

# Introdução a NODE JS

“ É um **ambiente** de execução, ou seja, nos permite executar **Javascript** fora de um **navegador**. ”



# Arquitetura **NODE JS**

Todos os navegadores possuem um **mecanismo Javascript** para ler e renderizar código JS. Isso torna o idioma dependente de um **navegador** para ser executado.

Os próprios navegadores usam motores diferentes. Por causa dessa variedade que às vezes o mesmo código JS pode se comportar de forma diferente, dependendo do navegador em que ele está em execução.



Motor  
**CHAKRA**



Motor  
**SpiderMonkey**



Motor  
**v8**

# Arquitetura **NODE JS**

O **NodeJS** é construído sob o motor **v8** do Google Chrome. Isso o torna um ambiente de execução Javascript e faz com que o idioma não dependa mais do navegador para ser executado.

Desta forma, podemos programar tanto o Front-end como o Back-end na mesma língua: **Javascript**.



# Instalando **NODE JS**

A primeira coisa a fazer é baixar o Node.js do seu site oficial: [nodejs.org/pt-br](https://nodejs.org/pt-br)

Junto com o Node.js vamos instalar o gerenciador de pacotes **NPM**, que veremos com profundidade mais tarde.

Para verificar se ele foi instalado corretamente, abra um terminal e execute o comando `node -v` OU `node --version` .

*Tenha em mente que, ao instalar o Node, não estamos instalando um software, mas sim um ambiente de execução.*



# Testando **NODE JS**

- Para testar o NodeJS, crie uma pasta chamada **Node**.
- Abra o editor de texto Visual Studio Code. Vá até a pasta **Arquivo/Abrir** e selecione a pasta que acabamos de criar.
- Crie um arquivo chamado test.js e escreva o seguinte script:

```
console.log('Testando Node!');
```

- Abra o terminal. Para fazer isso, vá até **Terminal/Novo terminal**, ou execute o atalho **ctrl + shift + n**.
- No terminal escreva o seguinte comando: `node test.js`
- Se tudo correr bem, veremos a mensagem no terminal:

**Testando Node!**

# INTRODUÇÃO A NODE JS

# ARQUITETURA **NODE JS**

Todos os navegadores possuem um **mecanismo Javascript** para ler e renderizar código JS. Isso torna o idioma dependente de um **navegador** para ser executado.

Os próprios navegadores usam motores diferentes. É por causa dessa variedade que às vezes o mesmo código JS pode se comportar de forma diferente, dependendo do navegador em que ele está em execução.



Motor  
**CHAKRA**



Motor  
**SpiderMonkey**



Motor  
**v8**



# ARQUITETURA **NODE JS**

O NodeJS é construído sob o motor **v8** do Google Chrome. Isso o torna um ambiente de execução Javascript e faz com que o idioma não dependa mais do navegador para ser executado.

Desta forma, podemos programar tanto o Front-end como o Back-end na mesma língua: **Javascript**.



# INSTALANDO NODE JS

A primeira coisa a fazer é baixar o Node.js no seu site oficial:  
[nodejs.org/pt-br](https://nodejs.org/pt-br)

Junto com o Node.js vamos instalar o gerenciador de pacotes **NPM**, que veremos com profundidade mais tarde.

Para verificar se ele foi instalado corretamente, abra um terminal e execute o comando

```
node -v node --version
```



*Tenha em mente que, ao instalar o Node, não estamos instalando um software, mas sim um ambiente de execução.*

# TESTE NODE JS

Para testar o NodeJS, crie uma pasta chamada **Node**.

Abra o editor de texto Visual Studio Code. Vá até a pasta Arquivo/Abrir e selecione a pasta que acabamos de criar.

Crie um arquivo chamado test.js e escreva o seguinte script:

```
console.log('Testando Node!');
```

Abra o terminal. Para fazer isso, vá até Terminal/Novo terminal, ou execute o atalho **ctrl + shift + n**.

No terminal escreva o seguinte comando:

```
node test.js
```

Se tudo correr bem, veremos a mensagem no terminal:

*Testando Node!*

# Node Package Manager NPM

“ O **NPM** é o **gerenciador** de **pacotes** Node, que nos permite baixar e **instalar** bibliotecas para incorporar ao nosso **projeto**. ”



# Introdução ao **NPM**

Quando instalamos o Node em nossos computadores, várias bibliotecas são instaladas para uso **global**. Uma delas é o **NPM**: Node Package Manager.

Através dele, instalamos as bibliotecas que consideramos necessárias para o funcionamento ou desenvolvimento da nossa aplicação.

Podemos instalá-las **localmente**, para uso em um projeto específico, ou **globalmente**, para utilizarmos em qualquer lugar do nosso computador.

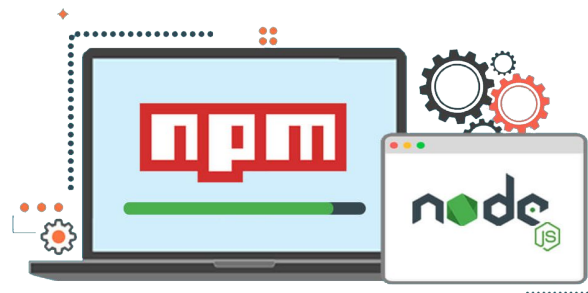


# O que são bibliotecas

São **bloco**s de **código** que nos permitem abordar soluções específicas dentro da aplicação que estamos desenvolvendo.

Em desenvolvimento web, há situações que se repetem em vários projetos. Alguns deles são gerenciar o upload de arquivos, validar um formulário ou restringir o acesso a um usuário que não está registrado.

As bibliotecas vêm para **facilitar** os problemas que sabemos que vamos encontrar ao desenvolver a nossa aplicação.



# Usando NPM

Quando o Node é instalado, é gerado o comando *npm* para uso no terminal. Para confirmar que a instalação está correta, podemos dar um dos comandos abaixo no terminal, que diz qual a versão do NPM está instalada.

```
>_ npm -v
```

```
>_ npm --version
```



# Usando NPM

A primeira coisa a fazer para usar o npm é **inicializar** nosso projeto Node, executando o comando:

```
>_ npm init
```

Este comando irá criar um arquivo **package.json**, dentro do qual todas as configurações do projeto serão salvas.

No momento, a propriedade que mais nos interessa neste arquivo é a *main*. Ela refere-se ao **entry point**, ou seja, ao ponto de entrada da nossa **aplicação**, onde colocaremos o nome do nosso arquivo principal, que, por convenção, costumamos chamar *app.js*

# package.json

```
{  
  "name": "app",  
  "version": "1.0.0",  
  "description": "",  
  "main": "app.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC"  
}
```

Caminho para **entry point**  
(ponto de entrada)

# Instalando bibliotecas

Para instalar uma biblioteca, usamos o seguinte comando:

```
>_ npm install PACKAGE --save
```

Onde iremos substituir a palavra *PACKAGE* pelo nome da biblioteca que queremos instalar.

A opção *--save* faz com que a biblioteca fique registrada no **package.json**, na propriedade *dependencies*.

# package.json

```
"main": "app.js",  
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1"  
},  
"author": "",  
"license": "ISC",
```

```
  "dependencies" {  
    "moment": "^2.24.0"  
  }  
}
```

**Referência** à(s)  
biblioteca(s) que  
instalamos em  
nosso projeto.

Dentro da pasta **node\_modules** serão criadas as pastas das bibliotecas que instalamos.

Cada um conterá os **arquivos necessários** para poder trabalhar com essa biblioteca dentro do projeto.



# Atualizando bibliotecas

Quando queremos que um pacote seja atualizado, precisamos executar o seguinte comando :

```
>_ npm update
```

Desta forma, ele irá atualizar os pacotes listados como dependências no seu arquivo **package.json**, baixar a versão mais atual para a pasta **node\_modules** e remover a versão anterior.

# Removendo bibliotecas

De vez em quando, podemos perceber que alguma biblioteca instalada não é mais necessária. Para remover, usamos o comando :

```
>_ npm uninstall PACKAGE
```

Este comando vai procurar essa dependência dentro do **package.json**, remover do arquivo e em seguida, apagar os arquivos da biblioteca que estão na pasta **node\_modules**.

Perceba que, diferente do comando **update**, aqui é obrigatório avisar qual o nome do pacote a ser removido. Sem esse nome, você terá um erro.

# Sistema de Módulos



“ São **estruturas de código** que, juntas, compõem toda a nossa aplicação e configuram sua usabilidade. ”



# Índice

O que é?

Como importar?

Módulo nativo

Módulo de terceiros

Módulo criado

# O que é um **módulo**?

Um módulo é um bloco de **código reutilizável**, uma unidade funcional, cuja existência não altera o comportamento de outros blocos de código.

A partir disso, o **Node** propõe atomizar nosso código, ou seja, **fragmentá-lo** em pequenos módulos, onde cada um terá uma funcionalidade específica para atingir um objetivo definido.



Existem **três tipos** de módulos:

Os **módulos nativos**, aqueles que já vieram instalados.

Os **módulos de terceiros**, aqueles que podemos instalar usando um gerenciador de pacotes (*por exemplo, o NPM*).

Os **módulos criados**, aqueles que nós mesmos definimos.



# Como importar um módulo?

Para requerer um módulo, não importa de que tipo seja, é preciso estar dentro do arquivo onde você quer incorporá-lo e fazer uso da função nativa do node `require()`. A função recebe como parâmetro uma string, que será o **nome** do módulo.

Esta função devolve um **objeto literal**, portanto, é importante salvar sua execução em uma **variável** para poder acessar, através da **dot notation**, todas as propriedades e funcionalidades do módulo.

```
{}  
let modulo = require('nomeModulo');  
modulo.propriedade;  
modulo.funcionalidade();
```

“ Por **convenção**, o **nome** da variável que armazena o módulo que estamos requerendo, geralmente recebe o **mesmo** nome do módulo, ou uma **abreviação** dele. ”



# Módulo **nativo**

Para requerer um módulo nativo, usamos a função `require()` e passamos como argumento o nome do módulo que vamos requerer.

[Neste link](#) você encontrará os módulos que estão inclusos quando instalamos o Node, listados em ordem alfabética à esquerda.

```
{ }  const fs = require('fs');
```

# Módulo de terceiros

Para requerer um módulo de terceiros, você deve primeiro instalá-lo usando o comando `npm install PACKAGE --save` .

Uma vez instalado, usamos a função `require()` e passamos como argumento o nome do módulo que instalamos.

```
>_ npm install moment --save
```

```
{ } const moment = require('moment');
```



# Módulo **criado**

Para requerer um módulo criado por nós, primeiro devemos criar um arquivo com extensão `.js` e dentro dele escrever o script que precisamos.

Uma vez definido o nosso código, temos que deixá-lo acessível para podermos importá-lo para a nossa aplicação. Para isso, temos que usar o **objeto nativo** `module` e sua propriedade `exports`. Atribuímos a ela o nome da variável que contém a informação que queremos exportar.

# {código}

```
const series = [  
  {titulo: 'Mad Men', temporadas: 7},  
  {titulo: 'Breaking Bad', temporadas: 5},  
  {titulo: 'Seinfeld', temporadas: 9},  
];
```

Definimos um conjunto de séries, onde em cada posição há um objeto literal com o título das séries e quantidade de temporadas.

# {código}

```
const series = [  
  {titulo: 'Mad Men', temporadas: 7},  
  {titulo: 'Breaking Bad', temporadas: 5},  
  {titulo: 'Seinfeld', temporadas: 9},  
];
```

```
module.exports = series;
```

Fazemos uso do objeto module e sua propriedade exports, e atribuímos a variável que queremos, neste caso, a série.

Tenha em mente que sempre que quisermos exportar módulos de um script, teremos que escrever esta linha no final do script.

# Módulo **criado**

Uma vez que exportamos nosso módulo, vamos ao arquivo para onde queremos importá-lo e usamos a função `require()` .

Neste caso, passamos como parâmetro a **rota** para o script onde encontramos o módulo a ser requerido. Para isso, usamos o `./` . Desta forma, estamos indicando ao Node que o caminho para aquele módulo **começa onde estamos** (app.js) **até o nome** do arquivo que passamos para ele.

