



Digital  
College

# FORMAÇÃO EM **DESENVOLVEDOR FULL STACK**

**PROGRAMAÇÃO FRONT-END EM JAVASCRIPT**

UNIDADE 2:

MÓDULO 2:

> **JAVASCRIPT** >

## Tipos Primitivos

- <https://developer.mozilla.org/en-US/docs/Glossary/Primitive>

JavaScript sabe manipular basicamente dois tipos de dados, definidos genericamente como tipos primitivos e objetos [veremos o que é um objeto, seus métodos (elementos que definem o comportamento de um objeto) e suas propriedades (elementos que definem o estado de um objeto) nos tópicos de orientação a objetos].

Tipos primitivos são elementos imutáveis (depois de criados não podem ser modificados) e a linguagem JavaScript possui sete. São eles:

- string
- number
- bigint
- boolean
- symbol
- undefined
- null

Iniciaremos nossos estudos pelo tipo primitivo string.

## O tipo Primitivo String

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)

A maioria das linguagens de programação tem a capacidade de declarar e manipular valores fixos dentro de seus códigos. Valores dessa natureza são chamados de LITERAIS.

EX:

<b>8080</b>	<b>VALOR NUMÉRICO INTEIRO</b>
<b>3.1415926</b>	<b>VALOR NUMÉRICO EM PONTO FLUTUANTE</b>
<b>TRUE</b>	<b>VALOR LÓGICO</b>
<b>FULL STACK NA DIGITAL COLLEGE</b>	<b>CADEIA DE CARACTERES</b>

```

//  

// VALOR NUMÉRICO INTEIRO  

//  

var meuPrimeiroProcessador = 8080;  

console.log(meuPrimeiroProcessador, ' -> ', typeof  

meuPrimeiroProcessador);  

//  

// SAÍDA  

//  

// 8080 -> number  

//  
  

//  

// VALOR NUMÉRICO EM PONTO FLUTUANTE  

//  

const PI = 3.1415926;  

console.log(PI, ' -> ', typeof PI);
//
```

```
// SAÍDA
//
// 3.1415926 -> number
//

//
// VALOR LÓGICO
//
const VERDADE = true;
console.log(VERDADE, ' -> ', typeof VERDADE);
//
// SAÍDA
//
// true -> boolean
//


//
// CADEIA DE CARACTERES
//
let melhorCurso = "Full Stack na Digital College";
console.log(melhorCurso, ' -> ', typeof melhorCurso);
//
// SAÍDA
//
// Full Stack na Digital College -> string
//
```

Para JavaScript um LITERAL formado por uma CADEIA DE CARACTERES também é chamada de string. Há várias formas de criar uma string em JavaScript e veremos as principais agora:

## Criando string usando ASPA DUPLA ("") - (DOUBLE QUOTE)

O símbolo de pontuação chamado de ASPA DUPLA é um dos símbolos usados para iniciar e finalizar uma string.

Ex:

```
"Full Stack na Digital College"
```

Ao fazer uso desse símbolo como demarcador de uma string, o mesmo não pode aparecer internamente nesta.

Ex:

```
//  
// SINTAX ERROR  
//  
  
let aspaDupla = "Full Stack na Digital College - o "MELHOR" curso";
```

As ASPAS DUPLAS que estão envolvendo a palavra MELHOR no exemplo acima não são permitidas. Essa construção gera um erro chamado de ERRO DE SINTAXE / SYNTAX ERROR.

## Usando ESCAPE CHARACTER em Strings

Caso seja necessária a inclusão de alguma ASPA DUPLA dentro de uma string delimitada por ASPAS DUPLAS ou a inclusão de alguma ASPA SIMPLES dentro de uma string delimitada por ASPAS SIMPLES, um dos procedimentos a ser usado é o que se chama ESCAPE CHARACTER.

O ESCAPE CHARACTER basicamente é um caractere que ao ser encontrado dentro de uma string não é interpretado como fazendo parte da mesma. Ele é interpretado como um elemento que está lá para indicar que o caractere seguinte é o que realmente faz parte da string.

Para o JavaScript o caractere correspondente ao ESCAPE CHARACTER é a barra invertida (\) ou BACKSLASH.

Vejamos alguns exemplos e como interpretá-los:

Usando ASPA DUPLA dentro de string

Ex:

```
//  
// ASPA DUPLA EM STRING DELIMITADA POR ASPA DUPLA USANDO ESCAPE CHARACTER  
//  
  
let aspaDupla = "Full Stack na Digital College - O \"MELHOR\" curso";  
console.log(aspaDupla);  
  
//  
// SAÍDA  
//  
//      Full Stack na Digital College - O "MELHOR" curso
```

Usando ASPA SIMPLES dentro de string

Ex:

```
//  
// ASPA SIMPLES EM STRING DELIMIT. POR ASPA SIMPLES USANDO ESCAPE  
CHARACTER  
//  
  
let aspaSimples = 'Full Stack na Digital College - O \'MELHOR\' curso';  
console.log(aspaSimples);  
  
//  
// SAÍDA  
//  
// Full Stack na Digital College - O 'MELHOR' curso  
//
```

## Mesclando ASPA DUPLA e ASPA SIMPLES sem usar ESCAPE CHARACTER

Há outras formas de fazer com que ASPA DUPLA ou ASPA SIMPLES façam parte de uma string sem fazer uso do ESCAPE CHARACTER.

Uma das outras formas de incluir ASPA DUPLA dentro de uma string é delimitá-la com ASPA SIMPLES.

Vejamos:

```
//  
// ASPA DUPLA DENTRO DE UMA STRING DELIMITADA POR ASPA SIMPLES  
//  
  
let aspaSimplesComDupla = 'Full Stack na Digital College - O "MELHOR"  
curso';  
console.log(aspaSimplesComDupla);  
  
//  
// SAÍDA  
//  
// Full Stack na Digital College - O "MELHOR" curso  
//
```

Para mais uma maneira de mesclar ASPA SIMPLES apareça dentro de uma string, basta usar ASPA DUPLA como o delimitador de tal string.

Há também a possibilidade de se querer incluir ASPA DUPLA dentro de uma string. Para tanto, outra maneira de fazê-lo é delimitar a string com ASPA SIMPLES.

Vejamos:

```
//  
// ASPA SIMPLES DENTRO DE UMA STRING DELIMITADA POR ASPA DUPLA  
//  
  
let aspaDuplaComSimples = "Full Stack na Digital College - O 'MELHOR'  
curso";  
console.log(aspaDuplaComSimples);  
  
//  
// SAÍDA  
//  
// Full Stack na Digital College - O 'MELHOR' curso  
//
```

## Caracteres Especiais

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/String#escape\\_sequences](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String#escape_sequences)

Há outros caracteres que precisam do ESCAPE CHARACTER para que o JavaScript possa reconhecê-los corretamente dentro de uma string.

A seguir veremos uma relação dos principais caracteres que precisam do ESCAPE CHARACTER:

<b>Backspace</b>	\b
<b>Form Feed</b>	\f
<b>New Line</b>	\n
<b>Carriage Return</b>	\r
<b>Horizontal Tabulator</b>	\t
<b>Vertical Tabulator</b>	\v
<b>Backslash</b>	\\\

EX:

```
//  

// OUTROS CARACTERES QUE PRECISAM DO ESCAPE CHARACTER  

// PARA SE FAZEREM PRESENTES EM UMA STRING  

//  

console.log("Backspace \b");  

console.log("Form Feed \f");  

console.log("New Line \n");  

console.log("Carriage Return \r");  

console.log("Horizontal Tabulator \t");  

console.log("Vertical Tabulator \v");
```

## concatenação de strings com o operador '+'

É muito comum que se precise unir (concatenar) strings distintas para formar uma outra única. A forma mais simples de fazê-lo é pelo uso do operador '+'. Entenda-se que quando usado com strings, o operador '+' não tem a finalidade da operação matemática de soma, mas sim a finalidade de LIGAR / CONCATENAR duas ou mais strings.

Vejamos:

```
//  
// CONCATENAÇÃO DE STRINGS (+)  
//  
  
let primeiroNome = "Otávio";  
let ultimoNome = "Medeiros";  
let nomeCompleto = primeiroNome + " " + ultimoNome;  
console.log(nomeCompleto);  
  
//  
// SAÍDA  
//  
//      Otávio Medeiros  
//
```

Sabendo-se que o operador '+' aplicado a strings funciona como um concatenador destas, o que esperar ao utilizar o operador '+' entre uma string e um outro elemento que não seja necessariamente uma string?

Vejamos:

```
//  
// CONCATENAÇÃO DE STRING COM OUTROS ELEMENTOS  
//
```

Sabendo-se que o operador '+' aplicado a strings funciona como um concatenador destas, o que esperar ao utilizar o operador '+' entre uma string e um outro elemento que não seja necessariamente uma string?

Vejamos:

```
//  
// CONCATENAÇÃO DE STRING COM OUTROS ELEMENTOS  
  
//  
  
console.log('casa ' + 1234);  
console.log('casa ' + String(1234));  
console.log('casa ' + new String(1234));  
console.log('casa ' + undefined);  
console.log('casa ' + null);  
  
//  
// SAÍDA  
  
// casa 1234  
// casa 1234  
// casa 1234  
// casa undefined  
// casa null  
//
```

Uma vez que um dos componentes da expressão que utiliza o operador '+' seja uma string, os demais componentes passarão automaticamente pelo processo de coerção (uma forma de conversão sem a definição explícita para qual tipo se deseja converter) e serão convertidos para string.

## Template String e Backsticks

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template\\_literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals)

Acontece que em algumas situações precisamos utilizar tanto a ASPA DUPLA quanto a ASPA SIMPLES para compor uma determinada string. Claro que se pode lançar mão do ESCAPE CODE para atingir esse objetivo, mas há soluções mais elegantes.

Imagine que se deseja armazenar o texto abaixo em uma variável chamada

```
'meuPrimeiroComputador':
```

```
O primeiro 'computador' pessoal que tive em mãos foi um equipamento  
produzido na Inglaterra por uma empresa chamada "Sinclair Research" e  
possuía apenas 2 Kb de memória RAM. Chamava-se 'Sinclair ZX80' e seu  
processador era um 'clone' do "Zilog Z80" de 8 bits o qual possuía um  
'clock' de 3,25 MHz
```

Perceba-se que nela há uma mistura de ASPAS DUPLAS com ASPAS SIMPLES. Para tornar a solução mais desafiadora, a string é formada por um conjunto de linhas.

**Vejamos uma possível solução usando ESCAPE CODE:**

```
//  
// STRING COMPLEXA  
//  
  
let meuPrimeiroComputador = "O primeiro 'computador' pessoal que tive em  
mãos foi um equipamento\nproduzido na Inglaterra por uma empresa chamada  
\\"Sinclair Research\\" e\npossuía apenas 2 Kb de memória RAM. Chamava-se  
'Sinclair ZX80' e seu\nprocessador era um 'clone' do \"Zilog Z80\" de 8  
bits o qual possuía um\n'clock' de 3,25 MHz";  
console.log(meuPrimeiroComputador);  
  
//  
// SAÍDA  
//  
// O primeiro 'computador' pessoal que tive em mãos foi um equipamento  
// produzido na Inglaterra por uma empresa chamada "Sinclair Research"  
e  
// possuía apenas 2 Kb de memória RAM. Chamava-se 'Sinclair ZX80' e seu  
// processador era um 'clone' do "Zilog Z80" de 8 bits o qual possuía  
um  
// 'clock' de 3,25 MHz
```

Muito trabalhoso. Certamente há outra forma de fazê-lo. E realmente há. Uma melhor alternativa seria utilizar TEMPLATE STRINGS, que nada mais é do que usar no lugar de ASPA DUPLA ou ASPA SIMPLES como os delimitadores de uma string, fazer uso da CRASE (`), chamada em inglês BACKTICKS.

Usa-se BACKTICKS da mesma forma que se usa ASPA DUPLA ou ASPA SIMPLES, ou seja, iniciando e terminando uma cadeia de caracteres (string). Nesse caso não há a necessidade de ESCAPE CHARACTER para usar ASPA DUPLA ou ASPA SIMPLES em uma string, quando necessário. Também não será mais necessário o uso do '\n' (New Line) quando for necessário escrever na linha seguinte.

Vejamos como ficaria a atribuição do texto dado como exemplo:

```
//  
// TEMPLATE STRING com BACKTICKS  
  
let meuPrimeiroComputadorComBackTicks = `  
O primeiro 'computador' pessoal que tive em mãos foi um equipamento  
produzido na Inglaterra por uma empresa chamada "Sinclair Research" e  
possuía apenas 2 Kb de memória RAM. Chamava-se 'Sinclair ZX80' e seu  
processador era um 'clone' do "Zilog Z80" de 8 bits o qual possuía um  
'clock' de 3,25 MHz`;  
console.log(meuPrimeiroComputadorComBackTicks);  
  
//  
// SAÍDA  
  
// O primeiro 'computador' pessoal que tive em mãos foi um equipamento  
// produzido na Inglaterra por uma empresa chamada "Sinclair Research" e  
// possuía apenas 2 Kb de memória RAM. Chamava-se 'Sinclair ZX80' e seu  
// processador era um 'clone' do "Zilog Z80" de 8 bits o qual possuía um  
// 'clock' de 3,25 MHz  
//
```

Muito mais elegante e manutenível, não?

Embora o TEMPLATE STRING ofereça uma alternativa bem interessante para o uso de strings, ele também passa pelo problema do uso interno de BACKTICKS, que não são permitidos diretamente. Para usar BACKTICKS em um TEMPLATE STRING é necessário lançar mão do ESCAPE CHARACTER.

Ex:

```
//  
// USO DO BACKTICKS DENTRO DE UM TEMPLATE STRING  
//  
  
let templateStringComBackticks = `Usando BACKTICKS (\`) em TEMPLATE  
STRING`;  
console.log(templateStringComBackticks);  
  
//  
// SAÍDA  
//  
// Usando BACKTICKS (`) em TEMPLATE STRING  
//
```

O TEMPLATE STRING apresenta usos mais avançados, como interpolação de variáveis e substituição de expressões. Vejamos alguns exemplos disso, mas para maiores informações, acessar o link abaixo:

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template\\_literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals)

## Interpolação de variáveis

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template\\_literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals)

Vejam a construção a seguir:

```
//  
// INTERPOLAÇÃO DE VARIÁVEIS COM TEMPLATE STRINGS  
  
let primeiroNome = 'José';  
let ultimoNome = 'Castanhedo';  
let nacionalidade = 'brasileira';  
let contrato = `Vimos por meio deste declarar que ${primeiroNome}  
${ultimoNome}  
de nacionalidade ${nacionalidade} ...`;  
console.log(contrato);  
  
//  
// SAÍDA  
//  
//      Vimos por meio deste declarar que José Castanhedo  
//      de nacionalidade brasileira ...  
//
```

Percebe-se que foi possível injetar variáveis (primeiroNome, ultimoNome e nacionalidade) em um TEMPLATE STRING, tornando a construção de strings muito mais natural. O mesmo resultado (saída) poderia ser obtido de várias outras formas, inclusive a partir de concatenação de strings por meio do operador '+', o qual já foi visto, mas a construção seria muito mais complicada e menos intuitiva.

