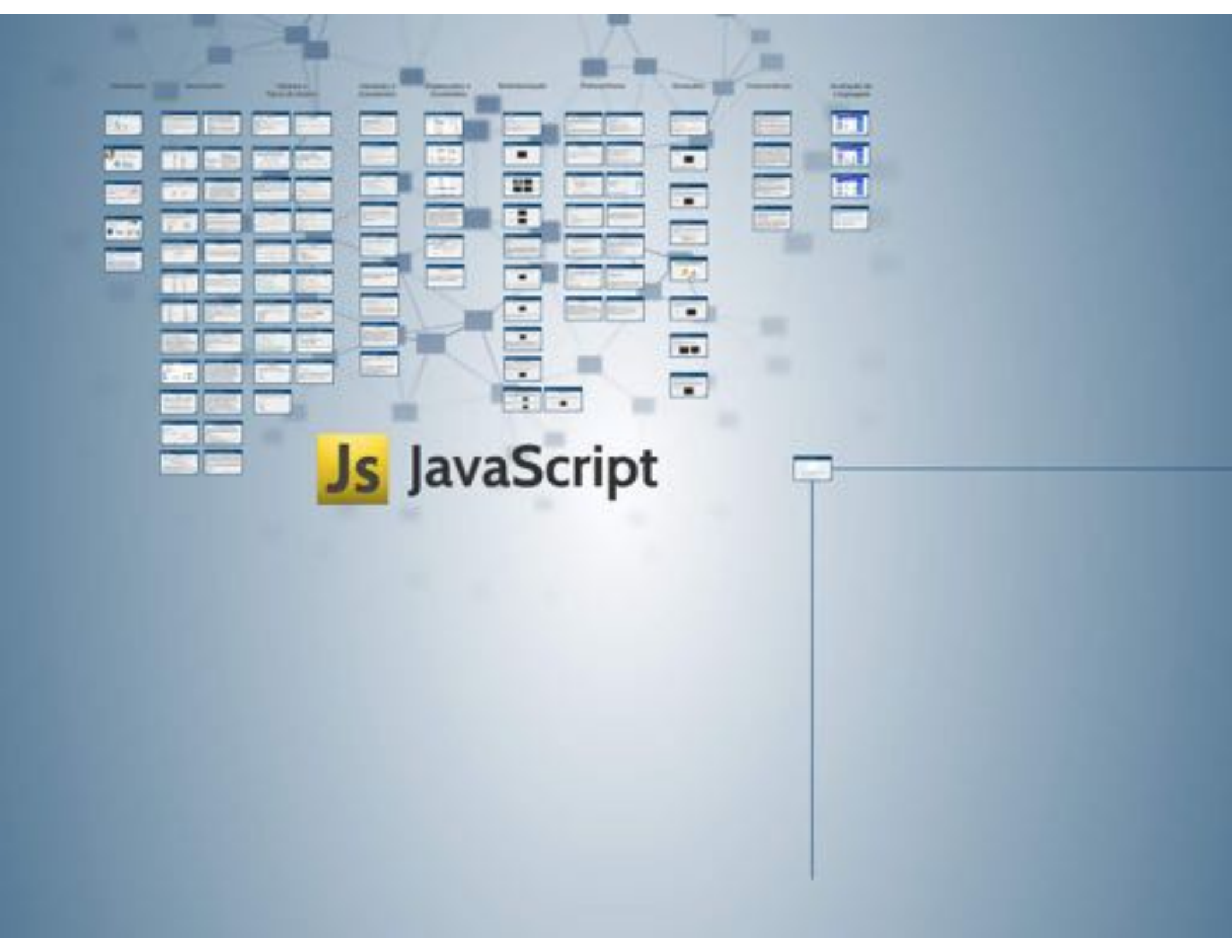
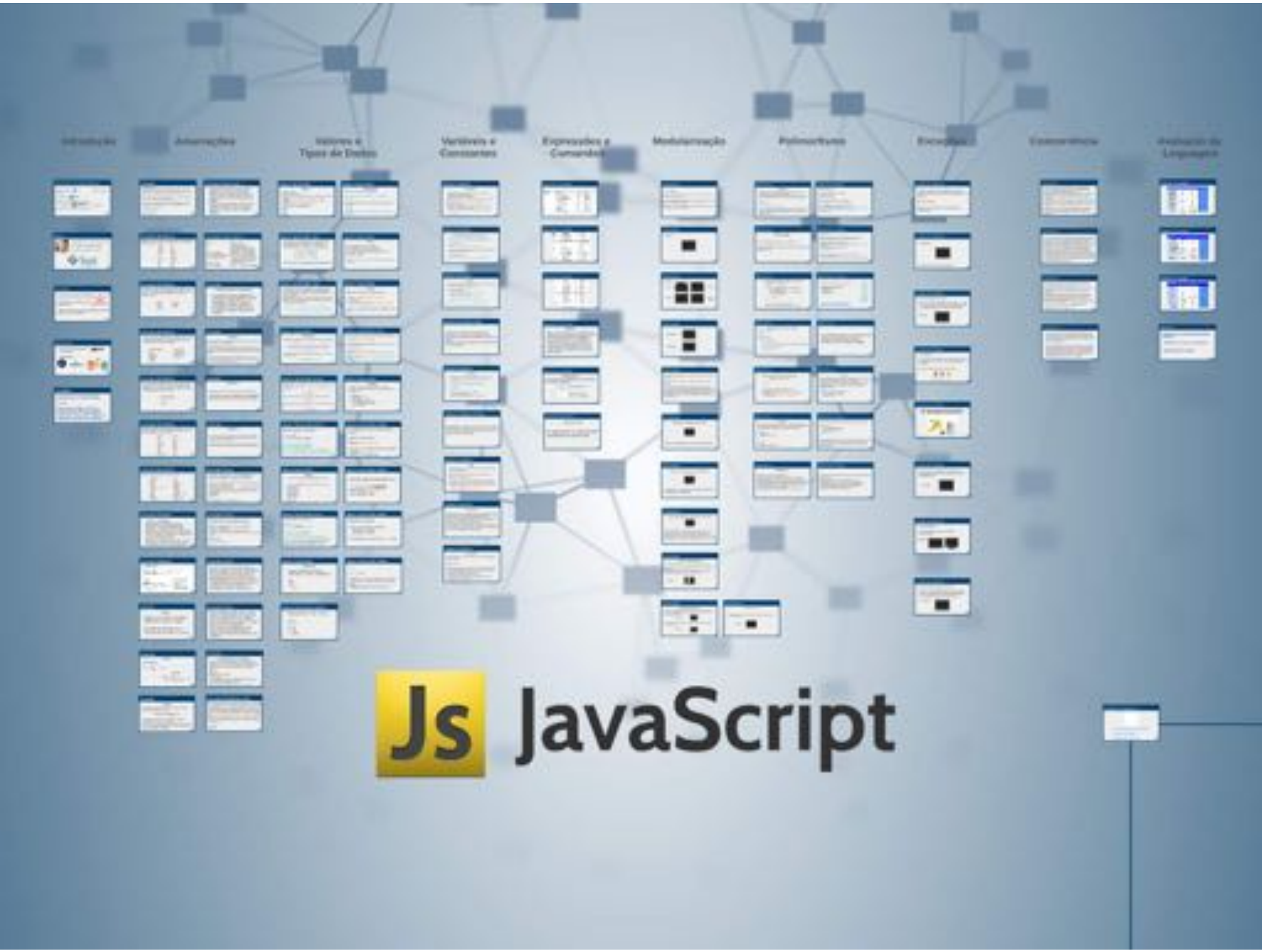

Js
JavaScript


Introdução
Anotações
Valores e Tipos de Dados
Variáveis e Constantes
Expressões e Comandos
Modularização
Polimorfismo
Encapsulamento
Exceções
Evolução da Linguagem
Js
JavaScript

Linguagens de Programação

- Leonardo Nascimento dos Santos
- Gustavo Costa Duarte
- Nicolas Ferreira de Aguiar

De onde veio a tecnologia JavaScript?

