

P.PORTO

Introdução a Javascript

Programação em Ambiente Web

Índice

Visão geral

A utilidade do Javascript

Javascript

ECMAScript6

Boas Práticas

Visão Geral

Inicialmente implementada apenas no lado do cliente, em browsers da Web

Atualmente também utilizada do lado do servidor

É uma linguagem de programação interpretada

Comporta-se como linguagens de *scripting*, executa as operações pela ordem em que aparecem e não tem uma função do tipo *main*

Visão Geral

O JavaScript é linguagem multi-paradigma

Suporta os estilos de programação:

- orientados a eventos
- orientado a objetos
- funcional
- imperativo

Tem APIs para trabalhar com texto, matrizes, datas, expressões regulares

Visão Geral

JavaScript **não** é java!

- Java foi desenvolvido na Sun Microsystems
- JavaScript foi desenvolvido no Netscape

Os programas JavaScript são executados num runtime que pode estar:

- num browser
(SpiderMonkey, V8, Chakra)
- instalados como uma aplicação no servidor ou desktop
(NodeJS)

Visão geral

Uma linguagem do “lado do cliente”

- Utilizada para dar “interatividade” às páginas web
- Inserir texto dinâmico em HTML
- Reagir a eventos
- Não inclui nenhuma E / S, armazenamento ou gráficos, delegando estas ações no ambiente *host* no qual está incorporado
- Obtém informação sobre o computador do utilizador

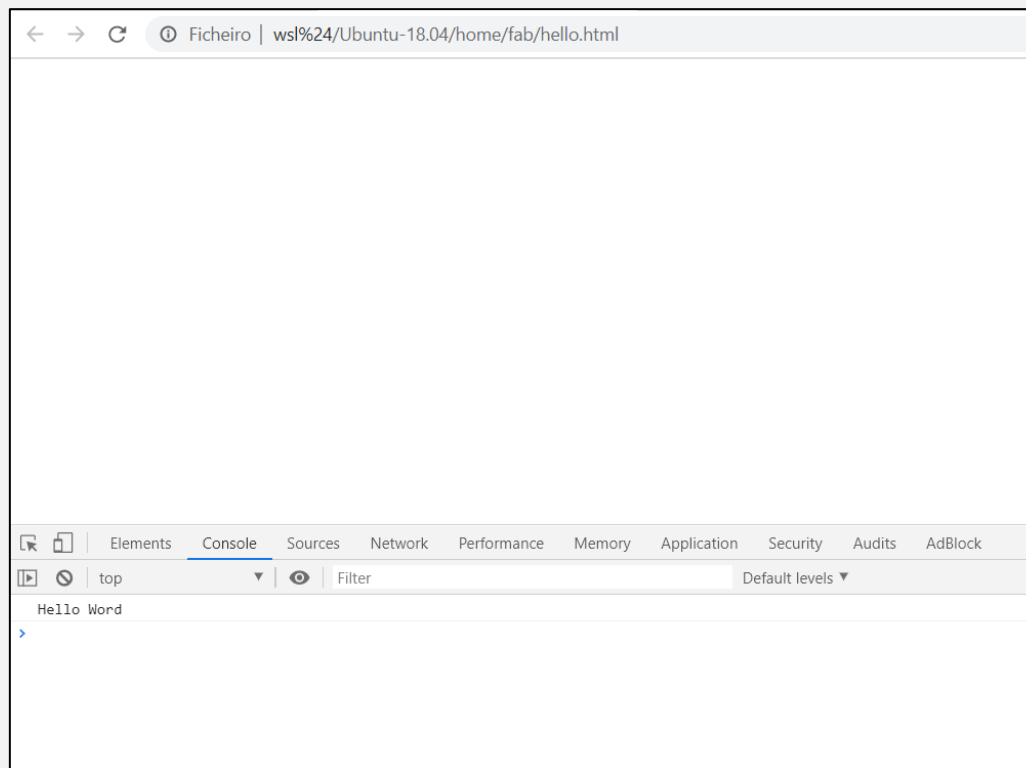
Visão Geral

Executa dentro de um ficheiro HTML dentro da tag script

```
hello.html > ...  
1  <!DOCTYPE html>  
2  <head></head>  
3  <body>  
4      <script>  
5          console.log("Hello Word");  
6      </script>  
7  </body>  
8  
9  
10
```

Visão Geral

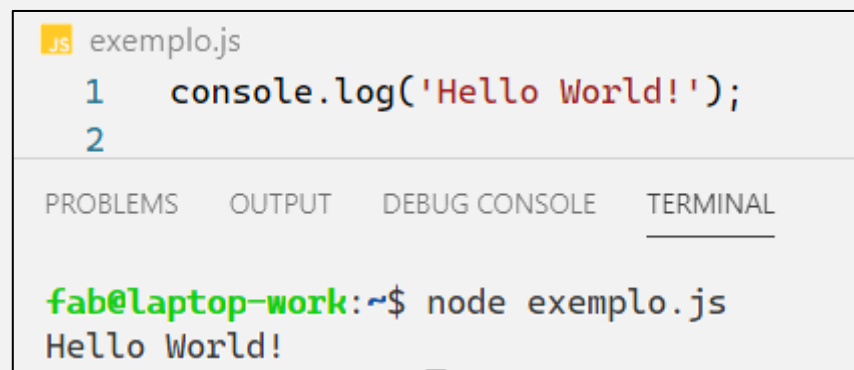
Executa dentro de um ficheiro HTML dentro da tag script



Visão geral

Uma linguagem do “lado do servidor”

- runtime NodeJS num servidor ou Desktop
- não tem acesso a objetos exclusivos do browser
- ficheiros com a extensão *.js e *.jsh
- executar o ficheiro num terminal com o comando:
 - node exemplo.js

