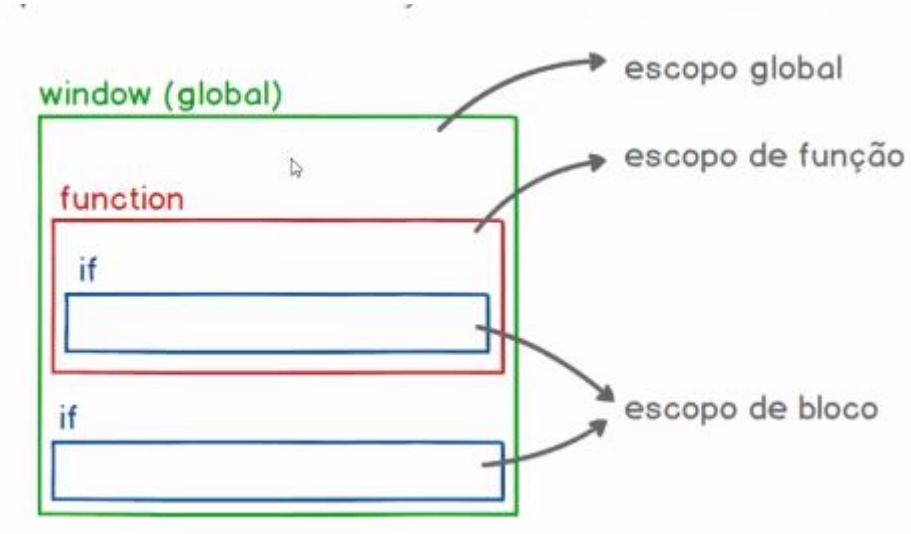
Ecma Script 6

Sumário

Operadores de declaração

Veremos as diferenças entre Var e Let, mas basicamente a única diferença é que o Let preserva o escopo onde a variável foi criada.

Antes, vamos ver um pouco sobre escopo. Existe 3 tipos de escopo: global, de função e de bloco.



Veja na prática alguns exemplos.

Variáveis declaradas com Var em escopo global, podem ser utilizadas em escopos de função e de bloco. Ou seja, está disponível para todos os escopos.

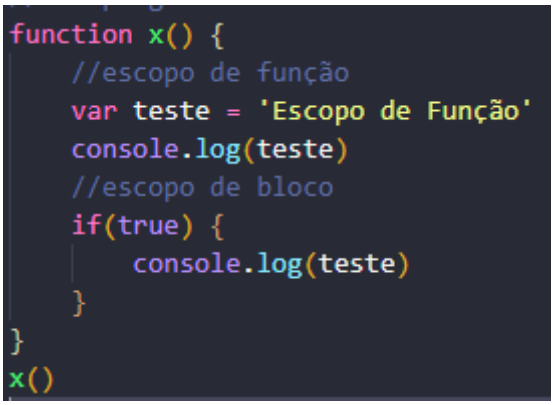
```
//escopo global
var teste = 'Escopo Global'

function x() {
    //escopo de função
    console.log(teste)
}
x()

if(true) {
    //escopo de bloco
    console.log(teste)
}
```

```
Escopo Global
Escopo Global
```

Variável Var declarada dentro de um escopo de função só poderá ser utilizada pela própria função ou por um escopo de bloco dentro da função.



```
function x() {
    //escopo de função
    var teste = 'Escopo de Função'
    console.log(teste)
    //escopo de bloco
    if(true) {
        console.log(teste)
    }
}
x()
```

```
Escopo de Função
Escopo de Função
```

Variáveis criada dentro de um escopo de bloco, elas forem elevação (*hoisting*). Se a variável foi criada em um escopo de bloco dentro do escopo global, ela sofre *hoisting* para o escopo global. Se foi criada no escopo de bloco dentro de um escopo de função, ela sofre *hoisting* para o escopo da função.

```
//escopo global
function x() {
    //escopo de função
    console.log(teste)
    //escopo de bloco
    if(true) {
    }
}


if(true) {
    //escopo de bloco
    var teste = "Escopo de Bloco"
}

x()

console.log(teste)
```

```
Escopo de Bloco
Escopo de Bloco
```

E como dito anteriormente, o Let preserva o escopo em que foi criado.

```
if(true) {
    //escopo de bloco
    let teste = "Escopo de bloco"
}
console.log(teste)
```

```
console.log(teste)
            ^

ReferenceError: teste is not defined
```

Como o Let não sofre *hoisting*, também não tem alteração em seu escopo, logo um erro de variável não definida é retornado.

Por fim, temos o Const. Ele funciona de forma semelhante ao Let, porém ele é uma constante, ou seja, não sofre alteração de valor ao longo do processamento do script.

```
const serie = "teste"

serie = "Mudar valor"

console.log(teste)
```

```
serie = "Mudar valor"
      ^

TypeError: Assignment to constant variable.
```

Lembrando que ela segue os mesmos critérios de escopo do Let.

