Revisão para o Checkpoint

Certified Developer
The Ultimate Tech Degree

DigitalHouse >
Coding School

Temas

1. Node.js
2. Tipos de dados
3. Funções, Arrow Functions e Callbacks
4. Condicionais
5. Arrays e Métodos de Arrays
6. Strings e Métodos de Strings
7. Objetos literais

1 | Node.js

Módulos e require

O que é um módulo? Um módulo é uma **unidade de código que se encarrega de resolver uma problemática em particular** e é, naturalmente, reutilizável. No futuro, utilizaremos módulos externos para ampliar as características de nossa aplicação.

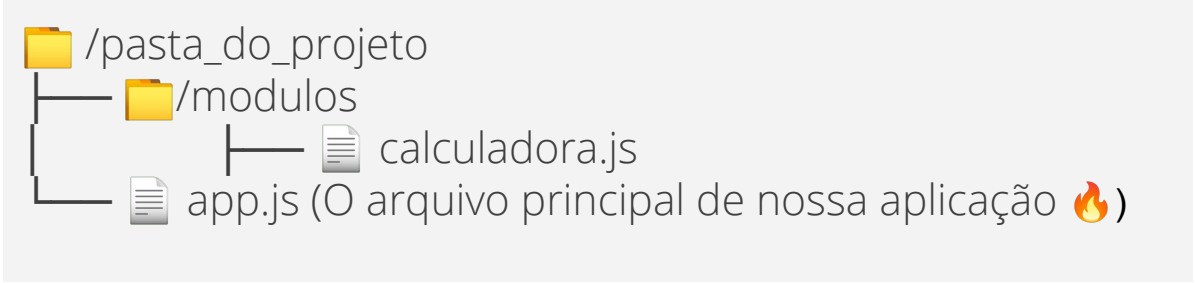
Assim como temos os módulos externos, também podemos criar nossos próprios módulos. A ideia é isolá-los em arquivos independentes para poder importá-los, controlá-los e, desta forma, dar-lhes manutenção de forma fácil e organizada.

Módulos e require

Imagine então que temos uma pasta que contém todo o nosso projeto. Dentro dela há um arquivo principal `app.js`, donde estaremos realizando várias tarefas. Se quiséssemos fazer um módulo que pode somar, subtrair, dividir e multiplicar, poderíamos então criar um módulo chamado `calculadora.js`, que vamos colocar fora de nosso arquivo principal. Para fazer isso, vamos criar uma pasta chamada **modulos** e aqui vamos colocar este e os outros módulos que criarmos.

Módulos e require

A árvore de arquivos para este projeto seria assim:



A ideia de cada módulo é **agrupar funções e dados** que vamos reutilizar no novo arquivo principal, ou em outros também. Então, nosso arquivo de **calculadora** seria mais ou menos assim:

JS

```
let calculadora = {
    somar: function(numeroA,numeroB) {
    return numeroA + numeroB
    },
    subtrair: function(numeroA,numeroB) {
    return numeroA - numeroB
    }
}
module.exports = calculadora;
```

Módulos e require

Na linha `module.exports = calculadora;` estamos esclarecendo que há algo dentro da variável **calculadora** que será o que queremos ter disponível para outros arquivos usarem. De alguma forma, estamos exportando o que está armazenado na variável calculadora. Mas não termina aqui.

Agora devemos - no arquivo onde queremos usar a calculadora - **requerer este módulo**. Como fazemos isso? Vamos para o arquivo `app.js` e vejamos a aparência do código.

```
// Primeiro vamos requerer o módulo e salvar o que é exportado em uma variável
let umaCalculadora = require('./modulos/calculadora')

// não há necessidade de colocar .js
console.log(umaCalculadora.somar(2,9))

// Se executarmos app.js no Terminal, o número 11 deve ser impresso no console
```

Documentação

Documentação do Node.js:
https://nodejs.org/dist/latest-v12.x/docs/api/

Módulo de FS - documentação:
https://nodejs.org/api/fs.html

Módulo process:
https://nodejs.org/dist/latest-v12.x/docs/api/process.html

2 | Tipos de dados

Numéricos (number)

```
let idade = 35; // número inteiro
let preco = 150.65; // número decimal
```

Como o JavaScript é em inglês, usaremos um ponto para separar os decimais.

