Thiago Delgado Pinto

thiago_dp (at) yahoo.com.br

Algumas Partes Úteis do

JavaScript 5

v0.3

Atualizações:

v0.3 de 25/10/2016 – exemplos adicionados; melhoria da seção sobre modularização.
v0.2 de 12/08/2015 – seção sobre arrays foi estendida.
v0.1 de 21/08/2014 – versão inicial.

agenda

funções

lambdas e closures

callbacks

escopo

arrays

objetos

pseudo-classes

encapsulamento

herança

modularização

funções

declaração comum

```
function digaOi() {
  console.log( "oi" );
}

function soma( x, y ) {
  return x + y;
}

digaOi();
console.log( soma( 10, 20 ) );
```

```
oi
30
```

função anônima para variável

```
var digaOi = function() {
  console.log( "oi" );
};

var soma = function( x, y ) {
  return x + y;
};

digaOi();
console.log( soma( 10, 20 ) );
```

```
oi
30
```

declaração e chamada direta


```
( function() {
  console.log( "oi" );
} )();
```

```
oi
```

declaração e chamada direta


```
console.log(
      ( function( x, y ) {
        return x + y;
      } )( 10, 20 )
);
```

```
30
```

lambdas e closures

lambda

lambda é uma declaração de uma expressão.

comumente, essa expressão é uma função anônima.

portanto, é correto dizer que uma função anônima é um lambda

mas não o contrário

closure

closure é quem recebe a declaração de um lambda

ou seja, uma **variável** ou **objeto** que recebe um lambda.

muitos desenvolvedores chamam (equivocadamente) funções anônimas de **closures**, apesar de serem **lambdas**

logo, muitas vezes esses termos são tratados como sinônimos

- apesar de haver essa notável diferença, podemos relevá-las

declaração e chamada direta


```
console.log(
      ( function( x, y ) {
        return x + y;
      } )( 10, 20 )
);
```

```
30
```

callbacks

callback


```
function fazAlgo( funcao ) {
  console.log( "Antes" );
  funcao();
  console.log( "Depois" );
}

function digaOi() { console.log( "oi" ); }

fazAlgo( digaOi );
```

é passado somente o nome da função, **sem** parênteses

```
Antes
oi
Depois
```

callback


```
function fazOperacao( funcao ) {
  console.log( "Antes" );
  console.log( funcao( 10, 20 ) );
  console.log( "Depois" );
}

function soma( x, y ) { return x + y; }

fazOperacao( soma );
```

```
Antes
30
Depois
```

função anônima em callback


```
function fazAlgo( funcao ) {
  console.log( "Antes" );
  funcao();
  console.log( "Depois" );
}

fazAlgo( function() { console.log( "oi" ); } );
```

```
Antes
oi
Depois
```

função anônima em callback


```
function fazOperacao( funcao ) {
  console.log( "Antes" );
  console.log( funcao( 10, 20 ) );
  console.log( "Depois" );
}

fazOperacao( function( x, y ) { return x + y; } );
```

```
Antes
30
Depois
```

número de argumentos de uma função

```
function digaOi() {
  console.log( "oi" );
}

var soma = function( x, y ) {
  return x + y;
};

console.log( digaOi.length );
console.log( soma.length );
console.log( function( a, b, c ) { }.length );
```

```
0
2
3
```

acessando os argumentos de uma função


```
function soma( x, y ) {
  console.log( arguments.length + " argumentos" );
  console.log( arguments[ 0 ] );
  console.log( arguments[ 1 ] );
  return x + y; // arguments[ 0 ] + arguments[ 1 ]
}

soma( 10, 20 );
```

`arguments` é uma palavra reservada. Procure nunca usá-la como nome de parâmetro.

```
2 argumentos
10
20
```

acessando os argumentos de uma função


```
function soma() {
  var resultado = 0;
  for ( var i = 0; i < arguments.length; ++i ) {
    resultado += arguments[ i ];
  }
  return resultado;
}
console.log( soma() );
console.log( soma( 10 ) );
console.log( soma( 10, 20 ) );
console.log( soma( 10, 20, 30 ) );
console.log( soma( 10, 20, 30, 40 ) );
```

```
0
10
30
60
100
```

comentários

lambdas com invocação imediata são úteis para evitar o escopo global

- declarações ficam no escopo da função e não no global

veremos isso a seguir

escopo

escopo global e de função

diferentemente de algumas linguagens, há dois tipos de escopo em JavaScript: **global** e de **função**

variáveis ou funções no **escopo global** são acessíveis por qualquer função

variáveis ou funções no **escopo de uma função** são acessíveis somente dentro dessa função

escopo global

```
var i = 0;

function fazAlgo( x ) {
  function incrementa() {
    return ++i;
  }
  return x + incrementa();
}

console.log( fazAlgo( 10 ) );
console.log( i );
```

```
11
1
```

escopo global

```
{
  var i = 0;
}

function fazAlgo( x ) {
  function incrementa() {
    return ++i;
  }
  return x + incrementa();
}

console.log( fazAlgo( 10 ) );
console.log( i );
```

```
11
1
```

escopo global

```
for ( var i = 0; i < 5; ++i ) {
  // nada
}

function fazAlgo( x ) {
  function incrementa() {
    return ++i;
  }
  return x + incrementa();
}

console.log( fazAlgo( 10 ) );
console.log( i );
```

```
16
6
```

escopo de função

```
var i = 0;

function fazAlgo( x ) {
  var i = 10;
  function incrementa() {
    return ++i;
  }
  return x + incrementa();
}

console.log( fazAlgo( 10 ) );
console.log( i );
```

```
21
0
```

escopo de função

```
function fazOutraCoisa() {
  if ( 0 == arguments.length ) {
    for ( var i = 0; i < 5; i++ ) {
       // nada
    }
  }
  console.log( i ); // pode ser acessível mesmo aqui!
}

fazOutraCoisa();
fazOutraCoisa( 100 );
```

```
5
undefined
```

evite o escopo global

usar o escopo global aumenta as chances de colisões de nome com outros scripts

- seus próprios scripts
- bibliotecas de terceiros
- widgets
- web analytics
- ...

