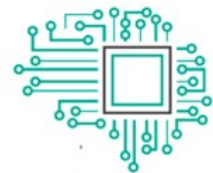


JAVASCRIPT

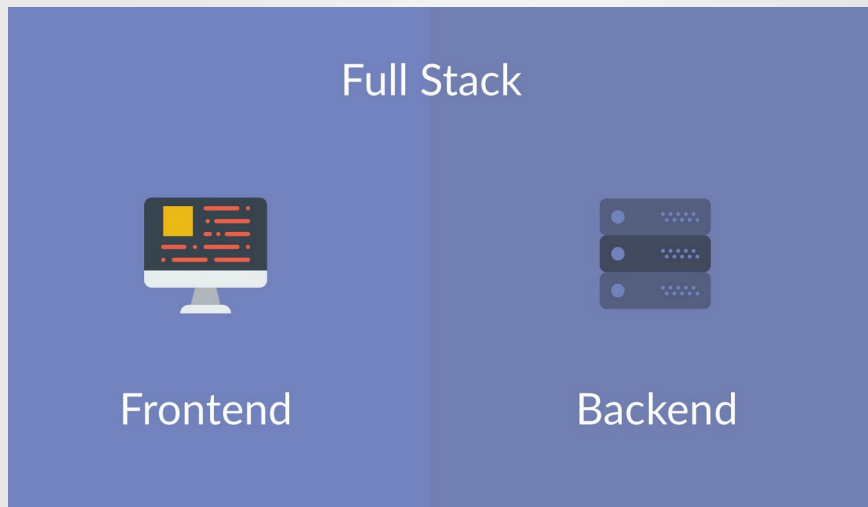
Pedro Moritz de Carvalho Neto



JCAVI

Observações

- Os conceitos deste curso serão abordados de maneira não-linear.
- JavaScript pode ser executado tanto no frontend quando no backend.




Observações

- Você pode utilizar qualquer IDE (Integrated Development Environment) durante o curso. Exemplos:
 - <https://www.sublimetext.com/>
 - <https://code.visualstudio.com/>
 - <https://atom.io/>
 - <https://www.jetbrains.com/webstorm/>
 - <https://codepen.io/pen>
 - <https://jsfiddle.net/>



Conteúdo Programático

- **Introdução ao JavaScript**
 - **Variáveis**
 - **Escopo**
 - **Funções**
 - **Closures**
 - **Tipos**
 - **Tipagem dinâmica**
 - **Operador typeof**
 - **Operadores aritméticos, lógicos e de comparação**
 - **Operador ternário**
 - **Arrays**
 - **Objetos**
 - **Eventos**
 - **Temporizadores**
 - **Ajax**
 - **DOM**
 - **Prototype**
- 



Introdução ao JavaScript

Introdução ao JavaScript

- Nasceu em 1995 com o nome de “Mocha” no Netscape Navigator.
- Em seguida, a linguagem foi renomeada para “LiveScript” e logo depois para “JavaScript”.
- O JavaScript não tem relação nenhuma com o Java, mas este último era uma linguagem extremamente promissora na época.



Introdução ao JavaScript

- A Microsoft fez uma engenharia reversa do JavaScript e criou sua própria implementação, denominada “JScript”, lançada em 1996.
- O JScript se comportava de maneira muito diferente do JavaScript, causando inúmeros problemas de compatibilidade.



Introdução ao JavaScript

- Em 1996 a Netscape submeteu o JavaScript ao ECMA (European Computer Manufacturers Association) para que fosse criado um padrão a ser seguido por todos os fabricantes de browsers.
- Em 1997 foi criado o padrão ECMA-262, que define a linguagem denominada “ECMAScript”.
- Sua atual versão é o ECMAScript 2019 (ou apenas ES2019):
<https://www.ecma-international.org/publications/standards/Ecma-262.htm>



Introdução ao JavaScript

- Algumas linguagens que seguem o padrão ECMAScript:
 - JavaScript
 - Jscript
 - ActionScript
 - QtScript
 - Google Apps Script



Introdução ao JavaScript

- Cada browser possui seu próprio interpretador de ECMAScript:
 - Chrome: v8
 - Firefox: SpiderMonkey
 - Safari: JavaScriptCore
 - IE e Edge: Chakra
- Existem implementações de interpretador de ECMAScript para o backend:
 - Node.js



Introdução ao JavaScript

- Existem muitas bibliotecas e plataformas que utilizam o ECMAScript:
 - jQuery
 - Bootstrap
 - AngularJS
 - React.js
 - Vue.js
 - Ember.js
 - ...



Introdução ao JavaScript

- Muitos produtos e tecnologias utilizam internamente a linguagem JavaScript:
 - MongoDB
 - Apache CouchDB
 - JSON
 - ...



Variáveis



Variáveis

- Variáveis são nomes simbólicos que referenciam valores
- Para se declarar uma variável utilizamos o comando **let**
- Para se atribuir um valor à uma variável utilizamos o operador **=**
- Variáveis no JavaScript podem ou não serem declaradas para serem usadas



Variáveis

- Regras para criação de nomes de variáveis no JavaScript:
 - Devem começar com uma letra, sublinhado (_), ou cifrão (\$)
 - Os caracteres subsequentes podem também ser dígitos (0-9)
 - As letras podem ser minúsculas ou maiúsculas
 - O nome das variáveis, assim como em todos os elementos no JavaScript, são “case-sensitive”

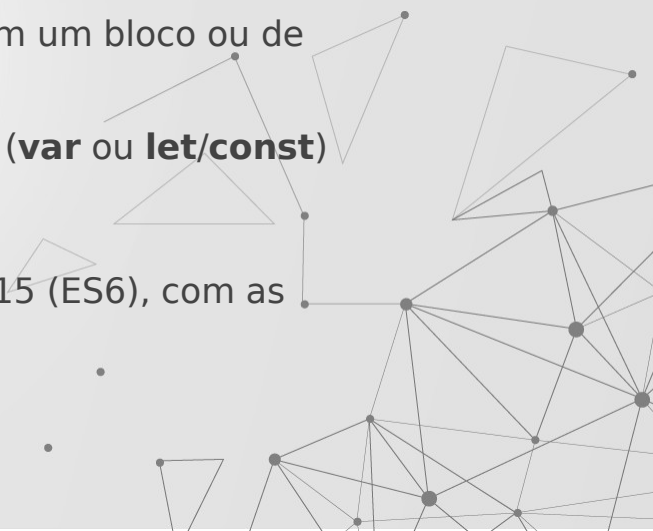


Escopo



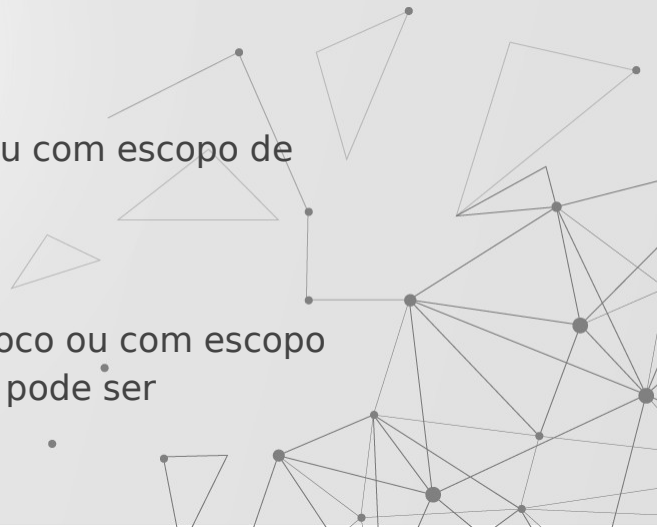
Escopo

- O escopo de uma variável é o espaço a partir do qual esta variável pode ser acessada
- O escopo de uma variável é definido por dois fatores:
 - pelo local onde ela é declarada (em uma função, em um bloco ou de maneira global)
 - pela instrução utilizada para declarar esta variável (**var** ou **let/const**)
- O conceito de escopo de bloco surgiu na ECMAScript 2015 (ES6), com as instruções **let** e **const**