Vejamos:

```
//  
// INTERPOLAÇÃO DE VARIÁVEIS SEM TEMPLATE STRINGS  
// (USANDO O OPERADOR '+')  
  
//  
  
let primeiroNome = 'José';  
let ultimoNome = 'Castanhedo';  
let nacionalidade = 'brasileira';  
let contrato = 'Vimos por meio deste declarar que ' + primeiroNome  
+ ' ' + ultimoNome + '\nde nacionalidade ' + nacionalidade + '  
...';  
console.log(contrato);  
  
//  
// SAÍDA  
//  
// Vimos por meio deste declarar que José Castanhedo  
// de nacionalidade brasileira ...  
//
```

Fica claro o maior nível de esforço e menor nível de clareza para se montar uma string dessa forma. Até mesmo a quebra de linha (\n) teve que ser tratada usando-se o artifício do ESCAPE CHARACTER (\nde nacionalidade).

## Substituição de Expressões

Outra facilidade ao se manipular strings por meio de TEMPLATE STRINGS é a possibilidade de se fazer uso de expressões matemáticas dentro da própria estrutura deste.

Vejamos:

```
//  
// SUBSTITUIÇÃO DE EXPRESSÕES EM TEMPLATE STRINGS  
//  
  
let saldo = 20000.15;  
let deposito = 4300;  
let saldoFinal = `Seu saldo era de ${saldo} e passou a ser de ${saldo +  
deposito}, após o depósito de ${deposito}`;  
console.log(saldoFinal);  
  
//  
// SAÍDA  
//  
// Seu saldo era de 20000.15 e passou a ser de 24300.15, após o depósito  
de 4300  
//
```

O uso de TEMPLATE STRING permitiu, de uma forma mais fluida, até mesmo injetar expressões (`(${saldo + deposito})`).

## Imutabilidade do tipo Primitivo string

Foi dito no início desse material que tipos primitivos são imutáveis (depois de criados não podem ser modificados). Vejamos isso na prática.

Uma vez que uma string é uma sequência de caracteres, é possível acessar cada caractere individual de uma string por meio de um índice que indique a posição desse caractere dentro da string. O índice do primeiro caractere dentro de uma string possui o valor 0 (zero).

D	i	g	i	t	a	I		C	o	I	I	e	g	e	length = 15
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	

Conhecendo esses fatos, podemos acessar o caractere 'C' da string acima e tentar alterá-lo para o caractere 'K'. Essa operação não terá sucesso, confirmando que o tipo primitivo string (por ser um tipo primitivo) é imutável. Os demais tipos primitivos apresentam a mesma propriedade de imutabilidade.

Vejamos:

```
//  
// IMUTABILIDADE DO TIPO PRIMITIVO STRING  
  
let strImutavel = "Digital College";  
console.log(strImutavel[8]); // c  
strImutavel[8] = 'K';  
console.log(strImutavel[8]); // c
```

## Acessando Propriedades e Métodos do objeto String em um Tipo Primitivo string

Iniciamos nosso material esclarecendo que string é um tipo primitivo. Entre outras consequências, isso significa dizer que string, enquanto tipo primitivo, não possui nem propriedades e nem métodos, características essas de um objeto. Entretanto é importante lembrar que JavaScript é uma linguagem fracamente tipada. Isso significa dizer que quando declaramos uma variável, uma constante ou mesmo um objeto, não precisamos informar qual o tipo de elemento estamos criando. Existem línguas que são fortemente tipadas e nessas, ao declararmos uma variável, uma constante ou mesmo um objeto, precisamos informar o tipo de elemento que estamos criando.

Vejamos:

```
//  
// LINGUAGEM FRACAMENTE TIPADA  
//  
  
let stringoNaoTipada = 'isso é uma string';  
  
//  
// SINTAX ERROR  
//  
// (A DECLARAÇÃO DO TIPO É USADA EM LINGUAGENS FORTEMENTE TIPADAS - JAVA)  
//  
// String stringoTipada = "isso é uma string";  
//
```

Uma vez que a linguagem JavaScript é fracamente tipada, a atribuição de uma string a uma variável ou a uma constante [de forma direta (string literal) ou de forma indireta (a partir de variáveis, constantes, funções ou objetos)] é que a torna uma string, sem que seja necessário dizer explicitamente para o interpretador que se trata de tal.

Dessa forma o trecho de código abaixo pode parecer estranho, mas está estritamente correto:

```
//  
// USO DE PROPRIEDADES E MÉTODOS EM TIPOS PRIMITIVOS STRING  
  
let tipoPrimitivoString = 'título do capítulo';  
console.log('Conteúdo da string : ', tipoPrimitivoString);  
console.log('Tamanho da string : ', tipoPrimitivoString.length);  
console.log('String em letras maiúsculas: ',  
tipoPrimitivoString.toUpperCase());  
  
//  
// SAÍDA  
//  
// Conteúdo da string : título do capítulo  
// Tamanho da string : 18  
// String em letras maiúsculas : TÍTULO DO CAPÍTULO
```

Percebe-se que, embora a variável de nome tipoPrimitivoString seja uma string (tipo primitivo), foi possível verificar a propriedade length e chamar o método toUpperCase(), os quais pertencem ao objeto global String.

Mesmo ainda não tendo falado sobre orientação a objeto em JavaScript, a ideia dessas informações é compreender um pouco mais a coerção na linguagem JavaScript

Todo tipo primitivo (com exceção dos tipos primitivos 'undefined' e 'null') possui um objeto 'wrapper' (objetos 'wrappers' serão vistos nos tópicos de orientação a objeto) correspondente. Na prática isso significa dizer que o tipo primitivo string possui um objeto wrapper chamado String.

Efetivamente a criação de um tipo primitivo string pode-se dar de várias formas distintas. As mais comuns são a partir de uma cadeia de caracteres (literal) ou a partir do uso do objeto String, como visto abaixo. Também é possível fazer a conversão de outros tipos para o tipo string, sendo a forma mais comum de fazê-lo por meio do método .toString().

Percebe-se que `String("cadeia de caracteres")` gera um tipo primitivo string, ao passo que `new String("cadeia de caracteres")` gera um objeto String.

Para, a partir de um objeto wrapper, obter-se seu tipo primitivo correspondente, basta usar o método `.valueOf()`.

```
//  
// CRIANDO STRINGS A PARTIR DE LITERAIS E DO OBJETO STRING  
  
  
let str1 = "cadeia de caracteres";  
let str2 = String("cadeia de caracteres");  
let str3 = new String("cadeia de caracteres");  
let str4 = str3.valueOf();  
console.log(`"${str1}" -> ${typeof str1}`); // cadeia de caracteres -> string  
console.log(`"${str2}" -> ${typeof str2}`); // cadeia de caracteres -> string  
console.log(`"${str3}" -> ${typeof str3}`); // cadeia de caracteres -> object  
console.log(`"${str4}" -> ${typeof str4}`); // cadeia de caracteres -> string  
  
  
let str5 = (3.1415926).toString();  
let str6 = String(3.1415926);  
let str7 = new String(3.1415926);  
let str8 = str7.valueOf();  
console.log(`"${str5}" -> ${typeof str5}`); // 3.1415926 -> string  
console.log(`"${str6}" -> ${typeof str6}`); // 3.1415926 -> string  
console.log(`"${str7}" -> ${typeof str7}`); // 3.1415926 -> object  
console.log(`"${str8}" -> ${typeof str8}`); // 3.1415926 -> string
```

Da mesma forma que o JavaScript executou uma coerção quando se utilizou o operador '+' para concatenar strings (ver o tópico concatenação de string com o operador '+'), o mesmo ocorre nos casos demonstrados acima. A coerção dá-se quando o interpretador encontra uma tentativa de acessar uma propriedade ou um método a partir de um tipo primitivo. Nesse caso, para um tipo primitivo do tipo string, o que ocorre é a coerção desse tipo primitivo para seu objeto wrapper correspondente (`String`). É isso que torna possível acessar propriedade e métodos a partir de tipos primitivos (sua coerção para objetos wrappers correspondentes).

Cientes agora que podemos acessar propriedades e métodos, elementos de objetos, mesmo em tipos primitivos, vejamos as principais propriedades e métodos associados ao objeto `String`, tomando como base a variável a seguir.

## Propriedade

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/String#instance\\_properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String#instance_properties)

Propriedade	Descrição
length	<b>Contém o comprimento de uma string (número de caracteres que a mesma possui).</b>

```
console.log(strBase.length); // 15
```

## Métodos

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)

Método	Descrição
charAT	Retorna o caractere em uma posição específica na string a partir de uma posição dada por um índice. Caso o índice esteja fora dos limites da string, retorna "" (string vazia).

```
console.log(`>${strBase.charAt(-1)}<`); // >< [string vazia (tamanho 0)]
console.log(`>${strBase.charAt(0)}<`); // >D<
console.log(`>${strBase.charAt(1)}<`); // >i<
console.log(`>${strBase.charAt(7)}<`); // > < [espaço em branco (tamanho 1)]
console.log(`>${strBase.charAt(strBase.length - 1)}<`); // >e<
console.log(`>${strBase.charAt(strBase.length)}<`); // >< [string vazia (tamanho 0)]
```

**indexOf, lastIndexOf**

Retorna a posição de uma substring específica na string ou a última posição da substring específica, respectivamente.

```
console.log(strBase.indexOf('x')); // -1 (não encontrado)
console.log(strBase.indexOf('g')); // 2
console.log(strBase.indexOf('g', 3)); // 13

console.log(strBase.lastIndexOf('x')); // -1 (não
encontrado)
console.log(strBase.lastIndexOf('g')); // 13
console.log(strBase.lastIndexOf('g', 12)); // 2
```

**startsWith, endsWith, includes**

Retorna se uma string começa, termina ou contém uma outra string específica.

```
console.log(strBase.startsWith('x')); // false
console.log(strBase.startsWith('Dig')); // true
console.log(strBase.startsWith('Col')); // false
console.log(strBase.startsWith('Col', 8)); // true

console.log(strBase.endsWith('x')); // false
console.log(strBase.endsWith('ege')); // true
console.log(strBase.endsWith('Dig')); // false
console.log(strBase.endsWith('Dig', 3)); // true

console.log(strBase.includes('x')); // false
console.log(strBase.includes('Coll')); // true
console.log(strBase.includes('Coll', 8)); // true [a partir de 8
(inclusive)]
console.log(strBase.includes('Coll', 9)); // false [a partir de 9
(inclusive)]
```

**concat**

concatena o texto de duas strings e retorna uma nova string.

```
console.log(strBase.concat(' tem o melhor curso de FULL STACK'));
    // Digital College tem o melhor curso de FULL STACK
console.log(strBase.concat(' tem', ' o melhor' + ' curso de' + ' FULL
STACK'));
    // Digital College tem o melhor curso de FULL STACK
```

**split**

Separa um objeto String em um array de strings, separando a string em substrings.

```
let split01 = strBase.split();
console.log(`valor: ${split01}
tipo: ${typeof split01}
nº elementos: ${split01.length}`);
// SAÍDA
// valor: Digital College
// tipo: object
// nº elementos: 1
let split02 = strBase.split(' ');
console.log(`valor: ${split02}
tipo: ${typeof split02}
nº elementos: ${split02.length}`);
// SAÍDA
//
// valor: Digital, College
// tipo: object
// nº elementos: 2
//
```

**slice**

Extrai uma seção de uma string e retorna uma nova string.

```
//  
// slice  
  
//  
// Usando apenas 'beginIndex'  
//  
console.log(`>${strBase.slice()}<`); // >Digital College<  
console.log(`>${strBase.slice(0)}<`); // >Digital College<  
console.log(`>${strBase.slice(3)}<`); // >ital College<  
console.log(`>${strBase.slice(strBase.length - 1)}<`); // >e<  
console.log(`>${strBase.slice(strBase.length)}<`); // >< [string vazia  
(tam. 0)]  
console.log(`>${strBase.slice(-5)}<`); // >llege<  
  
//  
// Usando 'beginIndex' e 'endIndex'  
//  
console.log(`>${strBase.slice(0, 5)}<`); // >Digit<  
console.log(`>${strBase.slice(2, 5)}<`); // >git<  
console.log(`>${strBase.slice(2, strBase.length - 1)}<`); // >gital  
College<  
console.log(`>${strBase.slice(2, strBase.length)}<`); // >gital  
College<  
console.log(`>${strBase.slice(2, strBase.length + 10)}<`); // >gital  
College<  
console.log(`>${strBase.slice(2, -8)}<`); // >gital<  
console.log(`>${strBase.slice(2, -12)}<`); // >g<  
console.log(`>${strBase.slice(2, -13)}<`); // >< [string vazia (tamanho  
0)]
```

**substring**

Retorna um subconjunto específico de uma string, definindo os índices inicial e final, ou definindo um índice e um tamanho.

```
//  
// substring  
//  
  
//  
// Usando apenas 'beginIndex'  
//  
console.log(`>${strBase.substring()}<`); // >Digital College<  
console.log(`>${strBase.substring(0)}<`); // >Digital College<  
console.log(`>${strBase.substring(3)}<`); // >ital College<  
console.log(`>${strBase.substring(strBase.length - 1)}<`); // >e<  
console.log(`>${strBase.substring(strBase.length)}<`); // >< [vazia  
(tam. 0)]  
console.log(`>${strBase.substring(-5)}<`); // >llege<  
console.log("\n");  
  
//  
// Usando 'beginIndex' e 'endIndex'  
//  
console.log(`>${strBase.substring(0, 5)}<`); // >Digit<  
console.log(`>${strBase.substring(2, 5)}<`); // >git<  
console.log(`>${strBase.substring(2, 2)}<`); // >< [string vazia  
(tamanho 0)]  
console.log(`>${strBase.substring(2, strBase.length - 1)}<`); // >gital  
Colleg<  
console.log(`>${strBase.substring(2, strBase.length)}<`); // >gital  
College<  
console.log(`>${strBase.substring(2, strBase.length + 10)}<`); //  
>gital College<  
console.log(`>${strBase.substring(7, 3)}<`); // >ital<  
console.log(`>${strBase.substring(2, -8)}<`); // >Di<  
console.log(`>${strBase.substring(-8, 2)}<`); // >Di<
```