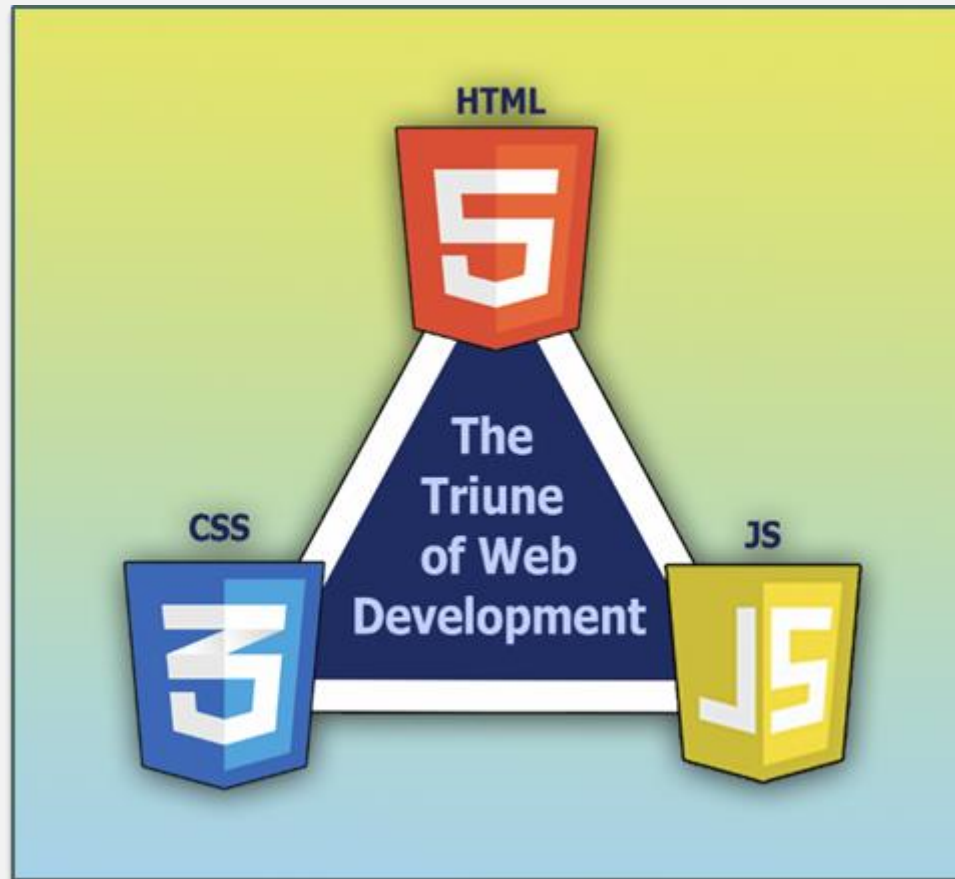


```
js exemplo.js
1 console.log('Hello World!');
2
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
fab@laptop-work:~$ node exemplo.js
Hello World!
```

A utilidade do JavaScript



A utilidade do JavaScript

Detetar e reagir a eventos iniciados pelo utilizador

Melhorar a navegação numa página web com ajudas, *scroll* automático, mensagens e alertas,, imagens dinâmicas, etc

Controlar a aparência da página de internet dinamicamente

A utilidade do JavaScript

Verificar se o utilizador tem plug-ins instalados

Validar o que o utilizador digitou num formulário antes de o enviar o ao servidor

Verificar endereços de e-mail válidos, números de segurança social, dados de cartão de crédito etc.

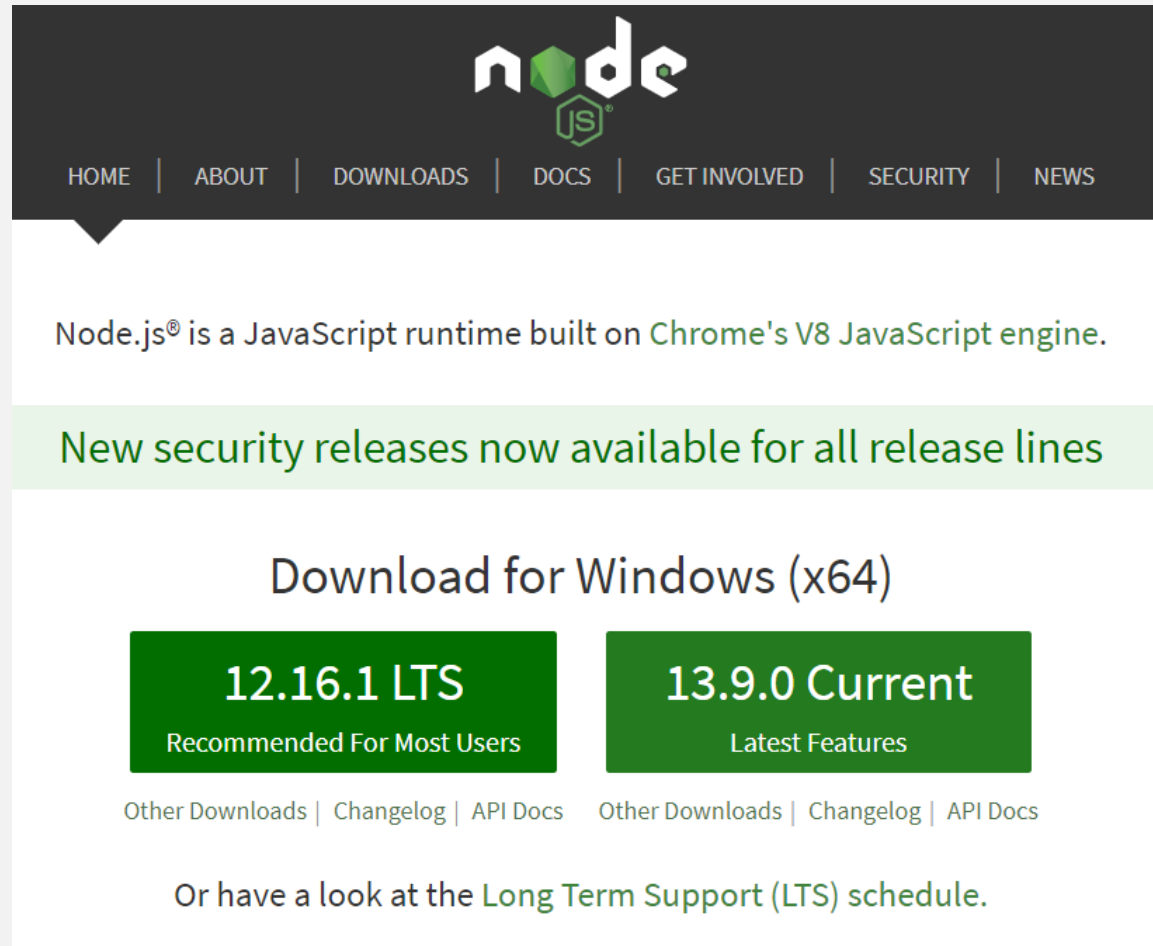
A utilidade do JavaScript

Cálculos aritméticos, manipulação de datas e horas e trabalha com matrizes, cadeias de caracteres e objetos.

