ECMAScript e JavaScript

Susana Natividade - Driven T10

O que é a ECMA e qual a diferença entre ECMAScript e JavaScript?

JavaScript é uma *Linguagem de Programação*,
criada por Brendan Eich, ele trabalhou como programador, arquiteto e especialista de sistemas operacionais em diversas empresas, até que em abril de 1995 foi convidado para trabalhar na Netscape Corporation, onde iniciou e alavancou o desenvolvimento do Javascript.

Netscape

{.js}
JavaScript

A linguagem Javascript teve seu primeiro protótipo criado em apenas 10 dias sendo batizada com o nome **Mocha.**

Porém, quando teve seu primeiro lançamento oficial em setembro de 1995, juntamente com a versão 2.0 do navegador Netscape, foi chamada de LiveScript.

E em dezembro do mesmo ano seu nome foi alterado para Javascript.

Está bem, e onde entra o ECMAScript nesta história toda?

Você sabia que a linguagem que popularmente conhecemos como Javascript não possui mais esse nome desde 1997?

Ela se chama, há 20 anos, ECMAScript.

Antes que o Javascript se tornasse popular, para que a linguagem evoluísse obedecendo a determinados padrões e normativas, em 1996 os criadores do Javascript se associaram ao **ECMA (European Computer Manufactures Association)** ou (Associação Europeia de Fabricantes de Computadores).

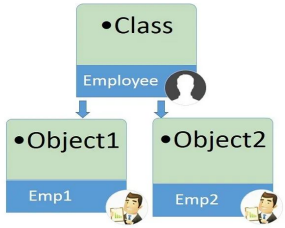
A ECMA International é uma associação industrial dedicada à padronização de sistemas de informação e comunicação.

Como o nome Javascript já havia sido patenteado pela Sun Microsystems (atual Oracle), eles definiram um novo nome à linguagem utilizando a junção das palavras ECMA e Javascript, surgindo então o ECMAScript.

ORACLE
Cloud Infrastructure

ECMAScript

O que é o ECMAScript?



Toda linguagem de programação precisa de uma padronização para reger as regras e o nivelamento da linguagem ao nível global, e para o JavaScript é isso que é o ECMAScript, é a versão oficial da linguagem, tanto que o nome JavaScript, na verdade, é uma tradição do mercado de desenvolvimento, sendo o nome oficial da linguagem ECMAScript.

Então o ECMA não passa de uma padronização criada para a linguagem do JavaScript.

A criação do ECMAScript adicionou novos recursos e funcionalidades à linguagem da programação. Dentre eles, iterar objetos, declarar variáveis utilizando *let* e *const*, bem como uma modularização de classes.

```
36
37  // Filtrando um array usando filter
38  let alunos = [
39      { nome: "Joao", nota: 90},
40      { nome: "Maria", nota: 80},
41      { nome: "Jose", nota: 50}
42  ]
43
44  var alunosAprovados = alunos.filter((aluno)=> {
45      return aluno.nota > 60;
46  });
47
48  console.table(alunos);
49  // Saída:
50  // [{ nome: "Joao", nota: 90},
51  //  { nome: "Maria", nota: 80}]
```

O ECMA funciona exatamente como o JavaScript, levando em consideração os mesmos comandos, recursos, atributos e resultados. A única diferença está na inovação e adaptação para os dias de hoje.

Afinal, para que serve o ECMA?

O ECMAScript tem o objetivo de trazer um maior dinamismo para todas as páginas que você encontra na internet. Ele possui uma linguagem de programação diretamente voltada para o *front-end*, a maior parte de seus recursos se ligam à construção de interfaces.

No entanto, com a criação das variações desse recurso, bem como de frameworks, torna-se possível utilizar o JavaScript até mesmo em uma versão *back-end*, para desenvolvimento *mobile*, que, querendo ou não, vem se tornando uma grande tendência entre as pessoas usuárias digitais.

Quais as diferenças entre ECMAScript e Javascript?

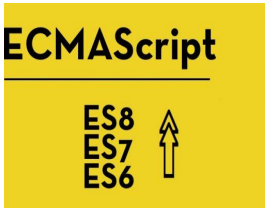
O termo ECMAScript é mais utilizado de maneira nostálgica, pelo seus criadores, a empresa Oracle.