Template String

Também conhecido como interpolação, é um método usado para usar um método diferente de concatenação.

```
let nome = "Allan Vigiani"
console.log('Nome é :' + nome)
```

Esse método era o que conhecemos até agora, porém vamos ver o uso do Template String.

```
let nome = "Allan Vigiani"
console.log(`O nome é ${nome}`)
```

Usa-se crases e a variável dentro de um '${}'.

Pode-se também formatar o texto que será exibido.

```
let nome = "Allan Vigiani"
console.log(`
    O nome
    é
    ${nome}`
)
```

```
O nome
é
Allan Vigiani
```

Arrow function

É uma forma mais curta de se escrever funções anônimas.

```
let quadrado = function (x) {
    return x * x
}

console.log(quadrado(2));
```

Função anônima normal, agora veremos essa mesma função em forma de uma *arrow function.*

```
let quadrado = x => {
    return x * x
}

console.log(quadrado(2));
```

Caso exista mais de um parâmetro, é necessário o uso dos parênteses.

Orientação a Objetos

Primeiro vamos ver um termo importante da orientação a objetos.

- Paradigma = é um padrão, conduta, de se fazer algo.

Vejamos a diferença de um paradigma procedural de um de orientação a objetos.

Na orientação a objetos nós criamos, com base em uma classe, um objeto que é uma abstração de algo, no caso acima, uma abstração de uma calculadora. E uma abstração é a forma de se interpretar alguma coisa.

Na programação orientada a objetos, existe 4 pilares: Abstração, Encapsulamento, Herança e Polimorfismo.

- Todos os códigos estarão disponíveis no meu repositório de acordo com o tema. Repositório.

Ao atender os 4 pilares, temos condições de criar aplicações orientadas a objetos.

Abstração

A ideia é entender alguma coisa no mundo real e trazer pra dentro da aplicação. Para entender a abstração deve-se conhecer outros 4 termos importantes.

- Entidade = É o objeto, o propósito principal da aplicação que veio do mundo real, como por exemplo uma conta bancária.

- Identidade = É a referência que se faz com a entidade, ou seja, com o objeto. Como por exemplo 'y = new ContaBancaria()' .

- Características = Também conhecidos como atributos, são realmente características de um objeto (entidade). Como por exemplo, agencia, número da conta, saldo, limite entre outras.

- Ações = Também conhecidas como métodos, são os comportamentos daquele objeto. Como por exemplo, depositar, sacar, consultar saldo, transferir e entre outras.

Em resumo, a abstração é modo de saber pensar, de saber como você transferirá elementos do mundo real para sua aplicação sabendo o que será importante ter em sua aplicação, dosando as características e ações que estarão presentes.

O termo *class* representa o modelo da entidade, ou seja, o modelo do objeto. Sendo, por exemplo, o a conta bancária o objeto em si.

Os atributos (características) precisam ser construídos dentro do objetos e para isso utiliza-se o método 'constructor()'. Utiliza-se também o '*this*', que faz com que um atributo seja conectado ao contexto do objeto.

Ações são definidas da mesma forma que uma função sem declarar a palavra '*function*', porém são chamadas de métodos.

Encapsulamento

Do ponto de vista conceitual, consiste em encapsular o objeto e torna-lo seguro, ou seja, o objeto terá condições de dizer o que está disponível para o sistema.

O encapsulamento permite que métodos e atributos sejam herdados por outros objetos e isso é conhecido como Herança, um dos pilares da programação orienta a objetos.

Vamos aprender sobre os métodos *getters* e *setters*. São muito utilizados em classes para recuperar e 'setar' atributos privados.

- get = indicamos um atributo que queremos recuperar e através de um função, retornamos o que queremos. O valor primeiramente atribuído, não poderá ser sobrescrevido. Em resumo, recupera um valor

- set = recebe um parâmetro e esse parâmetro é atribuído e sobrescreve o valor do *get.* Em resumo, atualiza um valor.

Herança

Trás dois pontos positivos, o primeiro é que nosso código se tornará reutilizável e com isso a manutenção se tornará mais simples e eficiente.

Vamos trabalhar com duas classes, cachorro e pássaro. Eles são objetos distintos, porém, eles possuem atributos em comuns como tamanho e cor, e também alguns métodos como, comer e dormir.

Nesse caso poderíamos criar uma terceira classe chamada 'Animal' que vai conter todas os atributos e métodos em comum.