Manipular animações e produzir aplicações e jogos baseados na web

Outros ...

A utilidade do JavaScript



The screenshot shows the Node.js website's download section for Windows (x64). At the top, the Node.js logo is centered, with a navigation bar below it containing links: HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, and NEWS. Below the navigation bar, a text block states: "Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine." A green banner below this text reads: "New security releases now available for all release lines". The main heading for the download section is "Download for Windows (x64)". Below this heading are two green buttons. The left button displays "12.16.1 LTS" and "Recommended For Most Users". The right button displays "13.9.0 Current" and "Latest Features". Below these buttons, a link "Other Downloads | Changelog | API Docs" is repeated twice. At the bottom, a text block says: "Or have a look at the Long Term Support (LTS) schedule."

node

HOME | ABOUT | DOWNLOADS | DOCS | GET INVOLVED | SECURITY | NEWS

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

New security releases now available for all release lines

Download for Windows (x64)

12.16.1 LTS
Recommended For Most Users

13.9.0 Current
Latest Features

Other Downloads | Changelog | API Docs Other Downloads | Changelog | API Docs

Or have a look at the Long Term Support (LTS) schedule.

A utilidade do JavaScript

Executar Scripts no Servidor/Desktop

Desenvolver aplicações web utilizando frameworks para a web

Interagir com o sistema operativo do dispositivo onde está instalado

Outros ...

JavaScript > Variáveis

As variáveis são criadas declarando-as. É lhes atribuído valores utilizando o operador de atribuição de JavaScript, '='

Atribuir um nome às variáveis em JavaScript tem de ter em conta as regras para atribuição de nomes bem como considerar a significância do nome.

Para declarar uma variável, deve usar a palavra-chave **var** ou **let**. Não se especifica o tipo de dados, é inferido pelo runtime javascript:

```
var variableName;
```


JavaScript > Variáveis

Para declarar a variável **numberOfOranges**:

```
var numberOfOranges;
```

Neste exemplo, há uma nova variável com o nome **numberOfOranges**. O ponto e vírgula termina a declaração. A variável **numberOfOranges** ainda não possui um valor atribuído

Variáveis JavaScript são case sensitive

numberOfOranges, **numeroforanges**, **NUMBERoFoRANGES** e **NumberOfOranges** são quatro variáveis diferentes.

JavaScript > Variáveis

Para atribuir um valor a uma variável, utilize o operador de atribuição do JavaScript, que é o símbolo igual a (=). Se pretender declarar uma variável e atribuir um valor a ela na mesma linha, use este formato:

```
var variableName = variávelValor;
```

Por exemplo, para nomear sua variável **numberOfOranges** e fornecer o valor numérico 18, utilize esta instrução:

```
var numberOfOranges = 18;
```

JavaScript > Variáveis

Pode declarar/atribuir múltiplas variáveis ao mesmo tempo

Para atribuir um valor a uma variável, use o operador de atribuição do JavaScript '='

Sem valores atribuídos:

```
var variableName1, variableName2, variableName3;
```

Com valores atribuídos:

```
var variableName1 = 10;  
var variableName2 = "doc", variableName3 = 3.14159;
```

JavaScript > Variáveis

Regras importante a ressaltar:

- um nome de variável deve começar com uma **letra** ou um caractere **'_'** ou um caractere **'\$'**. O nome da variável não pode começar com um número ou qualquer outro caractere que não seja uma letra (além do sublinhado).
- Palavras reservadas:

abstract	arguments	await	boolean
break	byte	case	catch
char	class	const	continue
debugger	default	delete	do
double	else	enum	eval
export	extends	false	final
finally	float	for	function
goto	if	implements	import
in	instanceof	int	interface
let	long	native	new
null	package	private	protected
public	return	short	static
super	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

JavaScript > Tipos de dados

Números

O JavaScript não exige que os números sejam declarados como números inteiros, números de virgula flutuante (decimal) ou qualquer outro tipo

O número permanecerá do mesmo tipo, até que a operação matemática force o javascript a a menos que você faça uma operação que force a alteração do tipo de número a considerar

Por exemplo, se usarmos um número inteiro numa variável, ela não terá casas decimais, a não ser que se execute uma operação que resulte num número decimal (ex: dividindo-se de forma desigual).

JavaScript > Tipos de dados

String

As *strings* são variáveis que representam uma cadeia de texto. A *string* pode conter letras, palavras, espaços, números, símbolos, etc.

As *strings* são definidas da seguinte forma:

```
var variableName = "stringText";
```

Em JavaScript, as *strings* definem-se colocando-as entre aspas

JavaScript permite que você use aspas duplas “ ” ou aspas simples ‘ ’ para definir o valor da string.

JavaScript > Tipos de dados

- **String**

- **Métodos:** `charAt`, `charCodeAt`, `fromCharCode`, `indexOf`, `lastIndexOf`, `replace`, `split`, `substring`, `toLowerCase`, `toUpperCase`

```
var s = "Connie Client";  
var fName = s.substring(0, s.indexOf(" ")); // "Connie"  
var len = s.length; // 13
```

JavaScript > Tipos de dados

Array

As matrizes são utilizadas para armazenar conjuntos de valores numa única variável

Podemos aceder aos valores através do índice

Criar um array:

Sintaxe

```
var array_name = [item1, item2, ...];  
array_name[0] = new_item;
```

ou

```
var array_name = new Array(item1, item2, ...);  
array_name[0] = new_item;
```


JavaScript > Tipos de dados

Array

As matrizes são um tipo especial de objetos. O operador **typeof** em JavaScript retorna "objeto" para matrizes.

As matrizes utilizam números para aceder aos seus "elementos". Neste exemplo, `pessoa[0]` retorna John:

```
var person = ["John", "Doe", 46];
```

Os objetos utilizam nomes para aceder aos seus "membros". Neste exemplo, `person.firstName` ou `person['firstName']` retornam John

```
var person = {firstName:"John", lastName:"Doe", age:46};
```

JavaScript > Tipos de dados

Array

Métodos: concat, join, pop, push, reverse, shift, slice, sort, splice, toString, unshift

```
var a = ["Stef", "Jason"]; // Stef, Jason
a.push("Brian"); // Stef, Jason, Brian
a.unshift("Kelly"); // Kelly, Stef, Jason, Brian
a.pop(); // Kelly, Stef, Jason
a.shift(); // Stef, Jason
a.sort(); // Jason, Stef
```

JavaScript > Tipos de dados

Booleanos

Uma variável booleana é aquela que possui o valor de **true** ou **false**.

```
var RexCodes = true;  
  
var RexIsNotCool = false;
```

As palavras **true** e **false** não precisam ser colocadas entre aspas.

