Curso de JavaScript, NodeJS e ES6+

Oferecimento: Student Branch IEEE UEM - https://www.ieeeuem.com.br

- **Rodolfo Lemes Saraiva - rodolfo_fero@hotmail.com**

Maringá, 27 de Julho de 2020

Ferramentas

Neste curso os recursos que iremos utilizar para realiza-lo são:

- VSCode: Será nosso editor de texto para arquivos JavaScript.
- NodeJS: Responsável por executar nossos arquivos JavaScript no computador, fora do navegador. Dentro da instalação do NodeJS, já está o npm.
- Yarn: Outra opção de packager manager, caso não queira utilizar o npm.

JavaScript

Introdução

> **JavaScript é uma linguagem de script orientada a objetos, multiplataforma. É uma linguagem pequena e leve. Dentro de um ambiente de host (por exemplo, um navegador web) o JavaScript pode ser ligado aos objetos deste ambiente para prover um controle programático sobre eles.**
> O que é JavaScript?. MDN Web Docs. Um dos melhores sites de documentação para desenvolvedores.

A linguagem pode ser utilizada no lado cliente, *front-end*, utilizando-se das ferramentas, bibliotecas e frameworks para isso. Além de ter integralidade total com o HTML.

A linguagem também pode ser utilizada no lado servidor, *back-end*, fornecendo ferramentas para comunicação de banco de dados, disponibilidade para comunicação entre aplicações e entre outros.

Para rodar os nossos códigos temos duas opções:

- Utilizar o NodeJS com o comando no terminal:

```
node nomedoarquivo.js
```

- Utilizar o console do navegador, apertando `f12` em seu navegador.

Variáveis

Por ser uma linguagem de programação não tipificada, temos três formas de declaração:

- `var` : Variável global
- `let` : Variável local ao seu escopo
- `const` : Constante local ao seu escopo

```
var a
let b
const c = "Sou uma constante"
```

Ou seja, com essas três formas de declaração é possível ter todos os tipos de variáveis, sendo elas:

- Numéricas
- Booleanas
- Strings
- Objetos
- Funções
- além dos tipos, `null` e `undefined`

Uma maneira de saber o tipo de variável que se esta trabalhando é utilizando o operador `typeof`

```
var função = function(a, b) { return a+b }
let forma = "redondo"
var tamanho = 1
const hoje = new Date()

typeof meuLazer  // retorna "function"
typeof forma     // retorna "string"
typeof tamanho   // retorna "number"
typeof hoje      // retorna "object"
typeof naoExiste // retorna "undefined"
```

Nome	Operador encurtado	Significado
Atribuição	x = y	x = y
Atribuição de adição	x += y	x = x + y
Atribuição de subtração	x -= y	x = x - y

Nome	Operador encurtado	Significado
Atribuição de multiplicação	x *= y	x = x * y
Atribuição de divisão	x /= y	x = x / y
Atribuição de resto	x %= y	x = x % y
Atribuição exponencial	x **= y	x = x ** y

Condicionais

Para a tabela a seguir, considere:

```
var var1 = 3;
var var2 = 4;
```

Comparação	Exemplos que retornam verdadeiro
Igual (==)	3 == var1 "3" == var1 3 == '3'
Não igual (!=)	var1 != 4 var2 != "3"
Estritamente igual (===)	3 === var1
Estritamente não igual (!==)	var1 !== "3" 3 !== '3'
Maior que (>)	var2 > var1 "12" > 2
Maior que ou igual (>=)	var2 >= var1 var1 >= 3
Menor que (<)	var1 < var2 "12" < "2"
Menor que ou igual (<=)	var1 <= var2 var2 <= 5

Seguindo a tabela acima temos a estruturas de condicionais:

```
if(condição) {
  façaAlgo()
} else if{
  façaOutroAlgoSe()
} else {
  FaçaOutroAlgo()
}
```

Existe uma outra forma que usamos para recriar um `if...else`, utilizando o operador **ternário**:

```
const string = (condição) ? "Se for true" : "se for false"
```

Exemplo:

```
let a = 2
let b = 3
let c = (a > b) ? (a + b) : (a - b)
console.log(c) // -1

a = 4
c = (a > b) ? (a + b) : (a - b)
console.log(c) // 7
```

Repetição

Existem diversas formas de realização uma estrutura de repetição no JavaScript, porém, iremos nós focar nas duas princípais: `for` e `while`.

Além disso, é possível utilizar o comando `break` para sair do laço repetição. Existe também o comando `continue`, que força ele a continuar o laço de repetição,

Começando pelo `while`, funciona da mesma forma que a maioria das linguagens:

> **Repita até a condição for satisfeita**

```
while(condição) {
  façaAlgo()
}
```

Existe também 3 formas de `for`, a forma mais simples:

```
for (expressaoInicial; condicao; incremento) {
  façaAlgo()
}
```

Exemplo:

```
for(let i = 0; i < 10; i++) {
  console.log(i) // 0 1 2 3 4 5 6 7 8 9
}
```

Outra forma de utilizar o `for` é através da declaração `for...in` e `for...of`, muito util para percorrer vetores.

```
let elementos = ['Germânio', 'Antimônio', 'Bismuto', 'Oganessônio']

for(let elemento in elementos) {
  console.log(elemento) // 0, 1, 2, 3
}

for(let elemento of elementos) {
  console.log(elemento) // Germânio, Antimônio, Bismuto, Oganessônio
}
```

Essas são as formas de repetição mais gerais que podemos utilizar, mais a frente iremos ver as funções de alto nível que realizamos nos arrays. como a função `map`, `filter` e `reduce`.