- 1992: A empresa 

comprou a empresa Nombas.
- A linguagem  .
- Renomeação da linguagem: ScriptEase.
- 1994: A empresa 

ficou muito interessada no projeto e decidiu utiliza-lo.

História

- 1995: Brendan Eich, co-fundador do Mozilla.
- Renomeação da linguagem: Mocha.
- Novamente renomeada: LiveScript.

Funcionou?

- Grande Jogada de Marketing.
- JavaScript é a mesma coisa que Java??

- 1996: A Microsoft criou a linguagem JScript. Será que ela é melhor?

Padronização

- 1997: Reconhecimento no mercado.

- Atualmente:

Curiosidades

É possível fazer muitas coisas utilizando JavaScript, basta ter vontade de aprender.

Exemplos:

Menu Teletype - ImageBox - Galeria 3D - Galeria Virtual 3D - Galeria 3D - ImagePress - ZoomGallery - Galeria 3D - Navegação 3D - Galeria - Book - Galeria Paineis - Galeria Infinty - Galeria Horizontal - Dock Menu - Galeria - Zoom - Menu - SlideShow - Animação Curling.

Amarrações

Identificadores:

- Um identificador JavaScript deve começar com uma letra, um sublinhado (_) ou um sinal de dólar ($). Os caracteres subsequentes podem ser letras, dígitos, sublinhados ou sinais de dólar.

- Para a portabilidade e facilidade de edição, é comum usar apenas ASCII letras e dígitos em identificadores.

- No entanto, o JavaScript permite que identificadores contenham letras e dígitos do conjunto de caracteres Unicode inteiro.

- Permitindo coisas como :

```
var sí = verdadeiro;
var π = 3,14;
```

Amarrações: Identificadores

JavaScript reserva um número de identificadores como as palavras-chave do próprio idioma. Você não pode usar essas palavras como identificadores em seus programas:

- Break
- Delete
- function
- Return
- typeof
- case
- do
- If
- switch
- var
- catch
- For
- Null
- try
- Else
- In
- This
- void
- continue
- False
- instanceof
- throw
- while
- Debugger
- finally
- new
- True
- With
- default

Amarrações: Identificadores

O JavaScript também reserva determinadas palavras-chave que não são usadas atualmente pelo idioma mas que pode ser usado em versões futuras.O ECMAScript 5 reserva as seguintes palavras

- class
- const
- enum
- export
- extends
- import
- super

Amarrações: Identificadores

- Além disso, as seguintes palavras, que são legais no código JavaScript comum, são reservadas em modo estrito:

  - implements
  - let
  - private
  - public
  - yield
  - interface
  - package
  - protected
  - static

Amarrações: Identificadores

O modo estrito também impõe restrições ao uso dos seguintes identificadores. Eles são não totalmente reservados, mas não são permitidos como nomes de variáveis, funções ou parâmetros:

- arguments
- eval

Amarrações: Identificadores

ECMAScript 3 reservou todas as palavras-chave da linguagem Java e, embora isso tenha sido amenizado no ECMAScript 5, você ainda deve evitar todos esses identificadores se você planeja executar seu código em uma implementação ECMAScript 3 do JavaScript:

- abstract
- double
- goto
- native
- static
- boolean
- enum
- implements
- package
- super
- byte
- export
- import
- private
- synchronized
- char
- extends
- int
- protected
- throws
- class
- final
- interface
- public
- transient
- const
- float
- long
- short
- volatile

Amarrações: Identificadores

JavaScript predefiniu um número de variáveis globais e funções, e você deve evitar. Usando seus nomes para suas próprias variáveis e funções:

- arguments
- encodeURI
- Infinity
- Number
- RegExp
- Array
- encodeURIComponent
- isFinite
- Object
- String
- Boolean
- Error
- isNaN
- parseFloat
- SyntaxError
- Date
- eval
- JSON
- parseInt
- TypeError
- decodeURI
- EvalError
- Math
- RangeError
- undefined
- decodeURIComponent
- Function
- NaN
- ReferenceError
- URIError

Amarrações: Declarações

- O var e o function
- Declarações de declaração
- Definem: identificadores que podem ser usados em outro local do programa e atribuem valores a esses identificadores
- Definem o significado das outras declarações em seu programa

Amarrações: Declarações

```
function hypotenuse(x, y) {
    return Math.sqrt(x*x + y*y);  // return is documented in the next section
}

function factorial(n) {           // A recursive function
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

```
var i;                                          // One simple variable
var j = 0;                                      // One var, one value
var p, q;                                       // Two variables
var greeting = "hello" + name;                  // A complex initializer
var x = 2.34, y = Math.cos(0.75), r, theta;     // Many variables
var x = 2, y = x*x;                             // Second var uses the first
var x = 2,                                      // Multiple variables...
    f = function(x) { return x*x },             // each on its own line
    y = f(x);
```

Amarrações

Loops:

- Utiliza o for, while e do/while identicos aos de C e Java
- De diferente possui uma variação de for-each e o for-in

Amarrações

For's

- **For-each:**

```
var a = ["a", "b", "c"];
a.forEach(function(entry) {
    console.log(entry);
});
```

- **For-in:**

```
for (key in a) {
    if (arrayHasOwnIndex(a, key)) {
        console.log(a[key]);
    }
}
```

Amarrações

Jumps

- Declaração de label, break, continue e return funcionam como em Java.

Ponto e virgula (;)

- O JavaScript usa o ponto-e-vírgula (;) como um separador ao final de uma instrução.
- Em JavaScript, você geralmente pode omitir o ponto e vírgula entre duas instruções se essas instruções são escritas em linhas separadas.

Amarrações: Ponto e virgula (;)

- Você deve entender sobre ponto e vírgula opcional em JavaScript.
- Observe que o JavaScript não trata cada quebra de linha como ponto-e-vírgula
- Trata quebras de linha como ponto-e-vírgula somente se não conseguir analisar o código sem os pontos-e-vírgulas.

Amarrações: Ponto e virgula (;)

```
var a
a
=
3
console.log(a)
```

Javascript interpreta esse codigo como:

```
var a;  a = 3;
console.log(a);
```

```
var y = x + f
(a+b).toString()
```

Mas os parenteses na segunda linha do código podem ser interpretados como uma chamada da função f da primeira linha, e o Javascript interpreta o código assim:

```
var y = x + f(a+b).toString();
```

Amarrações

Tempo de vida da váriavel

- O tempo de vida das variáveis em JavaScript começa quando elas são declaradas. Variáveis locais são descartadas quando a função é completada. Variáveis globais são descartadas quando uma página é fechada.

Amarrações

Escopo

- Como a maioria das linguagens de programação modernas, o JavaScript usa escopo estático.
- Para implementar o escopo estático, o estado interno de um objeto de função JavaScript deve incluir não apenas o código da função, mas também uma referência à corrente de escopo atual.
- Esta combinação de um objeto de função e um escopo (um conjunto de ligações variáveis) em que as variáveis da função são resolvidas é chamado de closure.

Amarrações

Closures

- São objetos e têm uma cadeia de escopo associada a elas.
- A maioria das funções são invocadas usando a mesma cadeia de escopo que estava em vigor quando a função foi definida, e realmente não importa que haja um encerramento envolvido.

Amarrações

Closures

- Os closures tornam-se interessantes quando são invocados sob uma cadeia de alcance diferente daquela que estava em vigor quando foram definidos.
- Isso acontece mais comumente quando um objeto de função aninhada é retornado da função dentro da qual ele foi definido.

Amarrações: Closures

- O primeiro passo para a compreensão de closures é revisar as regras de escopo estático para funções aninhadas. Considere o seguinte:

```
var scope = "global scope"; // Uma variavel global
function checkscope() {
 var scope = "local scope“;    // Uma variavel local
 function f() { return scope; }// Retorna o valor no
scope aqui
 return f();
}
checkscope()  => "local scope"
```

Amarrações: Closures

Agora vamos mudar o código apenas um pouco.
Você pode dizer o que esse código vai retornar?

```
var scope = "global scope"; // Uma variavel global
function checkscope() {
 var scope = "local scope"; // Uma variavel local
 function f() { return scope; } // Retorna o valor no scope aqui
 return f;
}
checkscope()()
```

Resposta: “local scope”

Amarrações: Closures

- Lembre-se da regra fundamental do escopo estático: as funções JavaScript são executadas usando a cadeia de escopo que estava em vigor quando foram definidas. A função aninhada f () foi definida sob uma cadeia de escopo na qual o escopo da variável foi vinculado ao valor "escopo local".

Amarrações: Closures

- Essa vinculação ainda está em vigor quando f é executado, onde quer que seja executado. Portanto, a última linha de código acima retorna "escopo local", não "escopo global".
- Eles capturam as amarrações das variáveis locais (e parâmetro) da função externa dentro da qual elas são definidas.

Amarrações

Declaração de variável

- Antes de usar uma variável em um programa JavaScript, você deve declarar. As variáveis são declaradas com a palavra-chave var, assim:
  var i;
  var sum;
- Tambem é possivel:
  var i, soma;
- E você pode combinar declaração de variável com inicialização de variável:
  var message = "hello"; Var i = 0, j = 0, k = 0;

Amarrações: Declaração de variável

- Se você não especificar um valor inicial para uma variável com a instrução var, a variável é declarada, mas seu valor é indefinido até que seu código armazene um valor nele.
- Diferente das linguagens estaticamente, como C ou Java, uma variável JavaScript pode conter um valor de qualquer tipo.
- Por exemplo, é perfeitamente legal em JavaScript atribuir um número a uma variável e depois atribuir uma string a essa variável:

```
var i = 10;
i = "dez";
```

Valores e Tipos de Dados

Tipos

- Tipos primitivos:
- Boolean - true e false;
- null - palavra-chave que indica valor nulo (diferente de Null e NULL);
- undefined - uma propriedade superior cujo valor é indefinido;
- Number - inteiros ou ponto flutuante;
- String - “uma palavra qualquer”;
- Objetos:
- array;
- Date;
- objetos;

Valores e Tipos de Dados: Tipos

Javascript é uma linguagem dinamicamente tipada (tipos de dados são convertidos automaticamente conforme a necessidade).

```
var x; // undefined
x = 13; // Number
x = "texto" // String
```

- Javascript é uma linguagem fracamente tipada.

Valores e Tipos de Dados: Tipos

- JavaScript é bem flexível quanto aos tipos de valor.
- Se JS espera um valor booleano e recebe qualquer outra coisa ele irá converter o valor de acordo com o que foi passado;
- Outro exemplo é que se o JS está esperando um valor string ele ira converter o valor passado para string, o mesmo vale para Number:

```
10 + " nota" // => "10 nota", 10 é convertido para string

"2" * "3" // 6, as duas strings são convertidas para Number
```

Valores e Tipos de Dados

Number

Números podem ser escritos com ou sem ponto

```
var x = 3.00;      var y = 3;      var z = .3;
```

Números podem ser precedidos de sinal:

```
var x = -3;
```

Números muito grandes ou pequenos podem ser escritos com expoentes:

```
var x = 123e5 // 12300000
```

Valores e Tipos de Dados: Number

Existe a representação de números em hexadecimal (16), octal (8), e binário (2);
Hexadecimal é representado com 0x na frente do número que vai de 0-9, a-f:

```
var x = 0x3A7;
```

Octal é representado com 0 na frente do número q vai de 0-7:

```
var x = 0453;
```

Binário é representado com 0b ou 0B na frente do número q vai de 0-1:

```
var x = 0b10001;
```

Valores e Tipos de Dados: Number

Números podem ser Objetos:

```
var x = 13;
var y = new Number(13);

// typeof x retorna number
// typeof y retorna object
// (x == y)  é true pois o valor é o mesmo
// (x === y) é false pois são de tipos diferentes
```

Valores e Tipos de Dados

String

A strings são declaradas entre aspas simples ou duplas:

```
var x = "texto";      x = 'texto';
```

Caracteres especiais são codificados com barra invertida:
\0 o caractere NULL
\' aspas simples
\" aspas duplas
\\ barra invertida
\n nova linha
\r carriage return
\v tab vertical
\t tab
\b backspace

Valores e Tipos de Dados: String

Strings podem ser objetos:

```
var x = "texto";
var y = new String("texto");

// typeof x retorna number
// typeof y retorna object
// (x == y)  é true pois o valor é o mesmo
// (x === y) é false pois são de tipos diferentes
```

Obs: strings podem ser comparadas com operadores de comparação(<,>,==,etc)

Valores e Tipos de Dados

Booleans

Aceita valor true ou false;
Tudo que é “real” é verdadeiro:

10
-12
“false”

Valores e Tipos de Dados: Booleans

Tudo que não é “real” é falso:

```
var x;
x = “”;
x = 0;
x = -0;
x = null;
```

Valores e Tipos de Dados

Undefined

Uma variável sem valor é undefined:

```
var x; // o valor de x é undefined, e o tipo de x é undefined
```

Você pode também limpar uma variável a igualando a undefined:

```
var x = 3;
x = undefined; // o valor de x é undefined, e o tipo de x é undefined
```

Valores e Tipos de Dados

Null

Em JavaScript null é “nada”;
Em JavaScript null é um objeto;
Você pode também limpar um objeto o igualando a null:

```
x = null; // o valor de x é null, mas o tipo de x é object
```

Valores e Tipos de Dados

Array

Pode ser criado de diversas formas:

```
var carro = [“Uno”,"Gol",”Fox”]
var carro [];
var caro = new Array(“Uno”,”Gol”,”Fox”);
```

O acesso a posições do array é igual a outras Lp’s;

Valores e Tipos de Dados: Array

Em JavaScript o mesmo array pode conter diferentes tipos

```
var pessoa = ["Val","Baiano",32]
```

Existem algumas formas de adicionar novos elementos em uma array e retirar:

```
var carro = ["Uno","Gol","Fox"]
carro.push("Fusca"); // push adiciona "Fusca" no final do array
carro[carro.length] = "Passat" // adiciona "Passat" no final do array
carro.pop(); // retira o último elemento "Passat" do array
```

Valores e Tipos de Dados

Objeto

Um objeto tem varias propriedades e metodos;
O objeto é declarado e dentro dele suas propriedades e metodos:

```
var jogador = {
    primeiroNome: "Val",
    ultimoNome : "Baiano",
    nomeCompleto : function() {
        return this.primeiroNome + " " +
    this.ultimoNome;
    }
};
```

Valores e Tipos de Dados: Objeto

As propriedades são acessada de duas formas:

```
jogador.primeiroNome;
ou
jogador["primeiroNome"];
```

O acesso aos métodos é feito da seguinte maneira:

```
jogador.nomeCompleto();
```

Valores e Tipos de Dados: Objeto

Criando Objeto utilizando new:

```
var carro = new Object();
carro.nome = "Fusca";
carro.idade = 50;
```

Valores e Tipos de Dados: Objeto

Utilizando construtor:

```
function carro(nome, idade) {
    this.nome = nome;
    this.idade = idade;
}
var meuCarro = new carro("Fusca",50);
```

Valores e Tipos de Dados: Objeto

Em:

```
var x = carro;
```

O objeto x não é uma cópia de carro, ele é carro. Tanto x quanto carro são o mesmo objeto.
Qualquer mudança em x muda carro.

Variáveis e Constantes

Variáveis

- A declaração de uma variável em JS é feita usando as palavras reservadas var e let;
- O nome de variáveis é identificador;
- Os identificadores de variáveis podem conter letras, dígitos, "$", "_";
- O identificador tem que começar com letra, "$", ou "_";
- JS é case-sensitive;
- Palavras reservadas não podem ser usadas como nomes;
- Igual (=) é o operador de atribuição;
- Pode-se declarar várias variáveis em uma mesma linha:

```
var x = 5, y = 6;
```

Variáveis e Constantes

Atribuição por Valor e Referência

- Variáveis do tipo primitivo são copiados por valor:

```
var x = 5, y = x;
y = 4; // => o valor de x é 5, e o de y é 4
```

- Quando se trabalha com tipos complexos (objetos) são copiados por referência:

```
var x = [0,1], y = x;
y[0] = 2 // x no índice 0 mudará para 2 ficando [2,1]
```

Variáveis e Constantes

Escopo

- Variáveis declaradas dentro de uma função são variáveis locais da função em que pertence:

```
function exemplo() {
    var x; //x existe aqui, y não existe aqui
    for(let y = 0; y < 2; y++){
        //x está visível aqui, y existe aqui
    }
    //x está visível aqui, y não está visível aqui
}
//x não está visível aqui, y não está visível aqui
```

Variáveis e Constantes: Escopo

- Usando var a variável tem escopo local;
- Usando let a variável tem escopo de bloco;
- Variáveis declaradas fora de funções tem escopo global;
- Declarar valor a uma variável não declarada a fará ter escopo global;

Variáveis e Constantes

Memória

- JavaScript aloca memória quando variáveis são criadas e automaticamente o libera quando não é mais usado:

```
var x = 123; // aloca memória para um number
var y = "texto"; // aloca memória para uma string
```

- Ciclo de vida:

1 - Aloca espaço quando é necessário;
2 - Usa a memória alocada;
3 - Libera a memória quando não é mais usado;

Variáveis e Constantes: Memória

- JavaScript conta com um coletor de lixo para liberar espaços de memória que não estão sendo utilizados;

Variáveis e Constantes

I/O

- JavaScript pode interagir com um usuário criando uma caixa de mensagem:

```
window.alert("Oi"); // Uma caixa de mensagem irá aparecer com a palavra Oi
```

- Para Ler uma entrada do teclado cria-se uma caixa de mensagem na tela com ou sem um texto, um lugar para entrar com a informação, e opções de "OK" e "Cancel":

```
var nome = prompt("Digite seu nome", "");
```

Variáveis e Constantes

Serialização

- JavaScript oferece as funções nativas JSON.stringify() e JSON.parse() para serializar e guardar objetos;
- JSON não consegue representar todos os valores de JS;
- Objetos, array, strings, números, true, false, e null podem ser serializados e guardados;
- NaN, Infinity e -Infinity ao serem serializadas viram null;
- Function, RegExp, Error objects e valores undefined não podem ser serializados ou guardados;

Variáveis e Constantes

Constantes

- Você pode criar uma constante por meio da palavra chave const:

```
const x = 5;
```

- Uma constante não pode alterar seu valor por meio de uma atribuição ou ao ser declarada novamente enquanto o script é executado;
- Deve ser inicializada com um valor;
- As regras de escopo para as constantes são as mesmas para as variáveis;

Expressões e Comandos

Operator	Operation	A	N	Types
++	Pre- or post-increment	R	1	lval→num
--	Pre- or post-decrement	R	1	lval→num
-	Negate number	R	1	num→num
+	Convert to number	R	1	num→num
~	Invert bits	R	1	int→int
!	Invert boolean value	R	1	bool→bool

Expressões e Comandos

delete	Remove a property	R	1	lval→bool
typeof	Determine type of operand	R	1	any→str
void	Return undefined value	R	1	any→undef
*, /, %	Multiply, divide, remainder	L	2	num,num→num
+, -	Add, subtract	L	2	num,num→num
+	Concatenate strings	L	2	str,str→str
<<	Shift left	L	2	int,int→int
>>	Shift right with sign extension	L	2	int,int→int
>>>	Shift right with zero extension	L	2	int,int→int
<, <=, >, >=	Compare in numeric order	L	2	num,num→bool
<, <=, >, >=	Compare in alphabetic order	L	2	str,str→bool
instanceof	Test object class	L	2	obj,func→bool
in	Test whether property exists	L	2	str,obj→bool

Expressões e Comandos

==	Test for equality	L	2	any,any→bool
!=	Test for inequality	L	2	any,any→bool
===	Test for strict equality	L	2	any,any→bool
!==	Test for strict inequality	L	2	any,any→bool
&	Compute bitwise AND	L	2	int,int→int
^	Compute bitwise XOR	L	2	int,int→int
|	Compute bitwise OR	L	2	int,int→int
&&	Compute logical AND	L	2	any,any→any
||	Compute logical OR	L	2	any,any→any
?:	Choose 2nd or 3rd operand	R	3	bool,any,any→any
=	Assign to a variable or property	R	2	lval,any→any
*=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=, >>>=	Operate and assign	R	2	lval,any→any
,	Discard 1st operand, return second	L	2	any,any→any

Lvalues

- Lvalue é um termo histórico que significa "uma expressão que pode aparecer legalmente no lado esquerdo de uma expressão de atribuição".
- Em JavaScript, variáveis, propriedades de objetos e elementos de arrays são lvalues.
- A especificação ECMAScript permite funções incorporadas para retornar lvalues mas não define quaisquer funções que se comportam dessa maneira.

Agregações

As agregações existentes em Javascript são arrays e objects.

