

ECMAScript 6

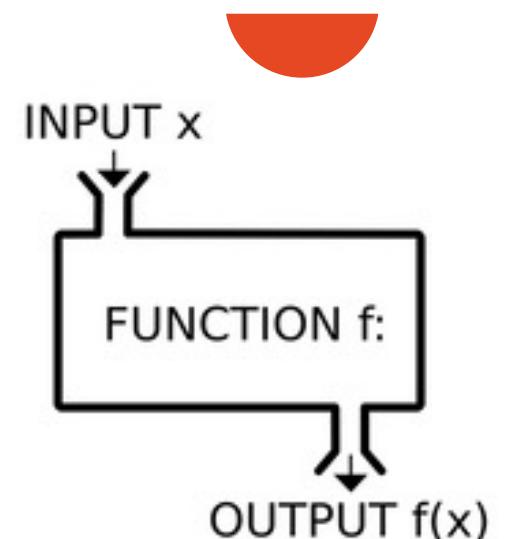
Rafael Escalfoni

*adaptado de ECMAScript6 - Entre de cabeça
no futuro do JavaScript*

Funções???

- Unidade de instrução com um propósito
- Exemplo

```
function addition(param1, param2) {  
    return param1 + param2;  
}  
  
// Calling functions  
console.log(addition(1)); //=> NaN  
console.log(addition(1, 2)); //=> 3  
console.log(addition(1, 2, 3)); //=> 3
```





Criando Funções em JavaScript

- Funções podem ser declaradas de 3 formas:
- Declaração de função:

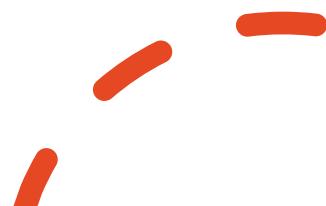
```
function nomeFunc() { /*instruções*/ }
```

- Expressão de função:

```
var nomeFunc = function() { /*instruções*/ }
```

- Invocando o construtor:

```
var nomeFunc = new Function(/*corpo da função */)
```



Declaração de funções em JavaScript

```
function a () {  
    ...  
}
```



The diagram shows a code snippet for a function declaration in JavaScript. The code is displayed on a dark background with white text. It consists of the keyword 'function', followed by the identifier 'a', then an empty parentheses '()' and a brace '{'. Below the identifier 'a', there are three dots '...', indicating that the function body contains multiple statements. An orange arrow points from a callout box containing the text 'Function name' to the identifier 'a'. The callout box has a black border and is positioned to the right of the identifier.

Declaração de funções em JavaScript

```
var a = function () {...}
```

Value of function assigned,
NOT the returned result!

No name defined

Arrow Functions





ECMAScript 6

Arrow function

- Parâmetros dentro de parênteses (...)
- “Fat arrow” (=>)
- Corpo da função entre chaves { }

```
(param1, param2 ...) => {  
    //corpo da função  
}
```





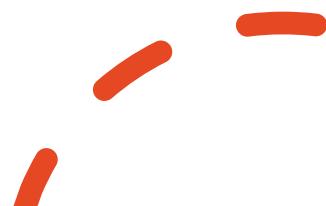
Comparando

```
var boasVindas = function(nome) {  
    return `Seja bem-vindo, ${nome}`;  
}
```

```
boasVindas("Luiz");
```

```
var boasVindas = (nome) => {  
    return `Seja bem-vindo, ${nome}`;  
}
```

```
boasVindas("Luiz");
```



Parâmetros Predefinidos em Funções

```
function imprimeNomeCompleto(nome, sobrenome, nomeDoMeio) {  
  if(nomeDoMeio === undefined){  
    console.log(` ${nome} ${sobrenome}`);  
  } else {  
    console.log(` ${nome} ${nomeDoMeio} ${sobrenome}`);  
  }  
}  
imprimeNomeCompleto("Alfredo", "Cairo", "Luiz"); // Alfredo Luiz Cairo  
imprimeNomeCompleto("Francisco", "Cairo"); // Francisco Cairo
```



Parâmetros Predefinidos em Funções

```
function imprimeNomeCompleto(nome, sobrenome, nomeDoMeio = "") {  
  console.log(` ${nome} ${nomeDoMeio} ${sobrenome}`);  
}  
imprimeNomeCompleto("Alfredo", "Cairo", "Luiz"); // Alfredo Luiz Cairo  
imprimeNomeCompleto("Francisco", "Cairo"); // Francisco Cairo
```