O JavaScript reconhece como valores booleanos.

Também podemos utilizar o número **1** para verdadeiro e o número **0** para falso

JavaScript > Tipos de dados

NULL

Nulo significa que a variável não tem valor. Não é um **espaço**, nem é um **zero**; é simplesmente **nada**.

Para definir uma variável com um valor null:

```
var variableName = null;
```

Assim como as variáveis booleanas, não é necessário colocar esse valor entre aspas

JavaScript > Tipos de dados

Undefined

Uma variável que ainda não recebeu um valor

```
var myVar;  
console.log(myVar); // undefined  
  
myVar = 200;  
console.log(myVar); // 200
```

JavaScript > Comparações

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
&&	logical and
	logical or
!	logical not

JavaScript > Atribuições

Há todo um conjunto de operadores de atribuição.

Diferente do sinal de igual, os outros operadores de atribuição servem como atalhos para modificar o valor das variáveis.

Por exemplo, uma maneira mais curta de dizer `x = x + 5` é dizer `x += 5`.

Operator	Example	Same As
=	<code>x = y</code>	<code>x = y</code>
+=	<code>x += y</code>	<code>x = x + y</code>
-=	<code>x -= y</code>	<code>x = x - y</code>
*=	<code>x *= y</code>	<code>x = x * y</code>
/=	<code>x /= y</code>	<code>x = x / y</code>
%=	<code>x %= y</code>	<code>x = x % y</code>
<<=	<code>x <<= y</code>	<code>x = x << y</code>
>>=	<code>x >>= y</code>	<code>x = x >> y</code>
>>>=	<code>x >>>= y</code>	<code>x = x >>> y</code>
&=	<code>x &= y</code>	<code>x = x & y</code>
^=	<code>x ^= y</code>	<code>x = x ^ y</code>
=	<code>x = y</code>	<code>x = x y</code>
**=	<code>x **= y</code>	<code>x = x ** y</code>

JavaScript > Condicionais

Código condicional: executa o código se algo for verdadeiro

```
if (condicional) {...} else {...}
```

Uma outra maneira de “afirmar o mesmo” é o seguinte:

```
(condicional) ? {...} : {...}
```


JavaScript > Ciclos

Ciclo For

```
var sum = 0;
for (var i = 0; i < 100; i++) {
    sum = sum + i;
}
```

```
var s1 = "hello";
var s2 = "";
for (var i = 0; i < s1.length; i++) {
    s2 += s1.charAt(i) + s1.charAt(i);
}
```

JavaScript > Ciclos

- **Ciclo While**

```
while (condição) {  
    statements;  
}
```

```
do {  
    statements;  
} while (condição);
```

JavaScript > Comentários

Como acontece em todas as linguagens de programação os comentários são uma parte importante do processo de programação.

Comentários numa linha:

```
// Isto é um comentário numa linha  
console.log("Hello"); // comentário depois do código
```

Comentários em bloco (multi-linha):

```
/*  comentário em bloco  
    e em várias linhas */
```

JavaScript > Espaços e newline

White spaces e Newlines

A maioria dos espaços em branco, como espaços, tabs e linhas vazias, são ignorados em JavaScript e geralmente apenas ajuda na legibilidade.

Em código de produção, todos os espaços em branco não essenciais são geralmente removidos para que os ficheiros de javascript sejam descarregados mais rapidamente.

JavaScript > objeto Math

O objeto Math

Metódos: `abs`, `ceil`, `cos`, `floor`, `log`, `max`, `min`, `pow`,
`random`, `round`, `sin`, `sqrt`, `tan`

Propriedadas: `E`, `PI`

```
var rand1to10 = Math.floor(Math.random() * 10 + 1);  
var three = Math.floor(Math.PI);
```

JavaScript > Hoisting

“Hoisting” é o comportamento por omissão do JavaScript que “move” a declaração de variáveis para o topo

Mesmo que uma variável seja “hoisted”, a sua inicialização não será

Em JavaScript, uma variável pode ser utilizada antes de ter sido declarada

```
function letTest(){  
  let x = 31  
  if (true){  
    let x = 71 // different variable  
    console.log(x) // 71  
  }  
  console.log(x) // 31  
}
```

JavaScript > Hoisting

De modo a evitarmos erros, devemos declarar todas as variáveis antes do início de cada “scope”

- O âmbito (“scope”) de uma variável declarada com “var” é o da função, caso seja declarada fora de qualquer função é global
- Com **ECMAScript 6**, o âmbito (“scope”) de uma variável declarada com “let” é ao nível do “bloco” dentro de uma função

```
function letTest(){  
  let x = 31  
  if (true){  
    let x = 71 // different variable  
    console.log(x) // 71  
  }  
  console.log(x) // 31  
}
```

JavaScript > Funções

Uma função consiste na palavra **function** seguida do nome da função, os argumentos e o corpo da função.

Exemplo:

```
function sayWhat() {  
    alert("JavaScript is good!");  
}
```


JavaScript > Funções

Na definição de uma função em JavaScript não se especifica o tipo de dados dos parâmetros

Uma função em JS não realiza a verificação de tipos (“type checking”) nos argumentos que lhe são passados

Uma função em JS não verifica o número de argumentos recebidos é o esperado

JavaScript > Funções

Se uma função é invocada com falta de argumentos (menos dos que declarados), os valores em falta são definidos como “undefined”

Se uma função é invocada com argumentos a mais (mais do que aqueles que foram declarados), estes argumentos podem ser acedidos utilizando o objeto dos argumentos

Funções em JS são “métodos de um objeto”

JavaScript > Funções

- Em JS argumentos de tipos primitivos são passados a funções por valor
- Os objetos são passados a funções por referência
- Há duas maneiras de declarar uma função em JS:
 - Por “declaration” (alerta: é “hoisted”, ou seja, “puxada para cima”)

```
//Function Declaration  
function square(x) {  
    return x*x;  
}
```

- Por “expression”

```
//Function Expression  
const square = function (x) {  
    return x*x;  
}
```

JavaScript > Funções

Alerta: o JavaScript é case sensitive!