```
[]        // An empty array: no expressions inside brackets means no elements
[1+2,3+4] // A 2-element array.  First element is 3, second is 7
```

```
var p = { x:2.3, y:-1.2 };  // An object with 2 properties
var q = {};                 // An empty object with no properties
q.x = 2.3; q.y = -1.2;      // Now q has the same properties as p
```

Agregações

agregações existentes em Javasc
arrays e objects.

```
[]          // An empty array: no expressions inside brackets means no elements
[1+2,3+4]   // A 2-element array.  First element is 3, second is 7
```

```
var p = { x:2.3, y:-1.2 };  // An object with 2 properties
var q = {};                 // An empty object with no properties
q.x = 2.3; q.y = -1.2;      // Now q has the same properties as p
```

agregações existentes em Javasc
o arrays e objects.

```
[]        // An empty array: no expressions inside brackets means no elements
[1+2,3+4] // A 2-element array.  First element is 3, second is 7
```

```
var p = { x:2.3, y:-1.2 };  // An object with 2 properties
var q = {};                 // An empty object with no properties
q.x = 2.3; q.y = -1.2;      // Now q has the same properties as p
```

Agregações

As agregações existentes em Javascript são arrays e objects.

```
[]        // An empty array: no expressions inside brackets means no elements
[1+2,3+4] // A 2-element array.  First element is 3, second is 7
```

```
var p = { x:2.3, y:-1.2 };  // An object with 2 properties
var q = {};                 // An empty object with no properties
q.x = 2.3; q.y = -1.2;      // Now q has the same properties as p
```

Curto-circuito

- As operações de curto-circuito acontecem em Javascript.

Modularização

Por que modularizar?

Se escrevermos tudo em apenas um arquivo, a manutenibilidade do código ficará comprometida.

Mesmo separando em várias partes, é preciso especificar muito bem todas as dependências existentes.

O objetivo de se utilizar a modularização é melhorar a compreensão, a manutenção e o reuso do código.

Modularização

Exemplo:

```
417
418
419
420 //Nesta função só é possível manipular os valores internamente
421 function numeros()
422 {
423     var x = 42;
424     var y = 2048;
425
426     return
427     {
428         magico: x;
429     }
430 }
431
432 //Só é possível visualizar o valor que a função retorna
433 var n = numeros();
434
435 n.x;     //undefined
436 n.y;     //undefined
437 n.magico;   //42
438
439
440
```

Modularização

Exemplos:

Classes ->

<- Heranças

Pacotes ->

<- Pacotes Private

```
405
406 function Pessoa(nome)
407 {
408     this.cumprimenta = function()
409     {
410         return "Oi, eu sou " +nome;
411     };
412 }
413
414 var p = new Pessoa("Léo");
415 p.cumprimenta();    //Oi, eu sou Léo
416
417
```

Modularização

Exemplos:

Classes ->

<- Heranças

Pacotes ->

<- Pacotes Private

```
419
420
421 function PessoaFisica(nome, cpf)
422 {
423     Pessoa.call(this, nome);
424
425     this.getCPF = function()
426     {
427         return cpf;
428     }
429 }
430
431
```

Modularização

Exemplos:

Classes ->

<- Heranças

Pacotes ->

<- Pacotes Private

```
435 //Desta maneira, todas as classes são públicas
436 var modelo =
437 {
438     Pessoa: function(nome)
439     {
440         ...
441     },
442
443     PessoaFisica: function(nome, cpf)
444     {
445         ...
446     },
447
448     Empresa: function(nome, cnpj)
449     {
450         ...
451     },
452     ...
453 };
454
455 var p = new modelo.Pessoa("Léo");
456
```

Modularização

Exemplos:

Classes ->

<- Heranças

Pacotes ->

<- Pacotes Private

```
458
459 //Tudo que fica dentro de uma função, só pode ser visualizado internamente.
460 function criaPacoteModelo()
461 {
462     //Parte private
463     function PessoaAjuda()
464     {
465         ...
466     }
467     //Parte pública
468     var exports =
469     {
470         Pessoa: function(nome)
471         {
472             ...
473         },
474
475         PessoaFisica: function(nome, cpf)
476         {
477             ...
478         },
479
480         Empresa: function(nome, cnpj)
481         {
482             ...
483         },
484         ...
485     };
486     return exports;
487 }
```

Modularização

Como utilizar?

Resolvendo...

```
494
495
496
497
498
499 //É preciso chamar a função para poder criar o pacote e utiliza-lo
500 var modelo = criaPacoteModelo();
501 var p = new modelo.Pessoa("Léo");
502
503
504 //Erro
505 var ph = new modelo.PessoaAjuda();
506
507
508
509
510
511
```

Modularização

Como utilizar?

Resolvendo...

```
var modelo =
(
    function()
    {
        //Parte private
        function PessoaAjuda()
        {
            ...
        }
        //Parte pública
        var exports =
        {
            Pessoa: function(nome)
            {
                ...
            },
            PessoaFisica: function(nome, cpf)
            {
                ...
            },
            Empresa: function(nome, cnpj)
            {
                ...
            },
            ...
        };
        return exports;
    }
());

//Agora é possivel utilizar de forma direta
var p = new modelo.Pessoa("Léo");
```

Modularização

A passagem de parâmetros em Javascript é feita por valor ou por referência?

Resposta: É feita por ambos, pois depende do tipo da variável passada.

De forma direta e resumida, o javascript utiliza passagem de parêmetros por VALOR quando a variável é de algum tipo primitivo (ou seja, quando typeof da variável é "number", "string", "boolean", "undefined" ou "null"), e utiliza passagem de parâmetros por REFERÊNCIA quando a variável é de algum tipo complexo ("object" ou "function").

Modularização

Exemplo de passagem por valor:

Após a passagem por valor, nada foi alterado.

```
399
400 var meuNumero = 10;
401 var minhaString = "Meu Nome";
402
403 var somarCinco = function(parametro)
404 {
405     parametro += 5;
406 }
407 var concatenarAlgo = function(parametro)
408 {
409     parametro += " - Algo Concatenado";
410 }
411
412 somarCinco(meuNumero);
413 concatenarAlgo(minhaString);
414
415 //Qual será o valor de meuNumero agora?
416 console.log(meuNumero); // Acertou quem disse 10.
417
418 //Qual será o valor de minhaString agora?
419 console.log(minhaString); // Acertou quem disse "Meu Nome".
420
```

Modularização

Exemplo de passagem por valor:

Após a passagem por valor, nada foi alterado.

Modularização

Exemplo de passagem por referência:

Neste caso, os valores foram alterados devido a passagem por referência.