além disso, variáveis globais podem ser alteradas em lugares adversos sem que você perceba, tornando difícil encontrar a causa de certos defeitos

use lambdas com invocação imediata

```
( function() {

  /// Retorna um valor inteiro >= min e <= max
  function randomInt( min, max ) {
    return Math.floor( Math.random() * ( max - min ) ) + min;
  }

  // Só precisou da função randomInt para esse trecho.
  // Logo, ela não foi declarada como global.
  console.log( randomInt( 0, 100 ) );

} )();
```

arrays

declaração usando colchetes

```
var a = [];

var b = [ 1, 2, 3 ];

var c = [ "java", "script" ];

console.log( a );
console.log( b );
console.log( c );
```

```
[]
[1,2,3]
["java", "script"]
```

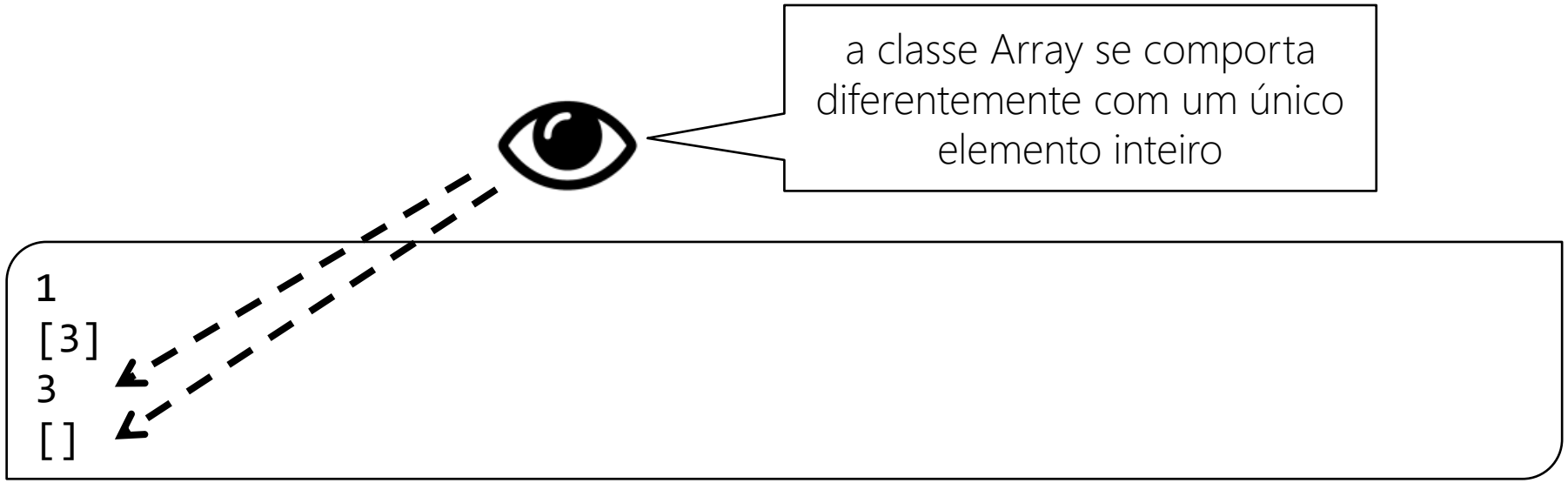
declaração instanciando classe

```
var a = new Array();

var b = new Array( 1, 2, 3 );

var c = new Array( "java", "script" );

console.log( a );
console.log( b );
console.log( c );
```

```
[]
[1,2,3]
["java", "script"]
```

porém....

```
var a = [ 3 ];
var b = new Array( 3 ); // efeito indesejado!

console.log( a.length );
console.log( a );

console.log( b.length );
console.log( b );
```



comentários

ao passar um único argumento do tipo **inteiro** para instanciar um objeto da classe **`Array`**, a classe entende que esse argumento é o **tamanho** do array e não seu único valor

logo, use **sempre a notação de colchetes**

você escreve menos, não perde legibilidade e o array continua sendo um objeto

adicionando elementos ao início

```
var frutas = [ "uva", "maçã" ];

frutas.unshift( "pêra" );
frutas.unshift( "goiaba" );

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

```
4
["goiaba", "pêra", "uva", "maçã"]
```

adicionando elementos ao fim – via método

```
var frutas = [ "uva", "maçã" ];

frutas.push( "pêra" );
frutas.push( "goiaba" );

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

```
4
["uva", "maçã", "pêra", "goiaba"]
```

adicionando elementos ao fim – via índice


```
var frutas = [ "uva", "maçã" ];

frutas[ 2 ] = "pêra";
frutas[ 3 ] = "goiaba";

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