Cadeias de caracteres (string)

```
let nome = 'Whesslley Rimar'; // aspas simples
let ocupacao = "Lider de Conteudo"; // aspas duplas têm o mesmo resultado.
```

Lógicos ou booleanos (boolean)

```
let status = true;
let x = false;
```

Objetos (object)

Ao contrário de outros tipos de dados que podem conter um único dado, os objetos são **coleções de dados** e todos os tipos anteriores podem existir dentro deles.

Podemos reconhecê-los porque são declarados com chaves **{ }**.

```
{}
let pessoa = {
  nome: 'Pedro', // string
  idade: 34, // number
  solteiro: true // boolean
}
```

Array

Assim como os objetos, os arrays são coleções de dados. Podemos reconhecê-los porque são declarados entre colchetes **[ ]**.

Os arrays são um tipo especial de objeto, por isso **não os consideramos apenas como mais um tipo de dado**.

Nós os mencionamos de maneira especial porque são muito comuns em todos os tipos de código.

```
{}
let comidasFavoritas = ['Pizza', 'Macarronada', 'Lasanha'];

let numerosSorteados = [12, 45, 56, 324, 452];
```

NaN (Not a Number)

Indica que o valor não pode ser convertido como um número.

```
{}  let malaDivision = "35" / 2; // NaN não é um número
```

Null (valor nulo)

Nós o atribuímos para indicar um valor vazio ou desconhecido.

```
{}  let temperatura = null; // Os dados não chegaram. Algo deu errado
```

Undefined (valor indefinido)

Indica a ausência de valor.

As variáveis têm um valor indefinido até que atribuamos algum valor a elas.

```
{}
let saudacao; // undefined, não há valor.
saudacao = "Olá!"; // Agora tem um valor
```

3 Funçõ̃es, Arrow Functions e Callbacks

Funções: estrutura básica

```
function somar (a, b) {
    return a + b;
}
```

Funções declaradas (function declaration ou function statement)

São aquelas que são declarados usando a **estrutura básica**. Podem receber um **nome**, escrito após a palavra reservada de **function**, por meio da qual podemos invocá-la.

```
{}
function fazerSorvete(quantidade) {
    return '🍦'.repeat(quantidade);
}
```

Funções expressas (Function Expression)

São aquelas que **são atribuídas como valor** de uma variável. Neste caso, a função em si não tem nome, ou seja, é uma **função anônima**.

Para invocá-la podemos usar o nome da variável que declaramos.

```
{}
let fazerSushi = function (quantidade) {
    return '🍣'.repeat(quantidade);
}
```

Invocando uma função

Antes de invocar uma função, precisamos que ela tenha sido declarada. Então, vamos declarar uma função:

```
function fazerSorvete() {
    return '🍦';
}
```

A forma de **invocar** (também se pode dizer chamar ou executar) uma função é escrevendo seu nome seguido da abertura e fechamento dos parênteses.

```
fazerSorvete(); // Retornará '🍦'
```

Invocando uma função

Se a função tem parâmetros, podemos passá-los dentro dos parênteses quando a invocamos. **É importante respeitar a ordem**, já que o JavaScript atribuirá os valores na ordem em que chegarem.

```
{}
function saudacao(nome, sobrenome) {
    return 'Olá ' + nome + ' ' + sobrenome;
}

saudacao('Whesslley', 'Rimar');
// retornará 'Olá Whesslley Rimar'
```

Invocando uma função

Também é importante ter em mente que quando temos parâmetros em nossa função, o JavaScript espera que os indiquemos ao executá-la.

```
function saudacao(nome, sobrenome) {
    return 'Olá ' + nome + ' ' + sobrenome;
}

saudacao(); // retornará 'Olá undefined undefined'
```

Neste caso, não tendo recebido o argumento de que precisava, o JavaScript atribui o tipo de dado **undefined** aos parâmetros *nome* e *sobrenome*.