**match, replace, search****Trabalha com expressões regulares.**

```
//  
// match  
//  
let regEx01 = /[A-Z]/;  
let match01 = strBase.match(regEx01);  
console.log(`>${match01}<`); // >D<  
console.log(`>${typeof match01}<`); // >object<  
console.log();  
let regEx02 = /[A-Z]/g;  
let match02 = strBase.match(regEx02);  
console.log(`>${match02}<`); // >D,C<  
console.log(`>${typeof match02}<`); // >object<  
console.log();  
// replace  
//  
  
let regEx03 = /college/;  
let repl01 = strBase.replace(regEx03, "Age");  
console.log(`>${repl01}<`); // >Digital College<  
console.log();  
  
let regEx04 = /College/;  
console.log(`>${strBase.replace(regEx04, "Age")}<`); // >Digital Age<  
console.log();  
  
let regEx05 = /college/i;  
let repl02 = strBase.replace(regEx05, "Age");  
console.log(`>${repl02}<`); // >Digital Age<  
console.log();  
// search  
  
let regEx06 = /college/i;  
console.log(strBase.search(regEx06)); // 8  
console.log();
```

**toLowerCase, toUpperCase**

Retorna a string com todos os caracteres em minúsculo, ou maiúsculo, respectivamente.

```
//  
// toLowerCase / toUpperCase  
  
console.log(`>${strBase.toLowerCase()}<`); // >digital college<  
console.log(`>${strBase.toUpperCase()}<`); // >DIGITAL COLLEGE<
```

**repeat**

Retorna uma string contendo os elementos do objeto repetidos pela quantidade de vezes dada.

```
//  
// repeat  
  
// console.log(`>${strBase.repeat(-1)}<`); // RangeError: Invalid count  
// value  
console.log(`>${strBase.repeat()}<`); // >< [string vazia (tamanho 0)]  
console.log(`>${strBase.repeat(0)}<`); // >< [string vazia (tamanho 0)]  
console.log(`>${strBase.repeat(2)}<`); // >Digital CollegeDigital College<
```

trim

Retira espaços em branco no começo e no final da string.

```
//  
// trim  
//  
let strLeftTrim = ` ${strBase}`;  
let strRightTrim = `${strBase} `;  
let strBothTrim = ` ${strBase} `;  
  
console.log(`>${strLeftTrim}<`); // > Digital College<  
console.log(`>${strLeftTrim.trim()}<`); // >Digital College<  
console.log();  
  
console.log(`>${strRightTrim}<`); // >Digital College <  
console.log(`>${strRightTrim.trim()}<`); // >Digital College<  
console.log();  
  
console.log(`>${strBothTrim}<`); // > Digital College <  
console.log(`>${strBothTrim.trim()}<`); // >Digital College<  
console.log();
```

## O Tipo Primitivo Number

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String)

Da mesma forma que string, number também é um tipo primitivo e ainda semelhantemente a string (na verdade a semelhança dos fatos descritos a seguir ocorre para todos os tipos primitivos – exceto para 'null' e 'undefined'), o tipo primitivo number também possui um objeto wrapper chamado Number.

As colocações sobre coerção e wrappers levantadas para strings no tópico Acessando Propriedades e Métodos do objeto String em um Tipo Primitivo string também podem ser trazidas para o contexto do objeto Number e do tipo primitivo number.

Há uma exceção. O tipo primitivo bigint, embora possua um objeto wrapper BigInt, não pode ser instanciado por meio do 'sintatic sugar' new (maiores esclarecimentos nos tópicos de orientação a objeto) ou seja, embora seja válido escrever **let big2 = BigInt(1234);** não é possível escrever **let big3 = new BigInt('1234');**. Nesse último caso ocorrerá um erro (**TypeError: BigInt is not a constructor**).

Vejamos:

```
//  
// BIGINT - EXCEÇÃO AO USO DO NEW  
//  
  
let big1 = 1234n;  
let big2 = BigInt(1234);  
//  
// TypeError: BigInt is not a  
constructor  
//  
// let big3 = new BigInt('1234');  
//  
console.log(`${big1} -> ${typeof  
big1}`); // 1234 -> bigint  
console.log(`${big2} -> ${typeof  
big2}`); // 1234 -> bigint  
  
//  
// CRIANDO NUMBERS A PARTIR DE  
LITERAIS E DO OBJETO NUMBER  
//  
  
let num1 = 3.1415926;  
let num2 = Number(3.1415926);  
let num3 = new Number(3.1415926);  
let num4 = num3.valueOf();  
console.log(`${num1} -> ${typeof  
num1}`); // 3.1415926 -> number  
console.log(`${num2} -> ${typeof  
num2}`); // 3.1415926 -> number  
console.log(`${num3} -> ${typeof  
num3}`); // 3.1415926 -> object  
console.log(`${num4} -> ${typeof  
num4}`); // 3.1415926 -> number
```

- [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Lexical\\_grammar#n%C3%BAmeros](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Lexical_grammar#n%C3%BAmeros)

O tipo primitivo number pode armazenar até 17 casas decimais de precisão e possui alguns limites máximos e mínimos de valores que ele pode armazenar. Esses valores podem ser obtidos por meio de propriedades estáticas (maiores esclarecimentos nos tópicos de orientação a objeto) do objeto Number.

Vejamos:

```
//  
// NUMBER LIMITS  
  
console.log(Number.MIN_VALUE); //  
5e-324  
console.log(Number.MAX_VALUE); //  
1.7976931348623157e+308  
console.log(Number.MIN_SAFE_INTEGER); // -9007199254740991  
console.log(Number.MAX_SAFE_INTEGER); // 9007199254740991  
console.log(Number.NEGATIVE_INFINITY); // -Infinity  
console.log(Number.POSITIVE_INFINITY); // Infinity
```

Há várias formas possíveis de se representar números.

Vejamos:

```
//  
// NUMBERS - FORMAS DE  
REPRESENTAÇÃO  
  
let decimal = 1234567;  
let binario1 = 0b1001101;  
let binario2 = 0B1001110;  
let octal1 = 0o74217;  
let octal2 = 0074220;  
let hexa1 = 0x0fff;  
let hexa2 = 0X1000;  
console.log(`${decimal} ->  
${typeof decimal}`); // 1234567 ->  
number  
console.log(`${binario1} ->  
${typeof binario1}`); // 77 ->  
number  
console.log(`${binario2} ->  
${typeof binario2}`); // 78 ->  
number  
console.log(`${octal1} -> ${typeof  
octal1}`); // 30863 -> number  
console.log(`${octal2} -> ${typeof  
octal2}`); // 30864 -> number  
console.log(`${hexa1} -> ${typeof  
hexa1}`); // 4095 -> number  
console.log(`${hexa2} -> ${typeof  
hexa2}`); // 4096 -> number
```

Vejamos os principais métodos estáticos (maiores esclarecimentos nos tópicos de orientação a objeto) associados ao objeto Number:

Método	Descrição
<b>isFinite</b>	Determina se o tipo e o valor passado é um número finito
<b>isInteger</b>	Determina se o tipo do valor passado é inteiro
<b>isNaN</b>	Determina se o valor passado é NaN (maiores informações no próximo tópico)
<b>isSafeInteger</b>	Determina se o tipo do valor passado é um inteiro seguro [número entre -(2 <sup>53</sup> -1) e 2 <sup>53</sup> -1]
<b>parseFloat</b>	Converte um string em um número de ponto flutuante
<b>parseInt</b>	Converte um string em um número inteiro

Agora vejamos os principais métodos de instância (maiores esclarecimentos nos tópicos de orientação a objeto) associados ao objeto Number:

Método	Descrição
<b>toExponential</b>	Retorna uma string representando o objeto Number por meio de notação exponencial
<b>toFixed</b>	Formata um número utilizando notação de ponto fixo
<b>toPrecision</b>	Retorna uma string que representa o valor do objeto Number com uma precisão específica
<b>toString</b>	Retorna uma string representando o objeto Number especificado
<b>valueOf</b>	Retorna o valor primitivo contido no objeto Number

## Propriedade global NaN (Not a Number)

- [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN)

Em se tratando de valores numéricos, há algumas operações nas quais valores numéricos são esperados, mas não é o que acontece, necessariamente. Para casos assim o JavaScript retorna um valor dado pela propriedade global de nome NaN (Not a Number).

Vejamos os casos em que isso pode acontecer:

```
//  
// OPERAÇÕES QUE RETORNAM NaN  
  
  
let nan1 = Number.parseInt("isso  
não é um número");  
let nan2 =  
Number.parseFloat("isso não é um  
número");  
let nan3 = Math.sqrt(-1);  
let nan4 = 3.14 * NaN;  
let nan5 = undefined +  
undefined;  
let nan6 = 0 * Infinity;  
console.log(`${nan1} -> ${typeof  
nan1}`); // NaN -> number  
console.log(`${nan2} -> ${typeof  
nan2}`); // NaN -> number  
console.log(`${nan3} -> ${typeof  
nan3}`); // NaN -> number  
console.log(`${nan4} -> ${typeof  
nan4}`); // NaN -> number  
console.log(`${nan5} -> ${typeof  
nan5}`); // NaN -> number  
console.log(`${nan6} -> ${typeof  
nan6}`); // NaN -> number
```

A propriedade global NaN apresenta algumas peculiaridades. Abaixo são apresentadas algumas delas:

```
//  
// COMPARAÇÕES COM NaN  
  
  
console.log(NaN === NaN); //  
false  
console.log(Number.NaN ===  
NaN); // false  
console.log(isNaN(NaN)); //  
true  
console.log(isNaN(Number.NaN))  
; // true  
console.log(Number.isNaN(NaN))  
; // true  
console.log(NaN !== NaN); //  
true  
console.log(Number.NaN !==  
Number.NaN); // true
```

## Arrays

Imagine a seguinte situação: é necessário calcular a altura média de uma determinada amostra de pessoas, além de identificar qual a maior e a menor alturas. Partindo da ideia que a amostra possui inicialmente 5 elementos, uma primeira abordagem para a solução do problema poderia ser a seguinte:

```
//  
// ARRAYS 01  
  
  
let altura1 = 1.78;  
let altura2 = 1.49;  
let altura3 = 1.65;  
let altura4 = 1.83;  
let altura5 = 2.11;  
  
let somaAlturas = altura1 +  
altura2 + altura3 + altura4 +  
altura5;  
let qtdAlturas = 5;  
let mediaAlturas = somaAlturas /  
qtdAlturas;  
let menorAltura =  
Math.min(altura1, altura2,  
altura3, altura4, altura5);  
let maiorAltura =  
Math.max(altura1, altura2,  
altura3, altura4, altura5);  
console.log(`  
A altura média entre as  
${qtdAlturas} alturas abaixo é  
${mediaAlturas}  
  
${altura1}  
${altura2}  
${altura3}  
${altura4}  
${altura5}
```

```
A menor altura é ${menorAltura} e  
a maior é ${maiorAltura}.  
`);  
  
//  
// SAÍDA  
//  
// A altura média entre as 5  
alturas abaixo é  
1.7719999999999998  
//  
// 1.78  
// 1.49  
// 1.65  
// 1.83  
// 2.11  
//  
// A menor altura é 1.49 e a  
maior é 2.11.  
//
```

O código desenvolvido atende perfeitamente à solicitação feita, mas qual nível de dificuldade surgiria caso a solicitação do cliente passasse de uma amostra de 5 alturas para uma de 5.000? Certamente os cabelos ficariam arrepiados.

A abordagem adotada não é nada eficiente para atender a esse nível de mudança e uma certeza a qual todos aqueles que se envolvem com desenvolvimento de sistemas têm é a de que **REQUISITOS MUDAM.**

É importante desenvolver códigos que atendam as possíveis mudanças que venham a surgir durante a vida útil do projeto. Isso nem sempre é simples, mas abordagens mais genéricas ao criar-se código é na maioria das vezes saudável. Aqui o termo 'na maioria das vezes' está sendo empregado porque pode acontecer de determinado código consumir energia e tempo excessivos na tentativa de torná-lo o mais genérico possível e depois identificar que o mesmo será usado para resolver problemas pontuais e que ocorrerão raramente. Essa percepção nem sempre está clara, principalmente no início dos projetos. É quando se diz que 'usou-se um canhão para matar uma mosca'.

Voltemos ao problema da média das alturas. Acredito que já deu para perceber que prosseguir com a abordagem inicial só traria dor de cabeça. Quando há a necessidade de manipulação de uma grande quantidade de dados, faz-se necessário lançar mão de algumas técnicas apropriadas para tal e também é importante que se use estruturas de dados adequadas.

A abordagem que será usada a seguir para resolver a demanda inicial fará uso de uma estrutura de dados chamada 'array'. Um array é uma estrutura de dados que tem as seguintes características básicas:

1. Armazena um conjunto de dados sob um único nome;
2. Cada elemento armazenado em um array possui um índice que indica a posição desse elemento dentro do array;
3. Para a linguagem JavaScript o primeiro elemento de um array é indicado pelo índice 0 (zero).

Vamos refatorar o código inicial para que o mesmo possa fazer uso de array na solução da demanda.

```
//  

// ARRAYS 02  

//  

//  

let alturas = [1.78, 1.49, 1.65,  

1.83, 2.11];  

let somaAlturas = 0;  

let qtdAlturas = alturas.length;  

let menorAltura =  

Number.POSITIVE_INFINITY;  

let maiorAltura =  

Number.NEGATIVE_INFINITY;  

for (let indice = 0; indice <  

qtdAlturas; indice++) {  

    somaAlturas += alturas[indice];  

    if(alturas[indice] <  

menorAltura) {  

        menorAltura = alturas[indice]  

    }  

    if(alturas[indice] >  

maiorAltura) {  

        maiorAltura = alturas[indice]  

    }  

}  

let mediaAlturas = somaAlturas /  

qtdAlturas;  

function listaAlturas(alturas) {  

    let lista = '';  

    for (let indice = 0; indice <  

alturas.length; indice++) {  

        lista += `  

${alturas[indice]}`;  

        if (indice < alturas.length - 1)  

            lista += `,`;  

    }  

    return lista;  

}
```

```
- 1) {
  lista += `\\n`;
}
}
return lista;
}
console.log(`  
A altura média entre as
${qtdAlturas} alturas abaixo é
${mediaAlturas}  
  
${listaAlturas(alturas)}
```

A menor altura é \${menorAltura} e  
a maior é \${maiorAltura}.  
`);  
  
//  
// SAÍDA  
//  
// A altura média entre as 5  
alturas abaixo é  
1.7719999999999998  
//  
// 1.78  
// 1.49  
// 1.65  
// 1.83  
// 2.11  
//  
// A menor altura é 1.49 e a  
maior é 2.11.  
//

A impressão inicial é de que a complexidade do código refatorado é bem maior que a da versão original. Na verdade, o que ocorreu foi o uso mais eficiente de elementos que já foram tratados anteriormente nesse material ou em módulos anteriores.

Um dos bônus dessa nova abordagem é a de que não faz mais diferença se a massa de dados de alturas contém 5 elementos, 5.000 ou 5.000.000. O código está preparado para atender aos requisitos fornecidos.