```
4
["uva", "maçã", "pêra", "goiaba"]
```

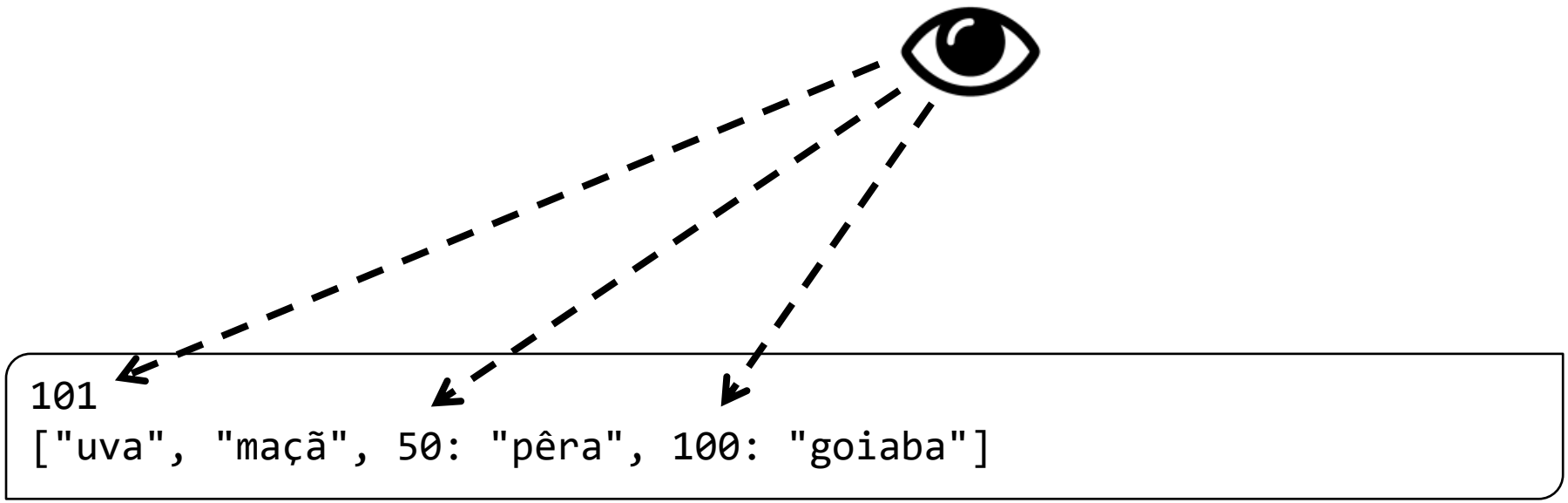
adicionando elementos ao fim – via índice


```
var frutas = [ "uva", "maçã" ];

frutas[ frutas.length ] = "pêra";
frutas[ frutas.length ] = "goiaba";

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

```
4
["uva", "maçã", "pêra", "goiaba"]
```

adicionando elementos nos índices desejados

```
var frutas = [ "uva", "maçã" ];

frutas[ 50 ] = "pêra";
frutas[ 100 ] = "goiaba";

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



concatenando

```
var gerais = [ "Bob", "Ana" ];
var regionais = [ "Pedro", "Marcio", "Carla", "Bia" ];

var todosOsGerentes = gerais.concat( regionais );
console.log( todosOsGerentes );
```

```
["Bob", "Ana", "Pedro", "Marcio", "Carla", "Bia"]
```

unindo elementos em uma string

```
var mensagens = [
      "Por favor, informe o nome.",
      "Por favor, informe o CPF."
      ];
var msg = mensagens.join( "\n" );
console.log( msg );
```

```
Por favor, informe o nome.
Por favor, informe o CPF.
```

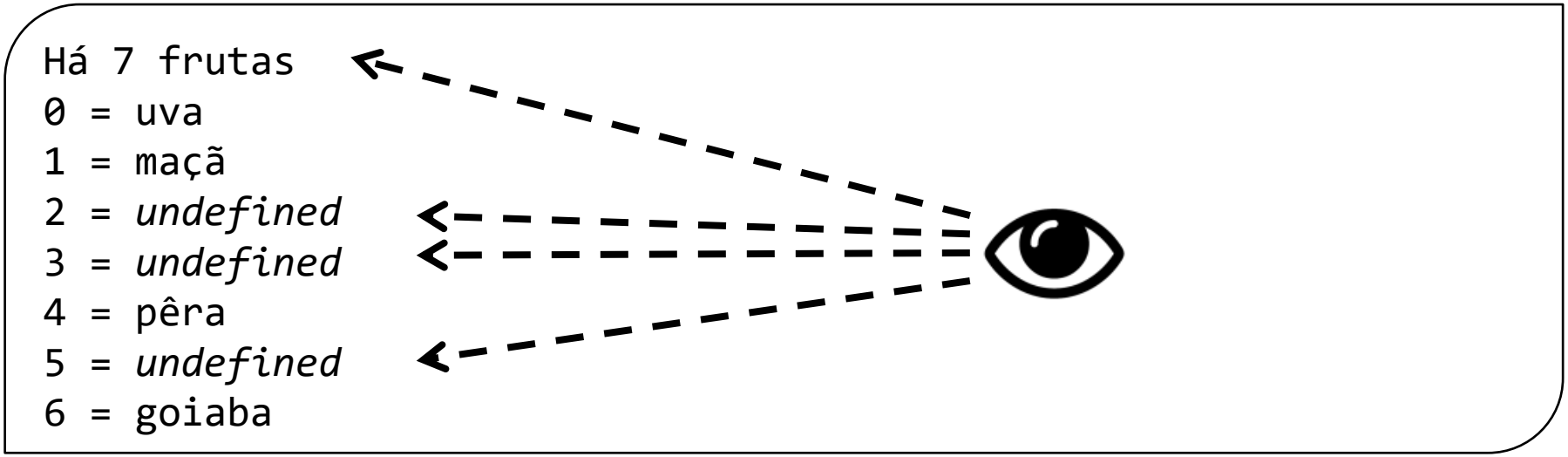
percorrendo elementos


```
var frutas = [ "uva", "maçã", "pêra", "goiaba" ];

console.log( "Há " + frutas.length + " frutas" );