*Como uma boa prática, normalmente armazenamos os módulos dentro de uma pasta com o mesmo nome do módulo que estamos criando.*



OS

# Módulo **criado**

Quando nos referimos a arquivos **Javascript**, não há necessidade de escrever a extensão.

```
{}
```

```
const series = require('./series/index');
```

▼ APP

> node\_modules

▼ series

JS index.js

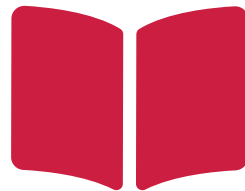
JS app.js

package-lock.json

package.json

“ Para poder ver **tudo** o que um módulo traz consigo, podemos fazer um `console.log()` da **variável** em que o armazenamos. ”





# Full Stack Node.js

## **Revisão: funções, condicionais e arrays**

# Antes de começarmos... onde estamos!

Metodologias de Desenvolvimento

Express

Sequelize

Introdução a Node.js

Como funciona a web

Banco de Dados



# Antes de começarmos... onde estamos!

03

## Introdução a NodeJS

O que é node.js,  
gerenciadores de  
dependências e sistema  
de módulos

04

## Revisão de funções, condicionais e arrays

Tipos de dados,  
métodos, condicionais,  
variáveis e variáveis  
arrays

05

## JSON, mais condicionais e ciclos

JSON, objetos literais,  
arrow functions, if  
ternário e ciclos

06

## Callback, mais ciclos e novos métodos

Callbacks, for in e for of,  
destructuring, objeto  
Date e spread operator

# O que vimos no Playground

- Variáveis
- Tipos de dados e operadores
- Funções
- Condicionais
- Arrays e Métodos de Arrays



# O que vamos ver hoje

- Variáveis e boas práticas em JS
- Tipos de dados
- Métodos de Arrays
- Continuação projeto CineHouse



# AS **VARIÁVEIS**

# Importante pontos sobre as variáveis:

- São espaços a memória em que podemos armazenar  
Informações (tipos de dados)
- Segue a padronização de nomenclatura **camelCase**, ou seja:

let nome-completo ❌

let nome\_completo ❌

let nomeCompleto ✅



# Importante pontos sobre as variáveis:

- Case-sensitive
- Podemos armazenar informações de 3 formas:

Let

Var

Const



MAS... E OS **ARRAYS**,

SÃO **VARIÁVEIS** OU NÃO?

# OS TIPOS DE DADOS



# Dentre os principais tipos de dados, temos:

- Number
- String
- Boolean
- Object
- Array
- Null



**Agora vamos praticar!**



# Atividade I - Variáveis para o CineHouse

- Voltando ao arquivo cinema que criamos na última aula, crie uma variável do tipo array.
- Esta variável armazenará um **array de objetos**, cada objeto representando um filme.

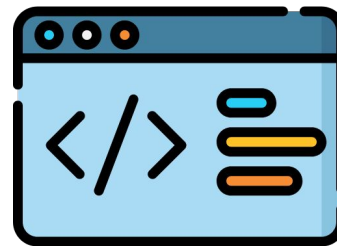
# Atividade I - Variáveis para o CineHouse

- Um filme (*objeto*) deverá ter as seguintes informações (*propriedades*):
  - **Código** (numérico - identificador do filme)
  - **Título** (string)
  - **Duração** (numérico - em horas)
  - **Atores** (array contendo nomes)
  - **Ano de lançamento** (numérico)
  - **Em cartaz** (booleano)
- Portanto, ao definir a variável, insira dois registros de filme de sua preferência

# OS MÉTODOS DE ARRAYS

# Alguns métodos de arrays

- `.push()`
- `.pop()`
- `.unshift()`
- `.indexOf()`
- `.find()`



# Atividade II - Funções para o CineHouse

- No mesmo arquivo, crie 3 funções:

- **adicionarFilme**

Esta função deve receber como parâmetros as informações de filmes e dentro de seu escopo, inseri-las na variável **catálogo** já existente (ao fim da lista).

- **buscarFilme**

Recebe o número identificador do filme e faz uma busca no array catálogo, deve retornar o objeto encontrado.

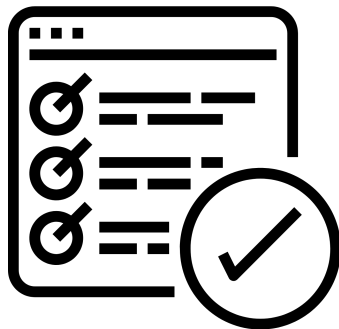
# Atividade II - Funções para o CineHouse

- **alterarStatusEmCartaz**

Função deve receber o número identificador, encontrar o objeto relacionado e alterar o status existente no campo **emCartaz** para **true**, caso o valor atual seja **false**, e **false**, caso o valor atual seja **true**



# Hora de compartilharmos o que criamos juntos



IDENTIFICARAM QUE PODEMOS  
**REAPROVEITAR FUNÇÕES** ?

# Variáveis

“

As **variáveis** são **espaços de memória** no computador onde podemos **armazenar** diferentes tipos de **dados.**

”



# Tipos de **variável**

Em Javascript, existem três tipos de variáveis:

- **var**
- **let**
- **const**

Para declarar uma variável escrevemos o tipo e o nome que queremos dar à variável. Vejamos cada parte com mais detalhe.

```
{}
```

```
var nome;  
let contador;  
const url;
```

# Declaração de uma variável

```
var nomeSignificativo;
```



**var**

a palavra reservada **var** indica ao Javascript que vamos **declarar uma variável**.



**Nome**

Só pode ser formado por letras, números e os símbolos \$ e \_ (sublinhado).  
Não pode começar com um número.  
Não deve conter ç ou caracteres com acentos.

É uma **boa prática** que os nomes das variáveis usem o formato **camelCase**, como **variavelExemplo** em vez de **variavelexemplo** ou **variavel\_exemplo**.



# Declaração de uma variável

```
var minhaVariavel;
```

não é o mesmo que

```
var MinhaVariavel;
```



*JavaScript é uma linguagem que **faz diferença entre MAIÚSCULAS e minúsculas**. Por isso, é bom seguir um padrão na hora de escrever os nomes.*

“

As **boas práticas**, embora não sejam obrigatórias para que nosso **código** funcione, irão permitir que ele seja **mais fácil de ler e de manter.**

”






# Atribuição de um valor

Quando declaramos uma variável, também podemos lhe atribuir um valor. Fazemos isso com o operador de atribuição.

```
var    meuApelido    =    'Hackerman' ;
```



## Nome

O nome vai nos ajudar a identificar nossa variável quando precisarmos usá-la.

## Atribuição

Diz ao JavaScript que queremos salvar o valor à direita na variável à esquerda.

## Valor

O que vamos salvar em nossa variável. Nesse caso, um texto.

# Atribuição de um valor

A **primeira vez** que declaramos uma variável, é necessária a palavra **var**.

```
{ } var meuApelido = 'Hackerman';
```

Uma vez que a variável já foi declarada, atribuímos valores sem **var**.

```
{ } meuApelido = 'Zezinho';
```



Nossa variável **salvará sempre o último valor** atribuído, isso quer dizer que **se nós atribuirmos um valor a ela novamente, perdemos o anterior.**

# Declaração com **let**

Estas variáveis se declaram de uma maneira similar, com a diferença que utilizamos a palavra reservada **let**.

```
{ } let contador = 1;
```

A principal diferença entre **var** e **let**, é que o **let** só será acessível no bloco de código em que foi declarada.

Os blocos de código são normalmente determinados pelas chaves { }.

**Vejamos um exemplo:**

# var

```
if (true) {  
  var nome = "João";  
}  
  
console.log(nome);  
// Ok, mostra "João"
```

Quando usamos **var**, o JavaScript ignora os blocos de código e converte nossa variável em global.

Isso quer dizer que se há outra variável **nome** em nosso código, seguramente estamos mudando seu valor.

# let

```
if (true) {  
  let nome = "João";  
}  
  
console.log(nome);  
// Error: nome não existe
```

Quando usamos **let**, o JavaScript respeita os blocos de código. Isso quer dizer que **nome** não poderá ser acessado fora do **if**.

Também quer dizer que podemos ter variável com o mesmo nome em diferentes blocos de nosso código.

# Declaração com **const**

As variáveis **const** se declaram com a palavra reservada **const**.

```
{} const email = "meu.email@hotmail.com";
```

As variáveis declaradas com **const** funcionam igual às variáveis **let**, e estarão disponíveis apenas no bloco de código em que foram declaradas.

Ao contrário do **let**, uma vez atribuído um valor, não podemos mudá-lo.

```
{} email = "meu.outro.email@hotmail.com";  
// Erro de atribuição, não se pode mudar o valor de um  
const
```

# Declaração com **let** ou **const**

Como falamos antes, tanto **let** como **const** são acessíveis dentro do bloco em que estão declaradas.

Por esta razão só podemos declarar uma vez. Se as declararmos novamente, o JavaScript retornará um erro.

```
{  
  let contador = 0;  
  let contador = 1;  
  // Erro de re-declaração da variável  
  
  const email = "meu.email@hotmail.com";  
  const email = "meu.novo.email@hotmail.com";  
  // Erro de re-declaração da variável  
}
```

“

As **palavras reservadas** como **var**, **let** e **const** só podem ser utilizadas para a finalidade em que foram criadas.

**Não podem ser** utilizadas como: **nome de variáveis, funções, métodos** ou **identificadores de objetos.** ”



# Os tipos de dados



# Índice

**Tipos de dados**

**Tipos de dados especiais**

# 1 | Tipos de dados

“ Os **tipos de dados** **permitem** que o JavaScript **conheça** as **características** e **funcionalidades** que estarão disponíveis para esses **dados**. ”



## Numéricos (number)

```
{}  
let idade = 35; // número inteiro  
let preco = 150.65; // decimais
```

Como o JavaScript está escrito em inglês, vamos usar um ponto para separar os decimais.

## Cadeias de caracteres (*string*)

```
{}  
let nome = 'PoP Vegan'; // aspas simples  
let ocupacao = "Mestre da pizza vegana"; // aspas  
duplas tem o mesmo resultado
```

## Lógicos ou booleanos (*boolean*)

```
{}  
let luzLigada = true;  
let temFeijoadaNoDomingo = false;
```

## Objeto (object)

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

Nós podemos reconhecê-los porque eles são declarados com chaves `{ }`.

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

## Objeto (object)

Para acessar um elemento dentro de um objeto, usamos a representação do ponto. Em outras palavras, o nome do objeto e o nome da propriedade separados por um ponto.

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

## Array

Como objetos, arrays são coleções de dados. Nós podemos reconhecê-los porque eles são declarados com colchetes `[]`.

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

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

```
{  
  let comidasFavoritas = ['Feijoada', 'Pizza', 'Sushi'];  
  let numerosSorteados = [12, 45, 56, 324, 452];  
}
```

## Array

Todo array terá seus dados organizados por posições (**index**) começando a partir da posição 0. Dessa forma, podemos selecionar facilmente cada valor em específico pela sua posição.

Para isso, basta informar o nome do array e entre colchetes [ ] colocar a posição desejada.

```
let numerosSorteados = [12, 45, 56, 324, 452];  
  
{} numerosSorteados[0] // 12  
    numerosSorteados[1] // 45
```



## 2 | Tipos de dados especiais

“

Os **tipos de dados especiais** permitem que o JavaScript determine **estados especiais** que os **dados** podem ter. ”



## **NaN** (Not A Number)

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

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

## **Null** (Valor nulo)

Nós o atribuímos para indicar um valor vazio ou desconhecido.

```
{ } let temperatura = null; // Não chegou um dado, algo falhou
```

## Undefined (valor indefinido)

Indica a ausência de um valor.

As variáveis têm um valor indefinido até lhes atribuirmos um valor.

```
{  
  let saudacao; // undefined, não tem valor  
  saudacao = "Olá!"; // Agora sim temos um valor
```



Os **comentários** são partes do nosso código que **não são executadas**. Eles começam sempre com duas barras inclinadas. **//**

**Nós os utilizamos para explicar** o que estamos fazendo, **e deixar informações úteis** para a nossa equipe ou para o nosso eu futuro.



```
// Math.round() retorna o valor arredondado para o inteiro mais próximo.
```

```
let arredondado = Math.round(20.49);
```

# Os operadores

“ Os **operadores** nos permitem **manipular o valor** das variáveis, realizar **operações** e **comparar** seus valores. ”



## Atribuição

Atribui o valor da direita na variável da esquerda

```
{ } let idade = 35; // Atribui o número 35 à idade
```

## Aritméticos

Nos permitem fazer operações matemáticas, retornando o resultado da operação.

```
{ } 10 + 15 // Soma → 25  
10 - 15 // Subtração → -5  
10 * 15 // Multiplicação → 150  
15 / 10 // Divisão → 1.5
```



## Aritméticos (continuação)

Nos permitem fazer operações matemáticas, e devolvem o resultado da operação.

```
{ } 15++ // Incremento, é igual a  $15 + 1 \rightarrow 16$   
15-- // Decremento, é igual a  $15 - 1 \rightarrow 14$ 
```

```
{ } 15 % 5 // Módulo, o resto de divisão 15 entre 5  $\rightarrow 0$   
15 % 2 // Módulo, o resto de divisão 15 entre 2  $\rightarrow 1$ 
```

O operador de módulo (%) nos retorna o resto de uma divisão.



“ Os **operadores** aritméticos sempre **retornarão** o **resultado numérico** da **operação** que se está realizando. ”



## De comparação simples:

Compara dois valores, retornando verdadeiro ou falso.

```
{}
```

```
10 == 15 // Igualdade → false  
10 != 15 // Desigualdade → true
```

## De comparação estrita:

Compara o valor e o tipo de dado também.

```
{}
```

```
10 === "10" // Igualdade estrita → false  
15 !== "15" // Desigualdade estrita → true
```

Veja no primeiro exemplo que, ao comparar 10 (Number) com 10 (String) usando a comparação estrita, teremos o resultado **false**. Isso porque, apesar de os valores serem iguais, os seus tipos são diferentes. Essa característica vale também para o operador de desigualdade, conforme o segundo exemplo.