Quanto mais recursos da linguagem JavaScript forem conhecidos, maiores as chances de se poder produzir códigos mais eficientes. Vejamos mais alguns recursos de arrays e tentemos melhorar ainda mais o código já escrito até aqui.

## Manipulando Elementos de um Array

Como já vimos, cada elemento de um array está associado a um índice e o índice inicial de um array é o 0 (zero). Para acessar um elemento específico de um array basta usar o nome dado ao array e informar qual o índice do elemento buscado, entre colchetes.

Ex:

**nomeArray[indice]**

Estejamos atentos ao fato de que uma das operações que fizemos usando um array foi identificar o número de elementos que a mesma possuía. Utilizamos para tanto a propriedade length. Isso já é o suficiente para levantar a desconfiança de que arrays são objetos, certo? Absolutamente correto.

Vejamos:

```
let alturas = [1.78, 1.49, 1.65,
1.83, 2.11];
console.log(typeof alturas); // object
```

Uma vez que arrays são objetos, além de propriedades, como length, os mesmos também possuem métodos.

Diante da compreensão de que arrays são objetos, passemos a executar algumas operações com arrays para que possamos verificar e discutir os resultados.

Para os tópicos a seguir, tomar-se-á como base o array abaixo:

```
let arrayBase = ["Digital",
"college"];
```

Vejamos:

### ACESSANDO ELEMENTO EM POSIÇÃO INEXISTENTE

```
console.log(arrayBase); // [
'Digital', 'college' ]
console.log(arrayBase[0]); // Digital
console.log(arrayBase[1]); // college
console.log(arrayBase[2]); // undefined
```

### ALTERANDO ELEMENTO POR SEU ÍNDICE

```
arrayBase[1] = 'College';
console.log(arrayBase); // [
'Digital', 'College' ]
```

### INSERIR ELEMENTO EM POSIÇÃO DEFINIDA PELO ÍNDICE

```
arrayBase[4] = 'GAPs';
console.log(arrayBase[0]); // Digital
console.log(arrayBase[1]); // College
console.log(arrayBase[2]); // undefined
console.log(arrayBase[3]); // undefined
console.log(arrayBase[4]); // GAPs
console.log(arrayBase); // [
'Digital', 'College', <2 empty items>, 'GAPs' ]
console.log(arrayBase.length); // 5
```

## REMOVER / INSERIR EM POSIÇÃO DEFINIDA PELO ÍNDICE E QUANTIFICADOR

```
// remover
console.log(arrayBase.splice(4,
1)); // [ 'GAPs' ]
console.log(arrayBase); // [ 'Digital', 'College', <2 empty items> ]
console.log(arrayBase.splice(2,
2)); // [ <2 empty items> ]
console.log(arrayBase); // [ 'Digital', 'College' ]
console.log(arrayBase.splice(0,
2)); // [ 'Digital', 'College' ]
console.log(arrayBase); // []

// inserir
arrayBase.splice(0, 0,
'JavaScript', 'Python',
'College');
console.log(arrayBase); // [ 'JavaScript', 'Python', 'College' ]
arrayBase.splice(1, 0, 'Go');
console.log(arrayBase); // [ 'JavaScript', 'Go', 'Python',
'College' ]
arrayBase.splice(0, 3,
'Digital');
console.log(arrayBase); // [ 'Digital', 'College' ]
```

## INSERIR ELEMENTO NO FINAL

```
arrayBase.push("JavaScript");
console.log(array2); // [ 'Digital', 'College',
'JavaScript' ]
```

## REMOVER ELEMENTO DO FINAL

```
console.log(arrayBase.pop()); // JavaScript
console.log(arrayBase); // [ 'Digital', 'College' ]
```

## REMOVER ELEMENTO DO FINAL DE UM ARRAY VAZIO

```
console.log([] .pop()); // undefined
```

## INSERIR ELEMENTO NO INÍCIO

```
arrayBase.unshift("JavaScript");
console.log(arrayBase); // [ 'JavaScript', 'Digital',
'College' ]
```

## REMOVER ELEMENTO DO INÍCIO

```
console.log(arrayBase.shift());
// JavaScript
console.log(arrayBase); // [ 'Digital', 'College' ]
```

## REMOVER ELEMENTO DO INÍCIO DE UM ARRAY VAZIO

```
console.log([] .shift()); // undefined
```

Temos manipulado, até agora, o que é chamado de array unidimensional. Isso significa dizer que nossos arrays, até então, possuem apenas uma posição para o índice. Avancemos um pouco mais e vejamos que podemos criar arrays com mais de uma dimensão, também chamados genericamente de matrizes, e entender a utilidade dessa abordagem.

Imaginemos agora que nosso cliente deseja associar as alturas informadas aos nomes das pessoas detentoras de cada altura e ao final, informar não somente a maior e a menor altura, mas os nomes das pessoas que as possuem. Sempre há mais de uma forma de atacar um problema e não estamos diante de uma exceção. Faremos a seguinte abordagem: criaremos um array adicional com os nomes das pessoas associadas às alturas, mas esse novo array precisa estar na mesma ordem do array das alturas, ou seja, o primeiro nome do array de nomes está associado à primeira altura do array de alturas, o segundo nome do array de nomes está associado à segunda altura do array de alturas e assim sucessivamente.

Ao final teremos não mais 1 array, mas 2, como colocado a seguir:

```
let arrayAlturas = [1.78, 1.49,
1.65, 1.83, 2.11];
let arrayNomes = ["Carlos",
"Madalena", "Henrique", "Joana",
"Pedro"];
```

Iniciaremos nossa abordagem criando um array onde o primeiro elemento do nosso novo array será o nosso array de alturas e o segundo elemento o nosso array de nomes. Percebem que agora cada elemento do nosso novo array não é mais um elemento único, mas um array de outros elementos. Vejamos como fazer isso e algumas formas de entender o que está acontecendo:

```
let arrayAlturas = [1.78, 1.49,
1.65, 1.83, 2.11];
let arrayNomes = ["Carlos",
"Madalena", "Henrique", "Joana",
"Pedro"];
let arrayAlturasNomes = [arrayAlturas, arrayNomes];
console.log(arrayAlturasNomes.length); // 2
console.log(arrayAlturasNomes);
// 
// SAÍDA
//
// [
//   [ 1.78, 1.49, 1.65, 1.83,
2.11 ],
//   [ 'Carlos', 'Madalena',
'Henrique', 'Joana', 'Pedro' ]
// ]
//
console.log(arrayAlturasNomes[0].length); // 5
console.log(arrayAlturasNomes[0]);
// [1.78, 1.49, 1.65, 1.83,
2.11]
console.log(arrayAlturasNomes[1].length); // 5
console.log(arrayAlturasNomes[1]);
// [ 'Carlos', 'Madalena',
'Henrique', 'Joana', 'Pedro' ]
```

Sigamos com a abordagem na tentativa de solucionar a demanda do cliente. O que faremos agora é a velha abordagem do copiar-colar. Utilizaremos como base para a solução do problema a abordagem que o cliente já havia aprovado (quando usamos array unidimensional, lembram?) e faremos a refatoração (acostumem-se com esse termo) do código para atendermos aos novos requisitos.

Vejamos:

```

let arrayAlturas = [1.78, 1.49,
1.65, 1.83, 2.11];
let arrayNomes = ["Carlos",
"Madalena", "Henrique", "Joana",
"Pedro"];
let arrayAlturasNomes =
[arrayAlturas, arrayNomes];
let somaAlturas = 0;
let qtdAlturas =
arrayAlturas.length;
let mediaAlturas = 0;
let menorAltura =
Number.POSITIVE_INFINITY;
let maiorAltura =
Number.NEGATIVE_INFINITY;
let nomeMenorAltura = '';
let nomeMaiorAltura = '';
for (let indice = 0; indice <
qtdAlturas; indice++) {
    somaAlturas += arrayAlturasNomes[0][indice];
    if (arrayAlturasNomes[0]
[indice] < menorAltura) {
        menorAltura =
arrayAlturasNomes[0][indice];
        nomeMenorAltura =
arrayAlturasNomes[1][indice];
    }
    if (arrayAlturasNomes[0]
[indice] > maiorAltura) {
        maiorAltura =
arrayAlturasNomes[0][indice];
    }
}
mediaAlturas = somaAlturas / qtdAlturas;
function
listaAlturas(arrayAlturas) {
    let lista = "";
    for (let indice = 0; indice <
arrayAlturas.length; indice++) {
        lista += `${arrayAlturasNomes[0][indice]}`;
        if (indice <
arrayAlturas.length - 1) {
            lista += `\n`;
        }
    }
    return lista;
}
console.log(`A altura média entre as
${qtdAlturas} arrayAlturas abaixo
é ${mediaAlturas}`)

${listaAlturas(arrayAlturas)}

```

```

nomeMaiorAltura      =
arrayAlturasNomes[1][indice];
}
}
mediaAlturas = somaAlturas /
qtdAlturas;
function
listaAlturas(arrayAlturas) {
    let lista = "";
    for (let indice = 0; indice <
arrayAlturas.length; indice++) {
        lista += `${arrayAlturasNomes[0][indice]}`;
        if (indice <
arrayAlturas.length - 1) {
            lista += `\n`;
        }
    }
    return lista;
}
console.log(`A menor altura é ${menorAltura} e
pertence a ${nomeMenorAltura}.
A maior altura é ${maiorAltura} e
pertence a ${nomeMaiorAltura}.
`);
console.log();

```

```

// SAÍDA
//
// A altura média entre as 5
arrayAlturas abaixo é
1.7719999999999998
//
// 1.78
// 1.49
// 1.65
// 1.83
// 2.11
//
// A menor altura é 1.49 e
pertence a Madalena.
// A maior altura é 2.11 e
pertence a Pedro.

```

Essa abordagem multidimensional não tem limite. Poderíamos, alterando um pouco a abordagem anterior, implementar uma solução que também fornecesse a idade associada a cada nome, mas fazendo com que essa idade esteja no mesmo array onde estão os nomes, tornado o array de nome também multidimensional.

Vejamos uma possível solução:

```

let arrayAlturas = [1.78, 1.49,
1.65, 1.83, 2.11];
let arrayNomesIdades = [
["Carlos", 22],
["Madalena", 17],
["Henrique", 35],
["Joana", 24],
["Pedro", 18],
];
let arrayAlturasNomesIdades =
[arrayAlturas, arrayNomesIdades];
console.log(arrayAlturasNomesIdades.length); // 2
console.log(arrayAlturasNomesIdades);
//
// SAÍDA
//
// [
//   [ 1.78, 1.49, 1.65, 1.83,
2.11 ],
//   [
//     [ 'Carlos', 22 ],
//     [ 'Madalena', 17 ],
//     [ 'Henrique', 35 ],
//     [ 'Joana', 24 ],
//     [ 'Pedro', 18 ]
//   ]
// ]
//
console.log(arrayAlturasNomesIdades[0].length); // 5
console.log(arrayAlturasNomesIdades[0]); // [1.78, 1.49, 1.65,
1.83, 2.11]
console.log(arrayAlturasNomesIdades[1].length); // 5
console.log(arrayAlturasNomesIdades[1]);

```

```

// 
// SAÍDA
//
// [
// [ 'Carlos', 22 ],
// [ 'Madalena', 17 ],
// [ 'Henrique', 35 ],
// [ 'Joana', 24 ],
// [ 'Pedro', 18 ]
// ]
//
console.log(arrayAlturasNomesIdades[1][0].length); // 2
console.log(arrayAlturasNomesIdades[1][0]); // [ 'Carlos', 22 ]
console.log(arrayAlturasNomesIdades[1][1].length); // 2
console.log(arrayAlturasNomesIdades[1][1]); // [ 'Madalena', 17 ]
console.log(arrayAlturasNomesIdades[1][2][1]); // 35

```

É importante observar que essa abordagem expõe uma outra característica de arrays em JavaScript:

- **Os elementos que compõem um array não precisam ser do mesmo tipo**

Cada elemento do array **arrayNomesIdades** é um array também. Peguemos seu primeiro elemento e o observemos:

```
arrayNomesIdades [ 0 ] -> [ "Carlos", 22 ]
```

Percebemos que o primeiro elemento do array **["Carlos", 22]** é uma string e o segundo elemento um number.

A partir da abordagem feita, pode-se perceber a enorme flexibilidade que podemos ter com o uso de arrays multidimensionais. Por outro lado, também fica evidenciada a dificuldade de compreensão para alguns casos. Vejam a construção necessária para acessar a idade do usuário Henrique. Iniciamos acessando o elemento de índice 1 do array

**arrayAlturasNomesIdades**

**(arrayAlturasNomesIdades[1]).**

Esse elemento trata-se do array **arrayNomesIdades**. De posse desse elemento (array), acessamos seu elemento de índice 2 **(arrayAlturasNomesIdades[1][2])**. Isso nos traz o elemento (o qual é um array) **["Henrique", 35]**. Finalmente podemos acessar a idade do usuário Henrique **(arrayAlturasNomesIdades[1][2][1])**. No caso obtemos o valor 35.

Há de se perceber que, embora útil, essa forma de acessar dados pode tornar-se bastante complexa, acompanhando o nível de complexidade no qual os dados estão organizados.

Certamente há formas menos complexas e mais intuitivas de fazer acesso a dados, mas isso será visto em tópicos futuros.

Passemos a explorar mais recursos de arrays.