Funções

As funções em JavaScript podem ser declaradas de duas formas, a forma convencional, utilizando o `function`, e do modo utilizando o *ES6*, chamadas de *Arrow functions*. Ambas são funções, podendo ou não retornar algo. Além de poderem ser utilizadas como funções anônimas.

```
// Modo convencional
function square(numero) {
  return numero*numero
}

// Modo Convencional da forma anônima
const constSquare = function(numero) {
  return numero * numero
}

// Modo Arrow Function, da forma mais reduzida possível
const arrowSquare = numero => numero * numero

// Modo Arrow Function, da forma expandida
const arrowFullSquare = (numero) => {
  return numero * numero
}

console.log(square(5)) // 25
console.log(constSquare(5)) // 25
console.log(arrowSquare(5)) // 25
console.log(arrowFullSquare(5)) // 25
```

Um detalhe interessante é que podemos passar funções como parâmetros para funções. Pegamos como exemplo a função de nível `map`, que é uma função dentro do objeto array.

```
function map(f,a) {
  var result = []; // Cria um novo Array
  var i;
  for (i = 0; i != a.length; i++)
    result[i] = f(a[i]);
  return result;
}
```

Note que a variável `f` é uma função, que nós que fornecemos para que ele execute em cada elemento do array que nós fornecemos também.

Então, em relação aos parâmetros que passamos em nossas funções, temos alguns *features* que podemos realizar nelas.

- Operador Rest (`...`)
  Com este operador podemos colocar quantos argumentos quisermos na nossa função, permitindo que o usuário coloque o número de argumentos que forem necessários para realizar determinada ação com a nossa função. Este operador é uma *feature* do *ES6*, sendo representado pelos `...`, e a variável que o recebe retorna um array.

  ```
  function func(a, b, ...c = [3, 4, 5]) {
    console.log(a, b, c)
    return a + b + c.length
  }

  func(1,2,3,4,5)
  ```

- Parâmetros Predefinidos
  Esta funcionalidade, vinda do *ES6*, permite que suas funções possam ter parâmetros que inicializem com algum valor padrão, caso esse não seja passado este argumento para função. Não é permitido usar com o operador rest.

  ```
  function func(a = 2, b = 3) {
    console.log(a, b)
    return a + b
  }

  func()
  func(4, 5)
  ```

Objetos

No JavaScript muitas coisas que usamos são objetos, como Strings, Arrays, Dates etc. Logo, eles possuem propriedades, variáveis intrínsecas, e métodos, funções intrínsecas, que podem ser utilizadas por esse objeto.

Como declarar um objeto:

```
const caneta = {
  quebrado: false,
  cor: "azul",
  parametros: {
    comprimento: 15,
    largura: 1
  },
  escrever: (x, y) => console.log('Escrevi em ' + x + ',' + y),
  borar: function() {
    return console.log('borar')
  },
  quebrar: () => {
    this.quebrado = true
  }
}
```

Como utilizar um objeto:

```
caneta.quebrado // false
caneta.parametros // { comprimento: 15, largura: 1 }
caneta.escrever(15, 12) // 'Escrevi em 15, 12'
```

Este modelo de objeto é usado em alguns modelos de API, pois este objeto é convertido em **JSON**, e vice-versa. **JSONs** são de extrema importância, pois é assim que costumamos enviar dados do backend para o frontend.

Arrays e Strings são tratadas como objetos javascript, como dito anteriormente. Logo, existem funções prontas para utilizar com esses dois tipos de dados.

```
const array = []
// Empurra um valor para dentro do array, esse valor
//sempre fica por último no array
array.push(3)
console.log(array) // [3]

let array2 = [3, 4, 5, 4]
// Percorre todo o array realizando a função que foi passada como
//parâmetro e retornando um novo array
let mappingArray = array2.map((element, index) => {
  return element**element
})
console.log(mappingArray) // [27, 256, 3125]

// Percorre todo o array e retorna apenas um array com apenas os
//valores que passaram no método
let filterArray = array2.filter(element => {
  return element === 4
})
console.log(filterArray) // [4, 4]
```

Callback, Promises e Async/Await

> **Para quem veio de uma linguagem síncrona como AdvPL, C, Python e tantas outras, é desafiador mudar a maneira de pensar. JavaScript é uma linguagem executada assincronamente; isso acontece porque quando o interpretador executa um comando que dependa de uma informação externa, como uma requisição, ele não bloqueia o prosseguimento do programa. Essa característica do JavaScript é extremamente utilizada no Node.js.**
> SANTOS, Victor F. dos; Entendendo funções callback em JavaScript. Medium - TOTVS Developers. 2019.