## De comparação (continuação)

Compara dois valores, retornando verdadeiro ou falso.

```
{  
  15 > 15 // Maior que → false  
  15 >= 15 // Maior ou igual que → true  
  10 < 15 // Menor que → true  
  10 <= 15 // Menor ou igual que → true  
}
```

*Sempre devemos escrever o símbolo **maior** (>) ou **menor** (<) antes do igual (>= o <=). Se fizermos o contrário (=> o =<), JavaScript lê primeiro o operador de atribuição = e então ele não sabe o que fazer com o maior (>) ou o menor (<).*



“

Os **operadores** de **comparação** sempre **retornarão** um booleano, ou seja, **true** ou **false**, como resultado.

”



## Lógicos

Permitem combinar valores booleanos, e o resultado também retorna um booleano.

Existem três operadores **e** (and), **ou** (or) e **negação** (not).

**AND** (&&) — **todos** os valores devem avaliar como **true** para que o resultado seja **true**.

```
{ } (10 > 15) && (10 != 20) // false
```



```
{ } (12 % 4 == 0) && (12 != 24) // true
```



**OR ( || )** — **ao menos uma** comparação deve retornar como **true** para que o resultado seja **true**.

```
{ } (10 > 15) || (10 != 20) // true
```

  
FALSE — TRUE → TRUE

```
{ } (12 == 12) || (12 % 5 == 0) // true
```

  
TRUE — FALSE → TRUE

**NOT ( ! )** — nega a condição; se era **true**, será **false**, e vice-versa.

```
{ } !false // true  
{ } !(20 > 15) // false
```

“ Os **operadores** de lógicos sempre **retornarão** um booleano, ou seja, **true** ou **false**, como resultado. ”





## Concatenação

Serve para unir duas strings, formando assim uma nova.

```
{}  
  let nome = 'Teodoro';  
  let sobrenome = 'Garcia';  
  let nomeCompleto = nome + ' ' + sobrenome;
```

Se misturarmos outros tipos de dados, eles serão convertidos em string.

```
{}  
  let fila = 'M';  
  let assento = 7;  
  let localização = fila + assento; // 'M7' como string
```

# Funções

# Índice

1. Declaração e estrutura
2. Execução
3. Escopo

# 1 | Declaração e estrutura

“

Uma função é um **bloco de código** que podemos usar todas as vezes que for necessário. Esse bloco pode realizar uma **tarefa específica** e **retornar** um valor.

Assim podemos **agrupar** o **código** que vamos **usar mais de uma vez**. ”



# Estrutura básica

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

## Palavra reservada

Usamos a palavra **function** para indicar ao Javascript que vamos escrever uma nova função.

# Estrutura básica

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

## Nome da função

Definimos um **nome** para podermos chamar nossa função no momento em que formos invocá-la.

# Estrutura básica

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

## Parâmetros

Escrevemos os parênteses e dentro deles colocamos os parâmetros da função. Se receber mais de um, os separamos usando vírgulas ,.

Mesmo se a função não receber parâmetros, escrevemos os parênteses sem nada dentro ().



# Estrutura básica

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

## Parâmetros

Dentro de nossa função podemos acessar os parâmetros como se fossem variáveis. Quer dizer, só de escrever os nomes dos parâmetros, poderemos trabalhar com eles.

# Estrutura básica

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

## Corpo

Entre as chaves de abertura e fechamento escrevemos a lógica da nossa função, ou seja, o código que queremos que seja executado cada vez que a chamarmos.

# Estrutura básica

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

## O retorno

É muito comum escrevermos uma função da qual queremos devolver externamente o resultado do processo que está sendo feito dentro dela.

Para isso utilizamos a palavra reservada **return** seguida do que queremos retornar.

# Funções declaradas

São aquelas que são declaradas usando a **estrutura básica**. Recebem um **nome formal** através do qual a invocaremos. Eles são carregados antes que qualquer código seja executado.

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

# Funções **expressas**

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

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

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

# 2 | Execução

“

Podemos imaginar as funções como se fossem máquinas.

Durante a **declaração**, cuidamos da **construção** da máquina e durante a **execução**, a colocamos em **funcionamento**.

”



# Executando uma função

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

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

A maneira de **executar** uma função é digitando seu nome, seguido pela abertura e fechamento dos parênteses:

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



# Executando uma função

Se a função espera por argumentos, podemos passá-los dentro de parênteses.

**É importante respeitar a ordem** se houver mais de um parâmetro, pois o Javascript irá atribuí-los na ordem em que chegam.

{}

```
function saudacao(nome, sobrenome) {  
    return 'Olá ' + nome + ' ' + sobrenome;  
}  
  
saudacao('Roberto', 'Rodríguez');  
// retornará 'Olá Roberto Rodríguez'
```

# Executando uma função

Também é importante ter em mente que, quando temos parâmetros em nossa função, o Javascript espera que passemos como argumentos ao executá-la.

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

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

# Executando uma função

Para este tipo de caso, o Javascript nos permite definir os **valores predefinidos**. Se adicionarmos um igual `=` depois do parâmetro, poderemos especificar o seu valor no caso de não chegar nenhum.

```
{  
  function saudacao(nome = 'visitante',  
    sobrenome = 'anônimo') {  
    return 'Olá ' + nome + ' ' + sobrenome;  
  }  
  
  saudacao(); // retornará 'Olá visitante anônimo'
```

# Armazenando os resultados

Caso queira salvar o que uma função retorna, você precisará armazená-la em uma variável.

```
function fazerSorvete(quantidade) {  
    return '🍦'.repeat(quantidade);  
}  
  
let meusSorvetes = fazerSorvete(3);  
console.log(meusSorvetes); // Mostrará no console  
'🍦🍦🍦'
```

“

Os **parâmetros** são as **variáveis** que escrevemos quando **definimos** a função.

Os **argumentos** são os **valores** que enviamos quando **executamos** a função.

”



# 3 | Escopo

“

O **escopo** se refere ao alcance que uma variável tem, ou seja, de onde podemos acessá-la.

Em Javascript, os escopos **são definidos** principalmente pelas **funções**.

”

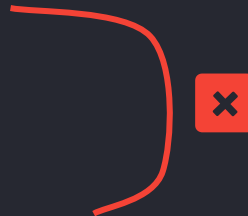


# Escopo **local**

O momento que nós declaramos uma variável **dentro** de uma função, **ela se torna local**. Ou seja, essa variável vive apenas **dentro** dessa função.

Se quisermos fazer uso da variável **fora** da função, não poderemos fazê-lo, dado que para **JavaScript** essa variável **não existe**.

```
function saudacao() {  
    // todo o código que escrevemos dentro  
    // da nossa função, tem escopo local  
}  
// Não conseguiremos acessar fora deste escopo
```





# {código}

```
function ola() {  
  let saudacao = 'Olá, tudo bem?';  
  return saudacao;  
}
```

**Definimos** a variável *saudacao* **dentro** da função *ola()*, por isso seu **escopo** é **local**.

Somente dentro desta função podemos ter acesso a ela.

```
console.log(saudacao);
```

# {código}

```
function ola() {  
    let saudacao = 'Olá, tudo bem?';  
    return saudacao;  
}
```

```
console.log(saudacao); // saudacao is not defined
```

Quando você quer fazer uso da variável de saudação fora da função, o Javascript não a encontra e retorna o seguinte erro:

**Uncaught ReferenceError:  
saudacao is not defined**

# Escopo **global**

O momento em que declaramos uma variável **fora** de qualquer função, a mesma passa a ter um **alcance global**.

Ou seja, podemos fazer uso dela de qualquer lugar no código em que estamos, e mesmo dentro de uma função, podemos acessar o seu valor.

```
{ }  
// todo o código que escrevemos fora  
// das funções é global  
function minhaFuncao() {  
    // Dentro das funções  
    // Temos acesso as variáveis globais  
}
```



# {código}

```
let saudacao = 'Olá, tudo bem?';
```

```
function ola() {  
    return saudacao;  
}
```

```
console.log(saudacao);
```

Declaramos a variável *saudacao* fora da nossa função, portanto o seu **escopo** é **global**.

Podemos acessá-la a partir de qualquer parte do código.

# {código}

```
let saudacao = 'Olá, tudo bem?';
```

```
function ola() {  
  return saudacao;  
}
```

Dentro da função *ola()* chamamos a **variável** *saudacao*.

Seu alcance é **global**, portanto o Javascript sabe a qual variável estou me referindo e executa a função com sucesso.

```
console.log(saudacao); // 'Olá, tudo bem?'
```

# Condicionais

IF / ELSE / ELSE IF

# Índice

1. Componentes de um if
2. Funcionamento de um if

# 1 | Componentes de um if



“

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 ser executado, caso seja verdadeira.

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

# Condicional com bloco **else**

Igual ao exemplo anterior, mas acrescenta um bloco de código a ser executado no caso da condição ser falsa.

É importante notar que o bloco `else` é opcional.

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

# Condicional com bloco **else if**

Igual ao exemplo anterior, só que adicionamos um **if** adicional. Ou seja, outra condição que pode ser avaliada no caso de a primeira ser falsa.

Podemos adicionar todos os blocos **else if** que quisermos, **mas só um pode ser verdadeiro**. Caso contrário, entrará em ação o outro bloco **else**, se existir.

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

1

# Funcionamento de um if

# {código}

```
let idade = 19;
```

```
let acesso = '';
```

```
if (idade < 16) {
```

```
    acesso = 'proibido';
```

```
} else if (idade >= 16 && idade <= 18) {
```

```
    acesso = 'permitido só acompanhado de um  
maior de idade';
```

```
} else {
```

```
    acesso = 'permitido';
```

```
}
```

# {código}

```
let idade = 19;  
let acesso = '';
```

Declaramos a variável **idade** e atribuímos o número 19.

```
if (idade < 16) {  
    acesso = 'proibido';  
} else if (idade >= 16 && idade <= 18) {  
    acesso = 'permitido só acompanhado de um  
maior de idade';  
} else {  
    acesso = 'permitido';  
}
```

# {código}

```
let idade = 19;
```

```
let acesso = '';
```

```
if (idade < 16) {  
    acesso = 'proibido';  
} else if (idade >= 16 && idade <= 18) {  
    acesso = 'permitido só acompanhado de um  
maior de idade';  
} else {  
    acesso = 'permitido';  
}
```

Declaramos a variável **acesso** e atribuímos uma string vazia, com a intenção de atribuir um novo valor segundo o resultado que atenderam uma das condicionais abaixo.



# {código}

```
let idade = 19;  
let acesso = '';
```

```
if (idade < 16) {  
    acesso = 'proibido';  
} else if (idade >= 16 && idade <= 18) {  
    acesso = 'permitido só acompanhado de um  
    maior de idade';  
} else {  
    acesso = 'permitido';  
}
```

Iniciamos com a condicional **if**.  
Nossa primeira condição avalia se  
a **idade** é menor que 16.

No caso de ser **verdadeira**, nós  
atribuímos a string '*proibido*' à  
variável **acesso**.

Neste caso, a **condição é falsa**,  
portanto o **Javascript** passa a  
**avaliar a próxima condição**.

# {código}

```
let idade = 19;  
let acesso = '';
```

```
if (idade < 16) {  
    acesso = 'proibido';  
} else if (idade >= 16 && idade <= 18) {  
    acesso = 'permitido só acompanhado de um  
maior de idade';  
} else {  
    acesso = 'permitido';  
}
```

Declaramos um bloco **else if** para contemplar uma **segunda condição**:

Esta condição vai ser composta e exigirá:

- Que a idade seja maior ou igual a 16;
- Que a idade seja menor ou igual a 18.

A condição novamente é **falsa**, portanto o Javascript continua lendo a condicional.

# {código}

```
let idade = 19;  
let acesso = '';
```

```
if (idade < 16) {  
    acesso = 'proibido';  
} else if (idade >= 16 && idade <= 18) {  
    acesso = 'permitido só acompanhado de um  
maior de idade';
```

```
    else {  
        acesso = 'permitido';  
    }  
}
```

Como **nenhuma** das condições anteriores **era verdadeira**, se executa o código dentro de else.

Portanto, agora a variável **acesso** é igual a string *'permitido'*.

“ É uma **boa prática** inicializar as variáveis com o **tipo de dado** que vão armazenar.



```
let texto = ''; // um texto vazio  
let numero = 0; // um número vazio
```

Dessa maneira, é mais claro para que eles serão usados.



# Arrays

“**Arrays** nos permitem gerar uma **coleção** de dados **ordenados**.”



# Estrutura de um **Array**

Utilizamos colchetes `[]` para indicar o início e fim de um **array**.

Usamos vírgulas `,` para separar os seus elementos.

