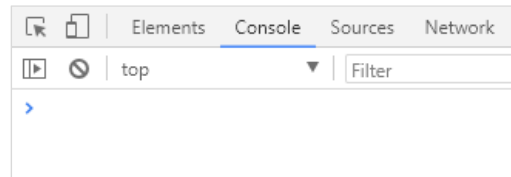
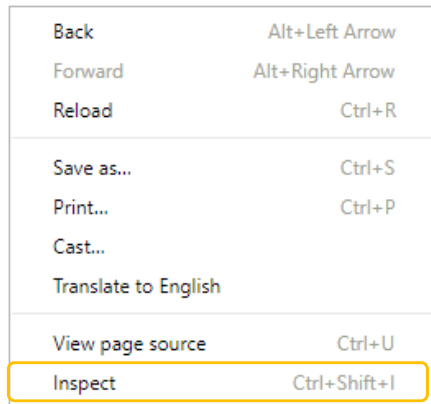
▪ Controlo da execução do script

- 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)

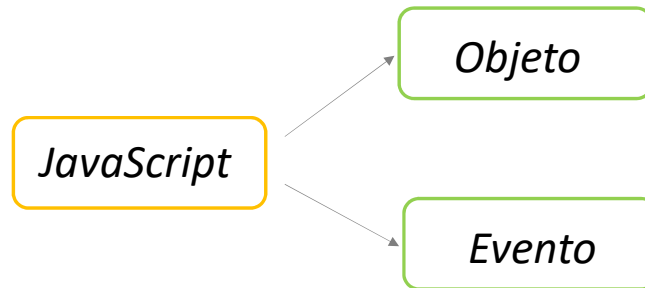
■ *Browser Console*

- Permite obter informação sobre o valor das variáveis, erros, *warnings*, *requests*, ...
- importante para efetuar o *debugging*/controlo completo dos scripts



Conceitos Chave

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>
```



■ 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.

<http://desenvolvimentoaparaweb.com/javascript/eventos-javascript/>

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

▪ *Expressions*

- a sua execução origina sempre um valor:
 - *numérico*
 - *string*
 - *boolean*
- podem ser parte de *statements*

```
3+5
declaredVariable="value"
functionCall()
```




<https://www.youtube.com/watch?v=WVvCr1cHi8>

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

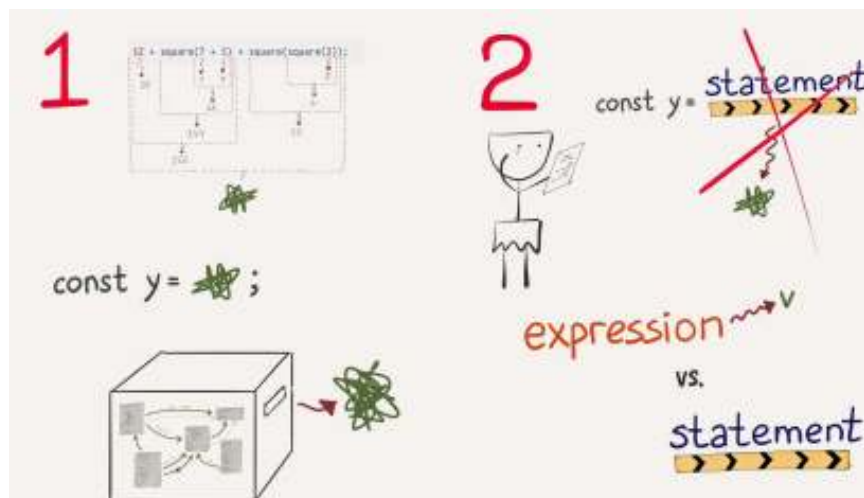
```
if (num > 0) {  
  return num;  
} else {  
  return 0;  
}  
  
while (counter <= n) {  
  result = result * result;  
}  
  
for (let counter = 1; counter <= n; counter++) {  
  result *= counter;  
}
```



Exemplos:

declaração de variáveis
declaração de função
if
if-else
while
do-while
for
switch
for-in

Sintaxe JS



As *expressions* originam um valor imediato

Os *statements* não e também não podem ser utilizados onde é esperada uma *expression*

```
console.log(const x); // error!  
let b = if (x > 10) { return 100; }; // error!
```

- 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*

- não é necessário ; no final de :

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

- necessário ; quando:

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

```
let x = 4;  
  
let x = function () {};  
  