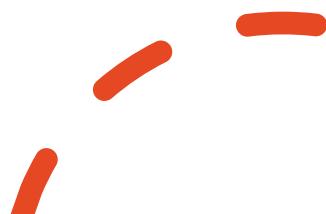




Parâmetros Predefinidos em Funções

OU

```
function imprimeNomeCompleto(nome, sobrenome, nomeDoMeio) {  
    let sobrenomeTratado = sobrenome || "";  
    let nomeDoMeioTratado = nomeDoMeio || "";  
  
    console.log(` ${nome} ${nomeDoMeioTratado} ${sobrenomeTratado}`);  
}  
  
imprimeNomeCompleto("Alfredo", "Cairo", "Luiz"); // Alfredo Luiz Cairo  
imprimeNomeCompleto("Francisco", "Cairo"); // Francisco Cairo
```





Valores *undefined*

```
function multiplicaPor(valor, fator = 2) {  
  return valor * fator;  
}
```

```
const valor = multiplicaPor(2,2);  
console.log(valor); // 4
```

```
const valor = multiplicaPor(2, undefined);  
console.log(valor); // 4
```

```
function printVal(valor = ""){  
  console.log(valor);  
}
```

```
printVal(); // ""  
printVal(null); // null
```





Tornando parâmetro obrigatório

```
function paramObrig(param) {  
    throw new Error(`O parâmetro ${param} é obrigatório!`)  
}  
  
function inserirNaTela(objeto = paramObrig('objeto')) {  
    // restante do código  
}  
  
inserirNaTela(); // Error: O parâmetro "objeto" é obrigatório!
```





Escopo

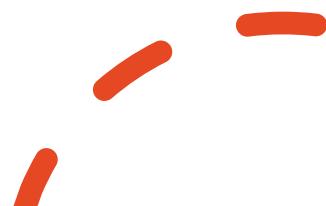
- Global
 - Variáveis e funções definidas aqui estarão disponíveis em qualquer lugar
- De função ou Léxico
 - Variáveis e funções definidas aqui só estarão visíveis dentro desta função





Cadeia de Escopos

- **Tudo** é executado em um **Contexto de Execução**
- A chamada de função **cria** um novo **Contexto de Execução**
- Cada Contexto de Execução tem:
 - Seu próprio **Ambiente de Variáveis**
 - Seu próprio objeto **this**
 - Referencia seu próprio Ambiente Externo
- Escopo global **não tem** um **Ambiente Externo** já que é o mais externo possível.





Cadeia de Escopos

- Variáveis referenciadas (não definidas) serão pesquisadas no seu escopo atual primeiro.
- Se não encontradas, o ambiente externo será pesquisado. Se não encontrar, o externo ao externo será pesquisado etc.
- Isto acontecerá até o escopo Global
- Se não achar nem no escopo Global, a variável é **undefined**



Global

```
var x = 2;  
A();
```

Function A

```
var x = 5;  
B();
```

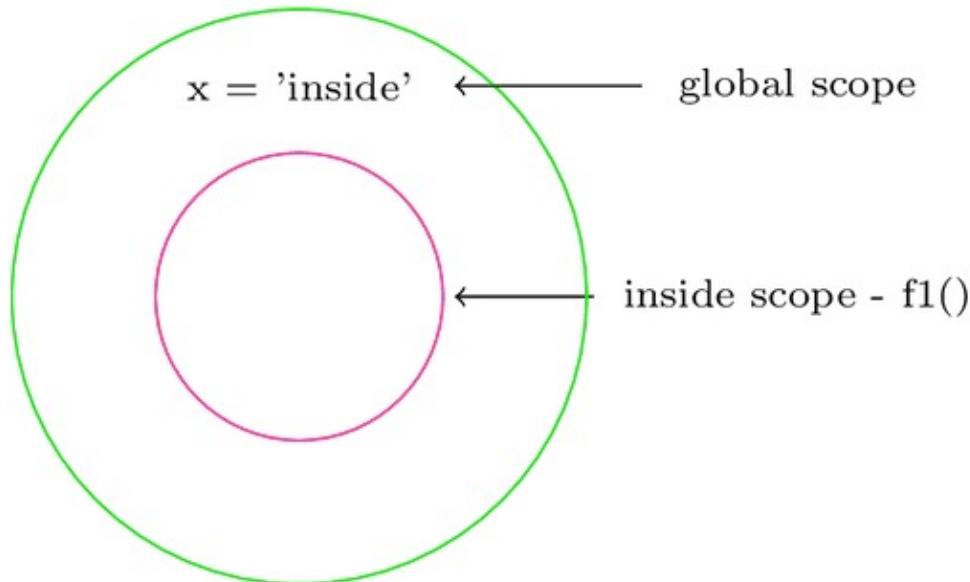
Even though
'B' is called
within 'A'