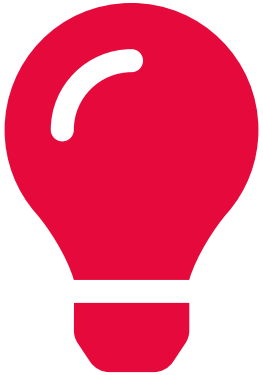
Invocando uma função

Para casos como o anterior, podemos definir **valores por padrão**.

Se adicionarmos um igual = após o parâmetro, poderemos especificar seu valor em caso de não chegar nenhum.

```
function saudacao(nome = 'Visitante',
    sobrenome = 'Anônimo') {
    return 'Olá ' + nome + ' ' + sobrenome;
}

saudacao(); // retornará 'Olá Visitante Anônimo'
```

"As **arrow functions** recebem este nome por conta do operador **=>**, que se parece com uma flecha.
Em inglês, geralmente é chamada de *fat arrow* para diferenciá-la da flecha simples **->**."



{código}

```
let saudacao = () => 'Olá Mundo!';

let dobro = numero => numero * 2;

let soma = (a, b) => a + b;

let horaAtual = () => {
    let data = new Date();
    return data.getHours() + ':' +
    data.getMinutes();
}
```

Arrow function sem parâmetros.

Requer parênteses para começar.

Por ter uma única linha de código, e essa é a que queremos retornar, o retorno é implícito.

{código}

```
let saudacao = () => 'Olá Mundo!';

let dobro = numero => numero * 2;

let soma = (a, b) => a + b;

let horaAtual = () => {
    let data = new Date();
    return data.getHours() + ':' +
    data.getMinutes();
}
```

Arrow function com um **único parâmetro** (não necessitamos dos parênteses para indicá-lo) e com um return implícito.

{código}

```
let saudacao = () => 'Olá Mundo!';

let dobro = numero => numero * 2;

let soma = (a, b) => a + b;

let horaAtual = () => {
    let data = new Date();
    return data.getHours() + ':' +
    data.getMinutes();
}
```

Arrow function com **dois parâmetros**.

Necessita dos parênteses e tem um return implícito.

{código}

```
let saudacao = () => 'Olá Mundo!';

let dobro = numero => numero * 2;

let soma = (a, b) => a + b;
```

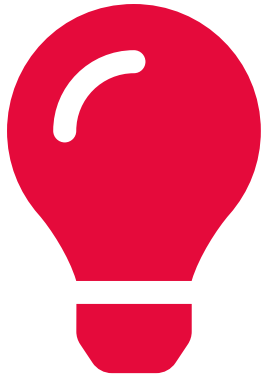
```
let horaAtual = () => {
    let data = new Date();
    return data.getHours() + ':' +
    data.getMinutes();
}
```

Arrow function **sem parâmetros** e com um **return explícito**.

Neste caso, fazemos uso das chaves e do **return** já que a lógica desta função se desenvolve em mais de uma linha de código.

Um **callback** é uma **função** que é passada como **parâmetro** de outra **função**.

A função que o recebe é quem se encarregará de **executá-lo** quando for necessário.



Invocando uma função

Neste caso, a função que passamos como **callback** não tem nome. Ou seja, é uma **função anônima**.