```
430 var meuObjeto = {};
431 meuObjeto.meuNumero = 10;
432 meuObjeto.minhaString = "Meu Nome";
433
434 var somarCinco = function(parametro)
435 {
436     parametro.meuNumero += 5;
437 }
438 var concatenarAlgo = function(parametro)
439 {
440     parametro.minhaString += " - Algo Concatenado";
441 }
442
443 somarCinco(meuObjeto);
444 concatenarAlgo(meuObjeto);
445
446 //Qual será o valor de meuObjeto.meuNumero agora?
447 console.log(meuObjeto.meuNumero); // Acertou quem disse 15.
448
449 //Qual será o valor de minhaString agora?
450 console.log(meuObjeto.minhaString);
451 // Acertou quem disse "Meu Nome - Algo Concatenado".
```

Modularização

Exemplo de passagem por referência:

Neste caso, os valores foram alterados devido a passagem por referência.

Modularização

Exemplo de atribuição de valores às variáveis:

No código acima, "b" estará com a propriedade "terceiroItem", mesmo sem nunca ter sido adicionada diretamente à ela.
O que acontece é que tanto "a" quanto "b" referênciam-se à mesma instância do objeto criado na inicialização de "a". Ou seja, modificações em "a" refletem-se diretamente em "b", e vice-versa.

```
455
456
457 //Inicializar um objeto e atribui-lo à uma váriável.
458 var a =
459 {
460     "primeiroItem": 1,
461     "segundoItem":2
462 };
463
464 // Atribuir esta variável à uma nova variável.
465 // O que acontecerá? Será que o javascript copia o objeto e
466 // fica com duas instancias dele na memória, sendo uma para "a" e outra para "b"?
467 var b = a;
468
469 // Adicionar uma nova propriedade ao objecto referenciado por "a"
470 a["terceiroItem"] = 3;
471
472 // E então, como estará "b" agora?
473 console.log(b);
474
475
476
```

Modularização

Exemplo de atribuição de valores às variáveis:

No código acima, "b" estará com a propriedade "terceiroItem", mesmo sem nunca ter sido adicionada diretamente à ela.
O que acontece é que tanto "a" quanto "b" referênciam-se à mesma instância do objeto criado na inicialização de "a". Ou seja, modificações em "a" refletem-se diretamente em "b", e vice-versa.

Modularização

Objeto de argumentos:

A variável arguments é como um array, mas não é um array.

Ela é utilizada para passar qualquer quantidade de argumentos para uma função.
Através da propriedade arguments.length, é possível saber a quantidade de parâmetros que a função recebeu.

Exemplo:

```
396
397 function concatenar(separador)
398 {
399     // inicializa a lista
400     var result = "", i;
401
402     // itera por meio de argumentos
403     for (i = 1; i < arguments.length; i++)
404     {
405         result += arguments[i] + separador;
406     }
407     return result;
408 }
409
410 // retorna "red, orange, blue, "
411 concatenar(", ", "red", "orange", "blue");
412
413 // retorna "elephant; giraffe; lion; cheetah; "
414 concatenar("; ", "elephant", "giraffe", "lion", "cheetah");
415
416 // retorna "sage. basil. oregano. pepper. parsley. "
417 concatenar(". ", "sage", "basil", "oregano", "pepper", "parsley");
418
```

Modularização

Parâmetros Padrão:

Parâmetros padrão são undefined. Antes era preciso testar os valores de parâmetros no corpo da função e atribuir um valor se eles fossem undefined.

Exemplo:

Agora já é possível colocar um valor no campo de declaração de parâmetros.

Exemplo:

```
427
428
429
430 function multiplicar(a, b) {
431   b = typeof b !== 'undefined' ?  b : 1;
432
433   return a*b;
434 }
435
436 multiplicar(5); // 5
437
438
439
440
441
```

Modularização

Parâmetros Padrão:

Parâmetros padrão são undefined. Antes era preciso testar os valores de parâmetros no corpo da função e atribuir um valor se eles fossem undefined.

Exemplo:

Agora já é possível colocar um valor no campo de declaração de parâmetros.

Exemplo:

```
427
428
429
430 function multiplicar(a, b = 1)
431 {
432     return a*b;
433 }
434
435 multiplicar(5); // 5
436
437
438
439
440
441
```

Modularização

Parâmetros Rest:

Permite representar um número indefinido de argumentos como um array.

Exemplo:

```
427
428
429
430 function multiplicar(multiplicador, ...args)
431 {
432     return args.map(x => multiplicador * x);
433 }
434
435 var arr = multiplicar(2, 1, 2, 3);
436 console.log(arr); // [2, 4, 6]
437
438
439
440
441
```

Polimorfismo

Coerção

- Em JavaScript, você pode realizar operações em valores de diferentes tipos sem causar uma exceção;
- O interpretador JavaScript converte implicitamente, ou força, um dos tipos de dados no outro e, em seguida, realiza a operação;

```
var x = 10;
var y = " nota";
var z = y + x; // z recebe "10 nota"
```

- Para se converter explicitamente uma cadeia de caracteres em um número inteiro, use-se a função parseInt, para se converter explicitamente uma cadeia de caracteres em um número, use-se a função parseFloat;

Polimorfismo

Sobrecarga

- Não suporta sobrecarga de operadores;
- Somente métodos podem ser sobrecarregados pelo programador;

Polimorfismo

Sobrecarga Independente de Contexto

- Exemplo de sobrecarga:

```
function carro(argumento1, argumento2) {
      var x = argumento1;
      if(typeof argumento2 !== “undefined”) {
          x += argumento2;
     }
          return x;
};

carro(“Gol”); // resultado = Gol
carro(“Gol ”,1000); // resultado = Gol 1000
```

Polimorfismo

Sobrecarga Dependente de Contexto

Exemplo de sobrecarga:

```
var Pessoa =
{
    Class: function()
    {
        this.tipo = function()
        {
            return this.tipo[arguments[0].constructor].apply(this, arguments)
        }
        this.tipo[String] = function()
        {
            alert("String");
        }
```

Polimorfismo

```
        this.tipo[Number] = function(){
                alert("Number");
            }
        }
    }
    var Jao = new Pessoa.Class
    Jao.tipo("Texto"); // alert  - "String"
    Jao.tipo(10); // alert - "Number"
```

Polimorfismo

Paramétrico

Em JS o polimorfismo paramétrico se manifesta através da sua definição de tipos, uma vez que o identificador var define/ declara qualquer estrutura, de primitivos a objetos;

```
var carro = {
    nomeCarro : “Fusca”,
    corCarro : “Azul”,
};

var x = 5;
var nome = “Val Baiano”;
var vet = [carro,x,nome];
```

Polimorfismo

Inclusão

- Característico de linguagens OO;
- JavaScript utiliza o modelo de herança prototipal;
- Cada objeto tem um link interno para um outro objeto chamado prototype, esse objeto prototype também tem um atributo prototype, e assim por diante até que null seja encontrado como prototype;
- O prototype de um objeto é uma super classe dele;

Polimorfismo: Inclusão

Exemplo de Herança:

```
var Pessoa = function(nome, email){
    this.nome = nome;
    this.email = email;
};
Pessoa.prototype.meuNome = function(){
   console.log("Meu nome é "+this.nomel);
};
var PessoaFisica = function(nome, email, cpf){
   Pessoa.call(this, nome, email); // Isso fará com que PessoaFisica
tenha todos os atributos que uma Pessoa teria;
   this.cpf = cpf;
};
```

Polimorfismo: Inclusão

Continuação:

Então, para herdarmos os métodos de Pessoa, basta setarmos o prototype da PessoaFisica para uma instância de Pessoa:

```
PessoaFisica.prototype = new Pessoa();
PessoaFisica.prototype.constructor = PessoaFisica; // corrige o ponteiro do construtor
PessoaFisica.prototype.dizCpf = function(){
   console.log(this.cpf);
};
```

Os métodos de PessoaFisica devem ser declarados depois de sobrescrever o prototype;
Esta é a maneira mais comum de se implementar herança em javascript;

Polimorfismo: Inclusão

Essa é a função, que foi incluída ao ECMAScript 5:

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}
```

/* Basicamente o que ela faz é criar uma função construtora F, setar o protótipo desta para o objeto passado como parâmetro e retornar uma instância de F. Ou seja, ela devolve um objeto vazio que herda objeto passado! */

http://blog.caelum.com.br/reaproveitando-codigo-com-javascript-heranca-e-prototipos/

Polimorfismo

- Os métodos podem ser sobrescritos;
- JS possui amarração tardia;
- JS não possui herança múltipla;

Polimorfismo

Classes

Apesar existir objetos puros, é possível, assim como em JAVA definir classes que tem suas próprias propriedades para armazenar ou definir seus estados e também possuem propriedades que definem seu comportamento;

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

Polimorfismo: Classes

Classes em JavaScript são introduzidas no ECMAScript 6:

```
class Carro {
    constructor(modelo, cor) {
        this.modelo = modelo;
        this.cor = cor;
    }
}
```

O método constructor é um tipo especial de método da criação e da inicialização de um objeto criado com uma classe, só pode existir um método epecial com o nome "constructor" dentro da classe;

Polimorfismo: Classes

- A palavra-chave static define um método estático de uma classe;
- Métodos estáticos são chamados sem a instanciação da sua classe e não podem ser chamados quando a classe é instanciada;
- JavaScript possui Metaclasses;

Exceções em JavaScript

- 3 tipos de erros em JavaScript: Erros de sintaxe, erros de tempo de execução e erros lógicos.
- Erro de sintaxe:
- Ocorrem no tempo de interpretação no JavaScript.

Exceções em JavaScript

Exemplo:

- O resto do código é executado.

```
1
2
3
4
5
6  <script language="javascript">
7
8      window.print( ;      // Falta um parêntese
9
10 </script>
11
12
13
14
15
```

Exceções em JavaScript

Exemplo:

- O resto do código é executado.

Exceções em JavaScript

- Erros de tempo de execução:
- Estes erros ocorrem durante a execução. A sintaxe pode estar correta, mas um erro pode ocorrer caso o programa chame um método que não exista.

Exemplo:

```
15
16
17
18
19 <script language="javascript">
20
21     window.printme(); // A função printme não existe
22
23 </script>
24
25
26
27
28
```

Exceções em JavaScript

- Erros lógicos:
- É o tipo de erro mais difícil de rastrear. Eles não são resultado de um erro de sintaxe ou de tempo de execução.
- Esses erros são resultado de que então

O programador!

Exceções em JavaScript

- Assim como a linguagem Java, o JavaScript também utiliza o try, catch e finally, assim como o operador throw.

Exemplo:

```
33
34 <script language="javascript">
35
36     try
37     {
38         processar();    //caso a função não exista
39     }
40     catch (e)
41     {
42         window.alert("Houve um erro de execução.");
43     }
44     finally
45     {
46         //O que for colocado aqui sempre será executado
47         window.alert("Processo concluído!");
48     }
49
50 </script>
51
```

Exceções em JavaScript

- A instrução throw:
- Ela é utilizada para lançar exceções personalizadas.

Exemplo:

```
<script language="javascript">

    //Função que retorna a mensagem de erro
    function MailFormatadoException(mail)
    {
        this.email = mail;
        this.mensagem = "E-mail inválido! Favor verificar.";

        this.toString = function()
        {
            return this.email + " " + this.mensagem;
        };
    }

    //Função que verifica se o email é válido
    function validarEMail(email)
    {
        if(email.indexOf("@") > -1)
        {
            return true;
        }
        else
        {
            throw new MailFormatadoException(email);
        }
    }
```

```
83
84
85
86
87      //Tenta validar o email
88      try
89      {
90          var mail = "netcoders.necoders.com";
91          var valido = validarEMail(mail);
92
93          if(valido)
94          {
95              window.alert("Endereço de e-mail correto.");
96          }
97      }
98      catch(e)
99      {
100         //Imprime a mensagem de erro
101         window.alert(e);
102     }
103
104 </script>
105
106
107
108
109
```

Exceções em JavaScript

- O manipulador de eventos Onerror:

- Surgiu com o objetivo de facilitar o tratamento de erros. Ele oferece 3 trechos de informações para identificar a natureza exata do erro.

Exemplo:

```
111
112
113 <html>
114     <head>
115         <script type="text/javascript">
116
117             window.onerror = function (msg, url, line)
118             {
119                 alert("Mensagem : " + msg );
120                 alert("Url : " + url );
121                 alert("Numero da linha : " + line );
122             }
123
124         </script>
125     </head>
126     <body>
127         <p>Clique para ver o resultado</p>
128
129         <form>
130             <input type="button" value="Clique aqui" onclick="funcaoInexistente();" />
131         </form>
132     </body>
133 </html>
```

Concorrência

- O núcleo da linguagem JavaScript não contém qualquer mecanismo de threading, e JavaScript do lado do cliente tradicionalmente não definiu qualquer um.
- Mesmo quando a execução simultânea é possível, o JavaScript do lado do cliente não pode detectar o fato de que ele está ocorrendo.
- A execução de um único thread permite um script muito mais simples: você pode escrever código com a garantia de que dois manipuladores de eventos nunca serão executados ao mesmo tempo.
- A execução de um único thread significa que os navegadores da Web devem parar de responder à entrada do usuário enquanto scripts e manipuladores de eventos estão sendo executados.

Concorrência

- Se seu aplicativo deve executar computação suficiente para causar um atraso notável, você deve permitir que o documento seja carregado completamente antes Executando essa computação, e você deve certificar-se de notificar o usuário de que a computação está em andamento e que o navegador não está pendurado.
- Se for possível quebrar seu cálculo em subtarefas discretas, você pode usar métodos como setTimeout () e setInterval () para executar as subtarefas em segundo plano enquanto atualiza um indicador de progresso que exibe feedback para o usuário.