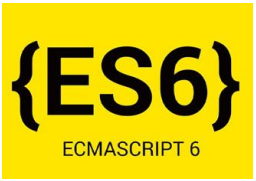
Embora dizemos que são a mesma coisa, há alguns pontos que podem diferir.
Dentre eles, estão:

- JavaScript é utilizado como base;
- ECMAScript surge como uma maneira de ajudar novos programadores;
- O ECMAScript representa uma especificação, enquanto o JavaScript se mostra uma implementação.

Em resumo, é como se o ECMAScript fosse a receita de um bolo e o JavaScript fosse o bolo de verdade, em sua forma mais pura e fiel à receita original. ActionScript, UnityScript e JScript seriam os “outros sabores”.

Versões do ECMAScript





O ECMAScript continua em evolução. Sempre que novas funcionalidades são adicionadas, o documento do registro ECMA-262 é atualizado.
Antes o nome das versões eram seguidos por um número, como ECMAScript 3 e ECMAScript 5.

Em 2015, durante o lançamento do que seria o ECMAScript 6, também conhecido como ES6, foi decidido que o nome das versões agora viriam com o número do ano de seu lançamento. Por isso que o ES6 também ficou conhecido como ECMAScript 2015 (ou ES2015).

Então é muito comum que ao fazer referência à linguagem, chamamos de JavaScript. Mas ao se referir a uma versão, chamamos de ECMA ou ES + ano.

Em 2015, com o lançamento do ES6, muitas mudanças começaram a acontecer no JavaScript

Objetivos do ES6

-› Ser uma linguagem melhor para construir aplicações complexas;
-› Tornar a linguagem mais acessível, versátil e enxuta;
-› Resolver problemas antigos do JavaScript;
-› Facilidade no desenvolvimento de *libraries*

Algumas das novas features:

-› Funções *Arrow Functions*;
-› Funções *map*, *filter* e *reduce;*
-› Funções *some* e *every*;
-› Função *find*;
-› Classes, módulos
-› Declaração de variáveis com let e const
-› Entre outros ...

Com o lançamento do ES6
Quais foram as mudanças?

- Default Parameters
- Template Strings
- Destructuring

Default Parameters ou Parâmetros padrão

Os parâmetros de funções tem *undefined* como valor *default*. Porém, em alguns casos, pode ser necessário utilizar um outro valor. Com a versão atual do JavaScript (ES5) nós já podemos fazer isso dessa forma:

```
var multiply = function(x, y) {
   y = y | 1;
   return x * y;
};

multiply(3, 2); // 6
multiply(3); // 3
```

default.js hosted with ❤ by GitHub

O ES6 introduziu uma nova forma, bem mais simples, de se fazer isso. Basta adicionar o valor *default* na definição do parâmetro desejado:

```
const multiply = (x, y = 1) => {
  return x * y
}

multiply(3, 2) // 6
multiply(3) // 3
```

default2.js hosted with ❤ by GitHub

Ou então, com apenas uma linha:

```
const multiply = (x, y = 1) => x * y

multiply(3, 2) // 6
multiply(3) // 3
```

default3.js hosted with ❤ by GitHub

Template Strings ou Cadeias de modelo

As **template strings** vieram para simplificar a interpolação de variáveis e expressões em strings.

Enquanto no **ES5** era feito da seguinte forma:

```
var name = 'Calos';
var country = 'Brazil';
var phrase = 'My name is ' + name + '.\n I\'m from ' + country;
```

No **ES6**:

```
const name = 'Calos';
const country = 'Brazil';
const phrase = `My name is ${name}.
I'm from ${country}`;
```

Note que as **template strings** são delimitadas pelos caracteres `` ` `` e as variáveis são interpoladas com `${variable}`. Note também que as quebras de linha são interpretadas.

Destructuring ou Desestruturando

No **ES5**, quando necessário atribuir variáveis com os valores de um **array** ou **objeto** tínhamos que fazer algo tipo —-›

```
const colors = ['red', 'green', 'yellow'];
const red = colors[0];

const dog = { name: 'Joe', owner: 'Carlos' };
const name = dog.name;
```

Uma declaração **destructuring** cria variáveis a partir dos valores de um **array** ou das chaves de um **objeto**. Exemplo com um **array** —-›

```
const colors = ['red', 'green', 'yellow'];
const [red, green, yellow] = colors;
```

Exemplo com um **objeto** —-›

```
const city = { name: 'Ribeirao Preto', temperature: 42 };
const { name, temperature } = city;
```

O que é Strict Mode?

```
'use strict';
```

O *strict mode* basicamente ajuda a melhorar a qualidade do código. Ou seja, o código será executado de forma mais rigorosa.

```
"use strict";

const title = "Sample";
```

Quando ativamos o *strict mode* somos obrigados a ser mais regrados na escrita de nossos códigos.