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

```
Há 4 frutas
0 = uva
1 = maçã
2 = pêra
3 = goiaba
```

percorrendo elementos


```
var frutas = [ "uva", "maçã" ];
frutas[ 4 ] = "pêra";
frutas[ 6 ] = "goiaba";
console.log( "Há " + frutas.length + " frutas" );
for ( var i = 0; i < frutas.length; ++i ) {
  console.log( i + " = " + frutas[ i ] );
}
```



percorrendo elementos


```
var frutas = [ "uva", "maçã", "pêra", "goiaba" ];
console.log( "Há " + frutas.length + " frutas" );

// Usando o laço for in
for ( var i in frutas ) {
  console.log( i + " = " + frutas[ i ] );
}
```

```
Há 4 frutas
0 = uva
1 = maçã
2 = pêra
3 = goiaba
```

percorrendo elementos


```
var frutas = [ "uva", "maçã" ];
frutas[ 4 ] = "pêra";
frutas[ 6 ] = "goiaba";
console.log( "Há " + frutas.length + " frutas" );

// Usando o laço for in
for ( var i in frutas ) {
  console.log( i + " = " + frutas[ i ] );
}
```

```
Há 7 frutas
0 = uva
1 = maçã
4 = pêra
6 = goiaba
```

comentários

use sempre o **for ... in**, exceto se você precisar percorrer os índices de um array que contém valores **indefinidos**

tenha atenção ao comprimento do array ao adicionar elementos em índices não contíguos

o comprimento de um array será sempre o índice de seu **último elemento + 1**

invertendo uma lista

```
var frutas = [ "uva", "maçã", "pêra", "goiaba" ];

frutas.reverse();

for ( var i in frutas ) {
  console.log( i + " = " + frutas[ i ] );
}
```

```
0 = goiaba
1 = pêra
2 = maçã
3 = uva
```

ordenando elementos

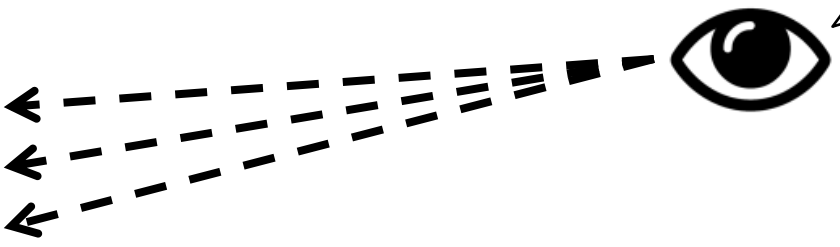

```
var frutas = [ "uva", "maçã", "pêra", "goiaba" ];

frutas.sort();

for ( var i in frutas ) {
  console.log( i + " = " + frutas[ i ] );
}
```

```
0 = goiaba
1 = maçã
2 = pêra
3 = uva
```

ordenando elementos


```
var frutas = [ "uva", "maçã" ];
frutas[ 4 ] = "pêra";
frutas[ 6 ] = "goiaba";

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

```
0 = goiaba
1 = maçã
2 = pêra
3 = uva
4 = undefined
5 = undefined
6 = undefined
```



elementos indefinidos são movidos para o fim

ordenando elementos decrescentemente

```
var frutas = [ "uva", "maçã", "pêra", "goiaba" ];

frutas.sort();
frutas.reverse();

for ( var i in frutas ) {
  console.log( i + " = " + frutas[ i ] );
}
```

```
0 = uva
1 = pêra
2 = maçã
3 = goiaba
```

ordenando elementos inteiros


```
var numeros = [ 100, 10, 1, 50, 20, 2 ];

numeros.sort(); // comportamento indesejado !

for ( var i in numeros ) {
  console.log( i + " = " + numeros[ i ] );
}
```

por default, inteiros são ordenados como strings!

```
1
10
100
2
20
50
```

ordenando elementos inteiros


```
var numeros = [ 100, 10, 1, 50, 20, 2 ];

function compara( a, b ) {
// Retorno: negativo = menor, 0 = igual, positivo = maior
      return a - b;
}

numeros.sort( compara );
for ( var i in numeros ) { console.log( numeros[ i ] ); }
```

```
1
2
10
20
50
100
```

ordenando elementos inteiros decrescentemente

```
var numeros = [ 100, 10, 1, 50, 20, 2 ];

function compara( a, b ) {
// Retorno: negativo = menor, 0 = igual, positivo = maior
      return b - a;
}

numeros.sort( compara );
for ( var i in numeros ) { console.log( numeros[ i ] ); }
```

```
100
50
20
10
2
1
```

removendo o último elemento

```
var frutas = [ "uva", "maçã", "pêra", "goiaba" ];

var removido = frutas.pop();

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

```
3
["uva", "maçã", "pêra"]
goiaba
```

removendo o primeiro elemento

```
var frutas = [ "uva", "maçã", "pêra", "goiaba" ];

var removido = frutas.shift();

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

```
3
["maçã", "pêra", "goiaba"]
uva
```

removendo elementos


```
var frutas = [ "uva", "maçã", "pêra", "goiaba" ];

var indiceInicial = 1; // maçã
var quantosQuerRemover = 1; // só maçã
var removidos = frutas.splice(
  indiceInicial, quantosQuerRemover );

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

```
3
["uva", "pêra", "goiaba"]
["maçã"]
```

