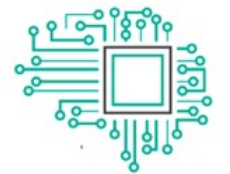


# JAVASCRIPT

---

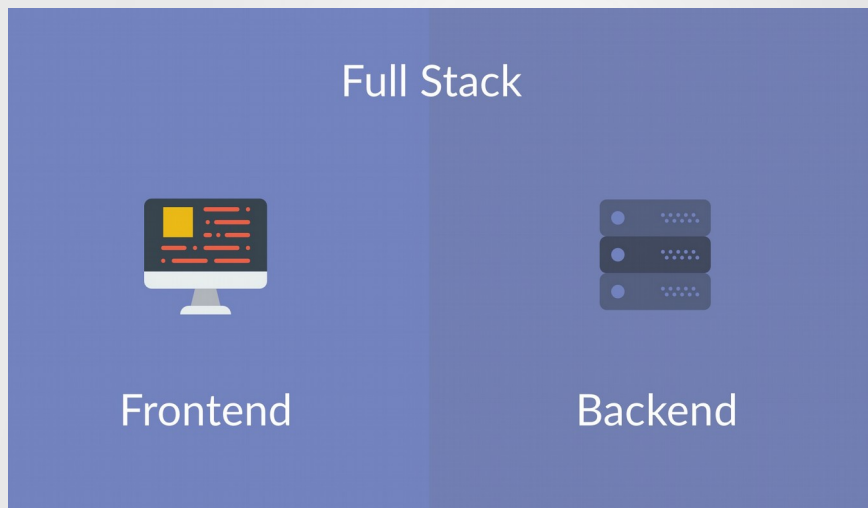
Pedro Moritz de Carvalho Neto



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.