(function() {}()) ;
```

Variáveis

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**

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

- A variável só é visível no bloco onde foi criada

■ **const**

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

* - slide 48

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*).

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

Primitive Types

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

Reference Types

Armazenados como objetos

Primitive Types

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



Primitive Types

- O JS possui 5 *primitive types*:
 - number
 - `var count=25;`
 - string
 - `var name="string exemplo";`
 - boolean
 - `var found=true;`
 - null
 - `var object = null;`
 - undefined
 - `var data;`
 - *variável sem inicialização definida*

Primitive Types

■ **number**

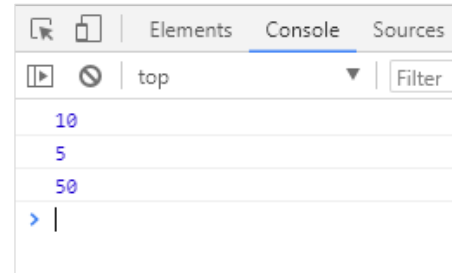
- Todos os números são representados através de *floats* de 64 bits

- Exemplos:

- `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>
```



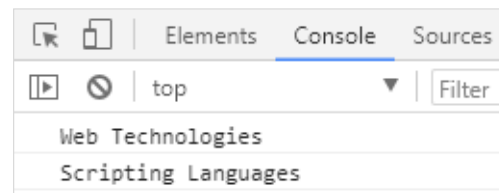
Primitive Types

■ **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

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

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

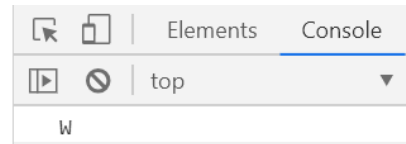


Primitive Types

▪ **string**

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

```
var strIndex ="Hello World!";  
console.log(strIndex[6]);
```



- Ao contrário de outras linguagens, ex: C, apesar de permitir indexação uma *string* não pode ser diretamente alterada
- Quando se pretende incorporar " ou ' numa *string*, deve declarar-se a *string* com o símbolo que não se pretende representar

"... just want to use double quotes in the string, you could surround the entire string in single quotes..."

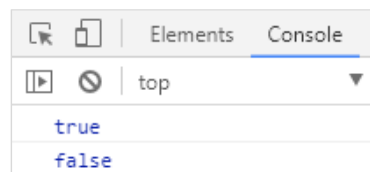
- Em alternativa pode recorrer-se a uma *backslash* \ antes da aspa ou da plica que se pretende representar.

Primitive Types

▪ **boolean**

- podem assumir apenas dois valores:
 - **true ; false**

```
<script>  
  var boolTrue=true;  
  var boolFalse=false;  
  
  console.log(boolTrue);  
  console.log(boolFalse);  
</script>
```



Reference Types *(objects)*

Reference Types

■ *Objetos*

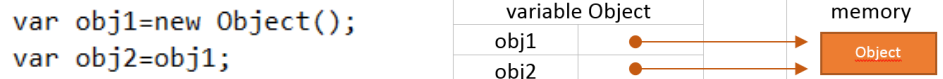
- Um objeto é uma lista não ordenada de **propriedades**, consistindo num nome (string) e num valor. Quando este valor é uma função, cria-se um **método**.
- Formas diferentes de criar objetos:
 - Operador **new** + constructor **Object()**
 - *constructor* é uma função que permite a criação de um objeto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

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.

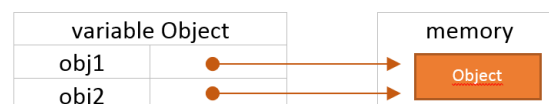


- Quando se atribui um objeto a uma variável, na realidade essa variável armazena um ponteiro que referencia o mesmo objeto
 - De facto existe apenas um objeto, o qual está a ser referenciado (apontado) por duas variáveis.



Reference Types

- O mesmo objeto referenciado por duas variáveis



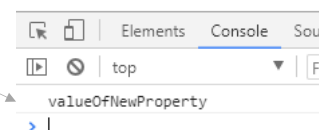
```
<script>
  var obj1=new Object();
  var obj2=obj1;
  obj1.newProperty="valueOfNewProperty";
  console.log(obj2.newProperty);
</script>
```

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 **};**

```
var books={  
  name: "JavaScript",  
  year:2014  
};
```

■ 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.

```
var books=new Object();  
  
books.name="JavaScript";  
books.year=2014;
```

Propriedades / Métodos objetos

(.)dot notation

■ Aceder a propriedades

■ *objectName.propertyName*

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

```
alert("Book title: " + book.title);
```