Dentro, podemos armazenar a quantidade de elementos que queremos, independentemente do tipo de dados de cada um.

Ou seja, podemos ter, no mesmo **array**, dados do tipo string, numérico, booleano e todos os outros.

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

# Posições dentro de um **Array**

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

```
{}
```

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

0

1

2

Para acessar um elemento pontual de um array, chame o array pelo **nome** e, entre **colchetes**, escreva o **índice** ao qual você deseja acessar.

```
{}
```

```
filmesFavoritos[2];
```

```
// Acessamos o filme Baby Driver, índice 2 da array
```



# Métodos de Arrays I

“

**Arrays** para JavaScript são um tipo especial de objeto.

Por esta razão, temos uma série de **métodos** muito úteis para trabalhar com a informação interna. ”



# .push()

**Adiciona** um ou mais **elementos** ao **final** do array.

- Receber um ou mais elementos como parâmetros
- Retorna o novo comprimento do array

{}

```
var cores = ['Roxo', 'Laranja', 'Azul'];  
cores.push('Violeta'); // retorna 4  
console.log(cores);  
// ['Roxo', 'Laranja', 'Azul', 'Violeta']  
cores.push('Cinza', 'Ouro');  
console.log(cores);  
// ['Roxo', 'Laranja', 'Azul', 'Violeta', 'Cinza', 'Ouro']
```

# .pop()

**Elimina** o **último** elemento de um array.

- **Não** recebe parâmetro
- **Retorna** o elemento eliminado

{}

```
var series = ['Sobrenatural', 'Breaking Bad', 'The Soprano'];  
  
// criamos uma variável para salvar o que ela retorna .pop()  
var ultimaSerie = series.pop();  
  
console.log(series); // ['Sobrenatural', 'Breaking Bad']  
console.log(ultimaSerie); // ['The Soprano']
```

# .shift()

Elimina o **primeiro** elemento de uma array.

- **Não** recebe parâmetro
- **Retorna** o elemento eliminado

{}

```
var nomes = ['Frida','Diego','Sofía'];

// criamos uma variável para salvar o que ela retorna .shift()
var primeiroNome= nomes.shift();

console.log(nomes); // ['Diego', 'Sofia']
console.log(primeiroNome); // ['Frida']
```

# .unshift()

**Adiciona** um ou mais **elementos** ao **início** de um array.

- **Recebe** um ou mais elementos como parâmetro
- **Retorna** o novo comprimento do array

{}

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

# .join()

**Junta os elementos de uma array** usando o separador que especificamos. Se não o especificar, use vírgulas.

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

{}

```
var diasDaSemana = ['Segunda', 'Terça', 'Quarta', 'Quinta'];  
var separadosPorVirgula = diasDaSemana.join();  
console.log(separadosPorVirgula);  
// 'Segunda,Terça,Quarta,Quinta'  
var separadosPorTraco = diasDaSemana.join(' - ');  
console.log(separadosPorTraco);  
// 'Segunda - Terça - Quarta - Quinta'
```

# .indexOf()

**Procura** no array pelo **elemento** que **recebe** como parâmetro.

- **Recebe** um elemento como parâmetro que será buscado no array.
- **Retorna** a **primeira posição** onde encontra o que estava procurando. Se não o encontrar, retorna com -1.

{}

```
var frutas = ['Maça', 'Pera', 'Morango'];  
frutas.indexOf('Morango');  
// Ele encontrou o que procurava.  
// Retorna 2, o índice do elemento  
  
frutas.indexOf('Banana');  
// Ele não encontrou o que procurava. Retorno -1
```



# .lastIndexOf()

Similar ao **.indexOf()**, exceto que ele começa procurando pelo elemento no **final do array** (de trás para frente).

Se houver elementos repetidos, ele retorna a posição do primeiro que encontrar (ou seja, o último se olharmos desde o início).

```
{}
```

```
var frutas = ['Maça', 'Pera', 'Morango', 'Pera'];  
frutas.indexOf('Pera');  
  
// Retorna 1, onde a primeira ocorrência é encontrada.  
frutas.lastIndexOf('Pera');  
  
// Retorna 3, onde a primeira ocorrência é encontrada, porém,  
// olhando de trás para frente.
```

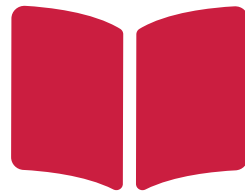
# .includes()

Também similar ao **.indexOf()**, mas retorna um booleano.

- **Recebe** um item para pesquisar no array
- **Retorna** verdadeiro se você encontrou o que estava procurando, falso se não.

{}

```
var frutas = ['Maça', 'Pera', 'Morango'];  
frutas.includes('Morango');  
// Ele encontrou o que procurava. Retorna true  
  
frutas.includes('Banana');  
// Ele não encontrou o que procurava. Retorna false
```



# Full Stack Node.js

## **JSON, mais condicionais e ciclos**

# Antes de começarmos... onde estamos!

Metodologias de Desenvolvimento

Express

Sequelize

**Introdução a Node.js**

Conceitos básicos de Javascript com Node.js, módulos nativos e módulos de terceiros.

Como funciona a web

Banco de Dados

# Antes de começarmos... onde estamos!

03

## Introdução a NodeJS

O que é node.js,  
gerenciadores de  
dependências e sistema  
de módulos

04

## Revisão de funções, condicionais e arrays

Tipos de dados,  
métodos, condicionais,  
variáveis e variáveis  
arrays

05

## JSON, mais condicionais e ciclos

JSON, objetos literais,  
arrow functions, if  
ternário e ciclos

06

## Callback, mais ciclos e novos métodos

Callbacks, for in e for of,  
destructuring, objeto  
Date e spread operator

# O que vimos no Playground

- Objeto Literal
- JSON
- Métodos de String
- For
- Arrow Function e If ternário



# O que vamos ver hoje

- Objeto Literal
- JSON
- For
- Arrow Function e If ternário

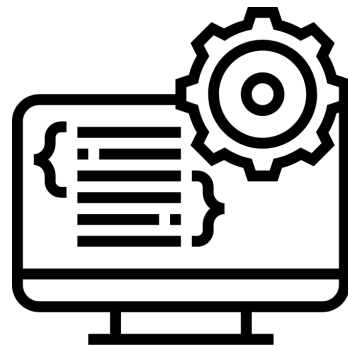


# OBJETOS LITERAIS



# Objetos Literais

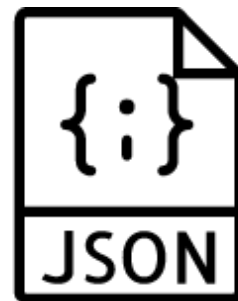
- É envolvido entre chaves { }
- Possui propriedades e seu valor { **propriedade : valor** }
- Possui métodos construtores (funções que acessam as propriedades e atribuem valor à tais)
- São instanciados (deixam de ser um molde para representarem algo real no sistema) **new NomeDoObjeto()**



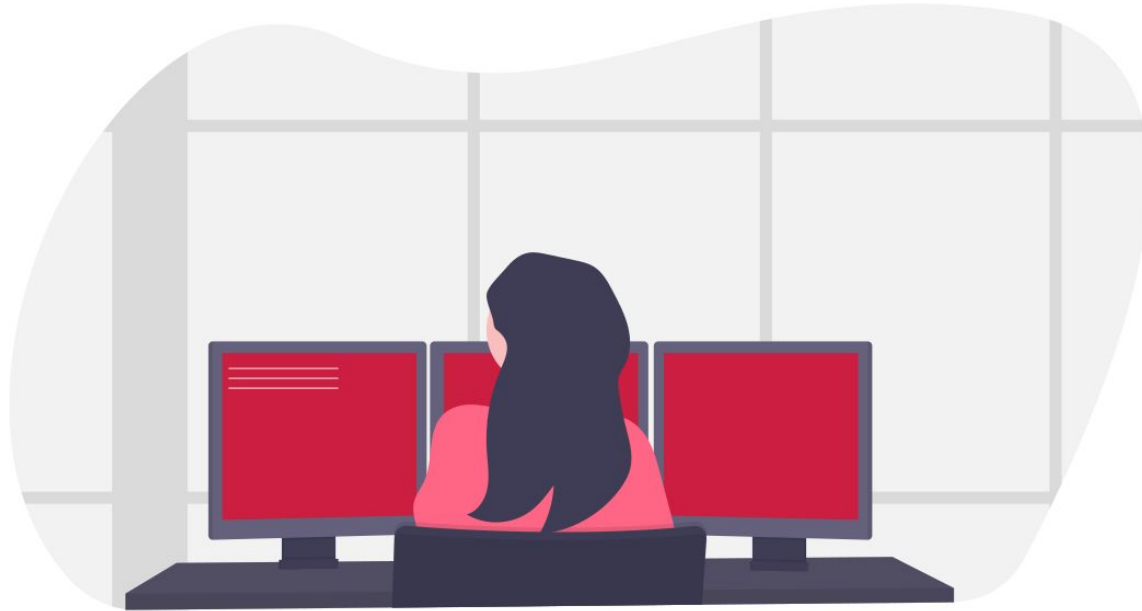
POR QUE COMPREENDER E  
SABER TRABALHAR COM **JSON** É  
TÃO IMPORTANTE?

# JSON

- É um tópico importantíssimo para compreensão, consumo e desenvolvimento de API's (aula que está porvir)
- Organização muito similar ao que estávamos acostumado (mas ainda sim são coisas diferentes)
- Possui dois métodos nativos para ajudar na transformação de dados:
  - `JSON.parse( textoAqui )`
  - `JSON.stringify( textoAqui )`



# Hora de praticar!



# Atividade - De objeto literal para JSON

- Modularizar o objeto criado por nós anteriormente
- Criar pasta chamada **database**
- Crie um arquivo chamado **catalogo.json**
- Passe as informações contidas no objeto para o arquivo
- Importe a base de dados de filmes para o arquivo **cinema.js**
- **Execute as funções do nosso mini-sistema verificando se o funcionamento de alguma delas foi prejudicado**

# OS **MÉTODOS DE STRINGS** E RESULTADOS OBTIDOS

# MÉTODOS

# RETORNO

**length()**

**(numérico)** - comprimento string

**indexOf()**

**(numérico)** - posição da string ou -1 caso não seja encontrada

**slice()**

**(array)** - lista contendo os elementos extraídos

# MÉTODOS

# RETORNO

**trim()**

**(string)** - remove espaços nas extremidades da string

**split()**

**(array)** - lista de strings divididas pelo separador indicado

**replace()**

**(string)** - nova string com os valores substituídos



# NOVA FORMA DE CRIAR UM LAÇO DE REPETIÇÃO: **FOR**

# Estrutura do loop:



for.js

```
for (início do contador; condição de repetição;  
atualização do contador ao fim de cada repetição) {  
    //código a ser executado a cada repetição  
}
```

# ATALHOS PARA ESTRUTURAS QUE JÁ CONHECEMOS: **ARROW FUNCTIONS E IF TERNÁRIO**

**Podemos usar IF TERNÁRIO e ARROW  
FUNCTIONS em qualquer contexto de código?**



# Hora de praticar!



# Atividade - Otimizando funções e condicionais

- Criar função **listarTodosOsFilmes** e **listarFilmesEmCartaz** (utilizando **for**)
- Alterar função **alterarStatusEmCartaz** fazendo adaptações necessárias para que o condicional implementado seja um **if ternário**

# JSON

“ É um formato de **texto simples** utilizado para troca de dados entre sistemas diferentes. ”



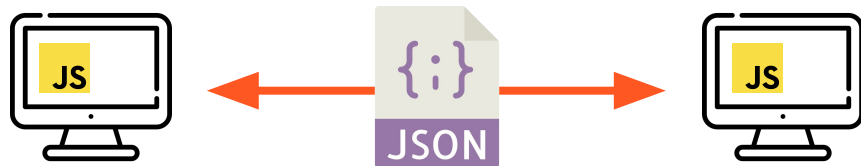


# A popularidade do JSON

Na web, a maioria das solicitações e suas respostas viajam como texto puro, ou seja, texto sem codificações especiais.

Como JSON é uma string simples, é um formato ideal para transmitir informações entre sites e aplicações web.

Especialmente considerando que o JavaScript está presente em todos os navegadores modernos.



# A popularidade do **JSON**

Uma outra vantagem do JSON é que qualquer linguagem de programação pode facilmente interpretá-lo. Na verdade, a maioria das linguagens web trabalham nativamente com JSON.



# Estrutura **JSON**

Como seu nome indica, o **JSON** é muito similar a um **objeto literal**. As diferenças entre eles são:

## **Objeto Literal**

- Admite aspas simples e duplas;
- As chaves do objeto ficam sem aspas;
- Podemos escrever métodos;
- Recomenda-se colocar uma vírgula na última propriedade.

## **JSON**

- Só podem ser utilizadas aspas duplas;
- As chaves vão entre aspas;
- Não suporta métodos, apenas propriedades e valores;
- Não podemos colocar uma vírgula no último elemento.