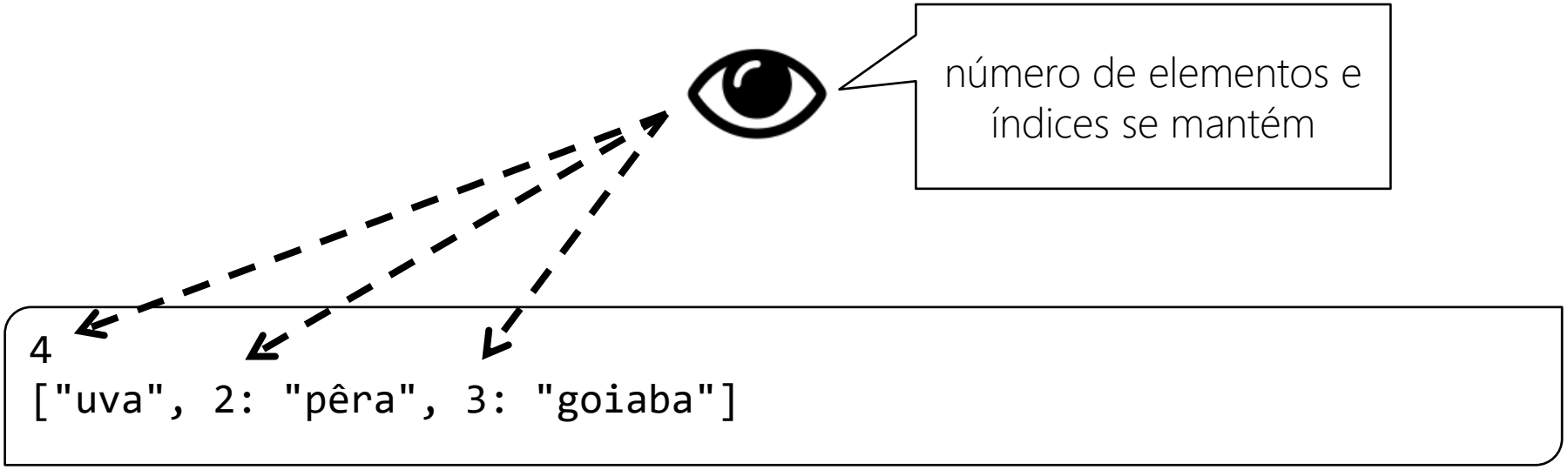
removendo elementos


```
var frutas = [ "uva", "maçã", "pêra", "goiaba" ];

var indiceInicial = 0; // uva
var quantosQuerRemover = 3; // uva, maçã, pêra
var removidos = frutas.splice(
  indiceInicial, quantosQuerRemover );

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

```
1
["goiaba"]
["uva", "maçã", "pêra"]
```

indefinindo elementos

```
var frutas = [ "uva", "maçã", "pêra", "goiaba" ];

delete frutas[ 1 ]; // indefine o elemento no índice 1

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



procurando elementos

```
var frutas = [ "uva", "maçã", "pêra", "uva", "goiaba" ];

var i = frutas.indexOf( "uva" );
var j = frutas.lastIndexOf( "uva" );

console.log( i );
console.log( j );
```

```
0
3
```

em suma

`concat`	retorna a concatenação de dois ou mais arrays
`indexOf`	retorna a posição de um elemento do array
`join`	une todos os elementos do array em uma string
`lastIndexOf`	retorna a última posição de um elemento do array
`pop`	remove o último elemento do array e retorna esse elemento
`push`	adiciona um elemento ao fim do array e retorna o novo comprimento
`reverse`	inverte a ordem do array
`shift`	remove o primeiro elemento do array e retorna esse elemento
`slice`	seleciona parte do array e retorna um novo array
`sort`	ordena os elementos do array
`splice`	adiciona ou remove um elemento do array
`toString`	retorna o array como uma string
`unshift`	adiciona um elemento ao início do array e retorna o novo comprimento
`valueOf`	retorna um novo array com os mesmos valores, mas índices contíguos

objetos

declaração em variável

```
var telefone = {
  ddd: "22",
  numero: "2527-1727",
  conteudo: function() {
    return '(' + this.ddd + ') ' + this.numero;
  }
};

console.log( telefone.ddd );
console.log( telefone.numero );
console.log( telefone.conteudo() );
```

```
22
2527-1727
(22) 2527-1727
```

agregando propriedades

```
var telefone = {};

telefone.ddd = "22";
telefone.numero = "2527-1727";
telefone.conteudo = function() {
      return '(' + this.ddd + ') ' + this.numero;
};

console.log( telefone.ddd );
console.log( telefone.numero );
console.log( telefone.conteudo() );
```

```
22
2527-1727
(22) 2527-1727
```

agregando propriedades como um array

```
var telefone = {};

telefone[ "ddd" ] = "22";
telefone[ "numero" ] = "2527-1727";
telefone[ "conteudo" ] = function() {
      return '(' + this.ddd + ') ' + this.numero;
};

console.log( telefone.ddd );
console.log( telefone.numero );
console.log( telefone.conteudo() );
```

```
22
2527-1727
(22) 2527-1727
```

acessando propriedades como um array

```
var telefone = {
  ddd: "22",
  numero: "2527-1727",
  conteudo: function() {
    return '(' + this.ddd + ') ' + this.numero;
  }
};

console.log( telefone[ "ddd" ] );
console.log( telefone[ "numero" ] );
console.log( telefone[ "conteudo" ]() );
```

```
22
2527-1727
(22) 2527-1727
```

acessando as propriedades em um laço


```
var telefone = {
  ddd: "22",
  numero: "2527-1727",
  conteudo: function() {
    return '(' + this.ddd + ') ' + this.numero;
  }
};

for ( var p in telefone ) {
  console.log( p );
}
```

```
ddd
numero
conteudo
```

acessando as propriedades em um laço


```
var telefone = {
  ddd: "22",
  numero: "2527-1727",
  conteudo: function() {
    return '(' + this.ddd + ') ' + this.numero;
  }
};

for ( var p in telefone ) {
  console.log( p + " = " + telefone[ p ] );
}
```

```
ddd = 22
numero = 2527-1727
conteudo = function() {
  return '(' + this.ddd + ') ' + this.numero;
 }
```

acessando as propriedades em um laço


```
var telefone = {
  ddd: "22",
  numero: "2527-1727",
  conteudo: function() {
    return '(' + this.ddd + ') ' + this.numero;
  }
};

for ( var p in telefone ) {
  console.log( typeof telefone[ p ] );
}
```

```
string
string
function
```