Escopo

- Nas declarações de variável em nível de função, o escopo fica restrito à função
- A instrução **var** declara variáveis com escopo global ou com escopo de função
- A instrução **let** declara variáveis com escopo de bloco ou com escopo de função
- A instrução **const** declara constantes com escopo de bloco ou com escopo de função (não podem ser redeclaradas e seu valor não pode ser modificado)



Escopo

keyword	const	let	var
global scope	NO	NO	YES
function scope	YES	YES	YES
block scope	YES	YES	NO
can be reassigned	NO	YES	YES

<http://www.constletvar.com>

Funções



Funções

- Uma função é um bloco de código que realiza uma tarefa específica
- É semelhante ao conceito de **procedure** ou **subrotina**, existente em outras linguagens de programação
- Funções sempre retornam um valor
- Se a instrução de retorno (**return**) não for utilizada dentro da função, ela retornará **undefined**



Funções

- Valores podem ser passados para uma função como parâmetros ou usados dentro da função se estiverem dentro do seu escopo
- Uma vez que a função executa o seu código de retorno, mais nenhum código declarado dentro da função é executado
- A função envia o resultado do seu retorno para a instrução que a invocou



Funções

- Funções possuem as seguintes características (antes do ECMAScript 2015):
 - É definida pela instrução **function**
 - Seu nome segue as mesmas regras de nomes de variáveis
 - Pode possuir parâmetros (listados dentro de parênteses)
 - O código a ser executado é declarado dentro de chaves (obrigatório)

```
function teste(parameter1, parameter2, parameter3) {  
    /* */  
}
```



Funções

- Uma função é executada quando é invocada dentro do algoritmo
- Podem ser invocadas imediatamente após sua criação (*Immediately-invoked Function Expression* ou **IIFE**):

```
(function() {  
    /**/  
})();
```

- Uma IIFE pode possuir um nome ou não:

```
(function teste() {  
    /**/  
})();
```



Funções

- Declaração de função:

O código a seguir é uma declaração de função. Esta função existe a partir do momento em que o script é carregado no browser.

```
function teste(parameter1, parameter2, parameter3) {  
    /**/  
}
```



Funções

- Expressão de função:

O código a seguir é uma expressão de função. Esta função só existe quando a linha onde está a expressão é executada.

```
let teste = function(parameter1, parameter2, parameter3) {  
  /**/  
}
```



Funções

- Funções declaradas podem ser executadas antes da declaração. Isto é possível devido ao mecanismo denominado *hoisting*
- Este mecanismo coloca as declarações de variáveis e funções no topo do código antes de sua execução, conforme o exemplo:

```
let output = teste(2, 4);  
console.log(output);
```

```
function teste(x, y) {  
    return x * y;  
}
```



Funções

- ECMAScript 2015 (ES6) também introduziu uma nova forma de criar funções: Arrow functions.

```
function teste(x, y) {  
  return x * y;  
}
```

```
let teste = function(x, y) {  
  return x * y;  
}
```

```
let teste = (x, y) => {  
  return x * y;  
}
```



Funções

- Se a arrow function não possui parametros, pode ser representada da seguinte forma:

```
let teste = () => {  
  return "Olá";  
}
```



Funções

- Se a arrow function possui apenas um parâmetro, pode ser representada assim:

```
let teste = x => {  
  return x * 2;  
}
```



Funções

- Se a arrow function possui apenas uma declaração, e esta retorna um valor, pode ser representada como no seguinte exemplo:

```
let teste = () => "Olá";
```

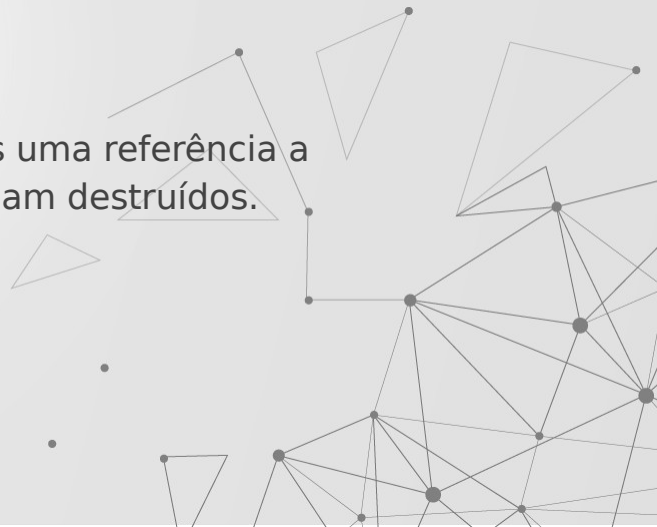


Closures



Closures

- A *closure* é um comportamento do JavaScript que permite o acesso do escopo de uma função externa a partir de uma função interna.
- O acesso ao escopo superior é possível mesmo que a função externa tenha sido encerrada.
- O escopo externo é preservado porque ainda mantemos uma referência a ele (a função interna), o que previne que seus dados sejam destruídos.



Closures

- Closures são úteis porque permitem que você associe dados com a função que trata estes dados.
- É possível traçar um paralelo com alguns aspectos da programação orientada a objetos, onde objetos permitem que seus dados (propriedades) fiquem associados aos métodos que tratam estes dados.



Closures: motivação

- Uma função pode acessar variáveis criadas dentro da função:

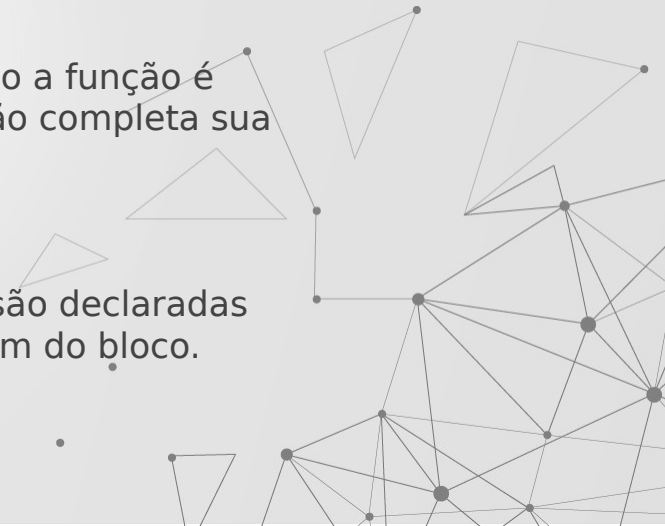
```
function myFunction() {  
    var a = 4;  
    return a * a;  
}
```

- Observação: variáveis criadas dentro de uma função possuem escopo local, ou seja, só podem ser acessadas dentro desta função.