Como as **funções anônimas** não podem ser chamadas por seu nome, devemos declará-la dentro da função que se encarregará de chamar o callback.

```
setTimeout( function(){
    console.log('Olá Mundo!')
} , 1000)
```

Callback definido

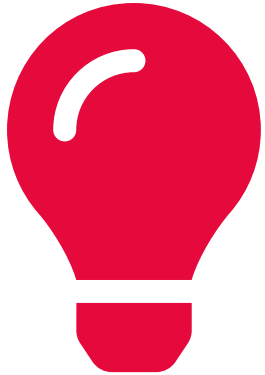
A função que passamos como **callback** pode ser uma função previamente **definida**. No momento de passá-la como parâmetro de outra função, nos referiremos a ela por seu nome.

```
{}
let myCallback = () => console.log('Olá mundo!');
setTimeout(myCallback, 1000);
```

Ao escrever uma função como parâmetro, o fazemos **sem os parênteses** para evitar que seja executada. Será a função que a recebe quem se encarrega de a executar.

4 | Condicionais

Nos permitem **avaliar condições** e realizar diferentes ações segundo o resultado dessas avaliações.



Condicional simples

Versão mais básica do `if`. Estabelece uma condição e um bloco de código a executar caso seja verdadeira.

```
if (condicao) {
  // código a executar se a condicao for verdadeira
}
```

Condicional com o bloco else

Igual ao exemplo anterior, mas agrega um bloco de código a executar em caso a condição seja falsa.

É importante ter em mente que o bloco `else` é opcional.

```
{}
if (condicao) {
  // código a executar se a condicao for verdadeira
} else {
  // código a executar se a condicao for falsa
}
```

Condicional com o bloco else if

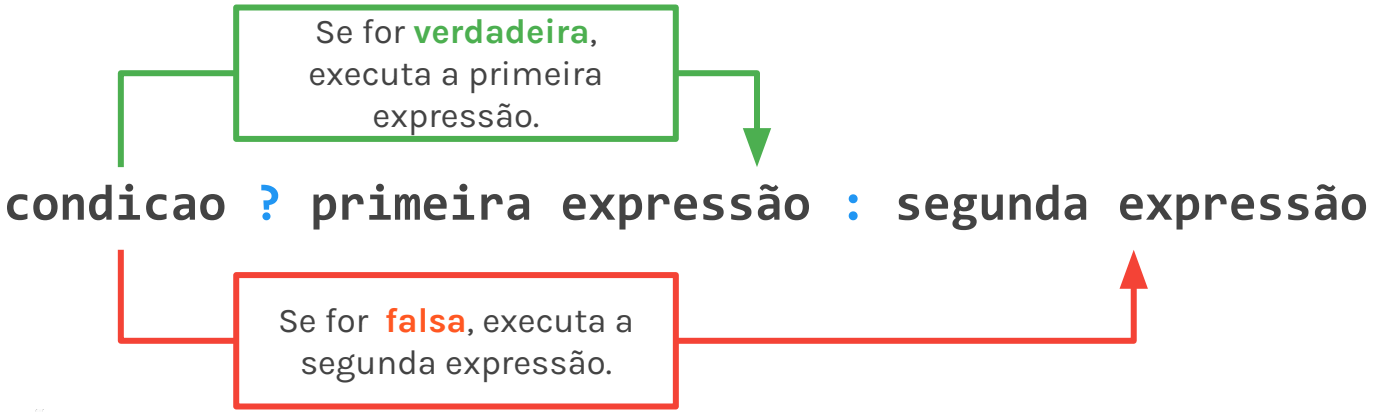
Igual ao exemplo anterior, mas agrega um `if` adicional. Ou seja, outra condição que pode ser avaliada caso a primeira seja falsa.