acessando as propriedades em um laço


```
var telefone = {
  ddd: "22",
  numero: "2527-1727",
  conteudo: function() {
    return '(' + this.ddd + ') ' + this.numero;
  }
};
for ( var p in telefone ) {
      var valor = "function" === typeof telefone[ p ]
            ? telefone[ p ]() : telefone[ p ];
        console.log( p + " = " + valor );
}
```

```
ddd = 22
numero = 2527-1727
conteudo = (22) 2527-1727
```

pseudo-classes

função como uma classe


```
function Pessoa( nome, sobrenome ) {
  this.nome = nome;
  this.sobrenome = sobrenome;
  this.nomeCompleto = function() {
    return this.nome + ' ' + this.sobrenome;
  };
}
```

ao usar o operador **new**, a função é entendida como um construtor.

```
var p = new Pessoa( 'Bob', 'Marley' );
console.log( p.nome );
console.log( p.nomeCompleto() );
```

```
Bob
Bob Marley
```

função como uma classe


```
var Pessoa = function( nome, sobrenome ) {
  this.nome = nome;
  this.sobrenome = sobrenome;
  this.nomeCompleto = function() {
    return this.nome + ' ' + this.sobrenome;
  };
}
```

ao usar o operador **new**, a função é entendida como um construtor.

```
var p = new Pessoa( 'Bob', 'Marley' );
console.log( p.nome );
console.log( p.nomeCompleto() );
```

```
Bob
Bob Marley
```

observação

Ao criar uma **função** que será usada como uma **classe**, o JavaScript disponibiliza um objeto **`prototype`**, com propriedades e métodos úteis, como:

```
constructor
apply()
call()
bind()
toString()
```

prototype

`constructor`

aponta para o construtor da classe

`apply( obj, [ argsArray ] )`

permite chamar um método de um objeto passando argumentos como um array

`call( obj, [ arg1, ..., argN ] )`

permite chamar um método de um objeto passando argumentos

`bind( obj, [ arg1, ..., argN ] )`

cria uma nova função cujo "this" será o objeto informado e irá conter os argumentos fornecidos

`toString()`

retorna "[object tipo]" onde tipo é o tipo do objeto

acessando o construtor

```
function Pessoa( nome, sobrenome ) {
  this.nome = nome;
  this.sobrenome = sobrenome;
  this.nomeCompleto = function() {
    return this.nome + ' ' + this.sobrenome;
  };
}
var p = new Pessoa( 'Bob', 'Marley' );
console.log( p.constructor );
console.log( p.constructor === Pessoa );
console.log( Pessoa.prototype.constructor === p.constructor );
```

o construtor do objeto faz referência à sua classe

```
function Pessoa( nome, sobrenome )
true
true
```

criando um objeto pelo construtor


```
function Pessoa( nome, sobrenome ) {
  this.nome = nome;
  this.sobrenome = sobrenome;
  this.nomeCompleto = function() {
    return this.nome + ' ' + this.sobrenome;
  };
}


var p = new Pessoa.prototype.constructor( 'Bob', 'Marley' );
var p2 = new p.constructor( 'Ziggy', 'Marley' );
console.log( p.nomeCompleto() );
console.log( p2.nomeCompleto() );
```

```
Bob Marley
Ziggy Marley
```

definindo a classe com prototype

```
function Pessoa( nome, sobrenome ) {
  this.nome = nome;
  this.sobrenome = sobrenome;
}

Pessoa.prototype.nomeCompleto = function() {
  return this.nome + ' ' + this.sobrenome;
};

Pessoa.prototype.digaOla = function() {
  return 'Olá, ' + this.nomeCompleto();
};


var p = new Pessoa( 'Bob', 'Marley' );
console.log( p.digaOla() );
```

prototype é uma propriedade especial dos objetos JavaScript

fazendo referências à métodos


```
function Pessoa( nome, sobrenome ) {
  this.nome = nome;
  this.sobrenome = sobrenome;
}
Pessoa.prototype.nomeCompleto = function() {
  return this.nome + ' ' + this.sobrenome;
};
var p = new Pessoa( 'Bob', 'Marley' );
var nc = p.nomeCompleto;
nc();
nc.call( p );
```

A variável **nc** referencia o método, mas não conhece o objeto **p**. Logo, o **this**, que é usado internamente pelo método, não funciona. Ao usar **call**, o objeto fornecido é usado como **this**.

```
undefined undefined
Bob Marley
```

fazendo referências à métodos


```
function Calculo() {
  this.soma = function( arr ) {
    for ( var s = 0, len = arr.length, i = 0; i < len; ++i ) {
      s += arr[ i ];
    }
    return s;
  };
  this.media = function( arr ) {
    var len = arr.length;
    return len > 0 ? this.soma( arr ) / len : 0;
  };
}

var c = new Calculo();
var s = c.soma;
s( [ 10, 20, 30 ] );          // 60
var m = c.media;
m( [ 10, 20, 30 ] );          // TypeError: soma is not a function
m.call( c, [ 10, 20, 30 ] );  // 20
```

encapsulamento

atributos ou métodos privados


```
function Calculo() {
  var soma = function( arr ) {
    for ( var s = 0, len = arr.length, i = 0; i < len; ++i ) {
      s += arr[ i ];
    }
    return s;
  };

  this.media = function( arr ) {
    var len = arr.length;
    return len > 0 ? soma( arr ) / len : 0;
  };
}
var c = new Calculo();
console.log( c.media( [ 10, 20, 30 ] ) );
console.log( c.soma( [ 10, 20, 30 ] ) ); // erro!
```

Declarações com **var** são todas **privadas** e não acessíveis com **this** dentro da classe.

```
20
Uncaught TypeError: c.soma is not a function
```

atributos ou métodos privados


```
function Pessoa( nome, sobrenome ) {
  this.nome = nome;
  var _sobrenome = sobrenome;

  this.nomeCompleto = function() {
    return this.nome + ' ' + _sobrenome;
  };
}

var p = new Pessoa( 'Bob', 'Marley' );
console.log( p.nome );
console.log( p._sobrenome );
console.log( p.nomeCompleto() );
```