- Ao declarar uma variável ou função, temos que ter atenção às suas letras maiúsculas e minúsculas. **myFunction** não é a mesma coisa que **MyFunction** ou **myFUNCTION** ou **myfunction**
- Devemos aceder objetos internos com o encapsulamento apropriado. **Math** e **Data** começam com letras maiúsculas, mas não com **window** e **document** em browsers não
- A maioria dos métodos internos são palavras combinadas, capitalizando todas, exceto a primeira. Exemplo: como **getElementById** (geralmente chamado de camelCase)

JavaScript > Erros

A instrução **try** permite testar um bloco de código para erros.

A instrução **catch** permite lidar com o erro.

A instrução **throw** permite criar erros personalizados.

A instrução **finally** permite executar o código, depois de tentar e capturar, independentemente do resultado.

```
try{  
    // Code  
}  
catch(err){  
    // Code  
}
```

```
try{  
    // Code  
}  
catch(err){  
    // Code  
}  
finally{  
    // Code  
}
```

JavaScript > Erros

Exemplo

```
function myFunction(x,message) {|
  try {
    if(x == "") throw "is empty";
    if(isNaN(x)) throw "is not a number";
    x = Number(x);
    if(x > 10) throw "is too high";
    if(x < 5) throw "is too low";
  }
  catch(err) {
    message.innerHTML = "Input " + err;
  }
  finally {
    document.getElementById("demo").value = "";
  }
}
```

JavaScript > Scope

Existem dois tipos de scope :

- scope local
- scope global

Cada função tem um scope próprio

O scope determina a acessibilidade (visibilidade) das variáveis.

Variáveis definidas dentro de uma função não são acessíveis (visíveis) de fora da função.

JavaScript > Scope

- Variáveis declaradas dentro de uma função JavaScript, tornam-se locais para a função.
- Uma variável declarada fora de uma função, torna-se GLOBAL.

```
// code here can NOT use carName

function myFunction() {
  var carName = "Volvo";

  // code here CAN use carName
}
```

```
var carName = "Volvo";

// code here can use carName

function myFunction() {

  // code here can also use carName
}
```


JavaScript > Strict Mode

Podemos executar javascript em strict mode

Tem como objetivo forçar a declaração e atribuição de valores a variáveis

Caso não seja cumprida a restrição é lançado um erro

```
"use strict";  
x = 3.14; // This will cause an error because x is not declared
```

```
x = 3.14; // This will not cause an error.  
myFunction();  
  
function myFunction() {  
  "use strict";  
  y = 3.14; // This will cause an error  
}
```

JavaScript > Objetos

Quem compreender o conceito de “objeto”, compreende JavaScript!

Em JS, quase tudo é um objeto

Booleanos, Números, Strings podem ser objetos (se definidos através da palavra reservada “new”)

Datas e Maths são sempre objetos

Arrays e funções são sempre objetos

Todos os valores, excepto os primitivos, são objetos

Um valor primitivo é um valor que não possui propriedades e métodos

String, Number, boolean, null, undefined

Objetos são **variáveis que contêm variáveis**

JavaScript > Objetos

As variáveis podem conter valores simples:

```
var person = "John Doe";
```

Os objetos também são variáveis, mas contêm valores como pares **chave: valor**

Um método de um objeto é uma propriedade do objeto contendo uma definição de função:

```
let car = {  
  brand: 'Smart',  
  model: 'forTwo',  
  year: 2013,  
  engine: {  
    horsepower: 45,  
    fuel: "diesel"  
  },  
  getAge: function() {  
    return (new Date()).getFullYear() - this.year;  
  }  
}
```

JavaScript > Objetos

Criação de um objeto:

- Utilizando um objeto literal

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

- Utilizando a palavra reservada “new”

```
var person = new Object();  
person.firstName = "John";  
person.lastName = "Doe";  
person.age = 50;  
person.eyeColor = "blue";
```

- Os objetos são **mutáveis**
 - os objetos são tratados por referência, não por valor

JavaScript > Objetos

As **propriedades** são a parte mais importante de qualquer objeto

As **propriedades** são os valores associados com o objeto

Um objeto em javascript pode ser visto como uma **coleção não ordenada de propriedades**

Acesso às propriedades de um objeto:

```
person.firstname + " is " + person.age + " years old.";
person["firstname"] + " is " + person["age"] + " years old.";
```

JavaScript > Objetos

- Utilizando um ciclo “for”

```
var person = {fname:"John", lname:"Doe", age:25};

for (x in person) {
  txt += person[x];
}
```

- Construtores de objetos

```
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}
```

```
var myFather = new Person("John", "Doe", 50, "blue");
var myMother = new Person("Sally", "Rally", 48,
"green");
```

JavaScript > Objetos Prototype

Todos os objetos “herdam” propriedades e métodos de um “prototype”

Anteriormente, foi mostrado como se utiliza um construtor de um objeto e **não podemos adicionar uma nova** propriedade a um construtor já existente

Exemplo: objetos “Date” herda de “Date.prototype”, objetos “Array” herda de “Array.prototype”, objeto “Person” herda de “Person.prototype”

Porém, a propriedade “prototype” javascript permite adicionar novas propriedades a um construtor de um objeto

```
function Person(first, last, age, eyecolor) {  
  this.firstName = first;  
  this.lastName = last;  
  this.age = age;  
  this.eyeColor = eyecolor;  
}  
Person.prototype.nationality = "English";
```

```
function Person(first, last, age, eyecolor) {  
  this.firstName = first;  
  this.lastName = last;  
  this.age = age;  
  this.eyeColor = eyecolor;  
}  
Person.prototype.name = function() {  
  return this.firstName + " " + this.lastName;  
};
```

JavaScript > Classes

Classes em javascript são essencialmente “açúcar sintático” sobre o conceito de herança baseada em “prototype”

Uma importante diferença entre a declaração de funções e a declaração de classes é que as declarações de funções são “hoisted” e as da classe não

É necessário primeiro declarar a classe e só depois acedê-la, caso contrário o código irá despoletar um “ReferenceError”

```
var car1 = new Car(); // ReferenceError  
  
class Car {}
```