Podemos agregar quantos blocos `else if` quisermos. **Somente um poderá ser verdadeiro.** Do contrário, entrará em ação o bloco `else`, se existir.

```
if (condicao) {
  // código a executar se a condicao for verdadeira
} else if (outra condicao) {
  // código a executar se a outra condicao for verdadeira
} else {
  // código a executar se todas as condições forem falsas
}
```

if ternário: estrutura básica

Ao contrário de um if tradicional, o **if ternário** é escrito **horizontalmente**. Assim como o if tradicional, possui o mesmo fluxo (se esta condição é verdadeira faça isto, se não, faça aquilo outro), mas neste caso não é necessário escrever a palavra **if** ou a palavra **else**.



if ternário: estrutura básica

Para o if ternário **é obrigatório** colocar código na **segunda expressão.** Se não quisermos que nada aconteça, podemos usar uma string vazia `''`.

```
{}  4 > 10  ? 'O 4 é maior' : 'O 10 é maior';
```

Condição

Declaramos uma expressão que será avaliada como true ou false.

Primeira expressão

Se a condição for verdadeira, se executa o código que está depois do sinal de interrogação.

Segunda expressão

Se a condição for falsa, se executa o código que está depois dos dois pontos.

É obrigatório escrevê-la.

switch: estrutura básica

O switch é composto por uma expressão a ser avaliada, seguida de diferentes casos (case), tantos quanto quisermos, cada um contemplando um cenário diferente.

Os casos devem terminar com a palavra reservada break para evitar que o próximo bloco seja executado.

```
switch (expressao) {
      case valorA:
            // código a executar se a expressao for igual a valorA
            break;
      case valorB:
            // código a executar se a expressao for igual a valorB
            break;
}
```

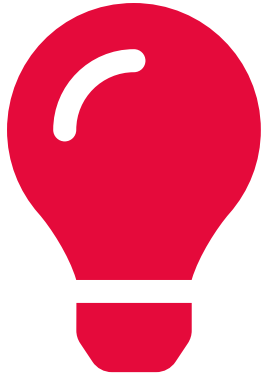
Agrupamento de casos

O switch também **nos permite agrupar casos** e executar um mesmo bloco de código para qualquer caso desse grupo.

```
switch (expressao) {
      case valorA:
      case valorB:
            // código a executar se a expressao for igual a ValorA ou valorB
            break;
      case valorC:
            //código a executar se valorC for verdadeiro.
            break;
}
```

5 Arrays e Métodos de Arrays

Os **arrays** nos permitem gerar uma **coleção** de **dados ordenados**.



Estrutura de um array

Utilizamos colchetes `[]` para indicar o **início** e o **fim** de um array. Utilizamos vírgulas `,` para **separar** seus elementos.

Dentro, podemos armazenar a quantidade de elementos que quisermos, sem importar o tipo de dado de cada um.

Ou seja, podemos ter em um mesmo array, dados do tipo string, number, boolean e todos os demais.

```
{}  let myArray = ['Star Wars', true, 23];
```

Posições dentro de um array

Cada dado de um array ocupa uma posição numerada conhecida como **índice**. A **primeira posição** de um array é **sempre 0**.

```
{}  let filmesFavoritos = ['Star Wars', 'Kill Bill', 'Alien'];
```

Para acessar um elemento pontual de um array, chamamos o array pelo seu nome e, **dentro dos colchetes**, escrevemos o **índice** ao qual queremos acessar.

```
{}  filmesFavoritos[2];
    // acessamos o filme Alien, o índice 2 do array
```

Tamanho de um array

Outra propriedade útil dos arrays é seu tamanho, ou quantidade de elementos. Podemos saber o número de elementos usando a propriedade. length.

```
let filmesFavoritos = ['Star Wars', 'Kill Bill', 'Alien'];
```

Para acessar o total de elementos de **um array**, chamamos o array pelo seu nome e, **seguido de um ponto** `.`, escrevemos a palavra **length**.

```
filmesFavoritos.length;
// Devolve 3, o número de elementos do array
```

.push()

Adiciona um ou vários elementos ao final do array.

- **Recebe** um ou mais elementos como parâmetros.
- **Retorna** o novo tamanho do array.

```
let cores = ['Vermelho','Laranja','Azul'];
cores.push('Violeta'); // retorna 4
console.log(cores); // ['Vermelho','Laranja','Azul','Violeta']
{}

cores.push('Cinza','Amarelo');
console.log(cores);
// ['Vermelho','Laranja','Azul','Violeta','Cinza','Amarelo']
```

.pop()

Elimina o último elemento de um array.