Declarações com **var** são todas **privadas** e não acessíveis com **this** dentro da classe.

```
Bob
undefined
Bob Marley
```

getters e setters


```
function Pessoa( nome, sobrenome ) {
  var _nome = nome;
  var _sobrenome = sobrenome;

  this.getNome = function() { return _nome; }
  this.setNome = function( valor ) { _nome = valor; }

  this.getSobrenome = function() { return _sobrenome; }
  this.setSobrenome = function( valor ) { _sobrenome = valor; }

  this.nomeCompleto = function() { return _nome + ' ' + _sobrenome; };
}

var p = new Pessoa( 'Bob', 'Dylan' );
p.setSobrenome( 'Marley' );
console.log( p.getSobrenome() );
```

Não é comum, em JavaScript, a declaração de métodos *getters* e *setters* separados.

getters e setters


```
function Pessoa( nome, sobrenome ) {
  var _nome = nome;
  var _sobrenome = sobrenome;

  this.nome = function( valor ) { // getter/setter
    if ( valor ) { _nome = valor; }
    return _nome;
  };

  this.sobrenome = function( valor ) { // getter/setter
    if ( valor ) { _sobrenome = valor; }
    return _sobrenome;
  };

  this.nomeCompleto = function() { return _nome + ' ' + _sobrenome; };
}

var p = new Pessoa( 'Bob', 'Marley' );
p.sobrenome( 'Dylan' );
console.log( p.sobrenome() ); // Dylan
```

É mais comum a declaração em um só método.

herança

herança

```
function Forma() {
  this.x = 0;
  this.y = 0;
}
Forma.prototype.mover = function( x, y ) {
  this.x += x;
  this.y += y;
};

function Retangulo() {
  Forma.call( this ); // Chama o construtor pai
}
// Cria um novo prototype que é a cópia do outro
Retangulo.prototype = Object.create( Forma.prototype );
// Ajusta o construtor para a classe correta
Retangulo.prototype.constructor = Retangulo;

var r = new Retangulo();
r.mover( 50, 100 );
console.log( r.x + ' ' + r.y ); // 50 100
```

herança e sobrescrita de método

```
function Animal( nome ) {
  console.log( 'Sou ' + nome );
}
Animal.prototype.som = function() { return '?'; };

function Gato( nome ) {
  Animal.call( this, nome ); // Chama o construtor pai
}
// Cria um novo prototype que é a cópia do outro
Gato.prototype = Object.create( Animal.prototype );
// Ajusta o construtor para a classe correta
Gato.prototype.constructor = Gato;
// Sobrescrita de método
Gato.prototype.som = function() { return 'Miau'; };

var a = new Animal( 'Rex' );          // imprime "Sou Rex"
console.log( a.som() );               // imprime "?"
var g = new Gato( 'Garfield' );       // imprime "Sou Garfield"
console.log( g.som() );               // imprime "Miau"
```

chamando métodos da classe pai (super)


```
function Animal() {}
Animal.prototype.som = function() { return '?'; };

function Gato() {
  Animal.call( this ); // Chama o construtor pai
}
Gato.prototype = Object.create( Animal.prototype ); // Herda
Gato.prototype.constructor = Gato; // Ajusta o construtor

// Sobrescreve o método pai
Gato.prototype.som = function() {
        // Chama o método pai
        var somPai = Animal.prototype.som.call( this );
        // Faz outra coisa
        return 'Miau' + somPai;
};

var g = new Gato( 'Garfield' );
console.log( g.som() );       // imprime "Miau?"
```

chamando métodos da classe pai (super)


```
function Animal() {}
Animal.prototype.som = function() { return '?'; };

function Gato() {
  Animal.call( this ); // construtor da classe pai
}
Gato.prototype = Object.create( Animal.prototype ); // Herda
Gato.prototype.constructor = Gato; // Ajusta o construtor
// Facilitador "parent", já que "self" é reservado para uso futuro
Gato.prototype.parent = Animal.prototype;

// Sobrescrita de método
Gato.prototype.som = function() {
        var somPai = this.parent.som.call( this );
        return 'Miau' + somPai;
};

var g = new Gato( 'Garfield' );
console.log( g.som() );       // imprime "Miau?"
```

modularização

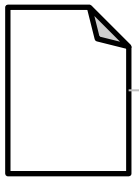
modularização

como vimos, podemos usar o **escopo de função** para realizarmos nossas declarações, evitando o escopo global

as **declarações** que devem ser **visíveis externamente** podem ser atribuídas a um **único objeto global**

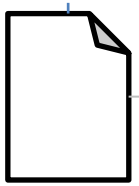
esse objeto global funciona como um ***namespace*** ou **módulo principal**

estrutura de exemplo



app.js

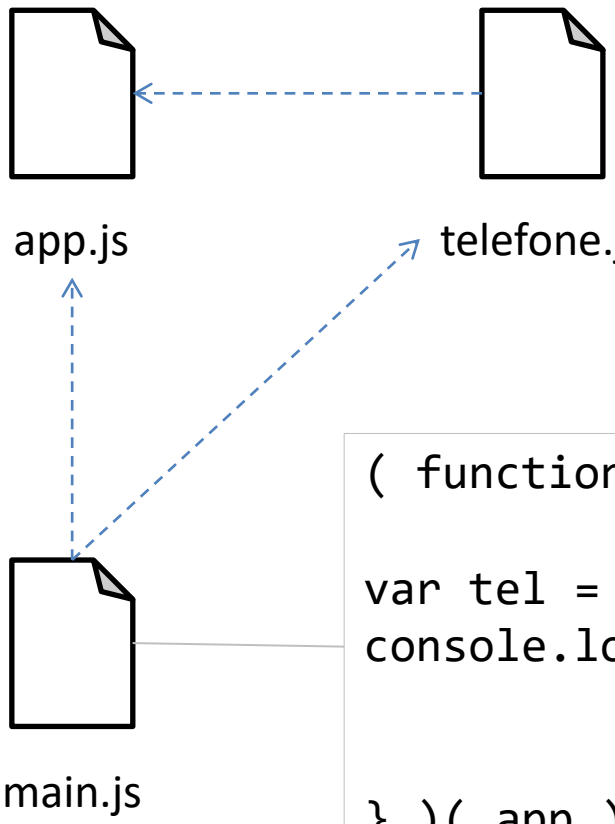
```
// "app" será nosso módulo/namespace principal
var app = {}; // global
```