JavaScript > Classes

Tal como nas funções, uma classe também pode ser definida através de uma expressão

Uma “class expression” é uma outra maneira de definir uma classe que podem ter um nome ou não

O nome dado uma “class expression” é local ao corpo da classe.

```
//or (unnamed)
var Moto = class {
  constructor(brand, model, year){
    this.brand = brand;
    this.model = model;
    this.year = year;
  }
};

//or (named)
var bicycle = class bicycle{
  constructor(brand, model, year){
    this.brand = brand;
    this.model = model;
    this.year = year;
  }
}
```

JavaScript > Classes

O método construtor é um método especial para criar e inicializar um objeto criado com a classe

Um construtor pode utilizar a palavra reservada “super” para evocar o construtor de uma classe pai

```
class Car {  
  constructor(brand, model, year){  
    this.brand = brand;  
    this.model = model;  
    this.year = year;  
  }  
  
  //Getter  
  get age(){  
    return this.calcAge()  
  }  
  
  calcAge(){  
    return (new Date()).getFullYear() - this.year;  
  }  
}  
  
const car1 = new Car('Smart', 'forTwo', 2013);  
document.write('The car: ${car1.year} has ${car1.age} years old <br />');
```

JavaScript > Classes

A palavra “static” define o método estático para uma classe

Os métodos estáticos são invocados sem serem instanciados pela sua classe e não podem ser invocados por uma instância da sua classe

Os métodos estáticos são habitualmente utilizados para criar funções utilitárias para uma aplicação

A palavra “extend” é utilizada nas declarações ou expressões de classe para criar uma classe como “filho” de outra classe através de herança

JavaScript > Debugger

A palavra debugger interrompe a execução do JavaScript e executa (se disponível) a função de depuração.

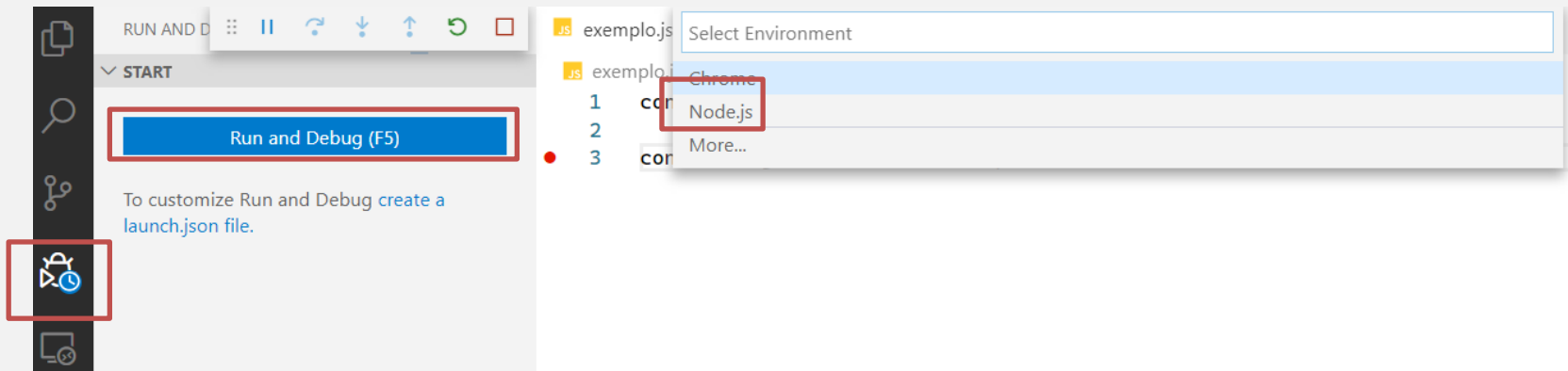
Tem a mesma função que definir um ponto de interrupção no debugger.

Se nenhum breakpoint estiver disponível, a instrução do debugger não terá efeito.

```
var x = 15 * 5;  
debugger;  
document.getElementById("demo").innerHTML = x;
```

