**ECMASCRIPT**

1. **O que é?**

Para um melhor entendimento, é necessário saber sobre **a origem do JavaScript**

- O JS for originalmente desenvolvido por Brendan Eich em 1995, para ser utilizado no Navigator2

- Rapidamente JavaScript ganhou popularidade e aceitação como linguagem de script, cliente side de páginas web

- Chamou a atenção da Microsoft que desenvolveu uma implementação da linguagem, que foi chamada de JScript para evitar problemas de licenciamento. O JScript foi inserido na internet em 1996

- Por volta de 1997, foi anunciado que o JavaScript seria enviado para uma equipe internacional, a Associação Européia dos Fabricantes de Computadores(ECMA), com o intuito de fazer do Javascript uma linguagem de padrão industrial

- A ECMA publicou então, o padrão 262, se popularizando como ECMAScript

- O ECMAScript é uma especificação, sendo assim, não está ligado diretamente aos navegadores, a ideia é aumentar a compatibilidade das aplicações em browsers diferentes

- A primeira versão foi disponibilizada em 1997, desde então, foram surgindo mais versões, sendo a mais significativa, a versão 5, padronizada em 2009, suportada de forma quase completa em browsers modernos, em junho de 2015, foi publicada a versão 6, ES6 ou ECMAScript 2015

1. **Babel**

**-** Um dos compiladores mais completos em termos de transcrição de código escrito em ES6 para ES5

- Com ele, os códigos escritos em ES6 serão transcritos para o ES6, versão suportada pelos browsers em 2018, porém, atualmente os browsers já suportam as features ES6, portanto, a configuração do Babel não será mais necessária

<https://babeljs.io/> > Setup > In the browser > babel-standalone > copiar cod js

1. **Var e Let**

A diferença entre eles, é que o Let preserva o escopo aonde a variável foi criada, ou seja, só pode ser utilizada depois do bloco de código que foi criada; diferentemente de variáveis declaradas com var podem ser utilizadas antes de sua declaração

São três os tipos de escopos: global, função e bloco

Em termos de escopo de bloco variáveis declaradas como var podem ser utilizadas dentro de funções e dentro de blocos

Variáveis, entre outras coisas declaradas no escopo global, fica disponível para tudo, o global é basicamente o objeto window

1. **Introdução a Orientação a Objetos – Considerações preliminares**

***Paradigmas de programação***

Paradigmas – Padrão; conduta; uma forma de se fazer alguma coisa

***Procedural/Estruturado/Imperativo***– Funcional/lógico

switch{}, if(){}else{}

***Orientado a Objetos***– Concorrente/reativo

While, do while

***Paradigma Procedural***

- Chamada de procedimentos (procedures) para manipulação de dados

***Paradigma de Orientação a Objetos***

- Estruturas de dados com comportamentos

Atributos são declarados dentro de objetos

Var > classe > objeto > atributo

**Pilares da orientação a objeto** – abstração, encapsulamento, herança, polimorfismo

***Abstração*** – **Entidade :** compreensão de que um objeto do mundo real deve passar para dentro da aplicação; **Identidade :**  identificação única da identidade; **Características :** atributos que especificam as características da entidade; **Ações :** métodos de comportamentos dos objetos na vida real, que fazem sentido no contexto da aplicação.

- Habilidade intuitiva de conhecimento das coisas

- Compreende coisas do mundo real e trás para o código

- As ações e características tem a necessidade de serem dosadas, para que que faça sentido na aplicação

Método **.this** – ajusta o contexto de um determinado atributo ou método, faz referência ao próprio elemento HTML que executa a função

**Encapsulamento**

Seguro, Reutilizável

**Conceitual -** Consiste em encapsular um objeto, de modo em que ele se torne um objeto seguro. Isso significa que o objeto terá condições de dizer o que há dentro dele e está disponível para o sistema. O encapsulamento é uma forma de dar visibilidade para os atributos e métodos que formam o objeto de acordo com nossas necessidades(um ex.: Definir um método ou atributo como publico ou privado).

A diferença que essa abordagem tem no código, é que se um método ou atributo for privado, só poderá ser acessado através do próprio objeto, contudo, percebemos que o encapsulamento cria uma camada de segurança evitando que a aplicação acesse recursos privados do objeto.