Closures: motivação

- O tempo de vida de uma variável depende de seu escopo:
 - Variáveis globais existem enquanto o script estiver sendo executado.
 - Variáveis locais em uma função são criadas quando a função é invocada e removidas da memória quando a função completa sua execução.
 - Variáveis locais em um bloco são criadas quando são declaradas dentro do bloco e são removidas da memória ao fim do bloco.



Closures: motivação

- Crie um contador que possa ser usado nas suas funções:

```
var counter = 0;
```

```
function add() {  
  counter += 1;  
}
```

```
add();
```

```
add();
```

```
console.log(counter);
```

- Problema: a variável *counter* pode ser alterada diretamente.



Closures: motivação

- Crie uma variável local na função:

```
var counter = 0;
```

```
function add() {  
  var counter = 0;  
  counter += 1;  
}
```

```
add();  
add();
```

```
console.log(counter);
```

- Problema: estamos acessando a variável errada.



Closures: motivação

- Retornando apenas a variável interna:

```
function add() {  
  var counter = 0;  
  counter += 1;  
  return counter;  
}
```

```
console.log(add());  
console.log(add());
```

- Problema: quando a função é invocada, a variável *counter* é inicializada.



Closures: motivação

- Testando uma hipótese: e se usarmos funções aninhadas?

```
function add() {  
  var counter = 0;  
  function plus() {  
    counter += 1;  
  }  
  plus();  
  return counter;  
}
```

```
console.log(add());  
console.log(add());
```

- Problema: counter ainda é inicializado a cada vez que a função é invocada.



Closures: motivação

- Usando uma **IIFE** (*Immediately-invoked Function Expression*):

```
var add = (function () {  
    var counter = 0;  
    return function () {  
        counter += 1;  
        return counter  
    }  
})();
```

```
console.log(add());  
console.log(add());
```

- Solucionamos o problema usando uma *closure*.

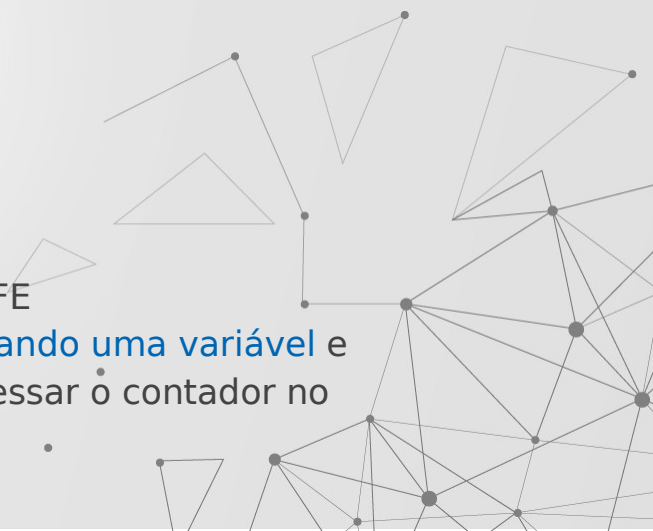


Closures: motivação

```
var add = (function () {  
    var counter = 0;  
    return function () {  
        counter += 1;  
        return counter  
    }  
})();
```

```
console.log(add());  
console.log(add());
```

- 1) A variável *add* receberá o resultado de uma função IIFE
- 2) Esta função será executada apenas uma vez, **inicializando uma variável** e **retornando uma expressão de função**, que consegue acessar o contador no escopo superior.



Closures

```
function retornaDobro(parametro) {  
  let numero = parametro;  
  return function() {  
    return numero * 2;  
  };  
}
```

```
let dobroDez = retornaDobro(10);  
let dobroCem = retornaDobro(100);  
let dobroMil = retornaDobro(1000);
```

```
console.log(dobroDez());  
console.log(dobroCem());  
console.log(dobroMil());  
console.log(dobroDez());
```



Closures

- Um exemplo de problema envolvendo *closures*:

```
let mostraltem = [];
```

```
for (var i = 0 ; i < 3; i++) {  
    mostraltem[i] = function() {  
        console.log("item " + i);  
    }  
}
```

```
mostraltem[0]();  
mostraltem[1]();  
mostraltem[2]();
```

- Problema: o escopo superior (variável *i*) persiste mesmo depois do fim do bloco ou da função (neste caso, do bloco *for*).



Closures

- Uma solução: capturar o valor de *i* e transportar para o escopo interno.

```
let mostraltem = [];
```

```
for (var i = 0 ; i < 3; i++) {  
  (function(i) {  
    mostraltem[i] = function() {  
      console.log("item " + i);  
    }  
  })(i);  
}
```

```
mostraltem[0]();  
mostraltem[1]();  
mostraltem[2]();
```



Tipos



Tipos

- Os tipos definem:
 - os valores que os dados de uma variável podem assumir
 - as operações que podem ser efetuadas sobre estes dados



Tipos

- Linguagens estaticamente tipadas: os tipos das variáveis devem ser declarados durante a codificação (Java, C, C++):

```
int num1 = 10;
```

```
String num2 = "5";
```

```
int num3 = num1 * num2; // erro
```



Tipos

- Linguagens dinamicamente tipadas: os tipos das variáveis dependem do seu conteúdo, atribuído durante a execução (PHP, Python, JavaScript):

```
$var1 = 120;
```

```
$var2 = "10";
```

```
$var3 = $var1 * $var2;
```