# Estrutura JSON

Como seu nome indica, o **JSON** é muito similar a um **objeto literal**. As diferenças entre eles são:

JS

JS

```
let o = {  
  texto: "Meu texto",  
  numero: 16,  
  array: [1,2,3],  
  booleano: true,  
  funcao: () => {return "olá"}  
}
```

VS.

{JSON}  
JavaScript Object Notation

JSON

```
{  
  "texto": "Meu texto",  
  "numero": 16,  
  "array": [1,2,3],  
  "booleano": true  
  /* JSON não suporta métodos */  
}
```

“

JavaScript nos fornece um **objeto JSON nativo** com dois métodos, que nos permitem converter o **formato** de um arquivo JSON para um objeto literal ou array, e vice-versa.

”



# Estrutura JSON

Converte um texto JSON para o tipo de dado JavaScript equivalente.

- Recebe uma string com formato JSON.
- Devolve o mesmo dado que recebeu em formato JavaScript.

JS

```
let dadosJson = '{"boate": "Vegas", "bairro": "Sé"}';  
let dadosConvertidos = JSON.parse(dadosJson);  
console.log(dadosConvertidos);  
// Um objeto literal será exibido no console  
// {  
//   boate: 'Vegas',  
//   bairro: 'Sé'  
// }
```

# Estrutura JSON

Converte um tipo de dado Javascript em um texto no formato JSON.

- Recebe um tipo de dado de Javascript.
- Devolve uma string com formato JSON.

```
let objetoLiteral = { nome: 'Carla', pais: 'Brasil' };  
let dadosConvertidos = JSON.stringify(objetoLiteral);
```

JS

```
console.log(dadosConvertidos);  
// Será visto no console os dados em uma string do tipo JSON  
// '{ "nome": "Carla", "pais": "Brasil" }'
```

“ Graças a estes dois métodos, poderemos gerar um **formato transaccional** de fácil compreensão **entre** diferentes **sistemas**. ”





# Métodos de Strings

“ Javascript nos oferece  
vários **métodos** e  
**propriedades** para  
trabalhar com **strings**. ”



# As strings em Javascript

Em muitos aspectos, para o Javascript, uma **string** não é nada mais do que um **array de caracteres**. Como em arrays, a primeira posição será sempre 0.

```
js let nome = 'Fran';
```

↑↑↑↑  
0123

Assim, podemos acessar o conteúdo da variável nome usando a mesma sintaxe dos arrays: variável [índice].

```
js nome[3]; // devolve 'n'
```

# .length

A propriedade `.length` retorna a **quantidade total** de caracteres na string, incluindo espaços.

Como é uma propriedade, não precisamos de parênteses quando a invocamos.

JS

```
let minhaSerie = 'Game of Thrones';  
minhaSerie.length; // retorna 15  
  
let arrayNomes = ['John', 'Sansa', 'Arya'];  
arrayNomes.length; // retorna 3
```

# Métodos de **strings**

**Dica!** Antes de conhecer os **métodos de strings**, é necessário entender um pouco sobre a definição de **função** e **método**, que veremos em profundidade mais tarde.



Uma **função** é um bloco de código que nos permite executar uma tarefa quantas vezes precisarmos. As funções podem receber dados para executar a tarefa e estes são conhecidos como **parâmetros**.

Quando uma função pertence a um objeto, nesse caso o nosso array, o chamamos de **método**.

# .indexOf()

Este método procura na string onde é aplicado, a string que recebe como parâmetro. Caso **o encontre**, retorna à primeira **posição** onde encontrou o elemento. No caso de não encontrá-lo, retorna **-1**.

JS

```
let saudacao = 'Olá! Estamos programando';  
saudacao.indexOf('Estamos'); // retorna 5  
saudacao.indexOf('vamos'); // retorna -1, não encontrou a palavra  
saudacao.indexOf('o'); // encontra a letra 'o' que está na posição 10,  
retorna 10 e pára a execução.
```

# .slice()

Este método **pega partes definidas de uma string e retorna** onde se aplica.

Recebe 2 números como parâmetros:

- O índice **de onde começa** o corte.
- O índice **para onde fazer o corte e é opcional**.

Ambos índices podem receber números negativos.

JS

```
let frase = 'Breaking Bad Rules!';  
frase.slice(9,12); // retorna 'Bad'  
frase.slice(13); // retorna 'Rules!'  
frase.slice(-10); // retorna 'Bad Rules!'
```

# .trim()

Este método **elimina os espaços** que estão no início e no fim de uma string. Se houver espaços no meio, não os remove.

**Não recebe parâmetros.**

JS

```
let nomeCompleto = '   Homer Simpson   ';\nnomeCompleto.trim(); // retorna 'Homer Simpson'\n\nlet nomeCompleto = '   Homer   Simpson   ';\nnomeCompleto.trim(); // retorna 'Homer   Simpson'
```



# .split()

Este método **divide uma string** em várias strings, utilizando a string que passamos como separador.

**Retorna um array** com as partes da string original.

```
JS let musica = 'And bingo was his name, oh! ';  
musica.split(' ');  
//retorna ['And', 'bingo', 'was', 'his', 'name,', 'oh!']
```

# .replace()

Este método **substitui uma parte de uma string** por outra.

Recebe os parâmetros:

- A string que queremos procurar;
- A string que vamos usar como substituto.

Retorna uma **string nova** com esta modificação.

```
JS let frase = 'Força, Python!'
   frase.replace('Python','JS') //retorna 'Força, JS!'
   frase.replace('Pyth','JS') // retorna 'Força, JSon!'
```

“

Ainda que **cada método** execute uma **ação muito simples**, quando **os juntamos**, podemos alcançar **resultados** muito **mais complexos e úteis**. ”

”



# Objetos Literais

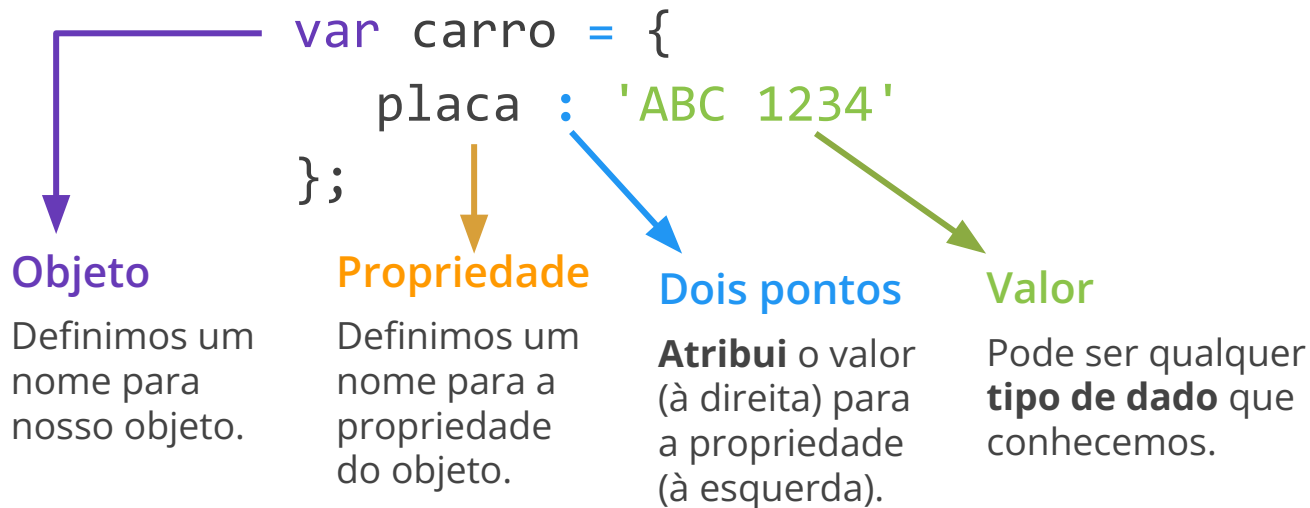
“ Podemos dizer que é a  
**representação** em  
**código** de um **elemento**  
da vida real. ”



# 1 | Estrutura do Objeto Literal

# Estrutura de um Objeto Literal

Um objeto é uma estrutura de dados que pode conter propriedades e métodos. Para criar um objeto, usamos chaves para abrir e para fechar { }.



# Estrutura de um **Objeto Literal**

Um objeto pode ter quantas propriedades desejarmos, se tiver mais de uma, separamos com vírgula.

Com a notação objeto.propriedade, nos dá o valor de cada um deles.

{}

```
var tenista = {  
  nome: 'Roger',  
  sobrenome: 'Federer'  
};  
  
console.log(tenista.nome) // Roger  
console.log(tenista.sobrenome) // Federer
```



# Método (ou Função) num Objeto Literal

Uma propriedade pode armazenar qualquer tipo de dado, até mesmo uma função. Se uma propriedade armazena uma função, dizemos que ela é um método desse objeto.

{}

```
let tenista = {  
  nome: 'Roger',  
  sobrenome: 'Federer',  
  cumprimentar: function(){  
    return 'Olá, Meu nome é Roger Federer';  
  }  
}
```

# Método (ou Função) num **Objeto Literal**

A palavra reservada **this** faz uma referência ao objeto em si. Com a notação **this.propriedade**, acessamos o valor de cada propriedade interna desse objeto.

{}

```
let tenista = {  
  nome: 'Roger',  
  sobrenome: 'Federer',  
  cumprimentar: function(){  
    return 'Olá, me chamo ' + this.nome;  
  }  
}  
  
console.log(tenista.cumprimentar()); // Olá, me chamo Roger
```

# Construtores de **Objetos**

Javascript nos dá uma opção de criarmos objetos utilizando uma função **construtora**. A função construtora nos permite montar um molde e criar todos os objetos que precisamos seguindo esse mesmo molde. O mais legal é que, como é uma função, ela pode receber parâmetros que podem ser usados para definir as propriedades de do objeto.


```
{}  
  
function Carro(marca, modelo){  
  this.marca = marca;  
  this.modelo = modelo;  
}
```

# Construtores de **Objetos**

## Objeto

Definimos um nome para a função, que será o nome do nosso objeto. Por convenção, iremos nomear apenas os objetos com a primeira letra maiúscula.

{ }




```
function Carro(marca, modelo){  
  this.marca = marca;  
  this.modelo = modelo;  
}
```

# Construtores de **Objetos**

## Parâmetros

Definimos os parâmetros que são necessários para a criação do nosso objeto.

{ }



```
function Carro(marca, modelo){  
  this.marca = marca;  
  this.modelo = modelo;  
}
```

# Construtores de **Objetos**

## Propriedades

Com a notação `this.propriedade`, definimos a propriedade do objeto que estamos criando neste momento. Normalmente, os valores das propriedades serão aqueles que vêm por parâmetros.

```
{}
```

```
function Carro(marca, modelo){  
    this.marca = marca;  
    this.modelo = modelo;  
}
```

# Instanciando **Objetos**

A função construtora *Carro()* espera dois parâmetros: *marca* e *modelo*. Para criar um objeto Carro, devemos usar a palavra reservada *new* e chamar a função passando os parâmetros que ela está esperando.

```
JS let meuCarro = new Carro('Ford', 'Fusion');
```

Quando executarmos o método *new* para criar um objeto, ele retorna uma **instância** do objeto Carro que será armazenada na variável *meuCarro*. Usando a mesma função, podemos instanciar quantos carros quisermos.

```
JS let seuCarro = new Carro('Honda', 'Civic');
```

# Arrow Functions



“

As funções serão muito utilizadas quando estivermos programando em Javascript.

As **arrow functions** nos permitem escrevê-las com uma **sintaxe** mais **compacta**. ”



# Índice

1. Declaração e estrutura
2. Exemplos

# 1 | Declaração e estrutura

# Estrutura básica

Vamos pensar em uma função simples que poderíamos programar de forma habitual, uma soma de dois números.

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

Agora, vejamos a versão reduzida da mesma função, transformando-a em uma arrow function.

```
{ } let somar = (a, b) => a + b;
```

# Nome de uma arrow function

As arrow functions são **sempre anônimas**, quer dizer, não possuem nome como as funções normais.

```
{ } (a, b) => a + b;
```

Se quisermos nomeá-las, é necessário escrevê-la como uma expressão de função, ou seja, atribuí-la como um valor a uma variável.

```
{ } let somar = (a, b) => a + b;
```

De agora em diante, podemos chamar nossa função pelo novo nome.

# Parâmetros de uma arrow function

Usamos parênteses para indicar os **parâmetros**. Se nossa função não recebe parâmetros, temos que escrevê-los mesmo assim.

```
{ } let somar = (a, b) => a + b;
```

Uma particularidade desse tipo de função é que, se ela recebe somente um **único parâmetro**, podemos omitir o uso dos parênteses.

```
{ } let dobro = a => a * 2;
```

# A seta de uma arrow function

Utilizamos a seta para indicar ao Javascript que vamos escrever uma função (substitui a palavra reservada `function`).

```
{ } let somar = (a, b) => a + b;
```

O que está a esquerda da flecha será a entrada da função (os parâmetros). O que está a direita, a saída (ou retorno).