Para os tópicos a seguir, tomar-se-á como base o array abaixo:

```
let arrayAlturas = [1.78, 1.49, 1.65, 1.83, 1.65, 2.11];
```

Vejamos:

Método	Descrição
<b>concat</b>	Retorna um novo array contendo todos os arrays ou valores passados como parâmetro

```
arrayAlturas = arrayAlturas.concat([1.74, 1.80]);
console.log(arrayAlturas); // [1.78, 1.49, 1.65, 1.83, 1.65, 2.11, 1.74, 1.80]
```

<b>every</b>	Testa se cada um dos elementos do array passa pelo teste implementado pela função fornecida
--------------	---

```
let alturaMinima = 1.8;
function menorQueAlturaMinima(element, index, array) {
  console.log(`element: ${element} | index: ${index} || array: ${array}`);
  return element < alturaMinima;
}
console.log(arrayAlturas.every(menorQueAlturaMinima));

// 
// SAÍDA
//
// element: 1.78 | index: 0 || array:
1.78,1.49,1.65,1.83,1.65,2.11
// element: 1.49 | index: 1 || array:
1.78,1.49,1.65,1.83,1.65,2.11
// element: 1.65 | index: 2 || array:
1.78,1.49,1.65,1.83,1.65,2.11
// element: 1.83 | index: 3 || array:
1.78,1.49,1.65,1.83,1.65,2.11
```

```
// false
//
let alturaMaxima = 1.45;
function maiorQueAlturaMaxima(element, index, array) {
  console.log(`element: ${element} | index: ${index} || array: ${array}`);
  return element > alturaMaxima;
}
console.log(arrayAlturas.every(maiorQueAlturaMaxima));

//
// SAÍDA
//
// element: 1.78 | index: 0 || array: 1.78,1.49,1.65,1.83,2.11
// element: 1.49 | index: 1 || array: 1.78,1.49,1.65,1.83,2.11
// element: 1.65 | index: 2 || array: 1.78,1.49,1.65,1.83,2.11
// element: 1.83 | index: 3 || array: 1.78,1.49,1.65,1.83,2.11
// element: 1.65 | index: 4 || array: 1.78,1.49,1.65,1.83,2.11
// element: 2.11 | index: 5 || array: 1.78,1.49,1.65,1.83,2.11
// true
//
```

**filter**

Cria um novo array com todos os elementos que passaram no teste implementado pela função fornecida

```
let alturaFiltro = 1.6;
let arrayAlturasFiltradas = arrayAlturas.filter(altura => altura >=
alturaFiltro );
console.log(arrayAlturasFiltradas); // [ 1.78, 1.65, 1.83, 1.65, 2.11 ]
```

<b>find</b>	Retorna o valor do primeiro elemento do array que satisfizer a função de teste provida. Caso contrário, é retornado 'undefined'
-------------	---

```

let alturaProcurada = 1.65;
let procurarAltura = (altura, index) => {
  if (altura == alturaProcurada) {
    console.log(`\nA altura de ${altura} foi encontrada no índice
${index}\n`);
    return true;
  }
}
console.log(
  arrayAlturas.find(procurarAltura)
);

// 
// SAÍDA
//
// A altura de 1.65 foi encontrada no índice 2
//
// 1.65
// 
```

<b>forEach</b>	Executa uma dada função em cada elemento de um array
----------------	--

```

let alturaMinima = 1.7;
let arrayResultados = [];
let retorno = arrayAlturas.forEach((altura, index) => {
  arrayResultados[index] = [altura, 'Aprovado'];
  if(altura < alturaMinima) {
    arrayResultados[index][1] = 'Reprovado';
  }
});
console.log('arrayAlturas:\n', arrayAlturas);
console.log('arrayResultados:\n', arrayResultados);
console.log('retorno:\n', retorno); 
```

```

// 
// SAÍDA
//
// arrayAltura:
// [ 1.78, 1.49, 1.65, 1.83, 1.65, 2.11 ]
// arrayResultados:
// [
//   [ 1.78, 'Aprovado' ],
//   [ 1.49, 'Reprovado' ],
//   [ 1.65, 'Reprovado' ],
//   [ 1.83, 'Aprovado' ],
//   [ 1.65, 'Reprovado' ],
//   [ 2.11, 'Aprovado' ]
// ]
// retorno:
// undefined
//

```

**includes**

Determina se um array contém um determinado elemento, retornando true ou false apropriadamente

```

let arrayAlturas = [1.78, 1.49, 1.65, 1.83, 1.65, 2.11];
console.log(arrayAlturas.includes(1.83)); // true
console.log(arrayAlturas.includes(1.22)); // false

```

**indexOf**

Retorna o primeiro índice em que o elemento pode ser encontrado no array, retorna -1 caso o mesmo não esteja presente

```

let arrayAlturas = [1.78, 1.49, 1.65, 1.83, 1.65, 2.11];
console.log(arrayAlturas.indexOf(1.83)); // 3
console.log(arrayAlturas.indexOf(1.22)); // -1

```

**map**

Invoca a função callback passada por argumento para cada elemento do Array e devolve um novo Array como resultado

```
let alturaMinima = 1.7;
let arrayResultados = arrayAlturas.map((altura, index) => {
  let elemento = [altura, "Aprovado"];
  if (altura < alturaMinima) {
    elemento = [altura, "Reprovado"];
  }
  return elemento;
});
console.log('arrayAlturas:\n', arrayAlturas);
console.log('arrayResultados:\n', arrayResultados);

// 
// SAÍDA
//
// arrayAlturas:
// [ 1.78, 1.49, 1.65, 1.83, 1.65, 2.11 ]
// arrayResultados:
// [
//   [ 1.78, 'Aprovado' ],
//   [ 1.49, 'Reprovado' ],
//   [ 1.65, 'Reprovado' ],
//   [ 1.83, 'Aprovado' ],
//   [ 1.65, 'Reprovado' ],
//   [ 2.11, 'Aprovado' ]
// ]
```

**reduce**

Executa uma função reducer (fornecida por você) para cada elemento do array, resultando num único valor de retorno

```
let somaAlturasSemValorInicialDefinido = arrayAlturas.reduce(  
  (alturaAnterior, alturaAtual) => {  
    return alturaAnterior + alturaAtual;  
  }  
);  
let somaAlturasComValorInicialDefinido = arrayAlturas.reduce(  
  (alturaAnterior, alturaAtual) => {  
    return alturaAnterior + alturaAtual;  
  },  
  10.51  
);  
console.log(somaAlturasSemValorInicialDefinido); // 10.51  
console.log(somaAlturasComValorInicialDefinido); // 21.02
```

Observem que os métodos **map** e **filter** devolvem um novo array, sem promover alterações no array original. Essa característica é bastante útil quando se deseja encadear operações. Um exemplo seria, a partir da execução do método **map** usar seu retorno (um novo array) na aplicação do método **filter**. É uma operação relativamente comum utilizar-se desse recurso e ainda incluir o método **reduce** para realizar tarefas encadeadas, que, de outra forma, deveriam passar por várias etapas individuais de processamento.

Uma demonstração dessa técnica será usada no projeto final desse material, pois serão usados recursos os quais veremos só um pouco mais a frente. O projeto em questão está no tópico **Projeto Horas Extras.**

## Objetos

- [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Object](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Object)

Para a linguagem JavaScript um objeto é um conjunto de pares '**chave x valor**' envoltos por chaves {}.

JavaScript não é uma linguagem de programação orientada a objeto (OOP) e baseada em **classes** (onde a **herança** ocorre por meio de **classes de objetos**), mas uma linguagem de programação orientada a **protótipos** (onde a **herança** ocorre por meio dos **próprios objetos**). Um protótipo é um objeto tomado como modelo a partir do qual obtém-se (herda-se) as propriedades iniciais para montar um novo objeto.

Objetos podem ser criados a partir de sua notação literal (inicializador de objetos), por meio de funções fábricas e por meio de funções construtoras.

Vejamos um exemplo do uso da notação literal:

```
let objExemplo = {
  chave1: "valor1",
  chave2: 3.1415926,
  chave3: false,
  chave4: function (param1, param2)
{
  return param1 + param2;
},
chave5: [1, 2, 3, 4],
chave6: {
  chave7: "valor7",
  chave8: 1 + 3
};
};
```

As **chaves** são usadas como elemento de acesso aos **valores**. Os **valores** podem ser de qualquer tipo reconhecido pelo JavaScript, como por exemplo **string, number, boolean, function, array** e mesmo **object**.

As duas principais formas de se acessar um valor dentro de um object são usando a notação ponto (.) ou usando a notação colchete ([]). Vejamos ambas:

### Usando Notação Ponto (.)

Para acessar um valor dentro de um objeto a partir da notação ponto (.), basta usar o nome do objeto seguido de um ponto (.) e o **nome da chave** que dá acesso ao valor que se pretende obter.

Vejamos:

```
console.log(`
objExemplo.chave1:
${objExemplo.chave1}
typeof objExemplo.chave1: ${typeof
objExemplo.chave1}`);
console.log(`
objExemplo.chave2:
${objExemplo.chave2}
typeof objExemplo.chave2: ${typeof
objExemplo.chave2}`);
```

```

console.log(`
objExemplo.chave3:
${objExemplo.chave3}
typeof objExemplo.chave3: ${typeof
objExemplo.chave3}`);
console.log(`
objExemplo.chave4:
${objExemplo.chave4}
typeof objExemplo.chave4: ${typeof
objExemplo.chave4}`);
console.log(`
objExemplo.chave5:
${objExemplo.chave5}
typeof objExemplo.chave5: ${typeof
objExemplo.chave5}`);
console.log(`
objExemplo.chave6:
${objExemplo.chave6}
typeof objExemplo.chave6: ${typeof
objExemplo.chave6}`);
console.log(`
objExemplo.chave6.chave7:
${objExemplo.chave6.chave7}
typeof objExemplo.chave6.chave7:
${typeof
objExemplo.chave6.chave7}`);
console.log(`
objExemplo.chave6.chave8:
${objExemplo.chave6.chave8}
typeof objExemplo.chave6.chave8:
${typeof
objExemplo.chave6.chave8}`);

// 
// SAÍDA
// 
// objExemplo.chave1: valor1
// typeof objExemplo.chave1:
string
// 
```

```

// objExemplo.chave2: 3.1415926
// typeof objExemplo.chave2: number
//
// objExemplo.chave3: false
// typeof objExemplo.chave3:
boolean
//
// objExemplo.chave4: function
(param1, param2) {
// return param1 + param2;
// }
// typeof objExemplo.chave4:
function
//
// objExemplo.chave5: 1,2,3,4
// typeof objExemplo.chave5: object
//
// objExemplo.chave6: [object
Object]
// typeof objExemplo.chave6: object
//
// objExemplo.chave6.chave7: valor7
// typeof objExemplo.chave6.chave7:
string
//
// objExemplo.chave6.chave8: 4
// typeof objExemplo.chave6.chave8:
number
// 
```

### Usando a notação colchete ([])

Também é possível acessar um valor dentro de um objeto a partir da notação colchete ([]). Nesse caso usa-se o nome do objeto seguido de **[*'nome da chave'*]**, onde **nome da chave** é o nome da chave que dá acesso ao valor que se pretende obter.

Vejamos:

```
console.log(`\nobjExemplo["chave1"]: ${objExemplo["chave1"]}\ntypeof objExemplo["chave1"]: ${typeof objExemplo["chave1"]}`);\nconsole.log(`\nobjExemplo["chave2"]: ${objExemplo["chave2"]}\ntypeof objExemplo["chave2"]: ${typeof objExemplo["chave2"]}`);\nconsole.log(`\nobjExemplo["chave3"]: ${objExemplo["chave3"]}\ntypeof objExemplo["chave3"]: ${typeof objExemplo["chave3"]}`);\nconsole.log(`\nobjExemplo["chave4"]: ${objExemplo["chave4"]}\ntypeof objExemplo["chave4"]: ${typeof objExemplo["chave4"]}`);\nconsole.log(`\nobjExemplo["chave5"]: ${objExemplo["chave5"]}\ntypeof objExemplo["chave5"]: ${typeof objExemplo["chave5"]}`);\nconsole.log(`\nobjExemplo["chave6"]: ${objExemplo["chave6"]}\ntypeof objExemplo["chave6"]: ${typeof objExemplo["chave6"]}`);\nconsole.log(`\nobjExemplo["chave6"]["chave7"]: ${objExemplo["chave6"]["chave7"]}\ntypeof objExemplo["chave6"]["chave7"]: ${typeof objExemplo["chave6"][\n"chave7"]}\n`);\nconsole.log(`\nobjExemplo.chave6["chave8"]:\n${objExemplo.chave6["chave8"]}\ntypeof objExemplo.chave6["chave8"]: ${typeof\nobjExemplo.chave6["chave8"]}`);\n\n//\n// SAÍDA\n//\n// objExemplo["chave1"]: valor1\n// typeof objExemplo["chave1"]: string\n//\n// objExemplo["chave2"]: 3.1415926\n// typeof objExemplo["chave2"]: number\n//\n// objExemplo["chave3"]: false\n// typeof objExemplo["chave3"]: boolean
```

```

// 
// objExemplo["chave4"]: function
(param1, param2) {
// return param1 + param2;
// }
// typeof objExemplo["chave4"]:
function
//
// objExemplo["chave5"]: undefined
// typeof objExemplo["chave5"]:
object
//
// objExemplo["chave6"]: [object
Object]
// typeof objExemplo["chave6"]:
object
//
// objExemplo["chave6"]["chave7"]:
valor7
// typeof objExemplo["chave6"]
["chave7"]: string
//
// objExemplo.chave6["chave8"]: 4
// typeof
objExemplo.chave6["chave8"]:
number
//

```

Imaginemos um caso no qual deseja-se criar não um, mas uma série de objetos que possuem basicamente a mesma estrutura. Para esses casos o uso da notação literal torna o código muito verboso, por conta da reescrita de código. Também poderão surgir problemas quando for necessário realizar alterações nas definições desses objetos, uma vez que determinada alteração poderá provocar a refatoração de uma quantidade razoável desses objetos.

Vejamos um exemplo:

```

let pessoa01 = { primeiroNome:
"Pierre", ultimoNome: "Sá", idade:
42 };
let pessoa02 = { primeiroNome:
"Germano", ultimoNome: "Rosa",
idade: 56 };
let pessoa03 = { primeiroNome:
"Carlyle", ultimoNome:
"Fernandes", idade: 57 };

```

Percebam a quantidade de código que se repete diante da necessidade da criação de objetos que representem uma 'pessoa'. Os nomes dos atributos ('primeiroNome', 'ultimoNome' e 'idade') estão sempre sendo repetidos. Uma forma de minimizar esse problema é por meio da utilização do que se chama 'função fábrica'. Uma 'função fábrica' é essencialmente uma função que retorna (fabrica) um objeto (conjunto de 'chaves x valores') e que recebe como parâmetros os valores que serão associados a cada chave presente no objeto a ser retornado.

Vejamos um exemplo:

```

function pessoa(primeiroNome,
ultimoNome, idade) {
    return {
        primeiroNome,
        ultimoNome,
        idade,
    };
}

let claudio = pessoa("Cláudio",
"Araripe", 32);
let marcia = pessoa("Márcia",
"Mendonça", 30);

```

```
console.log(claudio); // {
primeiroNome: 'Cláudio',
ultimoNome: 'Araripe', idade: 32 }
console.log(marcia); // {
primeiroNome: 'Márcia',
ultimoNome: 'Mendonça', idade: 30
}
```

Há também a possibilidade de criar-se objetos a partir do que se chama 'função construtora'. Nessa abordagem faz-se uso da palavra reservada 'new' (que é apenas um 'syntactic sugar').

Embora não faça diferença em termos de sintaxe, é recomendado que a primeira letra do nome de uma 'função construtora' seja maiúscula.

Vejamos um exemplo:

```
let Pessoa = function
(primeiroNome, ultimoNome, idade)
{
    this.primeiroNome =
primeiroNome;
    this.ultimoNome = ultimoNome;
    this.idade = idade;
};

let francisco = new
Pessoa("Francisco", "Alencar",
19);
console.log(francisco); // Pessoa
{ primeiroNome: 'Francisco',
ultimoNome: 'Alencar', idade: 19 }
```

O ECMAScript2015 trouxe a palavra reservada 'class', que acaba permitindo escrever código JavaScript mais próximo da sintaxe de uma linguagem orientada a objeto.