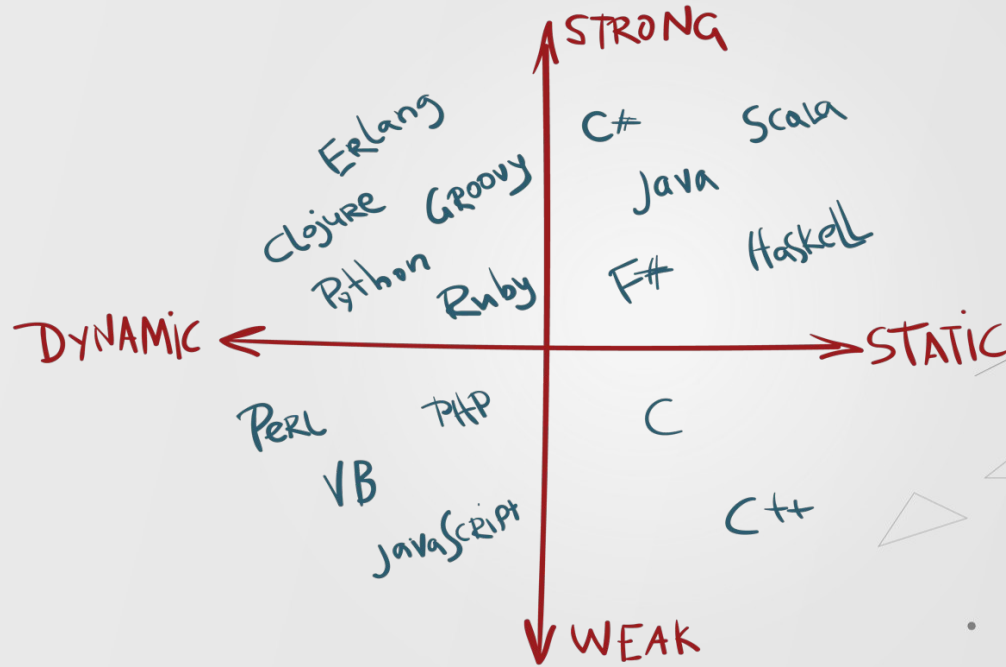


Tipos

- Linguagens fortemente tipadas: uma vez que um valor é atribuído à uma variável, esta mantém o tipo deste valor durante a execução do programa (Java, Python).
- Linguagens fracamente tipadas: o tipo de uma variável pode ser alterado durante a execução (C, C++, JavaScript, PHP).



Tipos



Tipos

- Algumas características do JavaScript em relação a tipos:

Tipagem dinâmica:

```
var resposta = 42;  
resposta = "Obrigado pelos peixes...";
```

Conversão dinâmica:

```
var x = "A resposta é " + 42;  
var y = "37" - 7;  
var z = "37" + 7;
```



Tipos

- Mecanismos para conversão explícita de dados:

- *parseInt()*

- *parseFloat()*

- *Operador +:*

- var a = "3";*

- var b = 1;*

- c = a + b; // "31"*

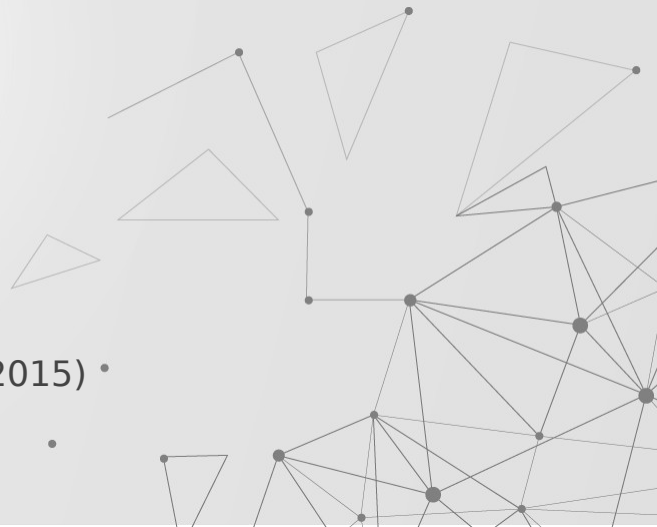
- c = +a + +b; // 4*



Tipos

- Tipos de dados (JavaScript):
 - boolean: true e false
 - null: valor nulo
 - undefined: valor indefinido
 - number: número - permite notação científica:

```
var y = 123e5;    // 123000000  
var z = 123e-5;   // 0.00123
```
 - string: cadeia de caracteres
 - symbol: cria um valor privado, interno e único (ES2015) •
 - Object: objeto



Tipos

- Object:

```
let pessoa = {nome:"Pedro", cpf:"923.483.949/87"};
```

- Array:

```
let pessoas = ["Pedro", "Joao", "Maria"];
```



Tipos

Object

Obtendo o nome das propriedades de um objeto:

Object.keys(obj);

Iterando sobre propriedades de um objeto:

```
for (const propriedade in pessoa) {  
  console.log(`${propriedade}: ${pessoa[propriedade]}`);  
}
```



Tipos

Array

Iterando sobre itens de um array:

```
for (index = 0; index < array.length; index++) {  
  console.log(array[index]);  
}
```



Tipos

Array

Iterando sobre itens de um array:

```
index = 0;  
  
while (index < array.length) {  
    console.log(array[index]);  
    index++;  
}
```



Tipos

Array

Iterando sobre itens de um array:

```
array.forEach(myFunction);  
  
function myFunction(item, index) {  
  console.log(item);  
}
```



Tipos

- Algumas funções nativas do Array:
 - `peessoas.length;`
 - `peessoas.sort();`
 - `peessoas.reverse();`
 - `peessoas.push("Aluno");`
 - `peessoas.indexOf("Aluno");`



Tipos

- Um array pode conter qualquer tipo de dado, inclusive elementos mais complexos:

```
var capitais = [  
    {nome: "Recife", estado: "Pernambuco", ano: "1823"},  
    {nome: "Boa Vista", estado: "Roraima", ano: "1890"},  
    {nome: "Palmas", estado: "Tocantins", ano: "1989"}  
];
```

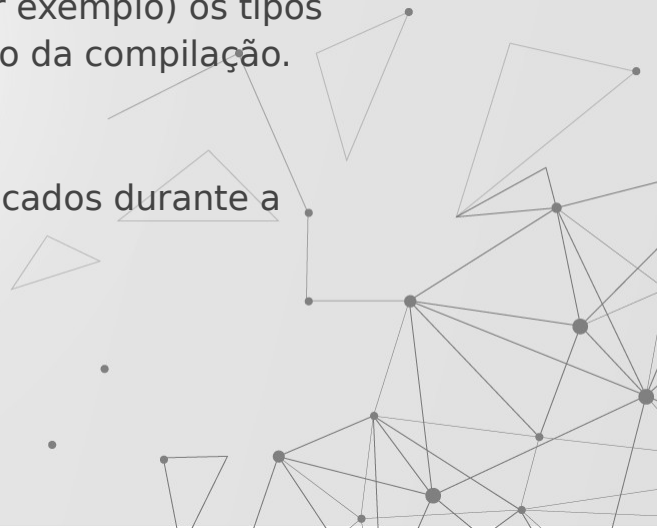




Tipagem Dinâmica

Tipagem Dinâmica

- Linguagens dinamicamente tipadas são aquelas (como o JavaScript) onde o interpretador (por exemplo, o browser) atribui um tipo para a variável no momento da execução do código com base no tipo do valor da variável.
- Nas linguagens estaticamente tipadas (como o Java, por exemplo) os tipos são checados antes da execução do código, no momento da compilação.
- Nas linguagens dinamicamente tipadas os tipos são checados durante a execução do código, no momento da interpretação.