'B' is defined
within Global

Function B

```
console.log(x);
```

Result: x = 2



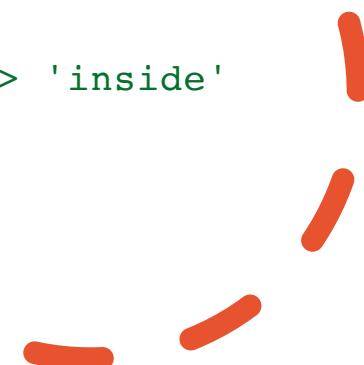
Scope	Hierarchy	Variable
global	g	x = 'inside'
inside	g>f1	

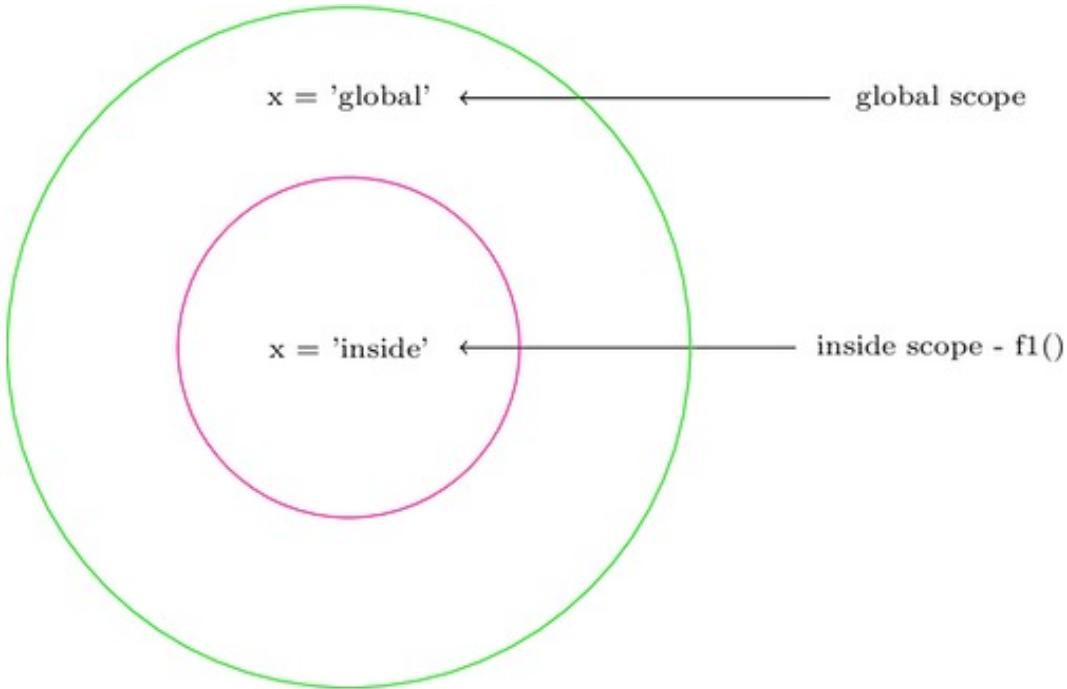
Escopo Global, Escopo Léxico

```
let x = 'global';

function f1() {
  x = 'inside';
}

console.log(x); //=> 'global'
f1();
console.log(x); //=> 'inside'
```