Ele nos força a desenvolver nossos códigos de forma mais correta, não permitindo que façamos algumas coisas do tipo: utilizar uma variável que não havia sido declarada.

Além de evitar problemas futuros com novas versões do JavaScript. Também nos ajuda a detectar erros em nosso código com mais facilidade.

Benefícios de utilizar o modo use strict

—-› O strict mode elimina alguns erros silenciosos que passariam batido do JavaScript e os faz emitir erros

—-› Corrige alguns erros que tornam o JavaScript difícil de ser otimizado

—-› Os códigos no modo estrito rodam mais otimizados e velozes do que os códigos no ‘modo normal’

—-› Em relação à segurança é bem vantajoso, pois ele proíbe ou marca como um erro quando ações não seguras são realizadas pelo código.

—-› O modo strict proíbe algumas sintaxes (modo de escrever o código) que provavelmente serão descontinuadas em versões futuras do ECMAScript

—-› Para iniciantes, o strict mode desabilita algumas funcionalidades que são confusas da linguagem, o que facilita bem dependendo do caso

Como usar o Strict Mode

Há **duas maneiras** para se utilizar o strict mode!

Usado no escopo global para o script inteiro :

```
"use strict";

const title = "Sample";
class Heading {
    render() {
        document.querySelector('body').innerHTML = title;
    }
}
export default Heading;
```

Aplicado individualmente em funções específicas :

```
function minhaFunção() {

  "use strict";

  a = 'teste' // isso já daria problema, a não está declarada

}
```

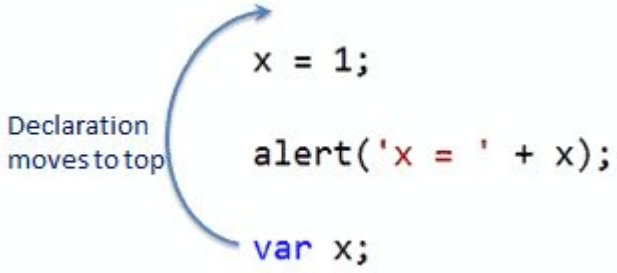
< Hoisting? />

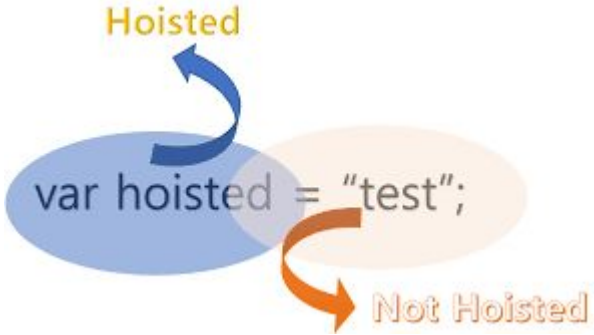
JS

O que é Hoisting?

Basicamente, quando o Javascript compila todo seu código, todas as declarações de variáveis usando `var` são hoistiadas/levadas ao topo de suas funções/escopo local (se declaradas dentro de uma função),
ou ao topo do escopo global (se declaradas fora de uma função) independentemente de onde a declaração foi feita.

É isso que quer dizer o "*hoisting*".





Exemplos básicos para demonstrar o impacto do hoisting :

O que o comando `console.log` mostrará?

```
console.log(meuNome);
var meuNome= 'Bob';
```

-> undefined

As variáveis são movidas pro topo de seus escopos quando o Javascript compila, porém a única coisa que é movida para o topo são as declarações de variáveis, não o valor real delas!

```
var meuNome;

console.log(meuNome);

meuNome= ‘Sunil’;
```

É por isso que o `console.log` devolverá **undefined,** pois ele reconhece que a variável **meuNome** existe, porém ela não teve um valor inserido até a terceira linha.

Qual a diferença de var, let e const?

Principais diferenças entre var, let e const são:

⟶ Variáveis de **var** podem ser atualizadas e declaradas novamente dentro de seu escopo (região do código).