Tipagem Dinâmica

- No JavaScript o operador `typeof` pode ser utilizado para informar o tipo de uma variável, como nos exemplos abaixo:

`typeof "Pedro"`

`typeof 3.14`

`typeof NaN`

`typeof false`

`typeof [1,2,3,4]`

`typeof {nome:'Pedro', cpf:'923.483.949-87'}`

`typeof new Date()`

`typeof function () {}`

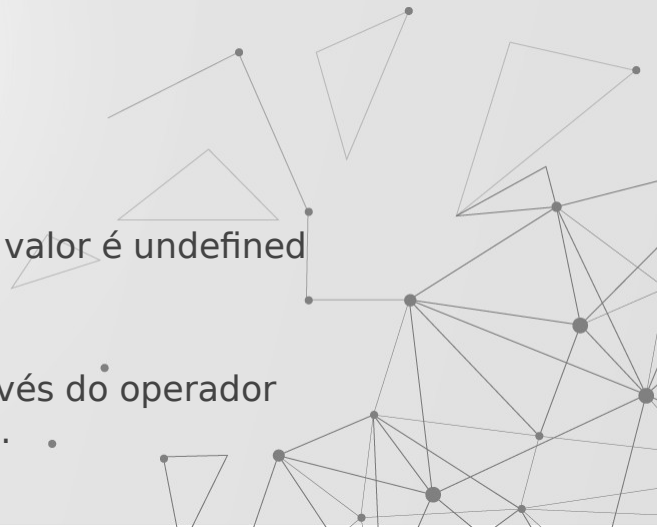
`typeof minhaVariavel`

`typeof null`



Tipagem Dinâmica

- Observações:
 - O tipo de NaN (*not a number*) é number
 - O tipo de um array é object
 - O tipo de uma data é date
 - O tipo de um null é object
 - O tipo de um undefined é undefined
 - O tipo de uma variável a qual não foi atribuído um valor é undefined
 - O operador typeof sempre retorna uma string
 - Não é possível saber se um objeto é um array através do operador typeof. Para isto usamos o operador Array.isArray().



Tipagem Dinâmica

- Alguns operadores de comparação também consideram o tipo dos dados:

`==` → verifica se os operandos são iguais

`===` → verifica se os operandos e os tipos são iguais

`!=` → verifica se os operandos são diferentes

`!==` → verifica se os operandos e os tipos são diferentes



Tipagem Dinâmica

- *Typecasting* ou coerção significa alterar o tipo de um valor.
- Uma coerção pode ser implícita, quando acontece de maneira automática por meio de uma operação, ou explícita, quando é intencional.
- O JavaScript suporta coerção para três tipos: string, boolean ou number.



Tipagem Dinâmica

- Coerção para string:
 - Para coerção explícita utilizamos o método `String()`:
`let a = 5;`
`let b = String(a);`
 - Há coerção implícita quando utilizamos o operador `+`:
`let a = 5;`
`let b = "10" + a;`



Tipagem Dinâmica

- Coerção para boolean:

- Para coerção explícita utilizamos o método Boolean():

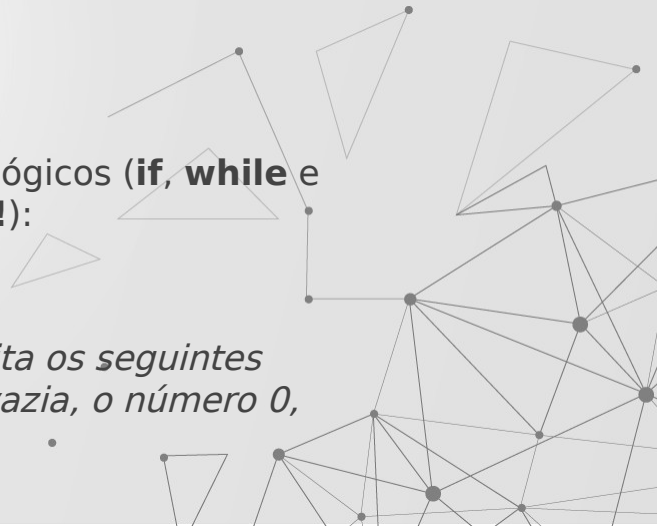
```
let a = 1;
```

```
let b = Boolean(a);
```

- Há coerção implícita quando utilizamos contextos lógicos (**if**, **while** e **for**, por exemplo) ou operadores lógicos (**||**, **&&** e **!**):

```
let a = !0;
```

*Observação: após uma coerção explícita ou implícita os seguintes valores serão convertidos para **false**: uma string vazia, o número 0, undefined e null.*



Tipagem Dinâmica

- Coerção para number:
 - Para coerção explícita utilizamos o método Number(), que invocará o método parseInt() ou parseFloat(), dependendo do valor informado:

```
let a = Number("5");
```

```
let b = Number("5.01");
```

- Há coerção implícita quando utilizamos operadores de comparação (>, <, ==, por exemplo) ou aritméticos (exceto o +):

```
let a = 4 > '5';
```

```
let b = 5 - '4';
```

```
let c = 5 == '5';
```



Operadores



Operadores

- **Operadores** são símbolos que executam operações sobre **operandos**.
- **Operandos** são os valores sobre os quais o **operador** trabalha.
- A combinação de **operadores** e **operandos** é uma **expressão**.

Exemplo:

```
var dias = 30;  
var horas = dias * 24;  
document.write(horas);
```

Onde:

dias é um operando (variável **dias** contendo o valor numérico **30**)

***** é um operador (operador aritmético da operação **multiplicação**)

24 é um operando (valor numérico **24**)



Operadores

- **Operadores de atribuição** atribuem um valor ao operando da esquerda com base no operando da direita.

Exemplos:

var **x = 1**;

x += 2; (mesmo que **x = x + 2**)

x -= 3; (mesmo que **x = x - 3**)

Referência:

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



Operadores

- **Operadores aritméticos** recebem valores numéricos (literais ou variáveis) como operandos e retornam um único valor numérico. Os operadores aritméticos padrão são os seguintes: adição (+), subtração (-), multiplicação (*) e divisão (/).

Exemplos:

```
var a = 2 + 3 - 1;
```

```
var b = 4 * 3 / 2;
```

```
var c = 11 % 3 ** 2;
```

Referência:

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



Operadores

- **Operadores de comparação** retornam um valor booleano comparando dois operandos entre si.

Exemplos:

1 == 1 (retorna **true**)

"1" == 1 (retorna **true**)

1 === 1 (retorna **true**)

"1" === 1 (retorna **false**)

1 != 2 (retorna **true**)

"1" != 1 (retorna **false**)

3 !== 4 (retorna **true**)

"3" !== 3 (retorna **true**)

4 > 3 (retorna **true**)

3 > 3 (retorna **false**)

3 >= 3 (retorna **true**)

3 < 4 (retorna **true**)

4 < 4 (retorna **false**)

4 <= 4 (retorna **true**)

Referência:

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

Operadores

- **Operadores lógicos** são geralmente utilizados com valores booleanos (true e false) e retornam um valor booleano. No entanto, os operadores lógicos **&&** e **||** retornam o valor de um dos operandos. Nesse caso, se foram valores não-boleanos, retornarão um valor não-boleano. Observação: a precedência destes operadores é **! → && → ||**.

Exemplos:

true && true

false || false && true

true && !false

Referência:

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

Operadores

- O **Operador Condicional** (ou operador ternário) é o único operador JavaScript que possui três operandos. Este operador é frequentemente usado como um atalho para a instrução if.

Exemplo:

```
var idade = 25;  
var msg = idade >= 18 ? "acesso permitido" : "acesso negado";  
document.write(msg);
```

Referência:

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

Operadores

- Todos os operadores possuem as seguintes características:
 - **Resultado**
 - **Aridade**
 - **Precedência**
 - **Associatividade**
- Alguns operadores possuem as seguintes características:
 - **Efeito colateral**
 - **Ordem de avaliação dos operandos**



Operadores

Resultado

Valor resultante da aplicação do operador sobre os seus operandos.