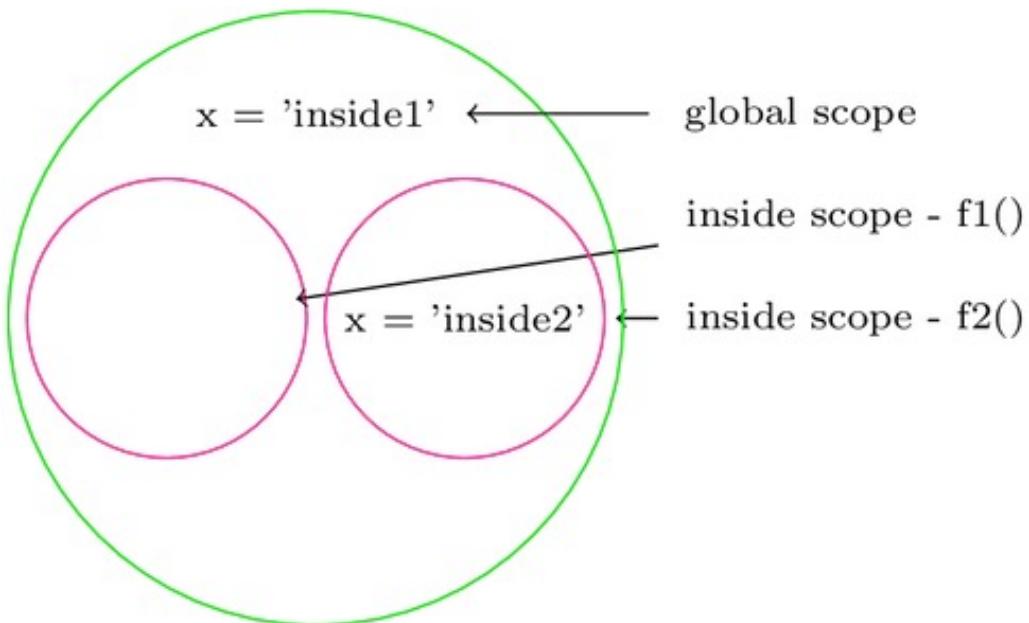
Scope	Hierarchy	Variable
global	g	x = 'global'
inside	g>f1	x = 'inside'

Escopo Global, Escopo Léxico – ex. 2

```
let x = 'global';

function f1() {
  let x = 'inside';
}

console.log(x); //=> 'global'
f1();
console.log(x); //=> 'global'
```



Scope	Hierarchy	Variable
global	g	x = 'inside1'
inside1	g>f1	
inside2	g>f2	x = 'inside2'

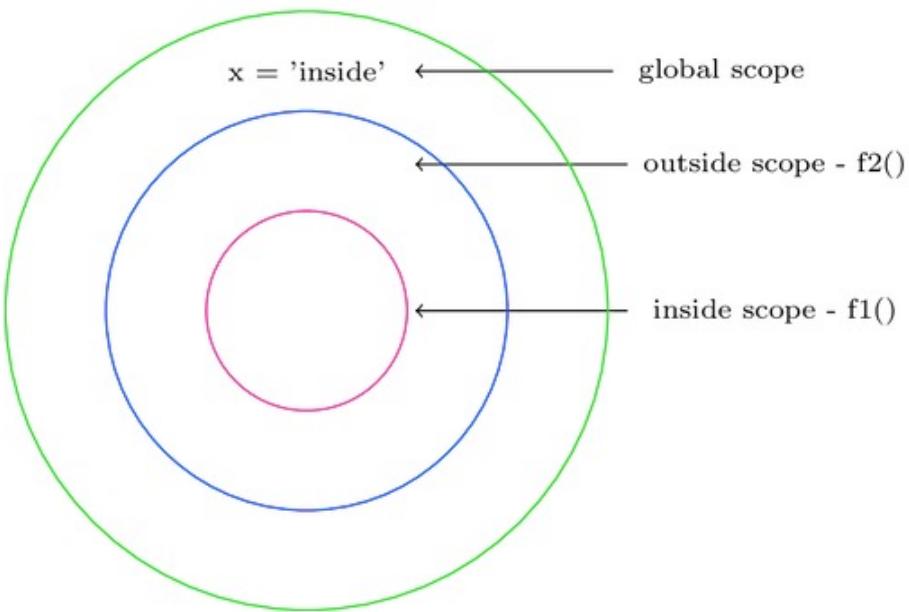
Escopo Global, Escopo Léxico – ex. 3

```
let x = 'global';

function f1() {
  x = 'inside1';
}

function f2() {
  let x = 'inside2';
  f1();
}

console.log(x); //=> 'global'
f2();
console.log(x); //=> 'inside1'
```