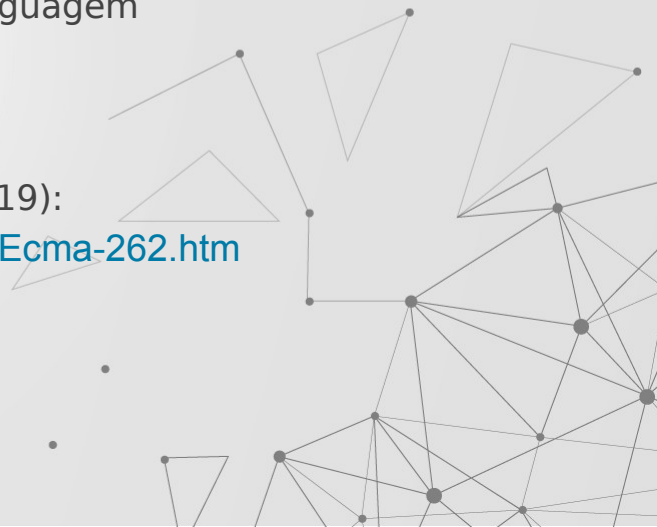
# Introdução ao JavaScript

- 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

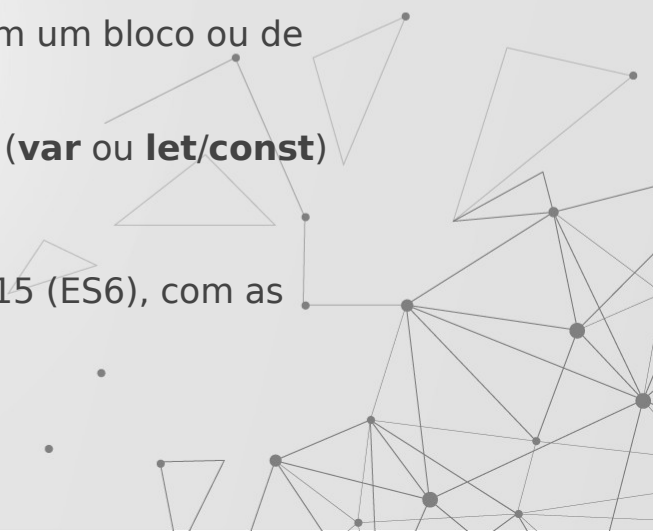
---



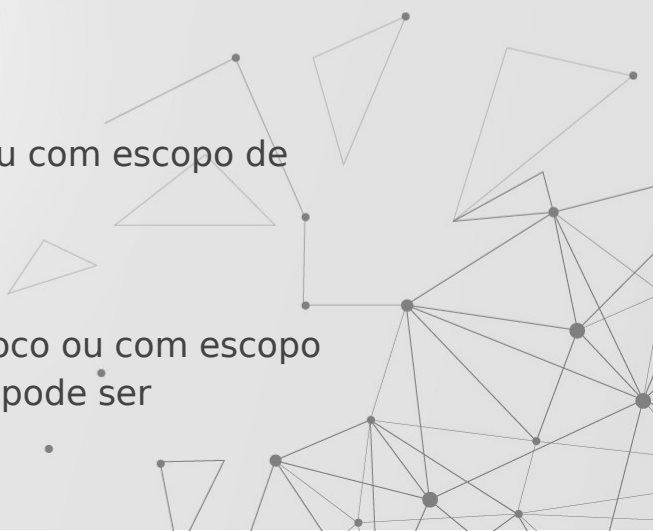


# Escopo

- 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

---





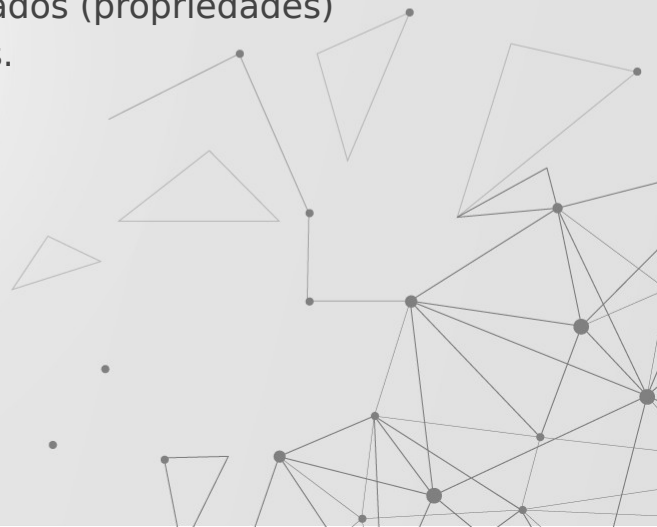
# Closures

- 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:

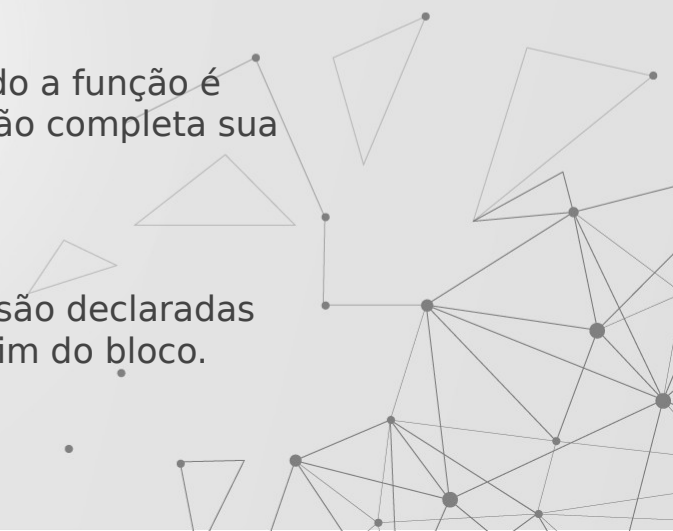
```
function myFunction() {  
    var a = 4;  
    return a * a;  
}
```

- 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]();
```



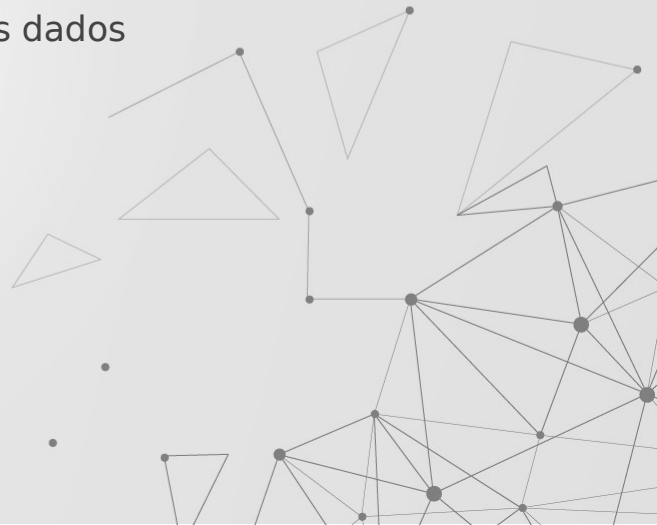
# Tipos

---



# 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;
```

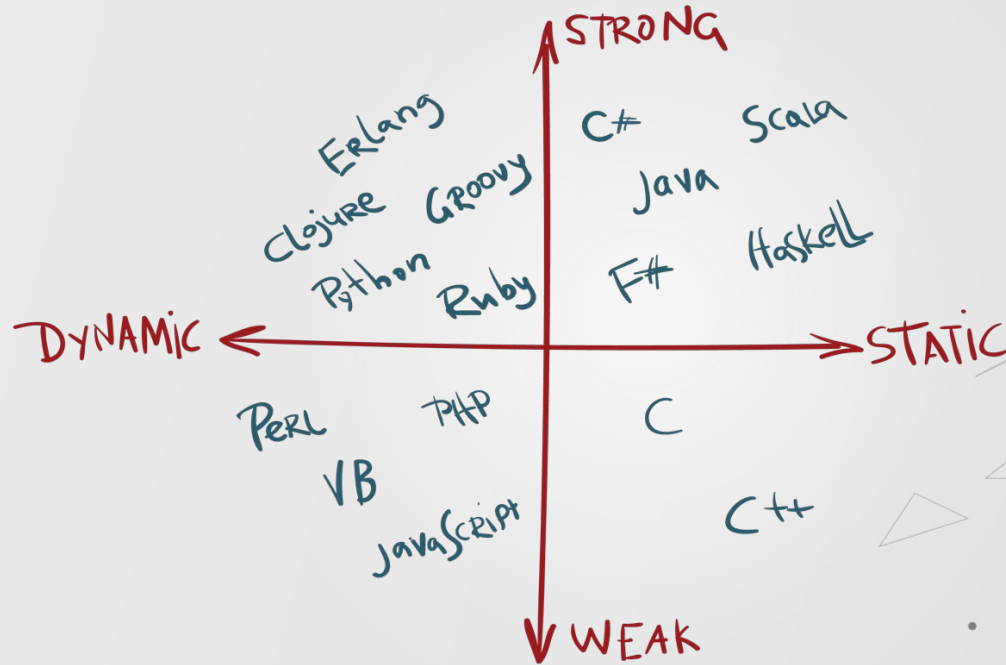


# Tipos

- 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"}  
];
```