- **Não recebe** parâmetros.
- **Devolve** o elemento eliminado.

```
let series = ['Mad Men','Breaking Bad','The Sopranos'];

// criamos uma variável para guardar o que o .pop() devolver
let ultimaSerie = series.pop();

console.log(series); // ['Mad men', 'Breaking Bad']
console.log(ultimaSerie); // ['The Sopranos']
```

.shift()

Elimina o primeiro elemento de um array.

- **Não recebe** parâmetros.
- **Devolve** o elemento eliminado.

```
let nomes = ['Ana','Beatriz','Silvia'];

// criamos uma variável para guardar o que o .shift() devolver
let primeiroNome = nomes.shift();

console.log(nomes); // ['Beatriz', 'Silvia']
console.log(primeiroNome); // ['Ana']
```

.unshift()

Adiciona um ou vários elementos ao início de um array.

- **Recebe** um ou mais elementos como parâmetros.
- **Retorna** o novo tamanho do array.

```
let marcas = ['Audi'];

marcas.unshift('Ford');
console.log(marcas); // ['Ford', 'Audi']

marcas.unshift('Ferrari','BMW');
console.log(marcas); // ['Ferrari','BMW','Ford', 'Audi']
```

.join()

Une os elementos de um array, utilizando o separador que lhe especificarmos. Se não o especificarmos, utilizará vírgulas, por padrão.

- **Recebe** um separador (string). **É opcional.**
- **Retorna** uma string com os elementos unidos.

```
let dias = ['Segunda','Terça','Quarta'];

let separadosPorVirgulas = dias.join();
console.log(separadosPorVirgulas); // 'Segunda,Terça,Quarta'

let separadosPorHifen = dias.join(' - ');
console.log(separadosPorHifen); // 'Segunda - Terça - Quarta'
```

.indexOf()

Busca no array o elemento que receber como parâmetro.

- **Recebe** um elemento a buscar no array.
- **Retorna** o primeiro índice onde encontrar o que buscava. Se não o encontrar, retorna -1.

{}

```
let frutas = ['Maçã','Pêra','Uva'];
frutas.indexOf('Uva');
// Encontrou o que buscava. Devolve 2, o índice do elemento

frutas.indexOf('Banana');
// Não encontrou o que buscava. Devolve -1
```

.lastIndexOf()

Similar ao `.indexOf()`, com a exceção de que ele começa a procurar o elemento no **final do array** (de trás para frente).

No caso de elementos repetidos, ele retorna a posição do primeiro que encontrar (ou seja, o último se olharmos desde o início).

```
let clubes = ['Flamengo','São Paulo','Palmeiras','Santos'];

clubes.lastIndexOf('Santos');
{} // Encontrou o que buscava. Devolve 3

clubes.lastIndexOf('Cruzeiro');
// Não encontrou o que buscava. Devolve -1
```

.includes()

Também similar ao `.indexOf()`, com a exceção de que retorna um booleano.

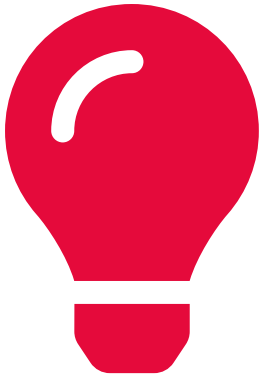
- **Recebe** um elemento a buscar no array.
- **Retorna** *true* se encontrar o que buscamos ou, *false* em caso contrário.

```
let frutas = ['Maçã','Pêra','Uva'];

frutas.includes('Uva');
// Encontrou o que buscava. Devolve true

frutas.includes('Banana');
// Não encontrou o que buscava. Devolve false
```

6 | Strings e Métodos de Strings

Para o JavaScript, os strings são como um array de caracteres.