Scope	Hierarchy	Variable
global	g	x = 'inside'
inside1	g>f1	
inside2	g>f1>f2	

Escopo Global, Escopo Léxico Interno e externo – ex. 4

```
let x = 'global';

function f2() {
  x = 'outside';
  function f1() {
    x = 'inside';
  }
  f1();
}

console.log(x); //=> 'global'
f2();
console.log(x); //=> 'inside'
```

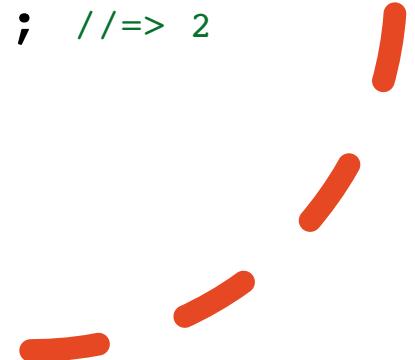
Hoisting

```
console.log(addition(1, 1)); //=> 2
function addition(param1, param2) {
    return param1 + param2;
}
```



Redefinição de Funções

```
function addition(param1, param2) {  
    return param1 + param2;  
}  
  
function addition(param) {  
    return param + 1;  
}  
console.log(addition(1)); //=> 2  
console.log(addition(1, 2)); //=> 2
```

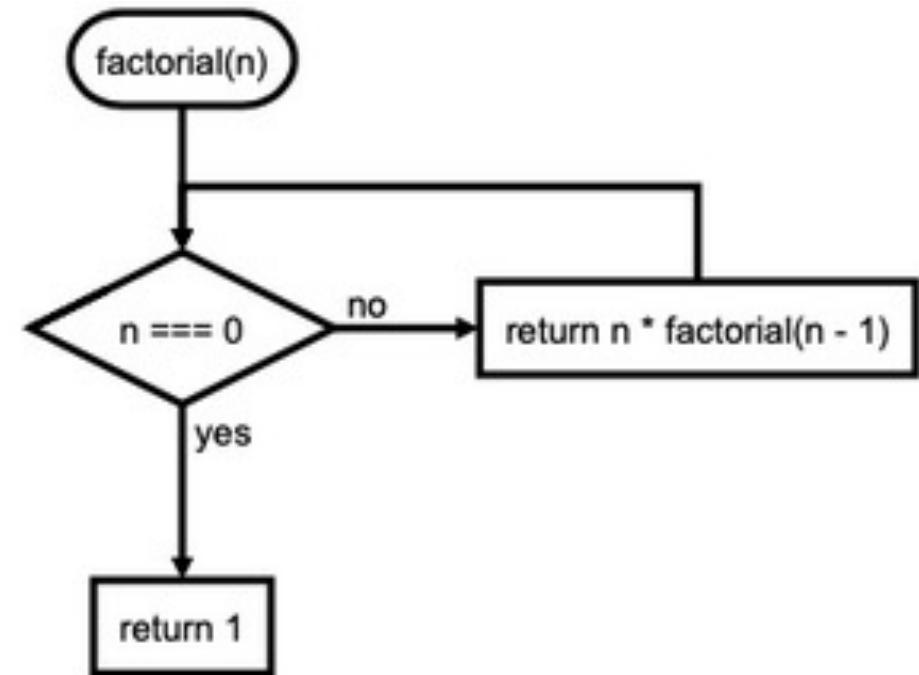


Recursão

$$n! = n * (n-1) * (n-2) * \dots * 1$$

$$4! = 4 * 3 * 2 * 1 = 24$$

```
function factorial(n){  
    return n === 0 ? 1 : n * factorial(n - 1);  
}  
console.log(factorial(4)); //=> 24
```





Contexto de execução

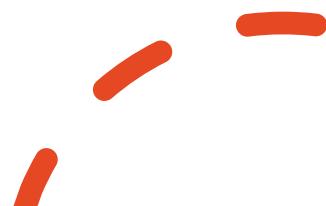
- Sempre que executamos uma função em JavaScript, ela é associada a um contexto de execução.
 - O contexto tem uma propriedade associada chamada `ThisBinding`, acessível através da palavra reservada `this`

```
console.log(this); // Window {...}
```