⟶ As variáveis de **let** podem ser atualizadas, mas não podem ser declaradas novamente.

⟶ As variáveis de **const** não podem ser atualizadas nem declaradas novamente (são utilizadas para valores constantes, como o valor de pi, por exemplo).

⟶ Enquanto **var** e **let** podem ser declarados sem serem inicializadas, variáveis declaradas com **const** devem ser inicializadas no momento da declaração.

var (**escopo global**) - escopo fora do bloco.

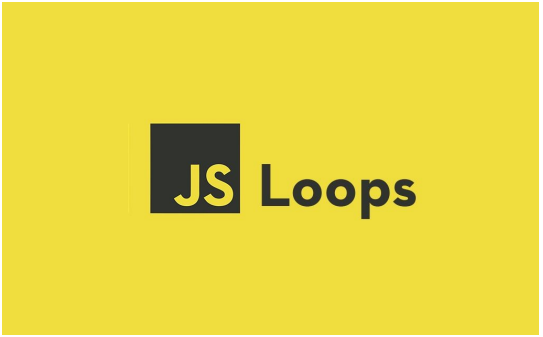
- a variável **var** é declarada e iniciada no escopo da função
- **não** respeitando bloco e permitindo a redeclaração e reatribuição.

let (**escopo local**) - escopo restrito ao bloco.

- a variável **let** é declarada no escopo da função, mas só é inicializada posteriormente
- respeitando bloco e permitindo reatribuição, mas **não** permite a redeclaração.

const (não permitindo reatribuição e nem redeclaração) - constante.

- a variável **const** é declarada no escopo da função, mas só é inicializada posteriormente
- respeitando bloco e **não** permitindo reatribuição **nem** redeclaração.



Qual a diferença entre for, forEach e for.. of ?

Loop For

O loop for JavaScript é usado para iterar através da matriz ou dos elementos por um número especificado de vezes. Se uma certa quantidade de iteração for conhecida, ela deve ser usada.

SINTAXE:

```
for (initialization; condition; increment)
{
   // code to be executed
}
```

EXEMPLO:

```
<script>
for (let i = 1; i <= 5; i++)
{
    document.write(i + "<br/>");
}
</script>
```

SAÍDA:

```
1
2
3
4
5
```

Loop ForEach

O método forEach() também é usado para percorrer arrays, mas usa uma função diferente do clássico “loop for”. Porém ele passa uma função de retorno de chamada para cada elemento de uma matriz junto com os parâmetros abaixo:

- Valor atual (obrigatório): o valor do elemento atual da matriz
- Índice (opcional): O número de índice do elemento atual
- Array (opcional): O objeto array ao qual o elemento atual pertence

Precisamos de uma função de retorno de chamada para percorrer uma matriz usando o método forEach. Para cada elemento da matriz, a função será executada. O callback deve ter pelo menos um parâmetro que represente os elementos de um array.

SINTAXE:

```
números = [1, 2, 3, 4, 5];

números.forEach(função(número) {
   // código a ser executado
});
```

SAÍDA:

```
1
2
3
4
5
```

ForEach com uma função de retorno de chamada.

←

SINTAXE:

```
numbers = [1, 2, 3, 4, 5];

numbers.forEach((number, index) => {
    console.log('Index: ' + index +
                ', Value: ' + number);
});
```

SAÍDA:

```
Índice: 0, Valor 1
Índice: 1, Valor 2
Índice: 2, Valor 3
Índice: 3, Valor 4
Índice: 4, Valor 5
```

ForEach com uso do parâmetro index no método forEach.

←

Loop For... of

O loop for...of cria um loop de iteração sobre objetos iteráveis, como por exemplo Objetos construídos com Array, String, Map, Set,... ele chama uma função com instruções a serem executadas para o valor de cada objeto distinto.

Variável
A cada iteração, um valor de uma propriedade diferente é atribuido à *variável*.

Iteravel
Objeto cujos atributos serão iterados.

```
for (variavel of iteravel) {
  declaração
}
```

Iterando sobre um Array

```
let iterable = [10, 20, 30];

for (let value of iterable) {
  console.log(value);
}
// 10
// 20
// 30
```

Iterando sobre uma String

```
let iterable = "boo";

for (let value of iterable) {
  console.log(value);
}
// "b"
// "o"
// "o"
```

Podemos usar const se você não for modificar a variável dentro do bloco.

Fim ...