Para uma subclasse herdar os atributos e métodos de uma superclasse usamos o *extends*.

```
class Cachorro extends Animal{
    constructor() {···
    correr() {···
    rosnar() {···
}
```

Utilizaremos também o métodos *super* dentro do *constructor()*, porém veremos ele em específico mais a seguir.

```
constructor() {
    super()
    this.orelha = 'Pequenas'
}
```

O operador *super()* fornece acesso para o método *constructor()* da superclasse, ou seja, ele é sempre colocado dentro das subclasses. Outro detalhe é que ele deve entrar sempre na frente dos demais atributos.

Ao passar os parâmetros desejados, para os atributos, o *constructor()* que receberá.

```
class Papagaio extends Passaro{
    constructor(sabeFalar) {
        super()
        this.sabeFalar = sabeFalar
    }
    falar() {
        console.log('Falar')
    }
}

let papagaio = new Papagaio(true)
```

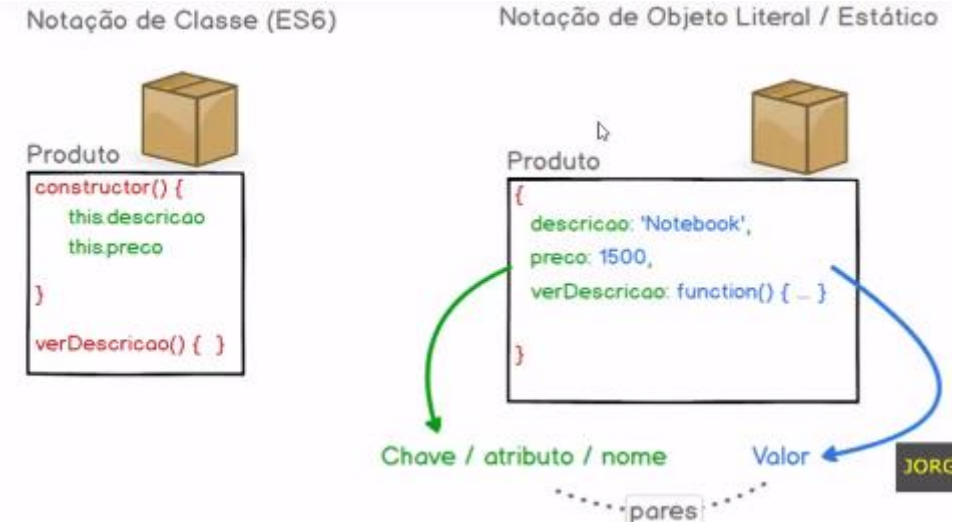
A partir do método *super()*, podemos passar parâmetros que serão recebidos pela classe pai (superclasse).

Polimorfismo

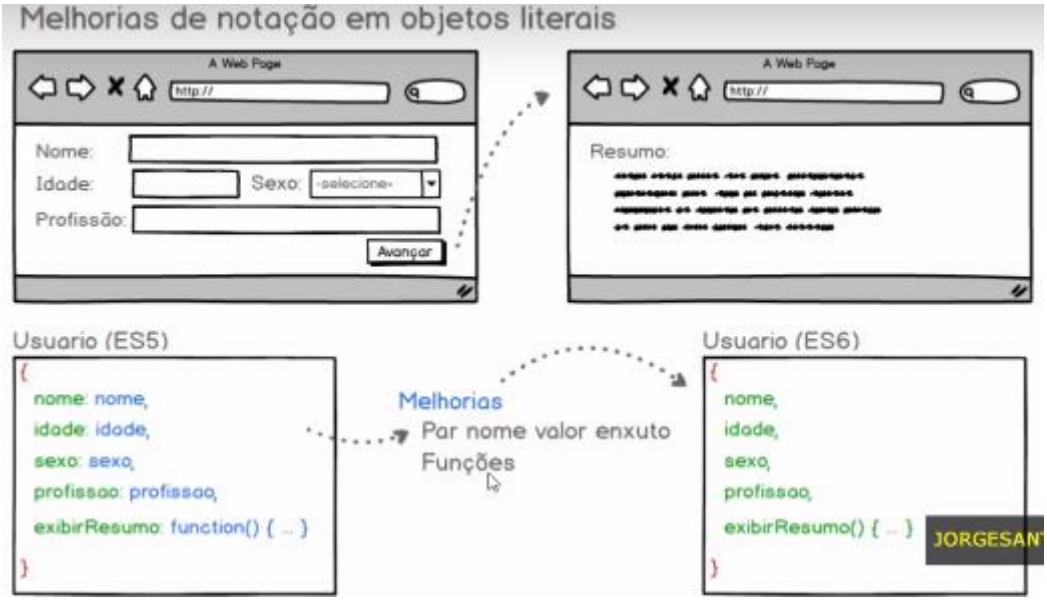
O polimorfismo é o principio da sobrescrita de métodos. Quando trabalhamos com a herança dos objetos filhos que herdam atributos e métodos do objeto pai, mas não necessariamente o objeto filho tem que se comportar como o objeto pai e assim usamos o pilar do polimorfismo.