**Técnico** – Permite que métodos e atributos sejam herdados por outros objetos. Esse procedimento se chama herança. Existe o conceito de encapsulamento e ele é importante para o paradigma, porém, o ES6 não suporta os operadores de visibilidade. Pelo menos na versão atualizada do ECMAScript 2015.

Portanto, para esse princípio ser suportado, foi estabelecida uma convenção, uma forma de escrever código, que indique a nossa intenção, porém, ainda assim, o código não reconhece.

***Restringindo atributo ao objeto:***

<script>

        class Tv {

            constructor() {

                this.\_relacaoCanais = [2,4,5,7,10]//underline restringe o acesso do atributo ao objeto

                this.\_canalAtivo = 5

                this.\_volume = 5

            }

        }

        let tv = new Tv()

        /\*console.log(tv.\_canalAtivo) - atributo ainda pode ser acessado assim, porém, quando coloca-se

        o underline, sugere-se que o programador, não tente acessar o objeto dessa forma.

        o ideal, seria acessá-lo com o método GET, que retorna o atributo para nós\*/

**MÉTODOS GETTERS E SETTERS (GET E SET)**

- Muito utilizados dentro de classes, para nos dar a possibilidade de ***recuperar*** e ***setar*** atributos privados.

**Herança**

Reutilizável, Manutenção

A herança trás duas vantagens para o código, são essas:

- O código se tornará mais reutilizável e menos redundante(excessivo – não haverá excesso de código)

- Manutenção posterior do código.

Um exemplo da utilização de **herança** no Javascript, é:

- Quando há atributos em comum entre duas classes, pode-se criar uma classe com os mesmos e criar outras apenas para os outros atributos. O mesmo pode acontecer com métodos.

**constructor** – serve para armazenar atributos relacionados a uma classe, com ele, usa-se o método this, que ajusta o contexto de um determinado atributo

constructor() {

this.atributo

this.atributo

}

 let cachorro = new Cachorro() //instância da base cachorro, baseada no modelo Cachorro, criando assim um objeto

        let passaro = new Passaro()

        //uso incorreto dos caracteres(letras maiúsculas e minúsculas podem gerar erros)

Uma relação de herança, é basicamente criada por uma Superclass e Subclass

Superclass – Classes que contém métodos e atributos herdadas por subclasses

Subclass – classes especializadas, ou seja, classes baseadas em uma outra classe, que especializam atributos e métodos para algo específico

indicar que uma classe é herança de hoje ex.:

 class Cachorro extends Animal {}

nesse exemplo, oque o código está dizendo, é:

crie um objeto baseado na classe Cachorro que estende/herda atributos e métodos da classe Animal

**Herança progride conforme extensão de classes, é como uma cascata**

Caso exista uma classe com muitos métodos ou atributos semelhantes a de uma subclass, mas que também é semelhante a uma superclass que herdou alguns atributos e métodos para a subclass, é possível utilizar o recurso **extends** a partir da subclass, deste modo, essa nova subclass, herdará dos atributos e/ou métodos da superclass e da subclass; pois como foi dito, a extensão de classes é como uma cascata.

um exemplo é:

class Papagaio extends Passaro{

            constructor() {

                this.sabeFalar = true

            }

        }

como papagaio, tem muitos atributos e métodos iguais ao da class passaro, ela pode herdar os mesmos dessa subclass

**Operador Super – super ()**

- Fornece acesso para o método *constructor()* da classe pai, portanto, sua inclusão é sempre dentro do *constructor()* da classe filho

- Se a classe tiver o operador *extends*, no constructor dessa respectiva classe, deve haver a inclusão do operador *super()*, e esse operador deve ficar antes de qualquer atributo *this.*

- O *super()* nas classes filho dá acesso ao *constructor()* da classe pai, tendo como colocar valores em parâmetros adicionados no *constructor()* dito.