Book title: Javascrit

OK

■ Aceder ao editor?

```
alert("Book Editor: " + book.editor);
```

Book Editor: O'Reilly

OK

(.)dot notation

■ Aceder a métodos

■ *objectName.methodName()*

```
book={  
  title:"Javascrit",  
  year:2018,  
  editor:"O'Reilly",  
  
  showDetails:function(){  
    return ("Book title: " + this.title + " ;    Book editor: " + this.editor);  
  }  
}
```

```
alert (book.showDetails());
```

Book title: Javascrit; Book editor: O'Reilly

OK

Reference Types

- Alterar o valor de propriedades:

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

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

```
...
}

book.editor="Wiley";
alert("Book Editor: " + book.editor);
```

Book Editor: Wiley

OK

Os objetos podem ser alterados (propriedades alteradas / adicionadas/removidas) em qualquer altura

Reference Types

- Adicionar Propriedades/Métodos

```
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

Os objetos podem ser alterados (propriedades alteradas / adicionadas/removidas) em qualquer altura

Built-in Types

(Reference Types)

Reference Types

■ Built-in types

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

- Object `var books=new Object();`

- Array `var 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*

- `var items=[];`

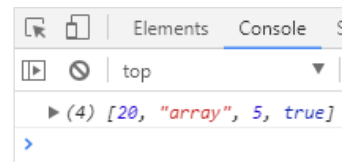
Reference Types

■ Array

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

```
<script>
  var values=[20, "array", 5, true];

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



- Baseado num *constructor*:

```
var values=new Array(20, "array", 5, true);
```

Array

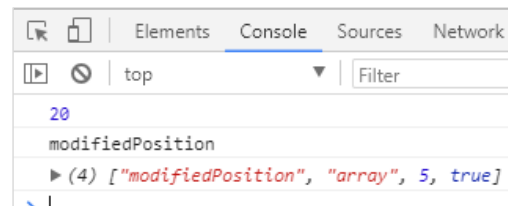
■ Indexação

- nomeArray [*posição*]
- Índices iniciam-se em **zero**
- Propriedade **length** é muito importante (retorna a dimensão do *array*)

```
<script>
  var values=[20, "array", 5, true];
  console.log(values[0]);

  values[0]="modifiedPosition";
  console.log(values[0]);

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



Array

Array Properties

Property	Description
constructor	Returns the function that created the Array object's prototype
length	Sets or returns the number of elements in an array
prototype	Allows you to add properties and methods to an Array object

Array Methods

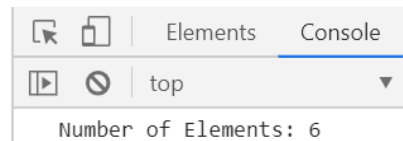
Method	Description
concat()	Joins two or more arrays, and returns a copy of the joined arrays
...	
map()	Creates a new array with the result of calling a function for each array element
pop()	Removes the last element of an array, and returns that element
push()	Adds new elements to the end of an array, and returns the new length
reduce()	Reduce the values of an array to a single value (going left-to-right)
reduceRight()	Reduce the values of an array to a single value (going right-to-left)
reverse()	Reverses the order of the elements in an array
shift()	Removes the first element of an array, and returns that element
slice()	Selects a part of an array, and returns the new array
some()	Checks if any of the elements in an array pass a test
sort()	Sorts the elements of an array
splice()	Adds/Removes elements from an array

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

Array

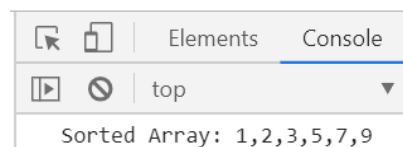
■ Propriedade: (exemplo)

```
<script>
let memberNumbs = [2, 5, 1, 7, 9, 3];
console.log("Number of Elements: " + memberNumbs.length);
```



■ Método: (exemplo)