Exemplos:

*A aplicação do operador **-** na expressão **7 - 2** resulta em 5.*

*A aplicação do operador **++** na expressão **10++** resulta em 11.*



Operadores

Aridade

Quantidade de operandos requeridos pelo operador, que pode ser unário (um operando), binário (dois operandos) ou ternário (três operandos).

Exemplos:

Na expressão **1++** o operador **++** (incremento) é um operador unário, pois requer apenas um operando.

Na expressão **6 - 2** o operador **-** (subtração) é um operador binário, pois requer dois operandos.

Na expressão **true ? 10 : 20** o operador **?:** (condicional) é um operador ternário, pois requer três operandos.

Operadores

Precedência

Determina a sequência com que um operador é aplicado. Operações com maior complexidade são executadas primeiro, mas o uso de parênteses altera as precedências.

Exemplos:

Na expressão $4 + 3 * 2$ o resultado é 10, pois o operador $*$ (multiplicação) possui precedência maior que o operador $+$ (adição).

Na expressão $(4 + 3) * 2$ o resultado é 14, pois os parênteses alteraram a precedência das operações.

Referência:

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

Operadores

Associatividade

Decide a ordem de aplicação de operadores quando estes possuem a mesma precedência.

Exemplos:

Na expressão **16 / 8 / 2** o resultado é 1, pois o operador / (divisão) possui associatividade à esquerda.

Na expressão **16 / (8 / 2)** o resultado é 4, pois os parênteses alteraram a precedência das operações.

A expressão **a = b = 10** possui o mesmo resultado que **a = (b = 10)**, pois o operador = (atribuição) possui associatividade à direita. .

Operadores

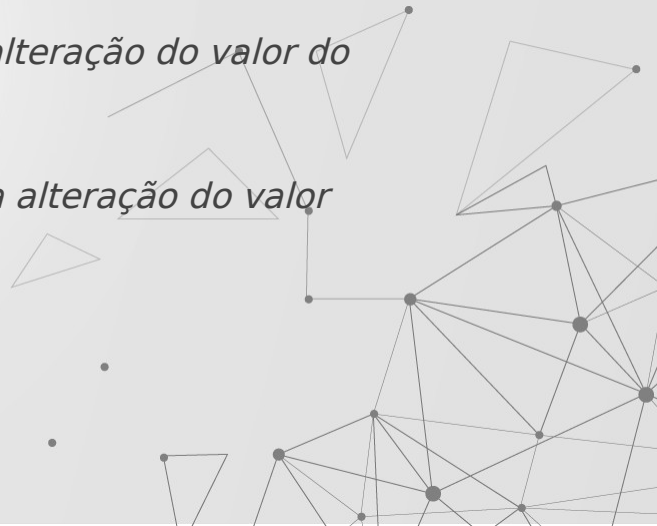
Efeito colateral

Alteração de valor produzido pelo operador em um dos seus operandos.

Exemplos:

Na expressão **a = 5** o operador **=** (atribuição) causa a alteração do valor do operando **a**.

Na expressão **b++** o operador **++** (incremento) causa a alteração do valor do operando **b**.



Operadores

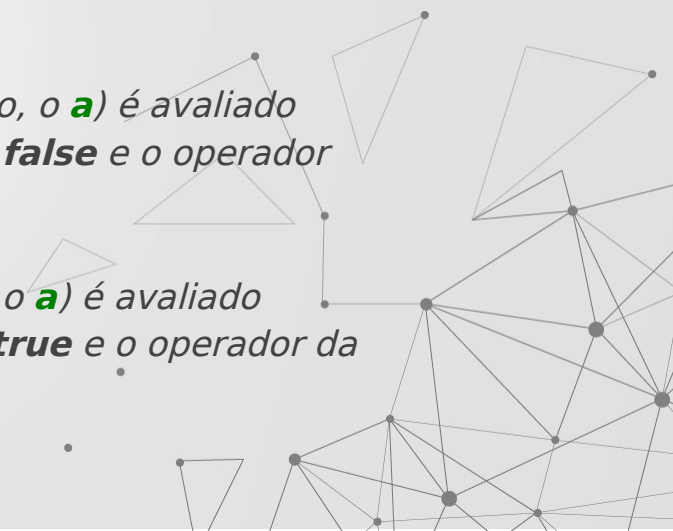
Ordem de avaliação dos operandos

Determina qual dos operandos é avaliado primeiro. Só existe em operandos onde a aridade é maior que um e não existe em operadores aritméticos.

Exemplos:

Na expressão ***a && b*** o operando da esquerda (no caso, o ***a***) é avaliado primeiro. Se este for ***false***, o resultado da expressão é ***false*** e o operador da direita (no caso, o ***b***) não é avaliado.

Na expressão ***a || b*** o operando da esquerda (no caso, o ***a***) é avaliado primeiro. Se este for ***true***, o resultado da expressão é ***true*** e o operador da direita (no caso, o ***b***) não é avaliado.



Arrays



Arrays

- Arrays são geralmente descritos como "listas de objetos"; são basicamente objetos que contem múltiplos valores armazenados em uma lista.
- Um objeto array pode ser armazenado em variáveis e ser tratado de forma muito similar a qualquer outro tipo de valor. A diferença está em podermos acessar cada valor dentro da lista individualmente.

Exemplo:

```
let minhaLista = ["abc", "fgh", "xyz"];
```

```
let meuElemento = minhaLista[2];
```



Arrays

- É possível criar um Array de duas formas diferentes:

```
let meuArray = [1, 2, 3];
```

```
let meuArray = new Array(1, 2, 3);
```

Observação: se for utilizado apenas um parâmetro numérico no construtor *new Array()*, este parâmetro determinará o tamanho do Array a ser criado.

```
let meuArray = new Array(4);
```



Arrays

- Acessando e modificando itens de um Array:

```
let meuArray = [1, "2", null, "teste"];
```

```
let item = meuArray[2]; // acessando um item
```

```
meuArray[2] = "novo item"; // modificando um item
```

```
console.log(meuArray);
```



Arrays

- Encontrando o comprimento de um Array:

```
let meuArray = [1, "2", null, "teste"];
```

```
let comprimento = meuArray.length;
```

```
console.log(comprimento);
```



Arrays

- Iterando sobre Arrays (usando *for*):

```
let frutas = ["Banana", "Laranja", "Manga"];
```

```
for (i = 0; i < frutas.length; i++) {  
    console.log(frutas[i]);  
}
```



Arrays

- Iterando sobre Arrays (usando *forEach*):

```
let frutas = ["Banana", "Laranja", "Manga"];
```

```
frutas.forEach(exibeFruta);
```

```
function exhibeFruta(fruta) { // a função permanece na memória  
  console.log(fruta);  
}
```

ou

```
frutas.forEach(function(fruta){ // a função é executada e sai da memória  
  console.log(fruta);  
});
```