Vejamos como o exemplo anterior ficaria se fosse usada a palavra reservada 'class':

```
class ClassPessoa {
    constructor(primeiroNome,
ultimoNome, idade) {
        this.primeiroNome =
primeiroNome;
        this.ultimoNome = ultimoNome;
        this.idade = idade;
    }
}

let paulo = new
ClassPessoa("Paulo", "Vidal", 49);
console.log(paulo); // ClassPessoa
{ primeiroNome: 'Paulo',
ultimoNome: 'Vidal', idade: 49 }
```

Observem que tanto na 'função construtora' quanto na 'classe' surgiu o elemento 'this'. É importante entender como usar esse elemento.

Em JavaScript 'this' refere-se a um objeto. A qual objeto ele se refere depende de como tal objeto está sendo invocado (usado ou chamado). A tabela abaixo traz um resumo das possibilidades:

**Quando 'this' aparece dentro de um método de um objeto, o 'this' refere-se a esse objeto**

**Sozinho, o 'this' refere-se ao objeto global**

**Dentro de uma função, o 'this' refere-se ao objeto global**

**Dentro de uma função, em 'strict mode', o 'this' é 'undefined'**

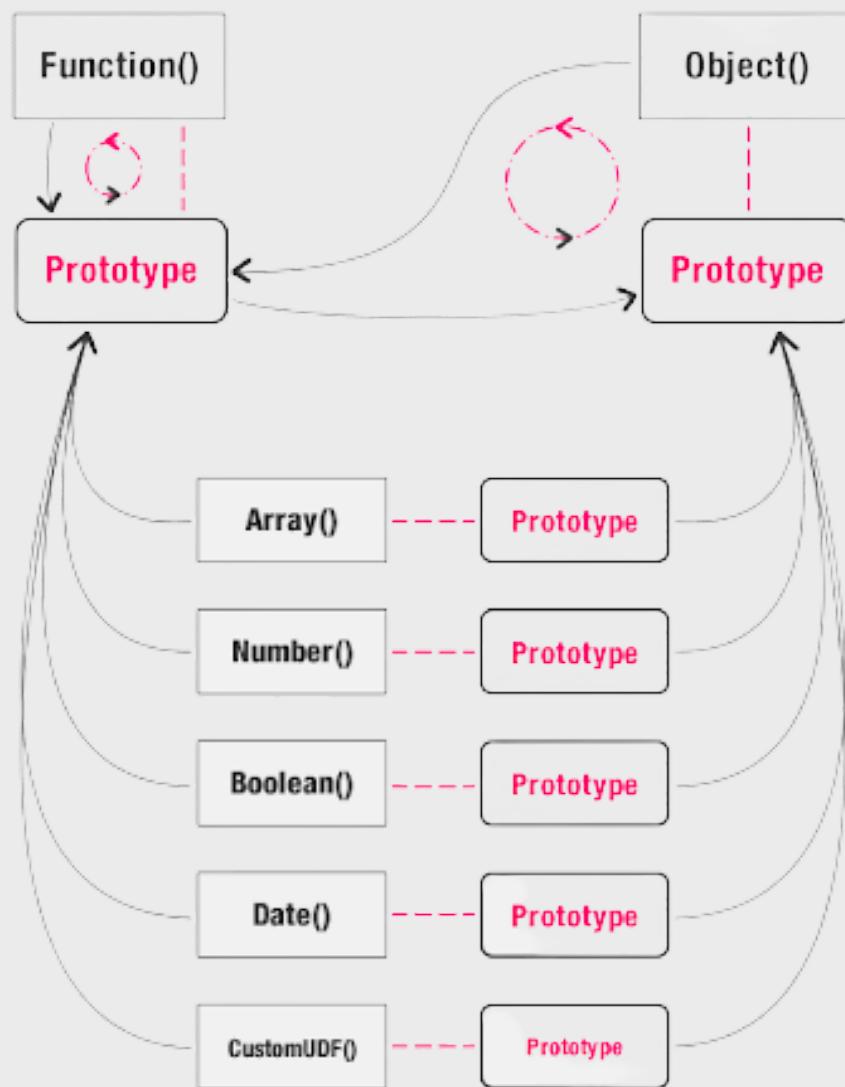
**Dentro de um 'event', o 'this' refere-se ao elemento que recebeu o 'event'**

**Usado em métodos como 'call()', 'applay()' e 'bind()' o 'this' pode referir-se a qualquer objeto**

Quando tratamos de objetos, um conceito importante é a **herança**. Em JavaScript a herança ocorre quando da criação de um objeto. Cada objeto criado possui um [[Prototype]]. Um [[Prototype]] é uma espécie de propriedade especial que **aponta** para o [[Prototype]] do objeto pai (objeto a partir do qual foi criado). Isso cria um encadeamento de [[Prototype]]s que termina no objeto de maior hierarquia no JavaScript. Esse objeto da maior hierarquia chama-se **Object**. A função construtora de um objeto também aponta para um [[Prototype]], mas nesse caso é o [[Prototype]] de uma função especial de mais alto nível hierárquico chamada **Function**. Vejamos a figura abaixo para melhor compreendermos:



## JavaScript Object Relationships



Uma vez que um objeto é uma coleção de 'chaves x valores', operações que accessem / manipulem tais elementos tornam-se essenciais. Vejamos algumas dessas operações.

- [https://developer.mozilla.org/ptBR/docs/Web/JavaScript/Enumerability\\_and\\_ownership\\_of\\_properties](https://developer.mozilla.org/ptBR/docs/Web/JavaScript/Enumerability_and_ownership_of_properties)

As operações a serem demonstradas tomarão como base a seguinte 'class' e sua instância 'paulo':

```
class ClassPessoa {
    constructor(primeiroNome, ultimoNome, idade) {
        this.primeiroNome = primeiroNome;
        this.ultimoNome = ultimoNome;
        this.idade = idade;
    }
}

let paulo = new ClassPessoa("Paulo", "Vidal", 49);
```

Operação	Descrição
for...in	Laço que interage sobre propriedades enumeradas de um objeto, na ordem original de inserção

```
for(const key in paulo) {
    console.log(` 
        key : ${key}
        value: ${paulo[key]}`
    );
}

// 
// SAÍDA
//
//    key : primeiroNome
//    value: Paulo
//
//    key : ultimoNome
//    value: Vidal
//
//    key : idade
//    value: 49
//
```

**for...of**

Percorre objetos iterativos (incluindo 'Array', 'Map', 'Set', o objeto 'arguments' e assim por diante), chamando uma função personalizada com instruções a serem executadas para o valor de cada objeto distinto. Não iteram, nativamente, sobre 'Object'

```
let arrIterable = ["Digital Collegea", 3.1415926, true];

for (const value of arrIterable) {
  console.log(value);
}

// 
// SAÍDA
//
// Digital Collegea
// 3.1415926
// true
//
```

**Object.keys()**

Retorna um array de propriedades enumeráveis de um determinado objeto, na mesma ordem em que é fornecida por um laço 'for...in'. A diferença é que um laço 'for...in' enumera propriedades que estejam na cadeia de protótipos

```
console.log(Object.keys(paulo)); // [ 'primeiroNome', 'ultimoNome', 'idade' ]
```

**Object.values()**

Retorna um array de valores das propriedades enumeráveis de um determinado objeto, na mesma ordem em que é fornecida por um laço 'for...in'. A diferença é que um laço 'for...in' enumera propriedades que estejam na cadeia de protótipos

```
console.log(Object.values(paulo)); // [ 'Paulo', 'Vidal', 49 ]
```

## Desestruturação de Objects e Arrays

- [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/Destructuring\\_assignment](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

Em JavaScript a desestruturação (destructuring) é uma operação por meio da qual se atribui valores presentes em um Array e/ou um Object a outras variáveis. A desestruturação fundamenta-se nas ideias de facilitação de codificação e compreensão dos códigos, tentando torná-los menos verbosos.

As possibilidades são bem generosas, mas mostraremos alguns exemplos de como esse recurso pode cumprir seu papel.

Partamos da ideia de que estamos diante de um array de objetos, sendo cada objeto a representação de um usuário, como definido abaixo:

```
let arrUsers = [  
  {  
    id: 3835,  
    name: "Chandraayan Ahuja Jr.",  
    email: "ahuja_jr_chandraayan@will.info",  
    gender: "male",  
    status: "inactive",  
  },  
  {  
    id: 3834,  
    name: "Ujjawal Kocchar",  
    email: "ujjawal_kocchar@morar-purdy.com",  
    gender: "female",  
    status: "inactive",  
  },  
  {  
    id: 3833,  
    name: "Bhuvaneshwar Khan II",  
    email: "bhuvaneshwar_khan_ii@kerluke.com",  
    gender: "male",  
    status: "active",  
  },  
];
```

Imaginemos agora que precisamos ter acesso ao segundo usuário desse array (aquele com índice 1). Teríamos essa construção:

```
let user01 = arrUsers[1];
```

Com o objeto **user01** em mãos, poderíamos querer acessar algumas de suas propriedades. Digamos que queremos ter acesso às propriedades **id**, **name** e **email** desse objeto. Uma construção válida seria:

```
let id = user01.id;
let name = user01.name;
let email = user01.email;

console.log(` 
  id: ${id}
  name: ${name}
  email: ${email}
`); 

// 
// SAÍDA
// 
// id: 3834
// name: Ujjawal Kocchar
// email: ujjawal_kocchar@morar-
purdy.com
//
```

Embora correta, diz-se que esse tipo de construção é muito **verbosa**. Para essas situações, na qual Arrays e/ou Objects estão envolvidos e precisamos obter alguns valores desses elementos, podemos lançar mão do conceito de **destructuring**.

Vejamos como a operação anterior torna-se bem mais **clean**.

```
let { id, name, email } = user01;

console.log(` 
  id: ${id}
  name: ${name}
  email: ${email}
`);

// 
// SAÍDA
// 
// id: 3834
// name: Ujjawal Kocchar
// email: ujjawal_kocchar@morar-
purdy.com
//
```

A operação de destructuring também pode ser aplicada a elementos de um Array. Observem que no código abaixo não estaremos usando essa técnica:

```
let arrNumerosPrimos = [1, 3, 5, 7, 11, 13, 17];

let primeiroPrimo = arrNumerosPrimos[0];
let quartoPrimo = arrNumerosPrimos[3];
let sextoPrimo = arrNumerosPrimos[5];

console.log(` 
primeiroPrimo: ${primeiroPrimo}
quartoPrimo: ${quartoPrimo}
sextetoPrimo: ${sextetoPrimo}
`);

// 
// SAÍDA
//
// primeiroPrimo: 1
// quartoPrimo: 7
// sextoPrimo: 13
```

Agora sim, usaremos o destructuring e poderemos confirmar como o código ficou mais leve:

```
let arrNumerosPrimos = [1, 3, 5, 7, 11, 13, 17];

let [primeiroPrimo, , , quartoPrimo, , sextoPrimo] =
arrNumerosPrimos;

console.log(` 
primeiroPrimo: ${primeiroPrimo}
quartoPrimo: ${quartoPrimo}
sextetoPrimo: ${sextetoPrimo}
`);

// 
// SAÍDA
//
// primeiroPrimo: 1
// quartoPrimo: 7
// sextoPrimo: 13
```

## Usando 'localStorage'

<https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>

JavaScript é uma linguagem de programação que tem sua origem voltada para a execução de código a partir de navegadores (browsers). Dentro desse contexto, é comum a necessidade de se manter valores entre as requisições. Existem algumas alternativas para atingir esse propósito. Há a possibilidade de usar-se 'cookies', 'sessionStorage' e 'localStorage'.

O uso de 'localStorage' e 'sessionStorage' traz uma vantagem significativa sobre o uso de 'cookies', no que diz respeito a capacidade de armazenamento de dados. Enquanto 'cookies' podem armazenar basicamente 4 KB de dados, o 'localStorage' e o 'sessionStorage' suporta até 5 MB de dados.

Uma característica que deve ser considerada no uso de 'localStorage' é o fato do mesmo permanecer armazenado no navegador até que uma ação de remoção de seu conteúdo seja realizada, diferentemente do 'sessionStorage' que se perde quando a sessão em uso pelo navegador é encerrada. Os 'Cookies' podem ser configurados para permanecerem ativo por um tempo determinado ou infinitamente, até que o mesmo seja removido.

Vejamos algumas operações usando o 'localStorage':

```
localStorage.setItem('theBest', 'Digital College');
localStorage.setItem('me', 'Otávio Medeiros');
localStorage.setItem('pi', 3.1415926);
console.log(localStorage.getItem('theBest')); // Digital College
console.log(localStorage.getItem('me')); // Otávio Medeiros
console.log(localStorage.getItem('pi')); // 3.1415926
console.log(localStorage.length); // 3
localStorage.removeItem('pi');
console.log(localStorage.getItem('pi')); // null
console.log(localStorage.length); // 2
localStorage.clear();
console.log(localStorage.length); // 0
```

## Usando o 'fetch'

[https://developer.mozilla.org/en-US/docs/Web/API/Fetch\\_API/Using\\_Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch)

A 'API Fetch' fornece uma interface JavaScript para acessar e manipular partes do 'pipeline' HTTP, tais como os 'requests' e 'responses'. Ela também fornece o método global 'fetch()' que fornece uma maneira fácil e lógica para buscar recursos de forma assíncrona através da rede.

Este tipo de funcionalidade era obtida anteriormente utilizando 'XMLHttpRequest'. O 'fetch' fornece uma alternativa melhor que pode ser facilmente utilizada por outras tecnologias como 'Service Workers'. O 'fetch' também provê um lugar lógico único para definir outros conceitos relacionados ao protocolo HTTP como CORS e extensões ao HTTP.

Vejamos um exemplo inicial onde o 'fetch' realiza uma requisição REST ao recurso 'todos' e, a partir dele, obter o elemento com 'id' 1:

```
fetch("https://jsonplaceholder.typicode.com/todos/1")
  .then((response) => response.json())
  .then((json) => console.log(json));

//  
// SAÍDA  
//  
// { userId: 1, id: 1, title: 'delectus aut autem', completed: false }
```

A ausência do verbo REST a ser utilizado indica que essa requisição usa o GET.

A função 'fetch()' aceita, além da URL, um parâmetro adicional que indica algumas configurações que devem ser informadas na requisição.

Para os exemplos a seguir, utilizaremos um documento HTML ('fetchExamples.html'), um arquivo 'JavaScript' ('fetchExamples.js') associado ao documento HTML e uma aplicação desenvolvida em 'JavaScript' ('httpServer.js') que deve ser executada com o 'Node'. O propósito é demonstrar algumas das possibilidades do 'fetch'.