```
<script>
let memberNumbs = [2, 5, 1, 7, 9, 3];
console.log("Sorted Array: " + memberNumbs.sort());
```



Reference Types

■ 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:

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

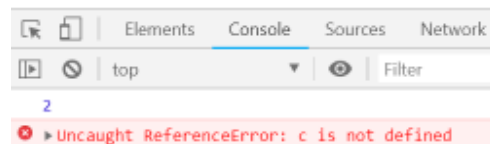


- À excepção do *built-in type Function* (notação literal) não existe uma forma correta ou errada de instanciar *built-in types*.

Declaração de Variáveis

■ let (JS ES6)

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

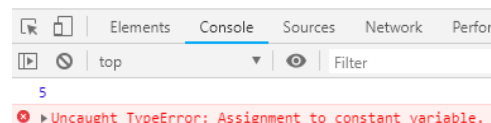


A variável só existe no bloco onde foi criada.

■ const (JS ES6)

- Não é possível alterar a atribuição do valor inicial

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



Primitive Wrapper Types

Primitive Wrapper Types

- Existem três **Primitive Wrapper Types**:
 - *String*
 - *Number*
 - *Boolean*
- Possibilitam o funcionamento com os *Primitive Types* da mesma forma (**dot notation**) que ocorre com os *Reference Types*
- Ao contrário dos *reference type*, um *primitive wrapper type* **não permite a adição de propriedades**, precisamente devido à criação de objetos temporários

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

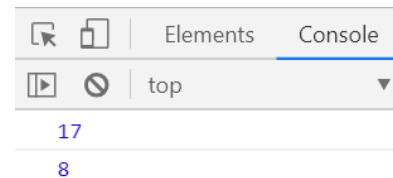
https://www.w3schools.com/jsref/jsref_obj_string.asp

Array

■ Propriedade: (exemplo)

```
<script>
```

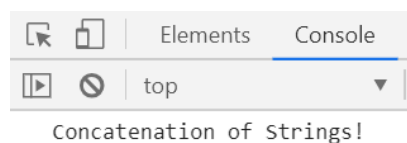
```
let firstStr='Concatenation of ';  
let secondStr='Strings!';  
console.log(firstStr.length);  
console.log(secondStr.length);
```



■ Método: (exemplo)

```
<script>
```

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



Variáveis

Primitive Types	Reference Types (Built-in)	Primitive Wrapper Types
Number	Object	Number
String	Array	String
Boolean	Date	Boolean
null	Error	
undefined	Function	
	RegExp	
	Math	
	...	

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

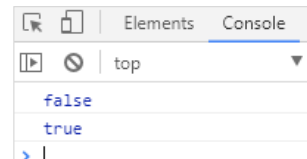
Operator	Description	Example
&&	and	(x < 10 && y > 1) is true
	or	(x==5 y==5) is false
!	not	!(x==y) is true

<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	false
		x==5	true
===	is exactly equal to (value and type)	x==="5"	false
		x==5	true
!=	is not equal	x!=8	true
!==	is not equal (neither value nor type)	x!="5"	true
		x!==5	false
>	is greater than	x>8	false
<	is less than	x<8	true
>=	is greater than or equal to	x>=8	false
<=	is less than or equal to	x<=8	true

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

Operadores

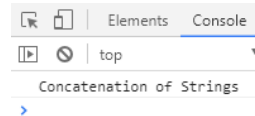
■ Strings

■ Concatenação (+)

- Operador utilizado muito frequentemente

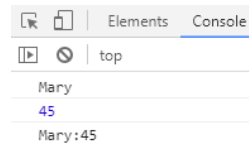
```
<script>
  var strC = "Concatenation " + "of" + " Strings";
  console.log(strC);
</script>
```

Concatenação



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

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



Operadores

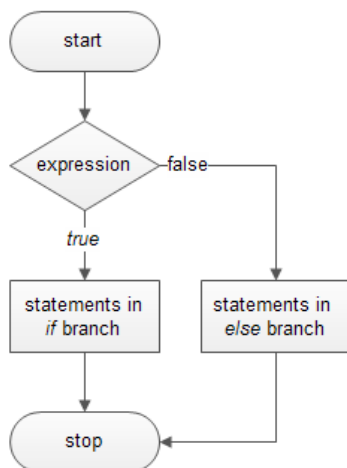
■ Precedência de Operadores

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 ...

Estruturas Condicionais

Seleção



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

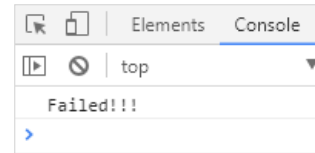
  if (teste == "verdadeiro")
    document.write("Condição Verdadeira!");
  else
    document.write("Condição Falsa!");
</script>
```

Seleção

▪ if ...else

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

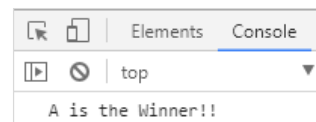
  if (grade>=threshold)
    {console.log("Aproved!!");}
  else
    {console.log("Failed!!");}
</script>
```



▪ 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>
```