Arrays

- Adicionando elementos a um Array:

```
let frutas = ["Banana", "Laranja", "Manga"];
```

```
frutas.push("Lima"); // adicionando ao fim do Array
```

```
console.log(frutas);
```

```
frutas.unshift("Morango"); // adicionando ao início do Array
```

```
console.log(frutas);
```



Arrays

- Removendo elementos de um Array:

```
let frutas = ["Banana", "Laranja", "Manga"];
```

```
frutas.pop(); // removendo o último elemento do Array
```

```
console.log(frutas);
```

```
frutas.shift(); // removendo o primeiro elemento do Array
```

```
console.log(frutas);
```



Arrays

- Procurando o índice de um item em um Array:

```
let frutas = ["Banana", "Laranja", "Manga", "Laranja"];
```

```
let indice = frutas.indexOf("Laranja");
```

```
console.log(indice);
```



Arrays

- Removendo um item pela sua posição no Array:

```
let frutas = ["Banana", "Laranja", "Manga", "Laranja"];
```

```
let indice = frutas.indexOf("Laranja");
```

```
frutas.splice(indice, 1); // removendo apenas um elemento
```

```
console.log(frutas);
```



Arrays

- Removendo diversos itens pela sua posição no Array:

```
let frutas = ["Banana", "Laranja", "Manga", "Laranja", "Morango"];
```

```
let indice = frutas.indexOf("Laranja");
```

```
frutas.splice(indice, 2); // removendo dois elementos
```

ou

```
frutas.splice(indice); // removeria todos os elementos a partir do índice
```

```
console.log(frutas);
```



Arrays

- Substituindo itens pela sua posição no Array:

```
let frutas = ["Banana", "Laranja", "Manga"];
```

```
let indice = frutas.indexOf("Laranja");
```

```
frutas.splice(indice, 1, "Morango"); // substituindo "Laranja" por "Morango"
```

```
console.log(frutas);
```



Arrays

- Diferença entre *splice()* e *slice()*:

splice(): modifica o array

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/splice

slice(): não modifica o array, apenas retorna os elementos solicitados.

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/slice



Arrays

- Exemplo de uso do *slice()*:

```
let frutas = ["Banana", "Laranja", "Limao", "Maçã", "Manga"];
```

```
let citricos = frutas.slice(1, 3);
```

```
// slice(<indice do primeiro elemento>, <limite da seleção>)
```

```
console.log(citricos);
```



Arrays

- Não é possível clonar (copiar o conteúdo) de um Array usando o operador de atribuição “=”. Isto acontece porque um Array é apenas uma referência aos valores que são contidos nesse Array.

Exemplo:

```
let a = [1, 2, 3];
```

```
let b = a;
```

```
let c = b;
```

```
console.log(c); // c possui os mesmos elementos de a
```

```
a[0] = 999; // alterando um valor de a
```

```
console.log(a); // a foi alterado
```

```
console.log(b); // b também foi alterado (porque é apenas uma referência)
```

```
console.log(c); // c também foi alterado (porque é apenas uma referência)
```



Arrays

- Para clonar um Array podemos utilizar diferentes métodos:

Exemplos:

```
let a = [1, 2, 3];  
let b = a.slice(); // aqui acontece uma clonagem  
let c = Array.from(b); // aqui acontece uma clonagem
```

```
console.log(c); // c possui os mesmos elementos de a  
a[0] = 999; // alterando um valor de a  
console.log(a); // a foi alterado  
console.log(b); // b não foi alterado porque é outro objeto  
console.log(c); // c não foi alterado porque é outro objeto
```



Arrays

- Convertendo um String para Array (função *split()*):

Exemplo:

```
let texto = "Santa Catarina,Paraná,Rio Grande do Sul";  
let estados = texto.split(",");
```

```
console.log(estados);
```



Arrays

- Convertendo um Array para String (função *toString()*):

Exemplo:

```
let estados = ["Santa Catarina", "Paraná", "Rio Grande do Sul"];  
let texto = estados.toString();
```

OU

```
let texto = estados.join(",");  
  
console.log(texto);
```



Arrays

- Ordenando Arrays com ordenação léxica:

```
let estados = ["Santa Catarina", "Paraná", "Rio Grande do Sul"];  
estados.sort(); // ordena os elementos (ordenação léxica)  
console.log(estados);
```

- Ordenando Arrays com ordenação numérica:

```
let numeros = [3, 1, 1000, 31, 0, 71];  
numeros.sort(function(a, b){  
    return a - b; // para ordenação decrescente utilize b - a  
});  
console.log(numeros);
```



Arrays

- Invertendo a ordem dos elementos de um Array:

```
let estados = ["Santa Catarina", "Paraná", "Rio Grande do Sul"];  
estados.reverse(); // inverte a ordem dos elementos  
console.log(estados);
```



Arrays

- Verificando se todos os itens do Array obedecem a um critério (função `every()`):

```
let numeros = [100, 7, 10, 10000, 33];
```

```
let resultado = numeros.every(function(item) {  
    return item > 10;  
})
```

```
console.log(resultado);
```



Arrays

- Verificando se ao menos um item do Array obedece a um critério (função *some()*):

```
let numeros = [100, 7, 10, 10000, 33];
```

```
let resultado = numeros.some(function(item) {  
    return item > 10000;  
})
```

```
console.log(resultado);
```



Arrays

- Verificando se o Array possui um determinado valor (função *includes()*):

```
let numeros = [100, 7, 10, 10000, 33];
```

```
let resultado = numeros.includes(7);
```

```
console.log(resultado);
```



Arrays

- Criando um novo Array a partir de um Array original (função *map()*):

```
let numeros = [100, 7, 10, 10000, 33];
```

```
let resultado = numeros.map(function(item) {  
  return item * 2;  
})
```

```
console.log(resultado);
```



Arrays

- Criando um novo Array a partir de um Array original (função *filter()*):

```
let numeros = [100, 7, 10, 10000, 33];
```

```
let resultado = numeros.filter(function(item) {  
  return item > 90;  
})
```

```
console.log(resultado);
```



Arrays

- Reduzindo os valores de Array a um único valor (função *reduce()*):

```
let numeros = [100, 7, 10, 10000, 33];
```

```
let reducao = numeros.reduce(function(acumulador, valor) {  
    return acumulador + valor;  
});
```

```
console.log(reducao);
```



Objetos



Objetos

- Um objeto é uma coleção de dados (propriedades) e/ou funcionalidades (métodos) relacionadas.
- Suas unidades de informação são os membros, onde cada um deles possui um nome e um valor:

```
var nomeDoObjeto = {  
  nomeMembro1: valorMembro1,  
  nomeMembro2: valorMembro2,  
  nomeMembro3: function() {  
    return 1;  
  }  
};
```



Objetos

- O valor de um membro de um objeto pode ser de qualquer tipo.
- Os membros podem ser listados pelo método **Object.keys(object)**:

```
var nomeDoObjeto = {  
  nomeMembro1: 123,  
  nomeMembro2: "abc",  
  nomeMembro3: function() {  
    return 1;  
  }  
};
```

```
console.log(Object.keys(nomeDoObjeto));
```



Objetos

- Pode ser criado de maneira literal ou instanciado a partir de uma classe.
- Exemplo de criação de um objeto de maneira literal:

```
let meuObjeto = {};
```