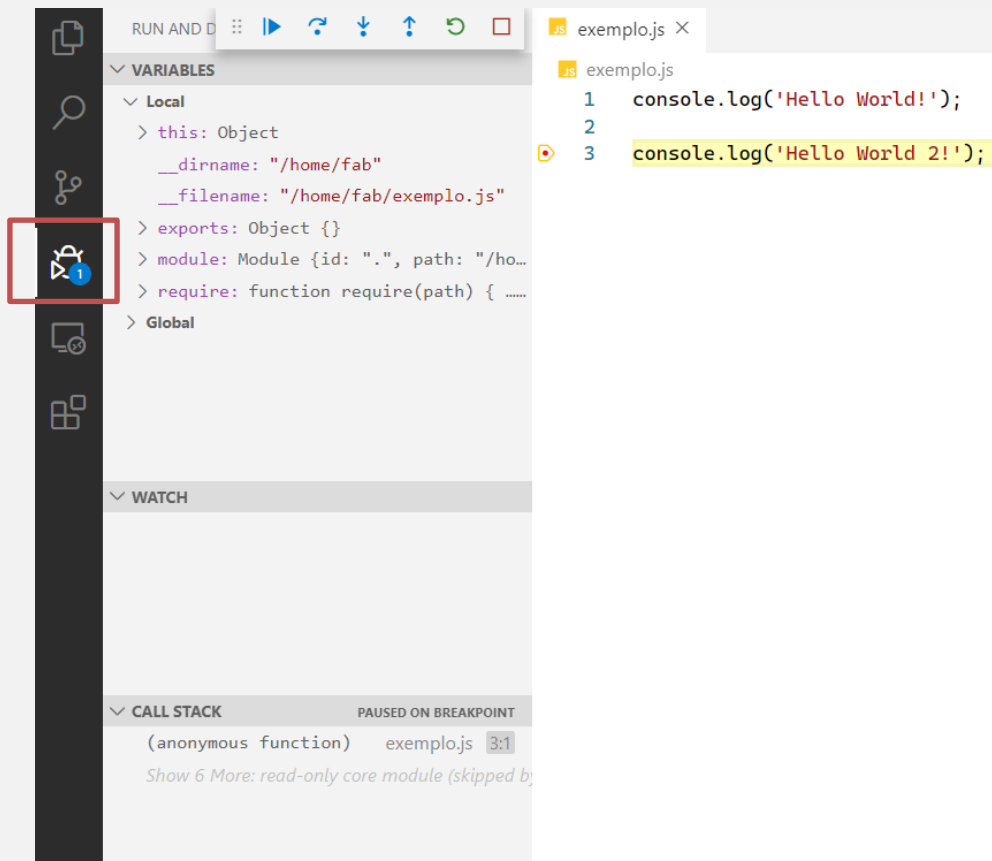
JavaScript > Debbuger

Utilizando o runtime NodeJS e VSCode



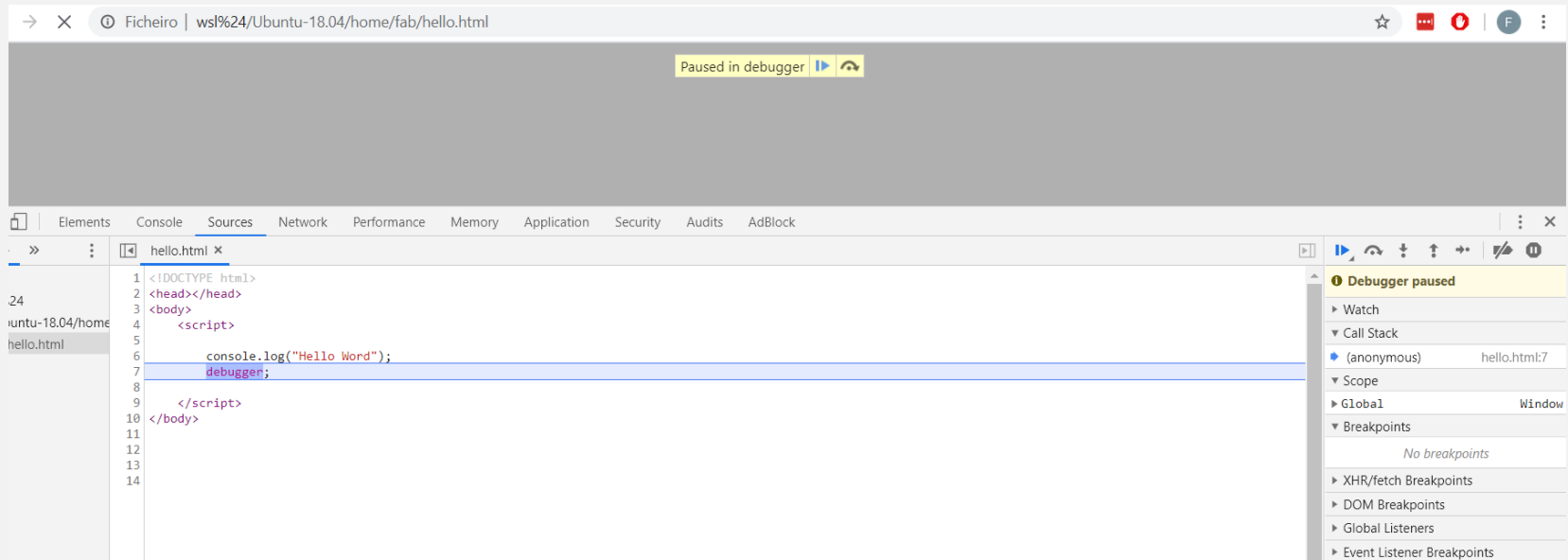
JavaScript > Debbuger

Utilizando o runtime NodeJS e VSCode



JavaScript > Debugger

Utilizando o runtime javascript no browser



ECMAScript6

A linguagem JavaScript contém atualmente várias versões incrementais

A versão ECMAScript 6 adiciona novas funcionalidades à linguagem Javascript, mas não quebra o suporte a versões anteriores

Os browsers modernos já contém suporte para JavaScript ECMAScript 6 mas as versões antigas não

ECMAScript6

Suporte para constantes (também conhecidas como "variáveis imutáveis")

```
const PI = 3.141593
```

Varáveis block-scoped sem hoisting (com recurso ao "let")

```
for (let i = 0; i < a.length; i++) {  
  let x = a[i]  
  ...  
}  
for (let i = 0; i < b.length; i++) {  
  let y = b[i]  
  ...  
}
```

```
var x = 10;  
// Here x is 10  
{  
  let x = 2;  
  // Here x is 2  
}  
// Here x is 10
```

ECMAScript6

Arrow functions

```
// ES6  
const f = (x, y) => x * y;  
  
// ES5  
var f = function(x, y) {  
    return x * y;  
}
```

Parâmetros por defeito em funções

```
function f (x, y = 7, z = 42) {  
    return x + y + z  
}  
f(1) === 50
```

ECMAScript6

Template Strings

```
// ES6
var customer = { name: "Foo" }
var card = { amount: 7, product: "Bar", unitprice: 42 }
var message = `Hello ${customer.name},
want to buy ${card.amount} ${card.product} for
a total of ${card.amount * card.unitprice} bucks?`

// ES5
var customer = { name: "Foo" };
var card = { amount: 7, product: "Bar", unitprice: 42 };
var message = "Hello " + customer.name + ",\n" +
"want to buy " + card.amount + " " + card.product + " for\n" +
"a total of " + (card.amount * card.unitprice) + " bucks?";
```

Declaração de objetos melhorada

```
// ES6
var x = 0, y = 0
obj = { x, y }

//ES5
var x = 0, y = 0;
obj = { x: x, y: y };
```

ECMAScript6

Suporte a Modules

```
// lib/math.js
export function sum (x, y) { return x + y }
export var pi = 3.141593

// someApp.js
import * as math from "lib/math"
console.log("2π = " + math.sum(math.pi, math.pi))

// otherApp.js
import { sum, pi } from "lib/math"
console.log("2π = " + sum(pi, pi))
```

POO mais intuitiva

```
class Shape {
  constructor (id, x, y) {
    this.id = id
    this.move(x, y)
  }
  move (x, y) {
    this.x = x
    this.y = y
  }
}
```

```
class Rectangle extends Shape {
  constructor (id, x, y, width, height) {
    super(id, x, y)
    this.width = width
    this.height = height
  }
}
class Circle extends Shape {
  constructor (id, x, y, radius) {
    super(id, x, y)
    this.radius = radius
  }
}
```