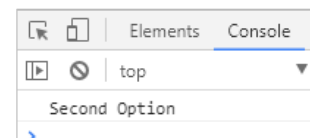
Seleção

▪ 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>
```

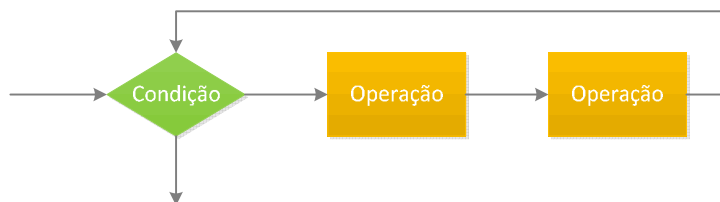


Estruturas de Repetição

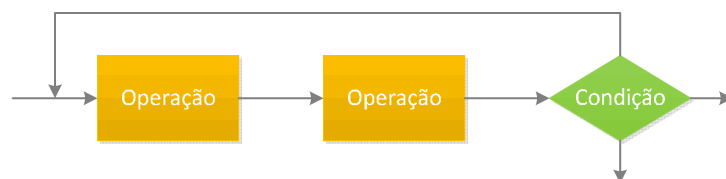
Repetição

■ Estruturas de Repetição / Ciclos (Loops)

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



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



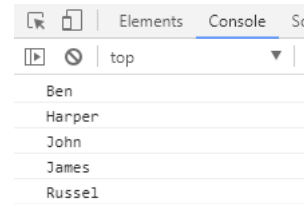
Repetição

■ Ciclos (Loops)

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

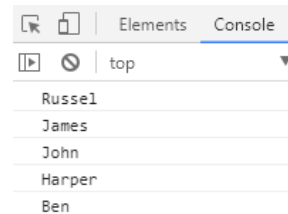
```
names=["Ben", "Harper","John", "James", "Russel"];

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]);
}
```

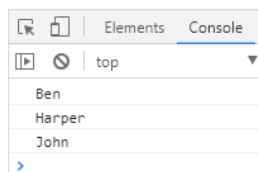


Repetição

■ Ciclos (Loops)

- **break**

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

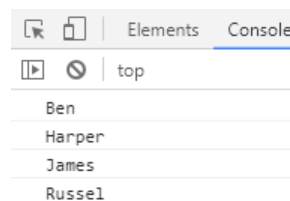


break: interrompe o funcionamento do ciclo

- **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

- valores tratados como *false*

Value	Description
var highScore = false;	valor booleano false
var highScore = 0;	número 0
var highScore = '';	<i>empty value</i>
var highScore = 10/'score';	<i>NaN (not a number)</i>
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	<i>resultado de um cálculo ≠ 0</i>
var highScore='true'; var highScore='0'; var highScore='false';	definidos como strings

J. Ducket, JavaScript and jQuery, 2014

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);
```

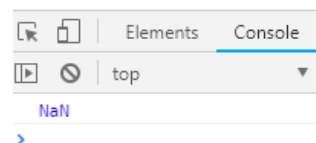
- Argumentos da função definidos através de variáveis

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

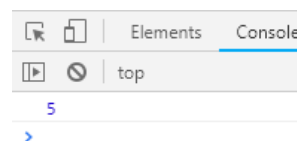
Funções

- 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

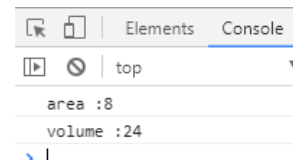
- Retorno de um valor único:



```
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*:

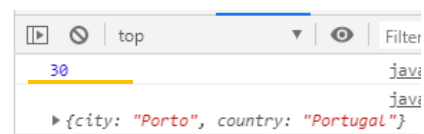
```
<script>  
  
    function getDimensions(w,h,d)  
    {    var area=w*h;  
        var volume=area*d;  
        var values=[area,volume];  
  
        return values;}  
  
    console.log("area : " + getDimensions(2,4,3)[0]);  
  
    console.log("volume : " + getDimensions(2,4,3)[1]);  
  
</script>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



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>
```



Funções

- O JavaScript admite formas diferentes de criar uma função:

Declaração de Função (statement)

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

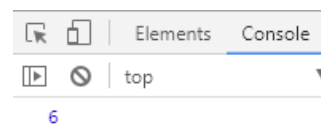
Function Expression (anonymous function)

```
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>
```



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)** (falar das arrows)
 - 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;}

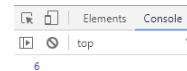
  var area=calculaArea(2,3);
  console.log(area);
</script>
```

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

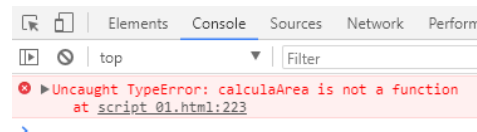
  var calculaArea=function(width,height){
    return width*height;}

  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**



- 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;
```