- Toda função declarada no escopo global possuo o objeto `window` como valor do `this`:

```
function imprimeMeuContextoDeExecucao() {  
    return this;  
}
```

```
imprimeMeuContextoDeExecucao() // Window {...}
```

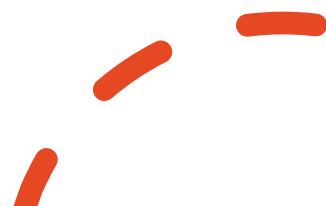




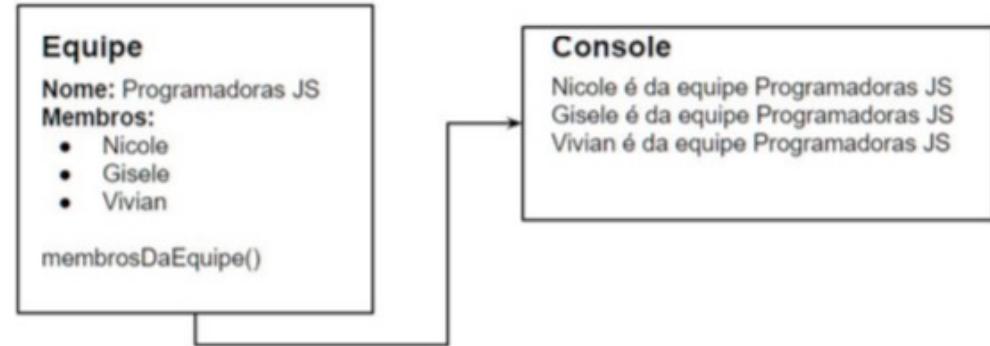
Contexto de execução

- Quando uma função representa um método de um objeto, `this` passa a ser o próprio objeto:

```
var objeto = {  
    meuContexto: function(){  
        console.log(this);  
    };  
};  
  
objeto.meuContexto(); // {meuContexto: [Function: meuContexto] }
```



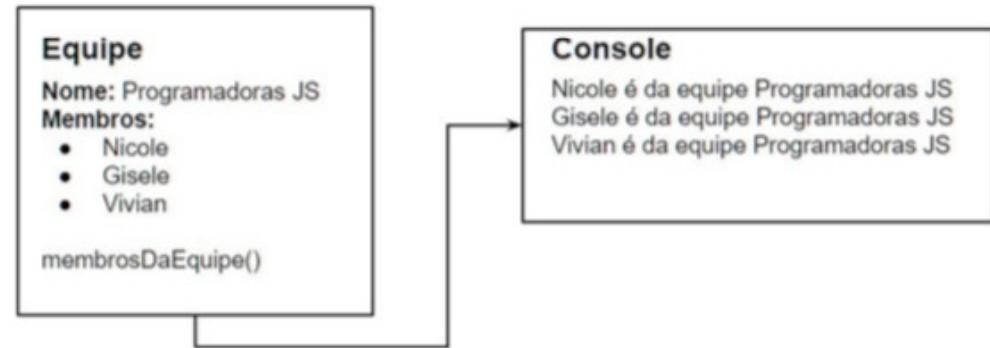
Contexto de execução



- `this` dentro de uma função de `callback` guarda o valor do objeto pai da função `callback` e não da função que recebe o `callback`.

```
const equipe = {  
    nome: 'Programadoras JS'  
    , membros: ['Nicole', 'Gisele', 'Vivian']  
    , membrosDaEquipe: function(){  
        this.membros.forEach(function(membro) {  
            console.log(` ${membro} é da equipe ${this.nome}`);  
        });  
    };  
};  
  
// Error: Cannot read property 'nome' of undefined
```

Contexto de execução



- A função anônima tem contexto de execução diferente do objeto. this tem outro valor.

```
const equipe = {  
    nome: 'Programadoras JS'  
    , membros: ['Nicole', 'Gisele', 'Vivian']  
    , membrosDaEquipe: function(){  
        const self = this;  
        this.membros.forEach(function(membro) {  
            console.log(` ${membro} é da equipe ${self.nome}`);  
        });  
    };  
};
```