- Esse parâmetro da classe pai, tem que ser também um atributo dentro da mesma, para funcionar

- Pode ser adicionado mais de um parâmetro

Exemplificando para melhor entendimento:

class Animal {

            constructor(cor, tamanho, peso) {

                this.cor = null

                this.tamanho = null

                this.peso = null

            }

            dormir() {

                console.log('Dormir')

            }

        }

        class Passaro extends Animal {

            constructor(bico, cor, tamanho, peso) {

                super(cor, tamanho, peso)

                this.bico = bico

            }

            falar() {

                console.log('Falar')

            }

        }

        class Papagaio extends Passaro{

            constructor(sabeFalar, cor, tamanho, peso) {

                super('Médio', cor, tamanho, peso)

                this.sabeFalar = null

            }

        }

        let papagaio = new Papagaio(true, 'Verde', 25, 350)

        console.log(papagaio)

        let papagaio2 = new Papagaio(false, 'Branco', 10, 80)

        console.log(papagaio)

Foram definidos os parâmetros ***cor, tamanho e peso*** na classe pai. Esses parâmetros poderiam ter seus respectivos valores, na classe filho, porém, foi levado em conta que pode haver muitos valores diferentes na classe papagaio. Por isso, foi definido apenas os parâmetros, e os valores seriam colocados na linha de código em que o objeto será exibido no browser, sendo assim, pode-se ter mais de um, como no exemplo acima.

***Polimorfismo***

*Reutilizável, Sobrescrita de métodos*

- O polimorfismo nada mais é do que a sobrescrita de métodos

- Na herança, tem um objeto pai e um objeto filho, sendo que o objeto filho herda os atributos e métodos no objeto pai, mas, não necessariamente o objeto filho tem que se comportar da mesma forma que o objeto pai. Nessas situações nós corrigimos esses comportamentos através da escrita de métodos.

1. **Objetos literais**

- Já estava disponível em versões anteriores do Javascript

- É um objeto que já é descrito dentro do código

***Para definir que é um objeto literal, é necessário ter:***

- abertura e fechamento de chaves

- pares de nome, atributo ou chave e valor(nome – atributo/método) – no caso de um método o valor será uma função, que é a ação executada pelo método

- esses pares funcionam como atributos do objeto

- esses conjuntos de nome e valor são separados por uma vírgula, isso indica que aquele par teminou

- nome e valor são separados por 2 pontos

- não é necessário um procedimento de instância, pois o objeto já está descrito

//acessar objeto

        produtoLiteral.verDescricao()

        //----------------------------------------as vezes não é necessário classes e usar um objeto literal facilita

        /\*formulario -> servidor

            recupera dados e monta um objeto Literal

                obj literal -> JSON -> HTTP -> Server -> Armazena\*/

        //JSON é algo diferente de obj literal.

***Objeto único***

Todo objeto literal é único. Um exemplo:

let assinatura = {

            idCliente: 1000,

            descricao: 'Acesso a internet',

            status() {

                console.log('Ativo')

            }

        }

        console.log(assinatura.descricao)

        /\*variável y, irá conter mesmos atributos e método que variável assinatura, funciona de forma contrária

        a herança

        os atributos e métodos, poderão ser acessados pela var 'y' ou 'assinatura', pois as duas

        contém os mesmos valores\*/

        let y = assinatura

        console.log(y.descricao)

        ///

        y.descricao = 'Internet + tv + telefone'

        console.log(assinatura.descricao)

        console.log(y.descricao)

        //o console das duas variáveis resultarão na mesma coisa

Por mais que eu passe uma variável que contém atributos e métodos específicos para outra, valerá a mesma quantidade de atributos e métodos.

1. **Funções construtoras**

Tem o papel de definição da classe do objeto que queremos

criar

**Criando objetos**

Pode-se criar várias instâncias desse objeto a partir desse modelo definido através de uma função construtora

Semelhante a sintaxe de Classe

        let carro = new Carro()//semelhante a como é feita a criação de objetos a partir de classe

**Encapsulando atributos e métodos**

- Operadores que permitem atribuir níveis de visibilidade para os atributos e métodos de um objeto;

- Em funções construtoras nós temos uma flexibilidade porque nós podemos criar atributos que farão parte do escopo da função;

- Os atributos que serão ativados eles são criados sem um operador this, ou seja, são realmente variáveis do contexto da função;

- Dessa forma será possível aplicar o princípio no encapsulamento de forma um pouco mais consistente dentro de funções construtoras já que graças a essa flexibilidade nós podemos criar atributos públicos e atributos que ficarão escondidos.