Por JavaScript ser uma linguagem assíncrona, tratamos o nosso código de maneiras diferentes. Por isso, existem as funções *callbacks*, que executam apos determinada ação ocorrer, um exemplo é quando queremos utilizar o modulo `fs` para executar a leitura de uma arquivo.

```
const fs = require('fs')

fs.readFile(path, 'utf-8', callback)
```

Essa função callback retorna a arquivo que lemos, porém, ele "demorará" para ser executado e isso interfere no seguimento do nosso código. Códigos executadas abaixo de função `fs.readFile()` serão

executados mais rápido que a função que chamamos, o código não irá esperar. Por este mótivo utilizamos as funções `callbacks` .

Porém, isso faz com que o nosso código fique em formato "barriga", pq ele cresce horizontalmente.

Para resolver isso, foram criadas as `Promise`, sendo elas objetos JavaScript que lidam de outra forma com os `callbacks`. Não irei abordar a fundo como funciona uma `Promise` e sua estrutura, irei deixar um artigo linkado para isso, O que são promessas em JavaScript?.

O que será passado de importante é como tratar um objeto `Promise` utilizando uma *feature* do *ES6*, chamada `async/await`.

Primeiramente, vamos supor que precisamos fazer uma requisição com `GET` para o site do google. Para realizar essa requisição usaremos o módulo chamado `axios` [1]. Então, sabendo que fazer uma requisição, solicitando dados de um site, é algo "demorado" o JavaScript irá entender isso e, então, teremos que tratar ele como um a `Promise`. Para isso, primeiramente criaremos uma função assíncrona no escopo que realizaremos a requisição, pois só assim é permitido usar o comando `await`.

```
const axios = require('axios')

async function main() {
  const response = await axios.get('https://www.google.com.br/')

  // resto do código
}

main()
```

Import e Export

ES6

A maneira ES6 de se exportar algo de um arquivo JavaScript é utilizando o comando `export`. Supondo um arquivo `modulo.js`.

```
// exporta uma constante, de maneira default
export default const raiz = Math.sqrt(2);

export function calcSoma(a, b) {
  return a + b
}

export function calcVezes(a, b) {
  return a * b
}
```

O comando `default` estabelece a exportação padrão do nosso módulo.

Agora, para importar essas coisas do arquivo exportado utiliza o comando `import`. No exemplo, importaremos do arquivo `modulo.js` em um outro arquivo `modulo2.js`, no mesmo diretório.

```
import { calcSoma, calcVezes } from './modulo.js'
// Com isso importaríamos apenas as duas funções exportadas

import quadrado from './modulo.js'
// Importa só o default, sem as duas funções criadas
```

CommonJS

A segunda maneira de utilizar exportações e importações de códigos em arquivos JavaScript, é através do `module.exports` e `require`. Supondo um arquivo `modulo3.js`.

```
const array = ["Zézinho", "Zézão", "Zé"]

function calcDiv(a, b) {
  return a / b
}

function calcMenos(a, b) {
  return a - b
}

// Utilizando desestruturação para exportar como um objeto
module.exports = {
  array,
  calcDiv,
  calcMenos
}
```

Então, para importar em um arquivo `modulo4.js` no mesmo diretório.

```
const modulo4 = require('./modulo4.js')
// { array, calcDiv, calcMenos }
```

NodeJS

Introdução

O NodeJS é um ambiente de execução JavaScript *server-side*, fornecendo a possibilidade de rodar códigos JavaScript sem um navegador.

NPM e YARN

Dentro desse ambiente existem o gerenciadores de pacotes, *packager manager*, como o `npm` e o `yarn`. Esses fornecem bibliotecas para os desenvolvedores, criados pelos próprios usuários.

Atualmente o npm é um dos maiores repositórios do mundo. O yarn executa a mesma coisa que o npm, porém, ele é mais rápido, mais seguro e mais organizado, na minha opinião.

Para instalar uma biblioteca em seu NodeJS globalmente, pode-se utilizar os seguintes comandos:

```
npm install -g <package_name>
```

ou

```
yarn global add <package_name>
```

Caso você queria instalar alguma biblioteca apenas em um projeto específico, pode-se utilizar os seguintes comandos:

```
npm install <package_name>
```

ou

```
yarn add <package_name>
```

Recomenda-se o uso de apenas um packager manager, não misture os dois.

Módulos

Como citados na sessão anterior, o ambiente do JavaScript contem diversas bibliotecas que podemos importar como módulos em nossos códigos. Para isso, tendo algum projeto que você queria utilizar determinado módulo, o primeiro passo é instalar esse módulo com o npm ou o yarn, de modo geral ou local, e então importar esse módulo no seu arquivo .js.

```
const nome_do_modulo = require('nome_do_modulo')
// ou, pelo modo ES6
import modulo from 'nome_do_modulo'
```

---

1. Isto tem relação com módulos em javascript, que iremos ver logo a frente. Porém, isto se trata de um módulo em NodeJS para realizar requisições, axios. ↩