Método bind

- Outra forma de resolver o problema anterior é usando o método **bind**

```
function mostraPropDoContexto(nomePropriedade) {  
  console.log(this[nomePropriedade]);  
}  
  
mostraPropDoContexto('location');
```

```
// equivalente a window.location  
{  
  replace: [Function],  
  assign: [Function],  
  hash: '',  
  search: '',  
  pathname: 'blank',  
  port: '',  
  hostname: '',  
  host: '',  
  protocol: 'about:',  
  origin: 'null',  
  href: 'about:blank',  
  ancestorOrigins: {  
    '0': 'https://replit.org',  
    '1': 'https://replit.org',  
    '2': 'https://repl.it'  
  },  
  reload: [Function: reload]  
}
```

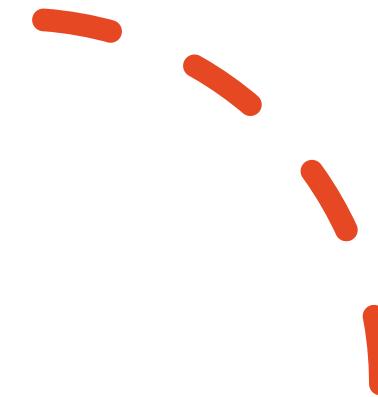
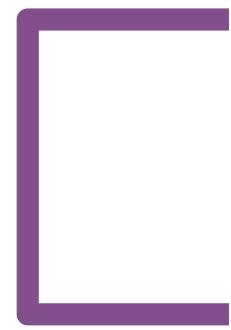
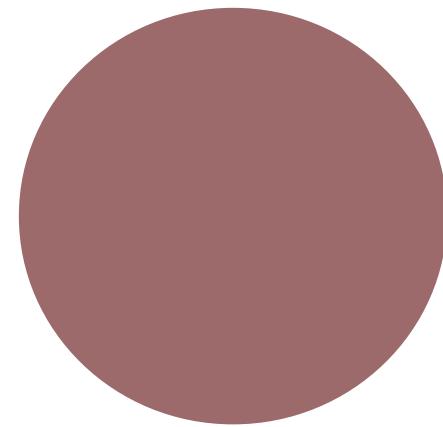
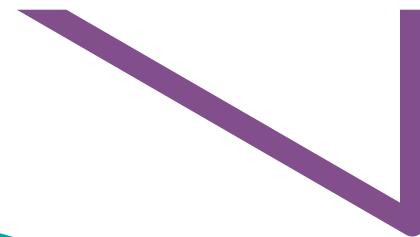


Usando o método bind

```
var mockLocation = {  
  location: 'fake-location'  
}  
var funcao = mostraPropDoContexto.bind(mockLocation);
```



Objetos Literais



JavaScript suporta Orientação a Objetos (por prototipagem)

- Em JavaScript, um objeto é uma coleção de propriedades
- Cada propriedade é uma associação entre chave/valor
- Para criar um objeto em JavaScript:
 - Função construtora ou objetos literais

```
// Função construtora
function Livro(titulo) {
  this.titulo = titulo;
}

var livro = new Livro("Mar Morto");
console.log(livro.titulo); // Mar Morto
```

JavaScript suporta Orientação a Objetos (por prototipagem)

```
// Objeto Literal
var livro = {
    titulo: "Mar Morto";
}
console.log(livro.titulo); // Mar Morto

var outroLivro = livro;
livro.titulo = "O Senhor dos Aneis";

console.log(outroLivro.titulo); // O Senhor dos Aneis
console.log(livro.titulo); // O Senhor dos Aneis
```



Objeto Literal vs JSON

- JSON (JavaScript Object Notation) é um formato baseado na notação de objetos JavaScript
 - As chaves PRECISAM estar entre aspas
 - Strings SEMPRE entre aspas duplas
 - Valores possuem limitações (não podem ser funções)
- Muitas APIs do tipo REST (Representational State Transfer) utilizam o formato JSON para gerar respostas:
 - Redes sociais (Facebook, Twitter, Google+)
 - Plataformas de pagamentos online (PayPal, Cielo, PagSeguro)
 - Serviços de localização (Google Maps, Foursquare)
 - Plataformas de comércio eletrônico (MercadoLivre, Amazon)