Vejamos:

**fetchExamples.html**

```
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script type="text/javascript" src="fetchExamples.js"></script>
    <title>fetchs</title>
  </head>
  <body>
    <header></header>
    <main>
      <article>
        <section>
          <button onclick="jornadas(true)">jornadas</button>
          <button onclick="jornadas(false)">clear</button>
          <pre id="jornadas"></pre>
        </section>
        <section>
          <button onclick="picture(true)">picture</button>
          <button onclick="picture(false)">clear</button>
          <br>
          <img id="picture">
        </section>
      </article>
    </main>
    <footer></footer>
  </body>
</html>
```

**fetchExamples.js**

```
function jornadas(show) {
  if (show) {
    const requestInfo = {
      method: "GET",
      headers: new Headers({
        "Content-Type": "application/json",
      }),
    };
    fetch("http://localhost:9095/jornadas", requestInfo)
      .then((response) => response.json())
      .then((arrJson) => {
        let jornadas = "";
        for (json of arrJson) {
          jornadas += JSON.stringify(json, undefined, 2);
        }
        document.getElementById("jornadas").innerText = jornadas;
      });
  } else {
    document.getElementById("jornadas").innerText = "";
  }
}

function picture(show) {
  if (show) {
    const requestInfo = {
      method: "GET",
      headers: new Headers({
        "Content-Type": "image/png",
      }),
    };
    fetch("http://localhost:9095/picture", requestInfo)
      .then((response) => response.blob())
      .then((imageBlob) => {
        const imageObjectURL = URL.createObjectURL(imageBlob);
        document.getElementById("picture").src = imageObjectURL;
      });
  } else {
    document.getElementById("picture").src = "";
  }
}
```

**httpServer.js**

```
import http from "http";
import fs from "fs";
import path from "path";
import url from "url";

const jornadas = (req, res) => {
  let arrJornadas = [
    {
      nome: "Otávio Medeiros",
      diaSemana: "segunda-feira",
      feriado: true,
      jornadaTrabalhadaEmMinutos: 540,
    },
    {
      nome: "Jaqueline Barbosa",
      diaSemana: "terça-feira",
      feriado: true,
      jornadaTrabalhadaEmMinutos: 600,
    },
    {
      nome: "Otávio Medeiros",
      diaSemana: "quarta-feira",
      feriado: false,
      jornadaTrabalhadaEmMinutos: 490,
    }
  ];
  res.setHeader("Access-Control-Allow-Methods", "OPTIONS, GET");
  res.writeHead(200, { "Content-Type": "application/json" });
  res.write(JSON.stringify(arrJornadas));
  res.end();
};

const redirecting = (req, res) => {
  let url = http://google.com;
  let body = `<p>Redirecting to <a href="${url}">${url}</a></p>`;
  res.setHeader("Access-Control-Allow-Methods", "OPTIONS, GET");
  res.writeHead(302, {
    "Content-Type": "text/html",
    Location: url,
  });
};
```

```
"Content-length": body.length,
});
res.write(body);
res.end();
};

const picture = (req, res) => {
  const __filename = url.fileURLToPath(import.meta.url);
  const __dirname = path.dirname(__filename);
  let imagePath = path.join(__dirname, "imgs", "escape_chars.png");
  let fileStream = fs.createReadStream(imagePath);
  res.setHeader("Access-Control-Allow-Methods", "OPTIONS, GET");
  res.writeHead(200, { "Content-Type": "image/png" });
  fileStream.pipe(res);
};

const app = (req, res) => {
  res.setHeader("Access-Control-Allow-Origin", "*");
  res.setHeader("Access-Control-Request-Method", "*");
  res.setHeader("Access-Control-Allow-Headers", "*");
  switch (req.url) {
    case "/jornadas":
      jornadas(req, res);
      break;
    case "/redirecting":
      redirecting(req, res);
      break;
    case "/picture":
      picture(req, res);
      break;
    default:
      break;
  }
}
http.createServer(app).listen(9095);
```

## Escopo e Funções

- <https://andy-carter.com/blog/variable-scope-in-modern-javascript>

O escopo determina a acessibilidade (visibilidade) de variáveis. JavaScript possui 3 tipos de escopos. São eles:

- Escopo Global
- Escopo de Função
- Escopo de Bloco

### Escopo Global

Vamos começar considerando o escopo global. Variáveis declaradas globalmente podem ser acessadas e modificadas em qualquer lugar no código (bem, quase, mas chegaremos às exceções mais tarde).

As variáveis globais são declaradas no código fora de qualquer definição de função, no escopo de nível superior.

```
var a = "Fred Flinstone";
function alpha() {
    console.log(a);
}
alpha(); // 'Fred Flinstone'
```

Neste exemplo, 'a' é uma variável global. Está, portanto, prontamente disponível em qualquer função em nosso código. Então, aqui podemos acessar/modificar o valor de 'a' internamente na 'function alpha()'. Quando chamamos 'alpha()' o valor 'Fred Flinstone' é escrito no console.

Ao declararmos variáveis globais em um navegador da web, elas também são propriedades do objeto global 'window'. Dê uma olhada neste exemplo:

```
var b = "Wilma Flintstone";
window.b = "Betty Rubble";
console.log(b); // 'Betty Rubble'
```

A variável 'b' pode ser acessada/modificada como propriedade do objeto 'window' ('window.b'). Claro que não é necessário atribuir o novo valor 'b' através do objeto 'window'. Isso é apenas para provar o ponto. É mais provável que escrevamos o código acima como:

```
var b = "Wilma Flintstone";
b = "Betty Rubble";
console.log(b); // 'Betty Rubble'
```

Tenha cuidado ao usar variáveis globais. Eles podem levar a um código ilegível que também é difícil de testar. É muito melhor passar variáveis como argumentos para funções do que confiar em globais. As variáveis globais devem ser usadas com moderação, se for o caso.

Se você realmente precisa usar o escopo global, é uma boa ideia colocar suas variáveis em um 'namespace' para que elas se tornem propriedades de um objeto global. Por exemplo, crie um objeto global e dê a ele um nome como 'app'.

```
var app = {};  
// Cria o objeto global de nome 'app'  
app.foo = "Homer";  
app.bar = "Marge";  
function beta() {  
    console.log(app.bar);  
}  
beta(); // 'Marge'
```

Se você estiver usando o 'NodeJS', o 'top-level scope' não será o mesmo que o escopo global. Se você usar 'var foobar' em um módulo 'NodeJS', a variável 'foobar' é local para esse módulo. Para definir uma variável global no 'NodeJS', precisamos usar o objeto de 'namespace' global chamado 'global'.

```
global.foobar = 'Hello World!';  
// A variável 'foobar' é global  
no 'NodeJS'  
console.log(foobar); // Hello World!
```

É importante estar ciente de que, se você não declarar uma variável usando uma das palavras-chave 'var', 'let' ou 'const', a variável declarada receberá um escopo global.

```
function gamma() {  
    c = "Top Cat";  
}  
gamma();  
console.log(c); // 'Top Cat'  
console.log(window.c); // 'Top Cat'
```

É uma boa prática sempre declarar variáveis inicialmente com uma das palavras-chave 'var', 'let' ou 'const'. Dessa forma, estamos no controle do escopo de cada variável dentro do nosso código.

## Escopo Local

As variáveis declaradas dentro de uma função têm escopo local para essa função. Nos exemplos abaixo, ambas as variáveis 'b' e 'c' são locais para suas respectivas funções. A variável 'b' tem escopo local para a função 'delta' por está sendo declarada como um argumento da mesma. Apenas o valor de 'a' está sendo passado para a função e não a própria variável. A variável 'c' é declarada localmente na função 'epsilon'.

```
var a = 'Daffy Duck';  
// A variável 'a' tem escopo global  
function delta(b) {  
    // O parâmetro 'b' tem escopo local à 'function delta(b)'  
    console.log(b);  
}
```

```
function epsilon() {  
    // A variável 'c' tem escopo local à 'function epsilon()'  
    var c = 'Bugs Bunny';  
    console.log(c);  
}  
delta(a); // Daffy Duck  
epsilon(); // Bugs Bunny  
console.log(b); // O valor de 'b' é 'undefined' para o escopo global
```

O que o código a seguir mostraria no console?

```
var d = "Tom";  
function zeta() {  
    if (d === undefined) {  
        var d = "Jerry";  
    }  
    console.log(d);  
}  
zeta();
```

A resposta é 'Jerry' que, a princípio, pode não ser óbvio. Dentro da função 'zeta' estamos declarando uma nova variável 'd' que tem escopo local. Ao usar 'var' para declarar variáveis, o JavaScript inicia a variável no topo do escopo atual, independentemente de onde ela foi incluída no código. Portanto, o fato de termos declarado 'd' localmente dentro da condicional é irrelevante. Essencialmente o JavaScript está reinterprettando este código como:

```
var d = "Tom";  
function zeta() {  
    var d;  
    if (d === undefined) {  
        d = "Jerry";  
    }  
    console.log(d);  
}  
zeta();
```

Isso é conhecido como 'Hoisting' (içamento). É um recurso do JavaScript que vale a pena conhecer, pois pode facilmente criar bugs nos códigos se as variáveis não forem iniciadas no topo de um escopo. Felizmente, agora temos 'let' e 'const'. Isso ajudará a evitar esses problemas no futuro. Então, vamos dar uma olhada em como podemos usar 'let' para variáveis de escopo de bloco.

## Escopo de Bloco

Com a chegada do ES6, há alguns anos, surgiram duas novas palavras-chave para declarar variáveis: 'let' e 'const'. Ambas as palavras-chave nos permitem definir o escopo para um bloco de código, que é qualquer coisa entre duas chaves {}.

### let

A palavra-chave 'let' é vista por muitos como um substituto para o 'var'. No entanto, isso não é estritamente verdade, pois eles abrangem variáveis de maneira diferente. Enquanto o uso de 'var' nos permite criar uma variável com escopo global ou local, o escopo do 'let' é de bloco. É claro que um bloco pode ser uma função que nos permite usar 'let' praticamente como usariammos 'var' anteriormente.

```
function eta() {
  let a = "Scooby Doo";
}
eta();
```

No exemplo abaixo 'b' pertence ao escopo de bloco do 'loop for' (que inclui o bloco condicional 'if' dentro dele). Portanto, ele exibirá os números ímpares 1 e 3, em seguida, lançará um erro porque não podemos acessar 'b' fora do 'loop' para o qual ele foi definido.

```
for (let b = 0; b < 5; b++) {
  if (b % 2) {
    console.log(b);
  }
}
console.log(b);           // 'ReferenceError: b is not defined'
```

E quanto à nossa pequena função estranha 'zeta' de antes, onde vimos o efeito bizarro de 'hoisting' ( içamento ) do JavaScript? Se reescrevermos a função para usar 'let' o que acontece?

```
var d = "Tom";
function zeta() {
  if (d === undefined) {
    let d = "Jerry";
  }
  console.log(d);
}
zeta();
```

Desta vez, a saída de 'zeta' é 'Tom' porque 'd' tem escopo de bloco dentro da condicional ('if'). Mas isso significa que o 'hoisting' não está acontecendo aqui? Não é isso. Quando usamos 'let' ou 'const' JavaScript ainda promove o 'hoisting' das variáveis para o topo do escopo, mas enquanto no 'hoisting' de variáveis declaradas com 'var' as mesmas são iniciadas com 'undefined', no 'hoisting' de variáveis declaradas com 'let' as mesmas permanecem não iniciadas. E elas existem em uma 'zona morta temporal'.

Vamos dar uma olhada no que acontece quando tentamos acessar uma variável com escopo de bloco antes que ela seja iniciada.

```
function theta() {
  console.log(e);      // 'undefined'
  console.log(f);      // 'ReferenceError: d is not defined'
  var e = "Wile E. Coyote";
  let f = "Road Runner";
}
theta();
```

Portanto, a chamada 'theta' produzirá 'undefined' para a variáveis 'e' com escopo local e lançará um erro para a variável 'f' com escopo de bloco. Não podemos usar 'f' até que seja iniciado, neste caso só quando definirmos seu valor como 'Road Runner'.

Há uma outra diferença importante entre 'let' e 'var' que precisamos mencionar antes de prosseguir. Quando usamos 'var' no nível mais alto do nosso código, ela se torna uma variável global e é adicionada ao objeto 'window' nos navegadores. Com 'let' a variável se tornará global no sentido de que tem escopo de bloco para toda a base de código. Ela não se torna uma propriedade do objeto 'window', nos navegadores.

```
var g = 'Pinky';
let h = 'The Brain';
console.log(window.g); // 'Pinky'
console.log(window.h); // 'undefined'
```

## const

Essa palavra-chave foi introduzida ao lado 'let' como parte do ES6. Em termos de escopo, funciona da mesma forma que o 'let'.

```
if (true) {
  const a = "Count Duckula";
  console.log(a); // 'Count
Duckula'
}
console.log(a); // 'ReferenceError: a is not
defined'
```

Neste exemplo o escopo de 'a' é definido para a instrução 'if', portanto, pode ser acessado dentro da condicional, mas é indefinido fora dela.

Ao contrário de 'let' e de 'var', as variáveis definidas por 'const' não podem ser alteradas por meio de reatribuição.

```
const b = "Danger Mouse";
b = "Greenback"; // 'TypeError:
Assignment to constant variable'
```

No entanto, ao trabalhar com 'arrays' ou 'objets' as coisas são um pouco diferentes. Ainda não podemos reatribuir, portanto, o seguinte código falhará:

```
const c = ["Sylvester",
"Tweety"];
c = ["Tom", "Jerry"]; // 'TypeError: Assignment to
constant variable'
```

Mas, podemos modificar um 'array' ou 'objeto' declarado como 'const', a menos que usemos 'Object.freeze()' na variável para torná-la imutável.

```
const d = ["Dick Dastardly",
"Muttle"];
d.pop();
d.push("Penelope Pitstop");
Object.freeze(d);
console.log(d); // ["Dick
Dastardly", "Penelope Pitstop"]
d.push("Professor Pat
Pending"); // Throws error
```

## Escopo Local + Global

Nós meio que vimos isso antes quando analisamos o 'hoisting', mas vamos dar uma olhada no que acontece quando redeclararmos uma variável no escopo local que já existe globalmente.

```
var a = "Johnny Bravo"; //  
Global scope  
function iota() {  
    var a = "Momma"; // Local  
    scope  
    console.log(a); // 'Momma'  
    console.log(window.a); //  
'Johnny Bravo'  
}  
iota();  
console.log(a); // 'Johnny  
Bravo'
```

Quando redeclararmos uma variável global no escopo local, o JavaScript inicia uma nova variável local. Neste exemplo, temos uma variável global 'a', mas dentro da função ocorreu o 'hoisting' da nova variável local 'a'. A nova variável local não modifica a variável global, mas se quisermos acessar o valor global de dentro da função, precisaremos usar o objeto global 'window'.