Por esta razão temos **propriedades** e **métodos** muito úteis ao trabalhar com as informações que estão dentro.



As strings em Javascript

De muitas maneiras, para o JavaScript, uma string nada mais é do que um **array de caracteres**. Como em arrays, a primeira posição sempre será 0.

```
{}  let nome = 'Fran';
```

Para acessar um carácter pontual de uma string, chamamos a string pelo nome e, **dentro dos colchetes**, escrevemos o **índice** ao qual queremos acessar.

```
{}  nome[2];
    // acessamos a letra a, o índice 2 da string.
```

.length

Esta **propriedade** retorna a **quantidade total de caracteres** da string, incluindo os espaços.
Como é uma propriedade, ao invocá-la, não precisamos dos parênteses.

```
let mySerie = 'Mad Men';
mySerie.length; // devolve 7

let arrayNomes = ['Bart', 'Lisa', 'Moe'];
arrayNomes.length; // devolve 3

arrayNomes[0].length; // Corresponde a 'Bart', devolve 4
```

.indexOf()

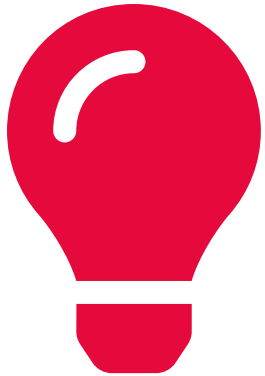
Busca na string, a string que receber como parâmetro.

- **Recebe** um elemento a buscar no array.
- **Retorna** o primeiro índice onde encontrar o que buscamos. Se não o encontrar, retorna um -1.

```
{}
let saudacao = 'Olá! Estamos programando';

saudacao.indexOf('Estamos'); // devolve 7
saudacao.indexOf('vamos'); // devolve -1, não o encontrou
```

Vimos antes que uma função é um bloco de código que nos permite agrupar funcionalidade para usá-la muitas vezes.

Quando uma função pertence a um objeto, nós a chamamos de método.



.slice()

Corta a string e retorna uma parte da string onde ela é aplicada.

- **Recebe** 2 números como parâmetros (podem ser negativos):
  - O índice de onde o corte começa.
  - O índice até onde fazer o corte (é opcional).
- **Retorna** a parte correspondente ao corte.

```
{}
let frase = 'Breaking Bad Rules!';

frase.slice(9,12); // devolve 'Bad'
frase.slice(13); // devolve 'Rules!'
```

.trim()

Elimina os espaços que estão no início e no final de uma string.

- **Não recebe** parâmetros.
- Não apaga os espaços do meio.

```
{}
```

```
let nomeCompleto = '   Homer Simpson   ';
nomeCompleto.trim(); // devolve 'Homer Simpson'

let nomeCompleto = '   Homer   J.    Simpson   ';
nomeCompleto.trim(); // devolve 'Homer    J.    Simpson'
```

.split()

Divide uma string em partes.

- **Recebe** uma string que será usada como o separador das partes.
- **Devolve um array** com as partes da string.

```
let cancao = 'And bingo was his name, oh!';

cancao.split(' ');
{} // devolve ['And', 'bingo', 'was', 'his', 'name,' , 'oh!']

cancao.split(', ');
// devolve ['And bingo was his name', 'oh!']
```

.replace()

Substitua uma parte de uma string por outra.

- **Recibe** duas strings como parâmetros:
  - A string que queremos buscar.
  - A string que usaremos como substituta.
- **Retorna** uma nova string com a substituição

```
{}
let frase = 'Olá Mundo!';
frase.replace('Mundo!', 'José'); // devolve 'Olá José!'
```

7 | Objetos literais

Objeto literal - Código

```
{}
let objetoLiteral = {
    texto: 'Meu texto',
    numero: 16,
    array: ['um', 'dois'],
    booleano: true,
    metodo: function() {return 'Olá'},
}
```


DigitalHouse>
Coding School