- Instanciação de um objeto a partir de uma classe:

```
let meuObjeto = new MinhaClasse();
```



Objetos

- Usando a **notação de ponto** para acessar os membros de um objeto:

```
let pessoa = {  
  nome: ['Carlos', 'Silva'],  
  idade: 32,  
  interesses: ['livros', 'jogos'],  
  saudacao: function() {  
    console.log('Oi! Eu sou ' + this.nome[0] + '.');  
  }  
};
```

```
let idade = pessoa.idade;  
let interesses = pessoa.interesses;  
pessoa.saudacao();
```



Objetos

- É possível criar **subnamespaces** usando a **notação de ponto**:

```
let pessoa = {  
  nome: {  
    primeiro: 'Carlos',  
    ultimo: 'Silva'  
  },  
  idade: 32,  
  interesses: ['livros', 'jogos'],  
  saudacao: function() {  
    console.log('Oi! Eu sou ' + this.nome.primeiro + '!');  
  }  
};
```

```
let primeiroNome = pessoa.nome.primeiro;
```



Objetos

- Usando a **notação de colchetes** para acessar os membros de um objeto:

```
let pessoa = {  
  nome: {  
    primeiro: 'Carlos',  
    ultimo: 'Silva'  
  },  
  idade: 32,  
  interesses: ['livros', 'jogos'],  
  saudacao: function() {  
    console.log('Oi! Eu sou ' + this.nome.primeiro + '!');  
  }  
};
```

```
let primeiroNome = pessoa['nome']['primeiro'];
```



Objetos

- A **notação de colchetes** permite a utilização de variáveis para a criação de novos membros:

```
let pessoa = {nome: 'Bob'};  
let myDataName = 'altura';  
let myDataValue = '1.75m';  
pessoa[myDataName] = myDataValue;
```

- O nome de um membro de um objeto pode ser **qualquer string JavaScript válida**, ou **qualquer coisa que possa ser convertida em uma string, incluindo uma string vazia**. No entanto, caso não obedeça as regras de criação de nomes de variável (por exemplo, contendo espaços ou iniciando com número), este membro só poderá ser acessado com a notação de colchetes.

Objetos

- É possível remover membros de um objeto com o comando **delete**:

```
let pessoa = {  
  nome: 'Bob',  
  idade: 32  
};
```

```
console.log(pessoa);  
delete pessoa.idade;  
console.log(pessoa);
```



Objetos

- A palavra-chave **this** é utilizada para se referir ao próprio objeto:

```
let pessoa = {  
  nome: {  
    primeiro: 'Carlos',  
    ultimo: 'Silva'  
  },  
  idade: 32,  
  interesses: ['livros', 'jogos'],  
  saudacao: function() {  
    console.log('Oi! Eu sou ' + this.nome.primeiro + '!');  
  }  
};  
  
pessoa.saudacao();
```



Objetos

- No JavaScript, objetos são um tipo de referência. Dois objetos distintos nunca são iguais, mesmo que tenham as mesmas propriedades:

```
var fruta1 = {nome: 'abacaxi'};  
var fruta2 = {nome: 'abacaxi'};  
console.log(fruta1 == fruta2);  
console.log(fruta1 === fruta2);
```

- Apenas comparando o mesmo objeto de referência com ele mesmo resulta em verdadeiro:

```
var fruta1 = {nome: 'abacaxi'};  
var fruta2 = fruta1;  
console.log(fruta1 == fruta2);  
console.log(fruta1 === fruta2);
```



Objetos

- É possível clonar um objeto usando o método **Object.assign(destino, origem)**. Ele copia os valores de todas as propriedades do objeto origem para o objeto destino, retornando o objeto de destino, como no exemplo abaixo:

```
let destino = { a: 1, b: 2 };
```

```
let origem = { b: 4, c: 5 };
```

```
let retornoDestino = Object.assign(destino, origem);
```

```
console.log(destino);
```

```
console.log(retornoDestino);
```



Objetos

- Desta maneira, se usarmos um objeto vazio como objeto destino, o método **Object.assign(destino, origem)** produzirá um novo objeto contendo todos os elementos do objeto origem, como no exemplo abaixo:

```
let destino = {};
```

```
let origem = { b: 4, c: 5 };
```

```
let retornoDestino = Object.assign(destino, origem);
```

```
console.log(destino);
```

```
console.log(retornoDestino);
```



Objetos

- É possível também clonar um objeto usando o método **Object.create(origem)**. Ele cria um novo objeto a partir do objeto de origem, como no exemplo abaixo:

```
let origem = { b: 4, c: 5 };
```

```
let retornoDestino = Object.create(origem);
```

```
console.log(retornoDestino);
```



Objetos

- Criando objetos a partir de uma classe (função) usando o comando **new**:

```
function Carro(marca, modelo, ano) {  
  this.marca = marca;  
  this.modelo = modelo;  
  this.ano = ano;  
}
```

```
let carro1 = new Carro("Renault", "Sandero", 2018);  
let carro2 = new Carro("Volkswagen", "1300", 1977);  
console.log(carro1);  
console.log(carro2);  
console.log(typeof Carro);  
console.log(typeof carro1);  
console.log(typeof carro2);
```



Objetos

- Otimizando a função para receber um objeto como parâmetro durante a criação da instância, aos invés de vários parâmetros:

```
function Carro(dados) {  
  this.marca = dados.marca;  
  this.modelo = dados.modelo;  
  this.ano = dados.ano;  
}
```

```
let carro1 = new Carro({marca: "Renault", modelo:"Sandero", ano:  
2018});
```

```
console.log(carro1);
```



Objetos

- Todos os métodos e propriedades são herdados de **Object.prototype** e modificações no **prototype** de uma classe são propagadas a todos seus objetos. Um exemplo disso é a inserção de um método em uma classe:

```
function Carro(marca) {  
  this.marca = marca;  
}
```

```
let carro1 = new Carro("Renault");
```

```
Carro.prototype.exibeDados = function(){  
  return "a marca deste carro eh " + this.marca;  
}
```

```
console.log(carro1.exibeDados());
```



Objetos

- O prototype é útil porque, em muitos casos, não é possível alterar a função pois não temos acesso à classe que constrói aquele tipo de objeto (exemplo: classes/funções nativas do JavaScript):

```
let minhaLista = [1, 2, 3, 4];
```

```
Array.prototype.retornaUltimoElemento = function() {  
  return this[this.length - 1];  
}
```

```
console.log(minhaLista.retornaUltimoElemento());
```



Objetos

- Desafio: criar uma função que compare os membros de um objeto e retorne **true** caso:
 - a) os objetos possuam a mesma quantidade de membros E
 - b) os objetos possuam membros de mesmo nome E
 - c) estes membros possuam o mesmo valor

Casos de teste:

compara({a: 1, b: 2}, {a: 1, b: 3}); // false

compara({a: 1, b: 2}, {a: 1, b: 2, c: 3}); // false

compara({a: 1, b: 2, c: 3, d: 4}, {a: 1, b: 2, c: 3}); // false

compara({}, {}); // true

compara({a: 1, b: 2, c: 3}, {a: 1, b: 2, c: 3}); // true