Esse código seria muito mais fácil de ler se tivéssemos usado um 'namespace' global para nossa variável global e reescrevesse nossa função para usar o escopo do bloco:

```
var globals = {};  
globals.a = "Johnny Bravo"; //  
Global scope  
function iota() {  
    let a = "Momma"; // Local  
    scope  
    console.log(a); // 'Momma'  
    console.log(globals.a); //  
'Johnny Bravo'  
}  
iota();  
console.log(globals.a); //  
'Johnny Bravo'
```

## Arrow Function

- [https://dmitripavlutin.com/differences-between-arrow-and-regular-functions/#:~:text=arguments%20object%20inside%20the%20regular,args%20\).](https://dmitripavlutin.com/differences-between-arrow-and-regular-functions/#:~:text=arguments%20object%20inside%20the%20regular,args%20).)

Em JavaScript você pode definir funções de várias maneiras.

A primeira maneira usual é usando a palavra-chave 'function':

```
// Declaração de Função (Função Regular)
function greet1(who) {
  return `Hello, ${who}!`;
}
console.log(greet1("João")); // Hello, João!

// Expressão de Função (Função Regular)
const greet2 = function (who) {
  return `Hello, ${who}!`;
}
console.log(greet2("Maria")); // Hello, Maria!
```

Passaremos a chamar a 'declaração de função' e a 'expressão de função' apenas de 'função regular'.

A segunda maneira, disponível a partir do ES6, é a sintaxe de 'arrow function':

```
// Arrow Function
const greet3 = (who) => {
  return `Hello, ${who}!`;
}
console.log(greet3("Marcio"));
// Hello, Marcio!
```

Dentro de uma 'função regular', o valor de 'this' (também conhecido como contexto de execução) é dinâmico.

O contexto dinâmico significa que o valor de 'this' depende de como a função é invocada. Em JavaScript, existem 4 maneiras de invocar uma 'função regular'.

Durante uma chamada simples o valor de 'this' é igual ao objeto global (ou 'undefined' se a função for executada em 'modo estrito'):

```
var a = "variável do objeto
global (window)";
function myFunction() {
  console.log(this.a);
  console.log(this === window);
}
// chamada simples
myFunction();
//
// SAÍDA
//
// variável do objeto global
(window)
// true
//
```

Durante a invocação de um método, o valor de 'this' é o objeto que possui o método:

```

const myObject = {
  a: `variável do objeto
'myObject',
  method: function () {
    console.log(this.a);
    console.log(this.a ===
`variável do objeto
'myObject`);
  },
};

// Invocação de Método
myObject.method();
//
// SAÍDA
//
// variável do objeto
'myObject'
// true
//

```

Durante uma chamada indireta usando 'regularFunction.call(thisVal, arg1, ..., argN)' ou 'regularFunction.apply(thisVal, [arg1, ..., argN])' o valor 'this' é igual ao primeiro argumento, ou seja, 'thisVal' define o contexto:

```

var name = "John";
const person = {
  name: "Nathan",
};
const regularFunction = function
(age, gender) {
  console.log(`name: ${this.name},
age: ${age}, gender: ${gender}`);
};
regularFunction.call(person, 25,
"male");
regularFunction.apply(person, [25,
"male"]);

```

```

//
// SAÍDA
//
//   name: Nathan, age: 25,
//   gender: male
//   name: Nathan, age: 25,
//   gender: male
//

```

Durante a invocação de uma função construtora usando a palavra-chave 'new', o 'this' é igual à instância recém-criada:

```

let Pessoa = function
(primeiroNome, ultimoNome,
idade) {
  this.primeiroNome =
primeiroNome;
  this.ultimoNome = ultimoNome;
  this.idade = idade;
};
let francisco = new
Pessoa("Francisco", "Alencar",
19);
console.log(francisco); //
Pessoa { primeiroNome:
'Francisco', ultimoNome:
'Alencar', idade: 19 }

```

O comportamento de 'this' dentro de uma 'arrow function' difere consideravelmente do comportamento da 'função regular'. A 'arrow function' não define seu próprio contexto de execução.

Não importa como ou onde está sendo executado, o valor do 'this' dentro de uma 'arrow function' sempre é igual ao valor do 'this' da função externa. Em outras palavras, a 'arrow function' resolve 'this' lexicalmente.

No exemplo a seguir, myMethod() é uma função externa da 'arrow function' de nome 'callback()':

```

const myObject = {
  a: `variável do objeto
'myObject'`,
  myMethod: function (items) {
    const callback = (item) => {
      console.log(`${
        item} -
${this.a}`);
    };
    items.forEach((item) =>
      callback(item));
  },
};

myObject.myMethod([1, 2, 3]);
// 
// SAÍDA
//
// 1 - variável do objeto
'myObject'
// 2 - variável do objeto
'myObject'
// 3 - variável do objeto
'myObject'
// 
```

Resolver o 'this' lexicalmente é uma das grandes características das 'arrow functions'. Ao usar retornos de chamada dentro de métodos, você tem certeza de que a 'arrow function' não define sua própria função. Não há mais a necessidade de soluções alternativas como 'const.self = this' ou 'callback.bind(this)'.

Ao contrário de uma função regular, a invocação indireta de uma 'arrow function' usando 'myArrowFunc.call(thisVal)' ou 'myArrowFunc.apply(thisVal)' não altera o valor de 'this': o valor de contexto é sempre resolvido lexicalmente.

```

var name = "John";
const person = {
  name: "Nathan",
};
const arrowFunction = (age,
  gender) => {
  console.log(`name: ${
    this.name}, age: ${age},
  gender: ${gender}`);
};

arrowFunction.call(person, 25,
  "male");
arrowFunction.apply(person, [25,
  "male"]);
// 
// SAÍDA
//
// name: John, age: 25, gender:
male
// name: John, age: 25, gender:
male
// 
// SAÍDA
//
// name: John, age: 25, gender:
male
// name: John, age: 25, gender:
male
// 
```

## Promises

Promises são um conceito essencial do JavaScript. Elas estão presentes em praticamente todo o ecossistema da linguagem.

Promises são um padrão de desenvolvimento que visam representar a conclusão de operações assíncronas. Elas não eram nativas do JavaScript até o ES6, quando houve uma implementação oficial na linguagem, antes delas, a maioria das funções usavam callbacks.

## Fluxo Assíncrono

O JavaScript por si só é tido como uma linguagem que tem que lidar com várias chamadas e execuções que não acontecem no momento que o programador executou o código, por exemplo, a leitura de um arquivo no NodeJS de forma síncrona:

```

console.log(">> 1\n");
const texto =
fs.readFileSync("./arquivo.txt")
;
console.log(texto.toString());
console.log("\n>> 2");
//
// SAÍDA
//
// >> 1
//
// [[ FILE CONTENT ]]
//
// >> 2
//

```

Esta função é uma função síncrona, ou seja, quando a chamarmos, vamos pausar o que quer que esteja sendo executado e vamos realizar este processamento, depois vamos retornar o valor final. Desta forma estamos fazendo uma operação completamente síncrona. No nosso caso, vamos parar a execução do programa para buscar e ler o arquivo e depois vamos retornar seu resultado ao fluxo normal do programa.

Como queremos que nossas operações e nosso código rodem o mais rápido possível, queremos paralelizar o máximo de ações que conseguirmos. Ações de leitura de arquivos são consideradas lentas porque I/O é sempre mais lento que processamento em memória, vamos paralelizar a nossa função dizendo que queremos ler o arquivo de forma assíncrona:

```

console.log(">> 1\n");
fs.readFile("./arquivo.txt",
(err, texto) => {
    if (err) {
        console.log("Ocorreu um
erro.");
    } else {
        setTimeout(() =>
            console.log(texto.toString()),
            2000);
    }
});
console.log(">> 2\n");
//
// SAÍDA
//
// >> 1
//
// >> 2
//
// [[ FILE CONTENT ]]
//

```

Agora o que estamos fazendo é passando um callback para a função readFile que deverá ser executado após a leitura do arquivo.

Basicamente estamos registrando uma ação que vai ser executada após uma outra ação ser concluída, mas não sabemos quando essa ação será concluída. O que sabemos é apenas que em um momento ela será concluída, então o JavaScript utiliza o EventLoop para registrar um callback. Essencialmente o que estamos dizendo é: "Quando a função X acabar, execute Y e me dê o resultado". Então estamos delegando a resolução de uma computação para outro método.

## Por que Promises?

Se já tínhamos uma implementação de funções assíncronas, por que houve a preocupação de criar todo um novo padrão para podermos ter exatamente a mesma coisa? A questão aqui é mais a organização do código do que a funcionalidade.

Imagine que temos uma função que lê um arquivo, após este arquivo ser lido ela precisa escrever em um outro arquivo e executar outras função assíncrona. Nossa código ficaria assim:

```
import fs from "fs";

fs.readFile("./arq_00.txt", (err, texto) => {
  if (err) {
    throw new Error(err);
  } else {
    fs.writeFile("./arq_01.txt", texto, (err) => {
      if (err) {
        throw new Error(err);
      } else {
        fs.readFile("./arq_01.txt", (err, texto) => {
          if (err) {
            throw new Error(err);
          } else {
            let txtUpp = texto.toString().toUpperCase();
            fs.writeFile("./arq_02.txt", txtUpp, (err) => {
              if (err) {
                throw new Error(err);
              } else {
                fs.readFile("./arq_02.txt", (err, texto) => {
                  if (err) {
                    throw new Error(err);
                  } else {
                    console.log(texto.toString());
                  }
                });
              }
            });
          }
        });
      }
    });
  }
});
```

Vejam que o código fica super complicado para ler. É a Isso que se chama callback hell

As Promises foram um passo seguinte para que pudéssemos melhorar um pouco a execução do nosso código. Primeiramente vamos melhorar nosso código anterior, podemos extrair as funções posteriores (callbacks) para outros blocos, melhorando um pouco a nossa visualização:

```
import fs from "fs";

const callbackReadFileArq_00 = (err, texto) => {
  if (err) {
    throw new Error(err);
  } else {
    fs.writeFile("./promises/arq_01.txt", texto,
callbackWriteFileArq_01);
  }
};

const callbackWriteFileArq_01 = (err) => {
  if (err) {
    throw new Error(err);
  } else {
    fs.readFile("./promises/arq_01.txt", callbackReadFileArq_02);
  }
};

const callbackReadFileArq_02 = (err, texto) => {
  if (err) {
    throw new Error(err);
  } else {
    let txtUpp = texto.toString().toUpperCase();
    fs.writeFile("./promises/arq_02.txt", txtUpp,
callbackWriteFileArq_02);
  }
};

const callbackWriteFileArq_02 = (err) => {
  if (err) {
    throw new Error(err);
  } else {
    fs.readFile("./promises/arq_02.txt", callbackReadFileArq_02_End);
  }
};
```

```
const callbackReadFileArq_02_End = (err, texto) => {
  if (err) {
    throw new Error(err);
  } else {
    console.log(texto.toString());
  }
};

fs.readFile("./promises/arq_00.txt", callbackReadFileArq_00);
```

Agora o problema é outro, estamos encadeando nossas funções e fica muito difícil entender todo o fluxo porque temos que passar em várias partes do código. Com Promises, nosso código ficaria assim:

```
import fs from "fs";

const promiseWriteFileArq_01 = (texto) => {
  return fs.promises.writeFile("./arq_01.txt", texto, "utf-8");
};

const promiseReadFileArq_01 = () => {
  return fs.promises.readFile("./arq_01.txt", "utf-8");
};

const promiseWriteFileArq_02 = (texto) => {
  let txtUpp = texto.toString().toUpperCase();
  return fs.promises.writeFile("./arq_02.txt", txtUpp, "utf-8");
};

const promiseReadFileArq_02 = () => {
  return fs.promises.readFile("./arq_02.txt", "utf-8");
};

const showFileArq_02 = (texto) => {
  console.log(texto);
};

const showError = (err) => {
  throw new Error(err);
};
```

Veja que agora, apesar de nosso código não ter reduzido muito em tamanho, ele está mais legível, porque temos a implementação do then, de forma que conseguimos ver todo o pipeline de execução.

## Async/Await

Async e await são keywords que foram introduzidas no ES8 em 2017. Basicamente é um syntax sugar. Uma firula de linguagem que foi adicionada somente para poder facilitar a escrita do then e do catch.

O motivo pela adição do async/await foi o mesmo da adição das Promises no JavaScript, o callback hell. Só que dessa vez tínhamos o Promise hell, onde ficávamos aninhando Promises dentro de Promises eternamente e isso tornava tudo muito mais difícil de se ler.

A proposta de funções assíncronas é justamente nivelar todo mundo em um único nível. Vejamos como fica o nosso código anterior (com promises) fazendo uso do async/await:

```
import fs from "fs";

const promiseWriteFileArq_01 = (texto) => {
  return fs.promises.writeFile("./arq_01.txt", texto, "utf-8");
};

const promiseReadFileArq_01 = () => {
  return fs.promises.readFile("./arq_01.txt", "utf-8");
};

const promiseWriteFileArq_02 = (texto) => {
  let txtUpp = texto.toString().toUpperCase();
  return fs.promises.writeFile("./arq_02.txt", txtUpp, "utf-8");
};

const promiseReadFileArq_02 = () => {
  return fs.promises.readFile("./arq_02.txt", "utf-8");
};

const showFileArq_02 = (texto) => {
  console.log(texto);
};
```

## Projeto Horas Extras

Chegou a hora de aplicarmos diversos recursos vistos nos tópicos anteriores. A proposta é construir uma aplicação com os seguintes requisitos:

- Receber um array contendo as informações abaixo sobre a jornada de trabalho dos colaboradores:
  - nome: string
  - diaSemana: string
    - segunda-feira
    - terça-feira
    - quarta-feira
    - quinta-feira
    - sexta-feira
    - sábado
    - domingo
  - feriado: boolean
  - jornadaTrabalhadaEmMinutos: number
- Calcular o total de horas extras a serem pagas a um determinado colaborador, conhecendo-se as seguintes regras:
  - Horas extras trabalhadas aos sábados valem 50% a mais
  - Horas extras trabalhadas aos domingos valem 100% a mais
  - Horas extras trabalhadas em feriados valem 100% a mais
  - Horas extras trabalhadas em dias úteis não sofrem qualquer majoração
  - O array que se recebe como dados possui a jornada de trabalho diária contabilizada em minutos e o resultado final deverá ser fornecido em horas.

É óbvio que há múltiplas formas de resolver esse problema, mas a solução apresentada deverá tentar usar os recursos já apresentados e juntá-los para compor a solução.

Abaixo está o link do repositório onde encontra-se o código do servidor a ser acessado via REST / GET. O recurso é o <http://localhost:9095/jornadas> e o mesmo retorna um JSON com os dados a serem manipulados no sentido de obter o resultado demandado.

**<https://github.com/prof-otavio-medeiros/trabalho-u02-m02.git>**



# Digital College

ENSINO DE HABILIDADES DIGITAIS

[digitalcollege.com.br](http://digitalcollege.com.br)