“ As **arrow functions** tem esse nome por causa do operador `=>` . Se olharmos para ele com um pouco de imaginação, perceberemos que ele se parece com uma flecha.

Em inglês, chamamos de **fat arrow** (flecha gorda) para diferenciá-lo de outra flecha simples `->` ”





# Corpo de uma arrow function

Escrevemos a lógica da função. Se a função tem somente uma linha de código e esta mesma linha **retorna** um resultado, podemos omitir as chaves e a palavra *return*.

```
{ } let somar = (a, b) => a + b;
```

Caso contrário, vamos precisar usar ambos. Isso geralmente acontece quando temos mais de uma linha de código em nossa função.

```
{ } let eMultiplo = (a, b) => {  
    let resto = a % b; // Obtemos o resto da divisão  
    return a == 0; // Se o resto é 0, são múltiplos  
};
```

# 2 | Exemplos

# {código}

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

```
let dobroDe = numero => numero * 2;
```

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

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

Arrow function  
**sem parâmetros.**

Precisa dos  
parênteses para ser  
iniciada.

Ao ter somente uma  
linha de código, e esta  
mesma seja a que eu  
quero retornar, o  
return fica implícito.

# {código}

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

```
let dobroDe = 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 precisamos dos parênteses para indicá-lo) e com um return **implícito**.

# {código}

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

```
let dobroDe = 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 com um  
return **implícito.**

# {código}

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

```
let dobroDe = 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 precisa de mais de uma linha de código.

# As condicionais

IF TERNÁRIO / SWITCH

# Índice

1. O if ternário
2. O switch



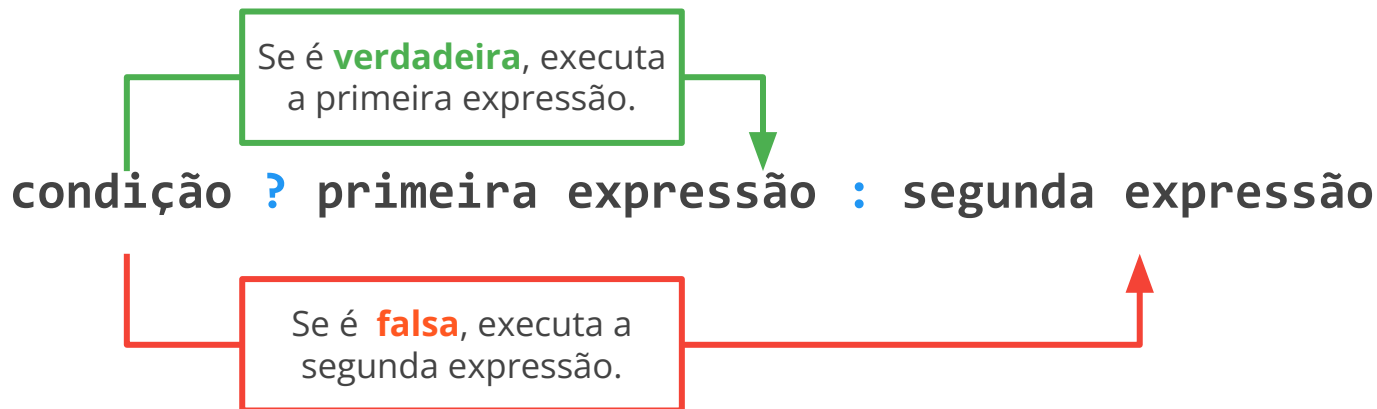
1 | 0 if ternario

“ Nos permite **avaliar condições** e realizar diferentes ações **segundo o resultado** dessas avaliações. ”



# Estrutura básica

Diferente de um **if** tradicional, o **if ternário** se escreve de forma **horizontal**. Ambas estruturas têm o mesmo fluxo interno (*se esta condição é verdadeira, faça isso, se não, faça esse outro*) mas, neste caso, não é necessário escrever a palavra **if** nem a palavra **else**.



# 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 ? '0 4 é maior' : '0 10 é maior';
```

## Condição

Declaramos uma expressão que é avaliada como true ou false.

## Primeira expressão

Se a condição for verdadeira, o código é executado após o ponto de interrogação.

## Segunda expressão

Se a condição for falsa, o código é executado depois dos dois pontos. É obrigatório escrevê-lo.

2 | 0 switch

“ O **switch** propõe uma sintaxe mais legível para os casos em que queremos avaliar muitas possibilidades de um único valor. ”



# Estrutura básica

O switch é composto por uma expressão a ser avaliada, seguida por diferentes casos (quantos 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 (expressão) {  
        case valorA:  
            // código a ser executado se a expressão é igual ao valorA  
            break;  
        case valorB:  
            // código a ser executado se a expressão é igual ao 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 (expressão) {  
        case valorA:  
        case valorB:  
            // código a executar se a expressão é igual ao valorA ou B  
            break;  
        case valorC:  
            //código a executar se o valorC for verdadeiro  
            break;  
    }  
}
```



# {código}

```
let idade = 5;
```

Definimos a variável idade e lhe atribuímos o número 5.

```
switch (idade) {  
  case 10:  
    console.log('Tem 10 anos');  
    break;  
  case 5:  
    console.log('Tem 5 anos');  
    break;  
}
```

# {código}

```
let idade = 5;
```

```
switch (idade) {  
  case 10:  
    console.log('Tem 10 anos');  
    break;  
  case 5:  
    console.log('Tem 5 anos');  
    break;  
}
```

Iniciamos a condicional com a palavra reservada **switch** e, entre parênteses, a expressão/condição que queremos avaliar.

Neste caso vamos avaliar **que valor tem a variável idade**.

# {código}

```
let idade = 5;
```

```
switch (idade) {
```

```
  case 10:
```

```
    console.log('Tem 10 anos');
```

```
    break;
```

```
  case 5:
```

```
    console.log('Tem 5 anos');
```

```
    break;
```

```
}
```

Para cada caso, escrevemos a palavra reservada **case** e depois o valor que queremos avaliar.

Neste caso, **perguntamos se** o valor da variável **idade** é 10.

Como este caso **NÃO é verdadeiro**, o Javascript ignora o código deste caso e passa a avaliar o caso seguinte.

# {código}

```
let idade = 5;
```

```
switch (idade) {  
  case 10:  
    console.log('Tem 10 anos');  
    break;  
  case 5:  
    console.log('Tem 5 anos');  
    break;  
}
```

Este caso **É verdadeiro**, portanto o **Javascript executa o código** que está associado: neste caso um `console.log()`.

A palavra reservada **break** corta as seguintes avaliações. Se esquecermos o `break`, os blocos ainda serão executados, independentemente do cumprimento ou não dos cases.

# O bloco **default**

Se quisermos reduzir a possibilidade de que nenhum dos dois casos seja verdadeiro, usamos a palavra reservada **default** seguida por dois pontos : e o bloco de código que queremos que seja executado.

Normalmente escrevemos o bloco default por último, neste caso, não é necessário escrever o break.

```
{}  
switch (expressão) {  
    case valorA:  
        // código a executar se valorA for verdadeiro  
        break;  
    default:  
        // código a executar se nenhum caso for verdadeiro  
}
```

# {código}

```
let fruta = 'wefwef';  
switch (fruta) {  
  case 'maça':  
    console.log('Que maçã deliciosa');  
    break;  
  case 'laranja':  
    console.log('Amo laranja!');  
    break;  
  default:  
    console.log('Que fruta é essa?');  
    break;  
}
```

Definimos a expressão que vamos avaliar no **switch**.

Neste caso, queremos perguntar pelo valor da variável **fruta**.

# {código}

```
let fruta = 'wefwef';
```

```
switch (fruta) {
```

```
  case 'maçã':
```

```
    console.log('Que maçã deliciosa');
```

```
    break;
```

```
  case 'laranja':
```

```
    console.log('Amo laranja!');
```

```
    break;
```

```
  default:
```

```
    console.log('Que fruta é essa?');
```

```
    break;
```

```
}
```

Este caso é **falso**, portanto não se executa seu código.

# {código}

```
let fruta = 'wefwef';  
switch (fruta) {  
  case 'maçã':  
    console.log('Que maçã deliciosa');  
    break;  
  
  case 'laranja':  
    console.log('Amo laranja!');  
    break;  
  
  default:  
    console.log('Que fruta é essa?');  
    break;  
}
```

Este caso também **é falso**, portanto, seu código não é executado.



# {código}

```
let fruta = 'wefwef';  
switch (fruta) {  
  case 'maçã':  
    console.log('Que maçã deliciosa');  
    break;  
  case 'laranja':  
    console.log('Amo laranja!');  
    break;  
  default:  
    console.log('Que fruta é essa?');  
}
```

Como nenhum caso foi verdadeiro, se executa o código dentro do bloco **default**.

# Os ciclos FOR

“ Os **ciclos** nos permitem **repetir instruções** de maneira simples. Podemos fazer isso uma determinada **quantidade de vezes** ou desde que se **cumpra** uma **condição**. ”



# Estrutura básica

A estrutura básica é composta por **3 partes** que definimos dentro dos parênteses. Em conjunto, essas partes nos permitem determinar de que maneira as **repetições** serão realizadas. Entre as chaves, definimos as **instruções** que queremos que sejam executadas em cada repetição.

```
{ } for (início; condição; modificador) {  
    //código que será executado em cada repetição  
}
```

# Estrutura básica

Neste exemplo, vamos contar de 1 até 5, inclusive.

```
{  
  for (let volta = 1; volta <= 5; volta++) {  
    console.log('Dando a volta ' + volta);  
  };  
}
```



```
Dando a volta 1  
Dando a volta 2  
Dando a volta 3  
Dando a volta 4  
Dando a volta 5
```

# Estrutura básica

```
{  
  for (let volta = 1; volta <= 5; volta++) {  
    console.log('Dando a volta ' + volta);  
  }  
};
```

## Início

Antes de iniciar o ciclo, estabelecemos o **valor inicial** do nosso contador.

# Estrutura básica

```
{  
  for (let volta = 1; volta <= 5; volta++) {  
    console.log('Dando a volta ' + volta);  
  }  
};
```

## Condição

Antes de executar o código em cada volta, se pergunta se a condição resulta como verdadeira ou falsa.

Se é **verdadeira**, continua com nossas instruções.

Se é **falsa**, interrompe o ciclo.

# Estrutura básica