telefone.js

```
// Recebe "app" como argumento
( function( app ) {

    // Nossa classe
    function Telefone( ddd, numero ) {
        ...
    }

    // Coloca a declaração em "app"
    app.Telefone = Telefone

} )( app ); // Passa o "app" global
```

usando nossa declaração




```
( function( app ) {

var tel = new app.Telefone( "22", "2527-1727" );
console.log( tel.conteudo() );


} )( app );
```

comentários

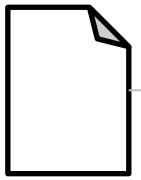
no exemplo anterior, "**app**" funcionou como nosso *namespace* para declarações de uma aplicação JavaScript

caso necessário, poderíamos criar ***subnamespaces*** em **app** para organizar as declarações dentro de assuntos específicos

esses *subnamespaces* são equivalentes à **submódulos**

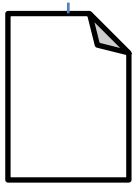
exemplo a seguir

exemplos

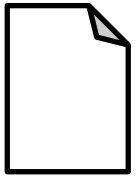


app.js

```
var app = {}; // namespace global

// subnamespace "fin" como módulo financeiro
app.fin = {};

// subnamespace "rh" como módulo recursos humanos
app.rh = {};
```



tributo.js



curriculo.js

```
( function( app ) {

function Tributo( ... ) {
    ...
}

app.fin.Tributo = Tributo;

} )( app );
```

```
( function( app ) {

function Curriculo( ... ) {
   ...
}

app.rh.Curriculo = Curriculo;

} )( app );
```

dependências externas

devemos usar **injeção de dependências** externas

ou seja, elas devem ser passadas como **argumentos**, de maneira a serem usadas como **variáveis locais**

regra: sempre transforme variáveis globais em locais

exemplo


```
// livro-ctrl.js
( function( app ) { // local "app"

      // dependência de um "jQuery" externo
      // só precisa ao instanciar LivroController
      function LivroController( jQuery ) {
            var $ = jQuery; // atalho
            ...
            this.desenhar = function( livro ) {
                  $( '#nome' ).val( livro.nome );
                  ...
            };
      };

      app.LivroController = LivroController;

} )( app ); // global "app" injetado
```

exemplo

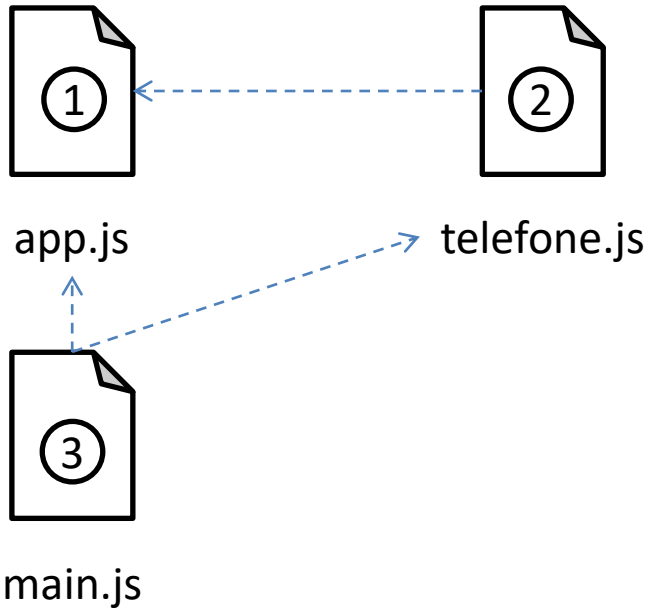

```
// usando a controladora, em algum arquivo javascript
( function( app, $ ) { // locais "app" e "$"

    var ctrl = new app.LivroController( $ );
    var livro = new app.Livro( "O Hobbit", "J.R.R. Tolkien" );
    ctrl.desenhar( livro );

} )( app, jQuery ); // globais "app" e "jQuery" injetados
```

uso no HTML

ordenação de dependências

infelizmente o JavaScript 5 não possui um modo nativo para **inclusão de arquivos** (como o #include de C)

ao incluir os arquivos em um arquivo HTML, deve-se respeitar a **ordem** das dependências




```
<script type="text/javascript"
  src="app.js" ></script>

<script type="text/javascript"
  src="telefone.js" ></script>

<script type="text/javascript"
  src="main.js" ></script>
```

1 2 3

index.html

dependências externas

convém sempre declarar arquivos de dependências externas **antes** dos arquivos da aplicação

obviamente também respeitando a ordem de dependência

```
<!-- Externo -->
<script type="text/javascript" src="jquery.js" ></script>
<script type="text/javascript" src="bootstrap.js" ></script>
...

<!-- Aplicação -->
<script type="text/javascript" src="app.js" ></script>
<script type="text/javascript" src="telefone.js" ></script>
...
<script type="text/javascript" src="main.js" ></script>
```

referências

Mozilla Developers' Network.
https://developer.mozilla.org

Meyers, Scott. **Lambdas vs. Closures.** Disponível em: http://scottmeyers.blogspot.com.br/2013/05/lambdas-vs-closures.html. Acesso em agosto/2014.