ECMAScript6

- POO mais intuitiva (continuação)

```
class Rectangle extends Shape {  
  ...  
  static defaultRectangle () {  
    return new Rectangle("default", 0, 0, 100, 100)  
  }  
}  
class Circle extends Shape {  
  ...  
  static defaultCircle () {  
    return new Circle("default", 0, 0, 100)  
  }  
}  
var defRectangle = Rectangle.defaultRectangle()  
var defCircle    = Circle.defaultCircle()
```

```
class Rectangle {  
  constructor (width, height) {  
    this._width  = width  
    this._height = height  
  }  
  set width  (width)  { this._width = width }  
  get width  ()       { return this._width }  
  set height (height) { this._height = height }  
  get height ()       { return this._height }  
  get area   ()       { return this._width * this._height }  
}  
var r = new Rectangle(50, 20)  
r.area === 1000
```

ECMAScript6

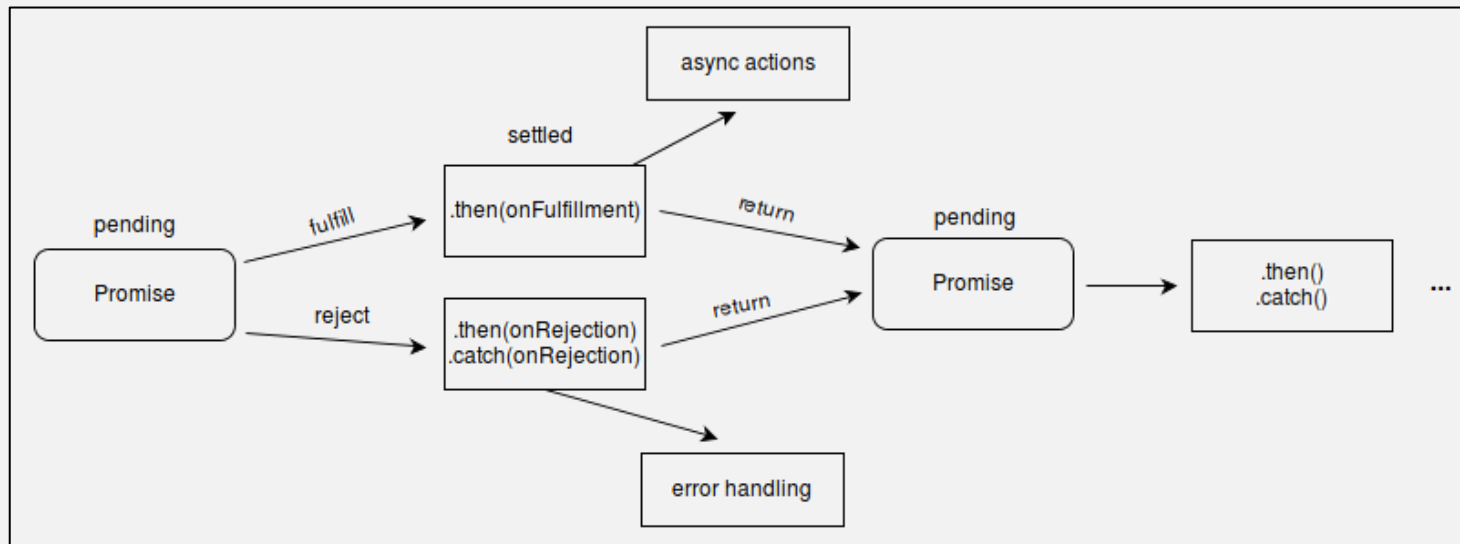
Promises

- Um Promise representa um proxy para um valor que não é necessariamente conhecido quando a promessa é criada
- Permite a associação de métodos para tratamento de eventos assíncronos em caso de sucesso e de falha
- Permite que métodos assíncronos retornem valores como métodos síncronos

ECMAScript6

Um Promise está em um destes estados:

- pending (pendente): Estado inicial, que não foi realizada nem rejeitada
- fulfilled (realizada): sucesso na operação
- rejected (rejeitado): falha na operação
- settled (estabelecida): Que foi realizada ou rejeitada



ECMAScript6

Exemplo

```
var promise = new Promise(function(resolve, reject) {  
  // do a thing, possibly async, then...  
  
  if (/* everything turned out fine */) {  
    resolve("Stuff worked!");  
  }  
  else {  
    reject(Error("It broke"));  
  }  
});  
promise.then(function(result) {  
  console.log(result); // "Stuff worked!"  
}, function(err) {  
  console.log(err); // Error: "It broke"  
});
```


Boas Práticas

Evitar variáveis globais

Declarar sempre variáveis locais

Declarações de variáveis no topo

Inicializar variáveis

```
// Declare at the beginning
var firstName, lastName, price, discount, fullPrice;

// Use later
firstName = "John";
lastName = "Doe";

price = 19.90;
discount = 0.10;

fullPrice = price * 100 / discount;
```

```
// Declare and initiate at the beginning
var firstName = "",
    lastName = "",
    price = 0,
    discount = 0,
    fullPrice = 0,
    myArray = [],
    myObject = {};
```

Boas Práticas

Não declarar objectos do tipo String, Boolean, Number, etc..

```
var x = "John";  
var y = new String("John");  
(x === y) // is false because x is a string and y is an object.
```

```
var x = new String("John");  
var y = new String("John");  
(x == y) // is false because you cannot compare objects.
```

Ter cuidado com a conversão automática de variáveis

Boas Práticas

Não usar new Object()

- Usar {} em vez de new Object()
- Usar "" em vez de new String()
- Usar 0 em vez de new Number()
- Usar false em vez de new Boolean()
- Usar [] em vez de new Array()
- Usar /(())/ em vez de new RegExp()
- Usar function (){} em vez de new Function()

```
var x1 = {};  
var x2 = "";  
var x3 = 0;  
var x4 = false;  
var x5 = [];  
var x6 = /(())/;  
var x7 = function(){};
```

Boas Práticas

Usar parâmetros por defeito em funções javascript

```
function myFunction(x, y) {  
  if (y === undefined) {  
    y = 0;  
  }  
  return x * y;  
}  
document.getElementById("demo").innerHTML = myFunction(4);
```

Referências

Tutorial Javascript

- <https://www.codecademy.com/learn/introduction-to-javascript>
- <https://www.tutorialspoint.com/es6/index.htm>
- [https://developer.mozilla.org/en-US/docs/Learn/Getting started with the web/JavaScript basics](https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/JavaScript_basics)

Especificação da linguagem Javascript

- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>

P.PORTO

Introdução a Javascript

Programação em Ambiente Web