```
for (let volta = 1; volta <= 5; volta++) {  
  console.log('Dando a volta ' + volta);  
};
```

## Modificador (incremento ou decremento)

Após executar nossas instruções, nosso contador é modificado da forma que especificamos. Neste caso, somamos 1, mas podemos adicionar a quantidade que quisermos.

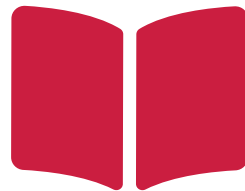


# O ciclo for em ação

Em cada ciclo, se verifica se o valor de **volta** é menor ou igual a 5. Nesse caso, se executa o **console.log()** e se aumenta o valor de **volta** em 1.

Quando a **volta** já não for menor ou igual a 5, o ciclo termina.

Iteração #	Valor da volta	Volta <= 5 ?	Executamos
1	1	true	✓
2	2	true	✓
3	3	true	✓
4	4	true	✓
5	5	true	✓
6	6	false	✗



# Full Stack Node.js

## Callback, mais ciclos e novos métodos

# Antes de começarmos... onde estamos!

Metodologias de Desenvolvimento

Express

Sequelize

**Introdução a Node.js**

Conceitos básicos de Javascript com Node.js, módulos nativos e módulos de terceiros.

Como funciona a web

Banco de Dados

# Antes de começarmos... onde estamos!

03

## Introdução a NodeJS

O que é node.js,  
gerenciadores de  
dependências e sistema  
de módulos

04

## Revisão de funções, condicionais e arrays

Tipos de dados,  
métodos, condicionais,  
variáveis e variáveis  
arrays

05

## JSON, mais condicionais e ciclos

JSON, objetos literais,  
arrow functions, if  
ternário e ciclos

06

## Callback, mais ciclos e novos métodos

Callbacks, for in e for of,  
destructuring, objeto  
Date e spread operator

# O que vimos no Playground

- Callbacks
- Métodos de Arrays II
- For in / For of
- Objeto Date
- Destructuring
- Spread Operator



# O que vamos ver hoje

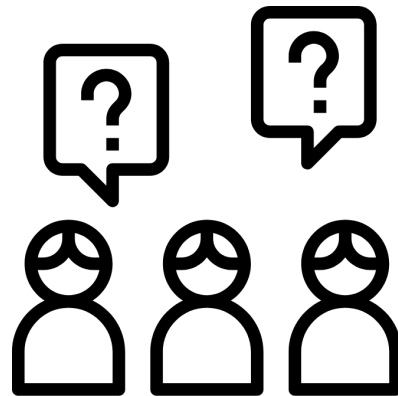
- Callbacks
- Novos métodos de arrays



QUAL A DIFERENÇA DE UMA  
**FUNÇÃO CALLBACK** PARA AS  
FUNÇÕES QUE CONHECEMOS?

# Uma função callback ...

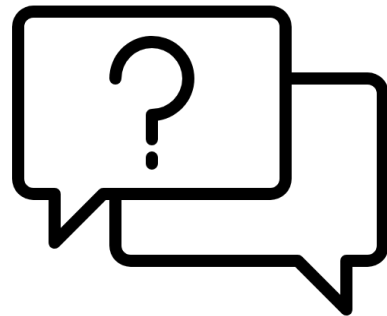
- ... é nativa da linguagem JS?  
(por exemplo, da mesma forma que **indexOf()** é nativa?)
- ... é obrigatoriamente chamada callback ?  
exemplo: **callback()**





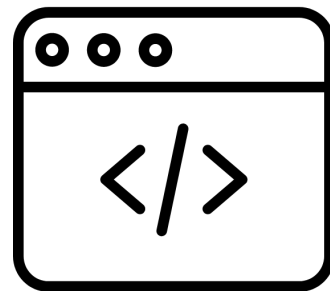
# Vamos realizar um exercício juntos?

- Como podemos aplicar esse uma função callback no projeto de cinema que estamos desenvolvendo?



# Vamos à prática!

- Abram o projeto e vamos observar a função `alterarStatusEmCartaz` que criamos na última aula.
- Ela faz uma busca na base de filmes com base no identificador que recebe como parâmetro, faz a busca e altera o status do filme encontrado. Porém já criamos uma função que se encarrega dessa parte inicial, a `buscarFilme`
- Vamos então utilizá-la como callback



NOVOS

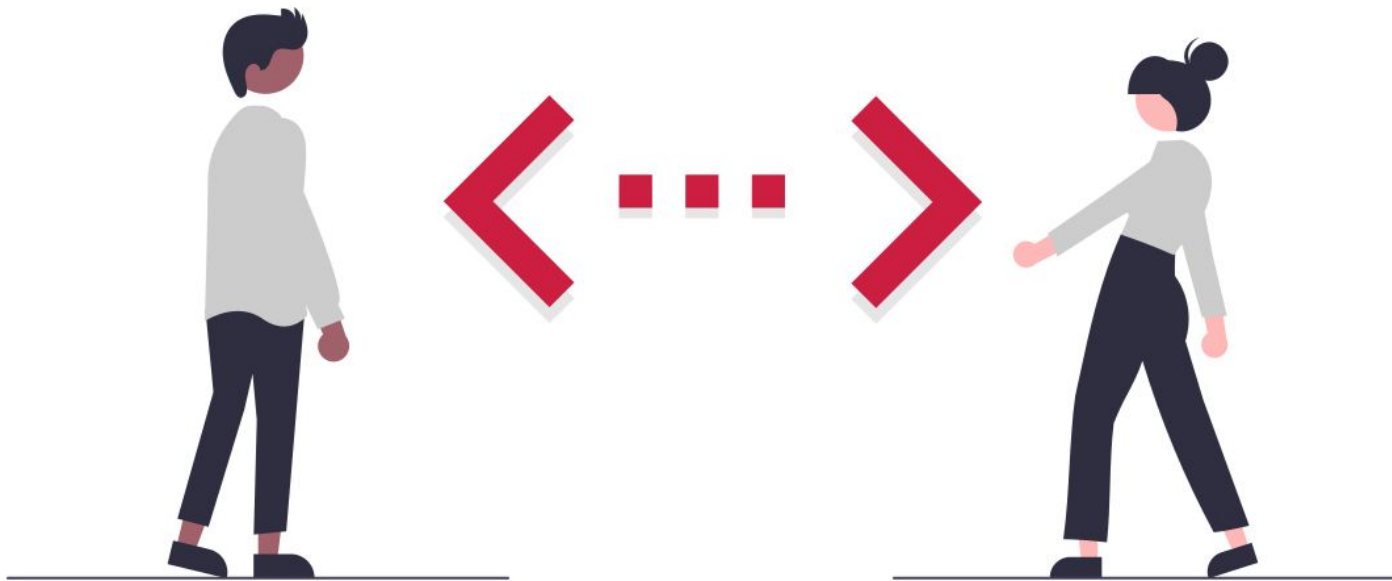
**MÉTODOS DE ARRAYS**

PARA FACILITAR O  
DESENVOLVIMENTO

# Você se lembra para que serve cada um?

<b>map()</b>	- novo array com novas modificações
<b>filter()</b>	- novo array com novas modificações baseadas no condicional
<b>reduce()</b>	- retorna valor acumulado do array percorrido
<b>forEach()</b>	- realiza uma ação programada por você no callback para cada item do array percorrido

# Hora de praticar!



# Atividade - Filtrando filmes

- Criar função **listarFilmesDeLongaDuracao** onde precisará filtrar e retornar para o usuário os filmes que possuem a partir de 2 horas.

PS: para esse exercício, insira alguns registros no catálogo de filmes que ultrapassem duas horas e outros de duração inferior.

# Atividade - Nova forma de percorrer uma lista

- Vamos alterar a função **listarTodosOsFilmes**, esta deve percorrer toda a lista de filmes armazenada no catálogo utilizando desta vez o método **forEach()** e retornar as informações de maneira amigável ao usuário.

# Callback



# Índice

O que é callback?

Callback com função anônima

Callback com função definida

“ **Callback** é uma **função** que se passa como **parâmetro** para outra **função**.

A função que recebe é que se encarrega de **executá-lo** quando for necessário. ”



# Tipos de **callback**

## Anônima

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

Como as funções anônimas não podem ser chamadas por seu nome, precisamos escrevê-la dentro da chamada da função callback.

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

# Tipos de callback

## Definida

A função que passamos como callback pode ser definida previamente. No momento em que a passarmos para outra função como parâmetro, nos referimos a ela pelo seu nome.

```
{}
```

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

Quando enviamos uma função como parâmetro, a escrevemos sem os parênteses, já que não queremos que se execute neste momento. Será a função que a recebe que se encarregará de executá-la.

# {código}

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

```
function saudar(nome, sobrenome, callback) {  
  return 'Olá ' + callback(nome, sobrenome) + '!';  
};
```

```
saudar('João', 'Neves', nomeCompleto);
```

# {código}

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

```
function saudar(nome, sobrenome, callback) {  
    return 'Olá ' + callback(nome, sobrenome) + '!';  
};
```

```
saudar('João', 'Neves', nomeCompleto);
```

Definimos a função **nomeCompleto()**.

A mesma se encarrega de unir o nome com o sobrenome, pondo um espaço entre eles.

Nos **retorna** uma **string**.

# {código}

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

```
function saudar(nome, sobrenome, callback) {  
  return 'Olá ' + callback(nome, sobrenome) + '!';  
};
```

```
saudar('João', 'Neves', nomeCompleto);
```

Definimos a função **saudar()**.

A mesma recebe um nome, um sobrenome e um **callback** como parâmetros.

Este último será função que vamos querer executar internamente.

# {código}

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

```
function saudar(nome, sobrenome, callback) {
```

```
    return 'Olá ' + callback(nome, sobrenome) + '!';
```

```
};
```

```
saudar('João', 'Neves', nomeCompleto);
```

O que queremos devolver é uma string completa.

Na primeira parte temos o return 'Olá'.

O restante da string virá do retorno do **callback** no momento em que ele seja executado.



# {código}

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

```
function saudar(nome, sobrenome, callback) {  
  return 'Olá ' + callback(nome, sobrenome)+'!';  
};
```

```
saudar('João', 'Neves', nomeCompleto);
```

Executamos a função **saudar**, passamos a ela como parâmetros um nome, um sobrenome e a função **nomeCompleto**.

**Primeiro** se executará o **callback**, que vai retornar o nome completo. Em seguida se executará a função **saudar** que irá retornar a saudação completa.

“

A função saudar só funciona se passarmos um callback à função nomeCompleto? **Não!**

Podemos passar a ela qualquer função que retorne uma string, já que na estrutura interna de saudar, definimos que ela opere com esse tipo de dados.

”



# {código}

```
function iniciais(nome, sobrenome) {  
    return nome[0] + sobrenome[0];  
};
```

```
function saudar(nome, sobrenome, callback) {  
    return 'Olá ' + callback(nome, sobrenome) + '!';  
};
```

```
saudar('João', 'Neves', nomeCompleto);
```

Poderíamos definir outra função que se encarregue de retornar as iniciais do nome e sobrenome de uma pessoa.

# {código}

```
function iniciais(nome, sobrenome) {  
    return nome[0] + sobrenome[0];  
};  
  
function saudar(nome, sobrenome, callback) {  
    return 'Olá ' + callback(nome, sobrenome) + '  
};
```

```
saudar('João', 'Neves', iniciais);
```

```
// Retornará 'Olá JN!'
```

Desta vez, quando executarmos a função **saudar**, passamos a ela a função iniciais como **callback**. Novamente se executará o callback que dessa vez retornará as iniciais do nome. Em seguida, a função saudar será executada que, por fim, retornará a saudação completa.

# Métodos de Arrays II

“ JavaScript nos fornece vários **métodos** para executar em arrays, dando-nos uma gama de ferramentas para trabalhar com eles. ”



# ALERTA DE SPOILER!

Antes de conhecer estes métodos, é necessário espiar um pouco a definição de callback.

Como já sabemos, as funções podem receber um ou mais parâmetros. No caso de recebê-los, eles podem ser, entre outros: um número, uma string, um booleano ou também... uma função!!

Quando um método ou função recebe uma função como parâmetro, essa função é conhecida como **callback**.



# Índice

`.map()`

`.find()`

`.filter()`

`.reduce()`

`.forEach()`



1 | .map()

# .map()

Este método recebe uma função como parâmetro (callback).

Percorre o array e **retorna** um **novo** array **modificado**.

As modificações serão aquelas que nós programamos em nossa função de retorno do callback.

```
{  
  array.map(function(elemento){  
    // definimos as modificações que queremos  
    // para aplicar em cada elemento da array  
  })  
}
```

# {código}

```
let numeros = [2, 4, 6];  
let dobroNumeros = numeros.map(function(num){  
    // Multiplicamos por 2 cada número  
    return num * 2;  
});  
  
console.log(dobroNumeros); // [4,8,12]
```

# {código}

```
let numeros = [2,4,6];
```

```
let dobroNumeros =  
numeros.map(function(num){  
    return num * 2;  
}));
```

```
console.log(dobroNumeros); // [4, 8, 12]
```

Definimos os números e armazenamos em um array com três números.

# {código}

```
var numeros = [2,4,6];
```

```
let dobroNumeros = numeros.map(function(num){  
    return num * 2;  
});
```

```
console.log(dobroNumeros); // [4, 8, 12]
```

Na variável **dobroNumeros**, vamos armazenar o array que o método map vai retornar para nós.

# {código}

```
let numeros = [2,4,6];
```

```
let dobroNumeros = numeros.map(function(num){  
    return num * 2;  
});
```

```
console.log(dobroNumeros); // [4, 8, 12]
```

Aplicamos o método do  
map ao array de números.

# {código}

```
let numeros = [2,4,6];
```

```
let dobroNumeros = numeros.map(function(num){
```

```
    return num * 2;
```

```
});
```

```
console.log(dobroNumeros); // [4, 8, 12]
```

Para o **map()** passamos uma função como **parâmetro (callback)**.

Esta função, por sua vez, recebe um parâmetro (pode ter o nome que quisermos).

O parâmetro representará cada elemento do nosso array, neste caso, um número.

# {código}

```
let numeros = [2,4,6];
```

```
let dobroNumeros = numeros.map(function(num){
```

```
    return num * 2;
```

```
});
```

```
console.log(dobroNumeros); // [4, 8, 12]
```

Nós definimos o comportamento interno que a função vai ter.

A função será executada 3 vezes: uma vez para cada elemento deste array e cada um será multiplicado por 2.



# {código}

```
let numeros = [2,4,6];
```

```
let dobroNumeros = numeros.map(function(num){  
    return num * 2;  
});
```

```
console.log(dobroNumeros); // [4, 8, 12]
```

Nós mostramos por console a variável **dobroNumeros**, que armazena um novo array com a mesma quantidade de elementos que o original, mas com valores **modificados**.

2 | .find()

# .find()

Este método também recebe uma função como **parâmetro**.

O **callback** precisa retornar um booleano (**true** ou **false**), para que o find saiba que encontrou o elemento adequado.

Retorna o valor do **primeiro elemento** do array buscado, caso não encontre nada, vai ser retornado **undefined**.

```
{}  
array.find(function(elemento){  
    // nós definimos a condição que queremos usar  
    // como uma busca para encontrar o elemento no array  
});
```

# {código}

```
let frutas = ["Uva", "Maçã", "Cereja", "Morango", "Abacaxi"];
```

```
let moraNoMar = frutas.find(function(fruta){  
    return fruta == "Abacaxi";  
});
```

```
console.log(moraNoMar); // [Abacaxi]
```

# 3 | .filter()

# .filter()

Este método também recebe uma função como **parâmetro**.

Desloca o array e **filtra** os elementos de acordo com uma condição que existe no callback. Assim como no find, o **callback** passado para o filter também precisa retornar um booleano (**true** ou **false**).

Filter **retorna** um **novo array**, que contém apenas os elementos que atenderam a essa condição. Isto significa que o nosso novo array pode conter menos elementos do que o original.