Objetos literais/estáticos

É um objeto descrito dentro do código. É criar um objeto através de um objeto literal. Veja a ilustração da diferença entre um objeto de classe e um objeto literal.

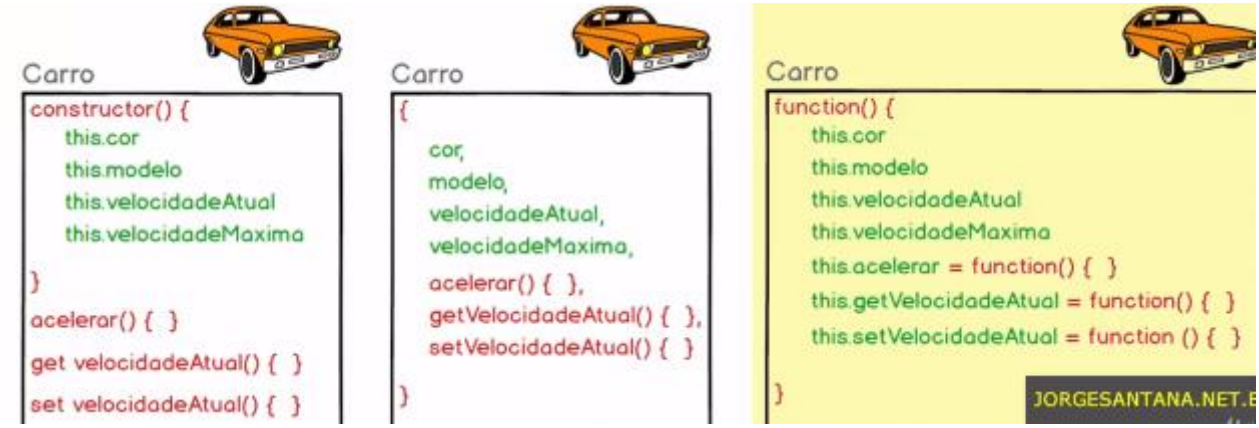


No ES6 temos também uma melhoria na forma de escrever, deixando mais enxuto.



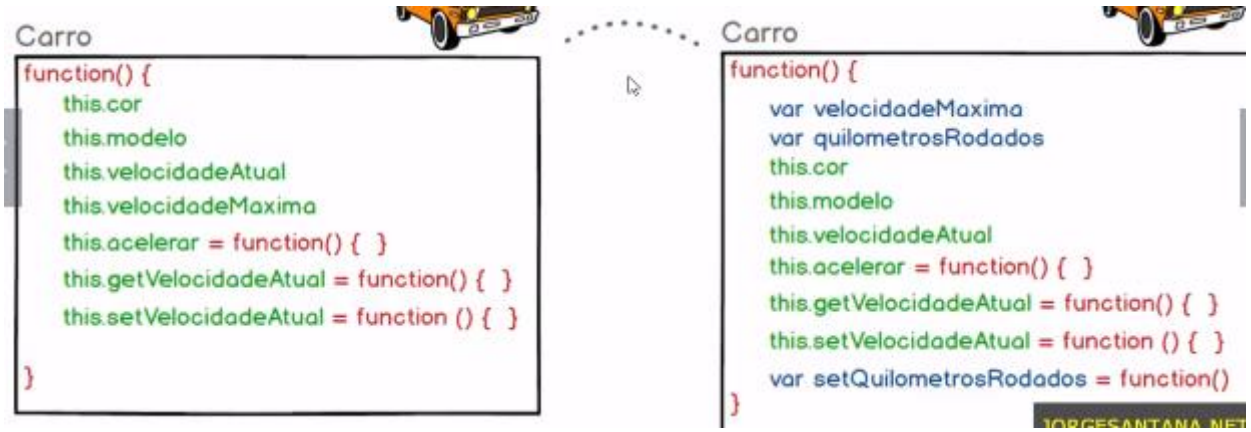
Funções construtoras

Veremos com criar objetos através de funções construtoras e não através de classes e nem de objetos literais. Veja a diferença entre elas.



Utilizaremos esse recurso pois a essência do Java Script são as funções, mesmo esse modo sendo antigo, ainda vale a pena.

Podemos também encapsular atributos e métodos. Veja na imagem.



Funções Factory

Podemos implementar uma fábrica para a criação de objetos durante a execução, passando parâmetros para a função.

```
//Factory Functions
let Bicicleta = function(cor, marcha, aro) {
    return {
        cor,
        marcha,
        aro,
        pedalar() {
            console.log('Pedalar')
        }
    }
}

let bicicleta = Bicicleta('Preta', '18', 12)
console.log(bicicleta)
```

Prototype

Todos os objetos em Java Script descendem de Object.