Concorrência

- SetTimeout () e setInterval () permitem que você registre uma função a ser invocada uma vez ou repetidamente após um determinado período de tempo decorrido.
- O método setTimeout () do objeto Window agenda uma função a ser executada após um número especificado de milissegundos decorridos.
- SetTimeout () retorna um valor que pode ser passado para clearTimeout () para cancelar a execução da função agendada.
- SetInterval () é como setTimeout (), exceto que a função especificada é chamada repetidamente em intervalos do número especificado de milissegundos:

```
setInterval(updateClock, 60000);  // Call updateClock() every 60 seconds
```

Chama a função alert depois de 2000 milissegundos

```
setTimeout(function() { alert("hello world"); }, 2000);
```

e pode ser passado para
cução da função agendada.
exceto que a função
ente em intervalos do número

```
setInterval(updateClock, 60000);  // Call updateClock() every 60 seconds
```

00 milissegundos

```
; }, 2000);
```

Concorrência

- SetTimeout () e setInterval () permitem que você registre uma função a ser invocada uma vez ou repetidamente após um determinado período de tempo decorrido.
- O método setTimeout () do objeto Window agenda uma função a ser executada após um número especificado de milissegundos decorridos.
- SetTimeout () retorna um valor que pode ser passado para clearTimeout () para cancelar a execução da função agendada.
- SetInterval () é como setTimeout (), exceto que a função especificada é chamada repetidamente em intervalos do número especificado de milissegundos:

```
setInterval(updateClock, 60000);  // Call updateClock() every 60 seconds
```

Chama a função alert depois de 2000 milissegundos

```
setTimeout(function() { alert("hello world"); }, 2000);
```

clear Timeout () para cancelar a execução da
SetInterval () é como setTimeout (), exceto q
especificada é chamada repetidamente em i
especificado de milissegundos:

setInterval(updateCl

Chama a função alert depois de 2000 milisse

```
setTimeout(function() { alert("hello world"); }, 2000);
```

Concorrência

- Como setTimeout (), setInterval () retorna um valor que pode ser passado para clearInterval () para cancelar quaisquer futuras invocações da função agendada.

- ClearTimeout () cancela a execução de código que foi adiada com o método setTimeout (). O argumento timeoutId é um valor retornado pela chamada para setTimeout () e identifica qual código diferido deve ser cancelado

Avaliação da Linguagem

Criterios Gerais	C	JAVA	JavaScript
Aplicabilidade	Sim	Parcial	Não
Confiabilidade	Não	Sim	Sim
Aprendizado	Não	Não	Não!!
Eficiência	Sim	Parcial	Não
Evolutibilidade	Não	Sim	Parcial
Método de projeto	Estruturado	OO	OO, Funcional
Reusabilidade	Sim	Sim	Sim
Integração	Sim	Parcial	Parcial
Portabilidade	Não	Sim	Sim
Custo	Depende da Aplicação	Depende da Aplicação	Depende da Aplicação

Avaliação da Linguagem

Criterios Específicos	C	JAVA	JavaScript
Escopo	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim
Gerenciamento de Memória	Programador	Sistema	Sistema
Persistência dos Dados	Biblioteca de Funções	Biblioteca de classes, Serialização	Serialização (JSON)
Passagem de Parâmetros	Lista Variável e por Valor	Lista Variável, por Valor e por Referência	Por Valor e por Referêcia

Avaliação da Linguagem

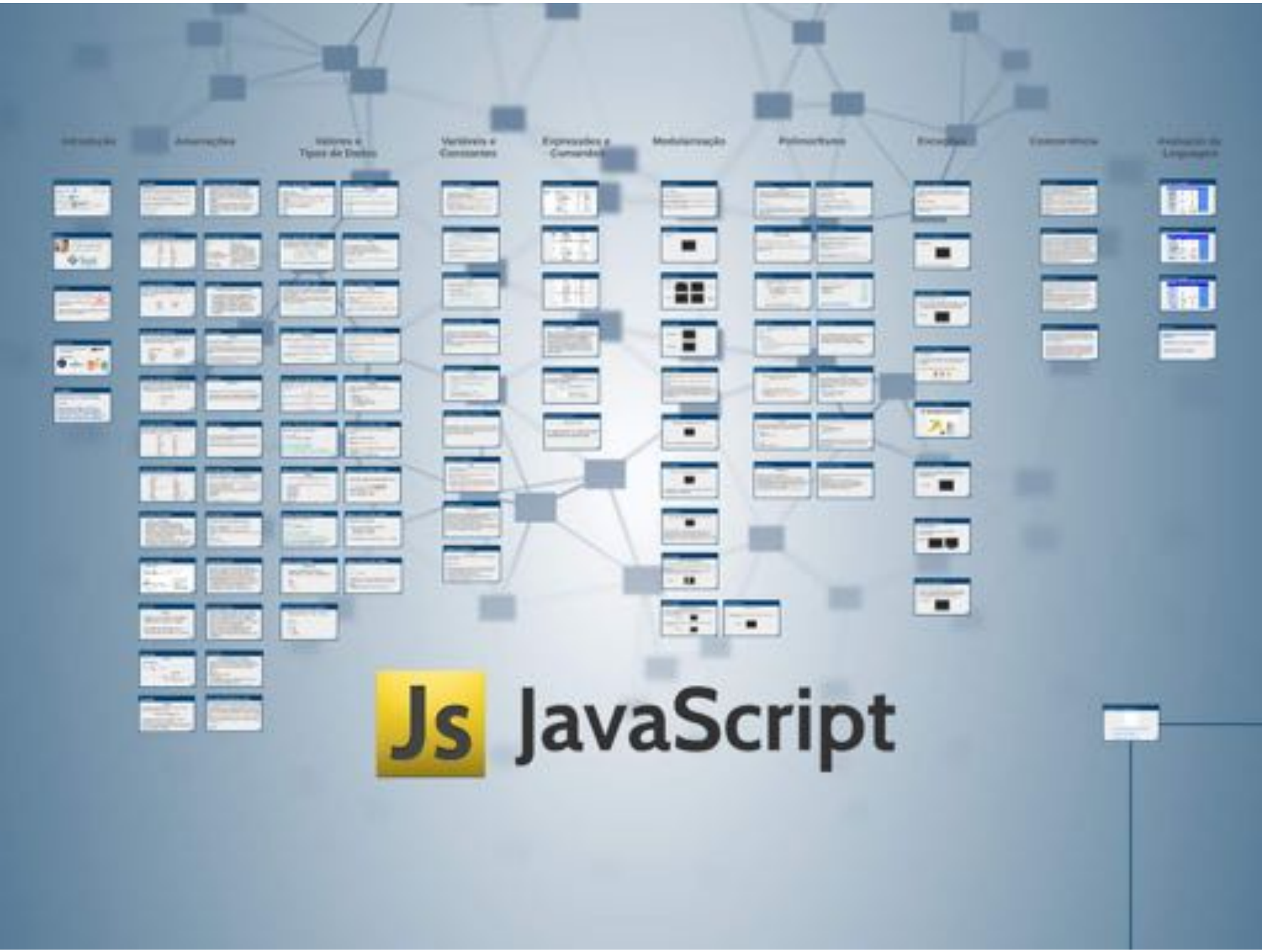
Criterios Específicos	C	JAVA	JavaScript
Encapsulamento e Proteção	Parcial	Sim	Sim
Sistemas de Tipos	Não	Sim	Sim
Verificação de Tipos	Estática	Estática/Dinamica	Dinâmica
Polimorfismo	Coerção e Sobrecarga	Todos	Todos
Exceções	Não	Sim	Sim
Concorrência	Não	Sim	Não

Referências

https://developer.mozilla.org/pt-BR/docs/Web/JavaScript

FLANAGAN, D. JavaScript: o guia definitivo.

http://www.w3schools.com/js/


Introdução
Anotações
Valores e Tipos de Dados
Variáveis e Constantes
Expressões e Comandos
Modularização
Polimorfismo
Encapsulamento
Exceções
Evolução da Linguagem
Js
JavaScript