```
{}  
array.filter(function(elemento){  
    // nós definimos a condição que queremos usar  
    // como um filtro para cada elemento do array  
});
```

# {código}

```
let idades = [22, 8, 17, 14, 30];  
let maiores = idades.filter(function(idade){  
    return idade > 18;  
});  
  
console.log(maiores); // [22, 30]
```

# {código}

```
let idades = [22, 8, 17, 14, 30];
```

```
let maiores = idades.filter(function(idade){  
    return idade > 18;  
});
```

```
console.log(maiores); // [22, 30]
```

Declaramos as idades variáveis e armazenamos em um array com cinco números.



# {código}

```
let idades = [22, 8, 17, 14, 30];
```

```
let maiores = idades.filter(function(idade){  
    return idade > 18;  
});
```

```
console.log(maiores); // [22, 30]
```

Na variável **maiores**, iremos armazenar em um novo array que o método **filter** retornará.

# {código}

```
let idades = [22, 8, 17, 14, 30];
```

```
let maiores = idades.filter(function(idade){  
    return idade > 18;  
});
```

```
console.log(maiores); // [22, 30]
```

Aplicamos o método **filter** ao array **idades**.

# {código}

```
let idades = [22, 8, 17, 14, 30];
```

```
let maiores = idades.filter(function(idade){  
    return idade > 18;  
});
```

```
console.log(maiores); // [22, 30]
```

Para o método **filter()** passamos uma função como parâmetro (**callback**).

Esta função, por sua vez, recebe um parâmetro (pode ter o nome que quisermos).

Representará todos os elementos da nossa array, neste caso, uma idade.

# {código}

```
var idades = [22, 8, 17, 14, 30];
```

```
var maiores = idades.filter(function(idade){
```

```
    return idade > 18;
```

```
});
```

```
console.log(maiores); // [22, 30]
```

Nós definimos o comportamento interno que esta função vai ter.

A função será executada 5 vezes: uma vez para cada elemento deste array, e os filtrará de acordo com a condição que nós definimos: que as idades são maiores que 18.

Isto significa que aqueles que não cumprirem a condição ( $idade > 18 == \text{falso}$ ), serão excluídos.

# {código}

```
var idades = [22, 8, 17, 14, 30];
```

```
var maiores = idades.filter(function(idade){  
    return idade > 18;  
});
```

```
console.log(maiores); // [22, 30]
```

Mostramos através do console da variável **maiores**, o array com os elementos que atendem a condição estabelecida.

“ É uma **boa prática** usar **nomes** que façam **sentido** para as nossas **variáveis**. Isto torna mais claro **para que servem**. ”



# 4 | .reduce()

# .reduce()

Este método percorre o array e retorna um **único** valor.

Recebe um callback que será executado em cada elemento do array. Ele, por sua vez, recebe dois parâmetros: um **acumulador** e o **elemento de corrente** pelo qual está passando.

```
{}
```

```
array.reduce(function(acumulador, elemento){  
  // definimos o comportamento que queremos  
  // para implementar no acumulador e no elemento  
});
```



# {código}

```
let numeros = [5, 7, 16];  
let soma = numeros.reduce(function(pilha, numero){  
    return pilha + numero;  
});  
  
console.log(soma); // 28
```

# {código}

```
let numeros = [5, 7, 16];
```

```
let soma = numeros.reduce(function(pilha, numero){  
    return pilha + numero;  
});
```

```
console.log(soma); // 28
```

Declaramos os números variáveis e atribuímos um array com três elementos.

# {código}

```
let numeros = [5, 7, 16];
```

```
let soma = numeros.reduce(function(pilha, numero){  
    return pilha + numero;  
});
```

```
console.log(soma); // 28
```

Na variável de soma nós armazenamos o que o método **reduce()** retorna ao aplicá-lo aos números do array.

# {código}

```
let numeros = [5, 7, 16];
```

```
let soma = numeros.reduce(function(pilha, numero){  
    return pilha + numero;  
});
```

```
console.log(soma); // 28
```

Nós aplicamos o método **reduce()** ao array de números.

# {código}

```
let numeros = [5, 7, 16];
```

```
let soma = numeros.reduce(function(pilha, numero){  
    return pilha + numero;  
});
```

```
console.log(soma); // 28
```

Para o `reduce()` passamos uma função como parâmetro (callback).

Esta função recebe dois parâmetros (podem ter o nome que quisermos).

O primeiro representará o acumulador, o segundo corre nesse momento, neste caso um número.

# {código}

```
let numeros = [5, 7, 16];
```

```
let soma = numeros.reduce(function(pilha, numero){
```

```
    return pilha + numero;
```

```
});
```

```
console.log(soma); // 28
```

Nós definimos o comportamento interno da função. Neste caso, queremos retornar a **soma total** dos elementos.

O acumulador armazenará o resultado e para cada iteração adicionar o elemento atual.

# {código}

```
let numeros = [5, 7, 16];
```

```
let soma = numeros.reduce(function(pilha, numero){  
    return pilha + numero;  
});
```

```
console.log(soma); // 28
```

Nós mostramos por console a variável soma que tem a **soma total** dos números do array.

# 5 | .forEach()



# .forEach()

A finalidade deste método é iterar sobre um array.

Ele recebe um callback como parâmetro que, ao contrário dos métodos anteriores, **não retorna nada**.

```
{}
```

```
array.forEach(function(elemento){  
    // nós definimos o comportamento que queremos  
    // para implementar em cada elemento  
});
```

# {código}

```
let paises = ['Argentina', 'Brasil', 'Colombia'];  
paises.forEach(function(pais){  
    console.log(pais);  
});
```

# {código}

```
let paises = ['Argentina','Brasil','Colombia'];
```

```
paises.forEach(function(pais){  
    console.log(pais);  
});
```

Declaramos a variável **paises** e atribuímos um array com três elementos.

# {código}

```
let paises = ['Argentina', 'Brasil', 'Colombia'];
```

```
paises.forEach(function(pais){  
    console.log(pais);  
});
```

Aplico o método **forEach()** ao array de **paises**.

# {código}

```
let paises = ['Argentina', 'Brasil', 'Colombia'];
```

```
paises.forEach(function(pais){  
    console.log(pais);  
});
```

O método **forEach()** tem uma função como parâmetro (callback).

Esta função recebe um parâmetro que representará cada elemento do array, neste caso, cada país.

# {código}

```
let paises = ['Argentina', 'Brasil', 'Colombia'];
```

```
paises.forEach(function(pais){
```

```
    console.log(pais);
```

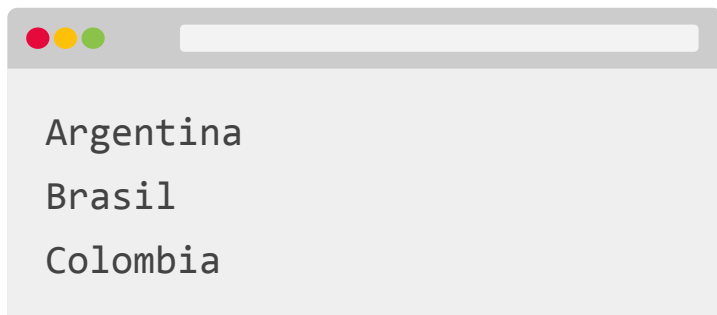
```
});
```

Definimos o comportamento interno da função. Neste caso, queremos mostrar cada país por console.

# {código}

```
let paises = ['Argentina', 'Brasil', 'Colombia'];  
paises.forEach(function(pais){  
    console.log(pais);  
});
```

O método `forEach()` neste caso irá imprimir todos os elementos do array por console.





# for/in for/of



“

Estas sentenças em Javascript nos permitem **repetir** elementos usando uma **sintaxe clara e simples.** ”



# Índice

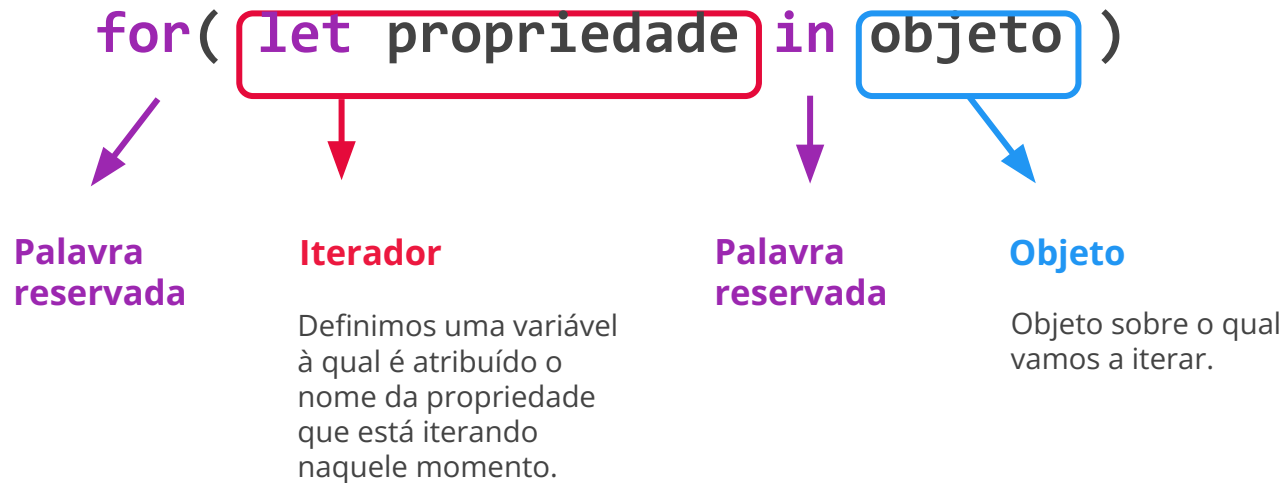
Ciclo: for in

Ciclo: for of

# 1 | for/in

# Estrutura do for/in

O loop **for ... in** nos permite **iterar** (lembrando que iterar é percorrer) em cada uma das **propriedades** de **um objeto**.



# {código}

```
var pessoa = {  
  nome: 'Guilherme',  
  idade: 23  
};
```

Declaramos a variável pessoa e armazenamos um objeto literal com o nome das propriedades e idade.

```
for (let dados in pessoa) {  
  console.log(dados, pessoa[dados]);  
};
```

# {código}

```
var pessoa = {  
  nome: 'Guilherme',  
  idade: 23  
};
```

```
for (let dados in pessoa) {  
  console.log(dados, pessoa[dados]);  
};
```

Declaramos a estrutura  
do laço for ... in.

# {código}

```
var pessoa = {  
  nome: 'Guilherme',  
  idade: 23  
};
```

```
for (let dados in pessoa) {  
  console.log(dado, pessoa[dado]);  
};
```

Declaramos a variável que representará cada propriedade do objeto durante a iteração.

O mesmo pode ser declarado usando **var** ou **let** e pode ter o nome que quisermos.

# {código}

```
var pessoa = {  
  nome: 'Guilherme',  
  idade: 23  
};
```

```
for (let dados in pessoa) {  
  console.log(dado, pessoa[dado]);  
};
```

Chamamos o objeto sobre o qual queremos iterar. Neste caso, **pessoa**.



# {código}

```
var pessoa = {  
  nome: 'Guilherme',  
  idade: 23  
};  
  
for (let dado in pessoa) {  
  console.log(dado, pessoa[dado]);  
};
```

Vamos lembrar que, para cada turno, na variável de dados é atribuído o **nome** da **propriedade** que está iterando naquele momento.

Usando o **nome** do objeto e **colchetes**, podemos acessar o **valor de cada propriedade**.

# {código}

```
var pessoa = {  
  nome: 'Guilherme',  
  idade: 23  
};
```

```
for (let dado in pessoa) {
```

```
  console.log(dado, pessoa[dado]);
```

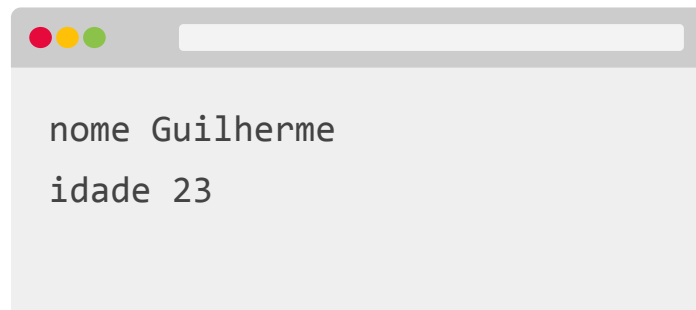
```
};
```

Mostramos no console o nome e o valor de cada propriedade.

# {código}

Assim você terá pelo console o nome e valor de cada propriedade:

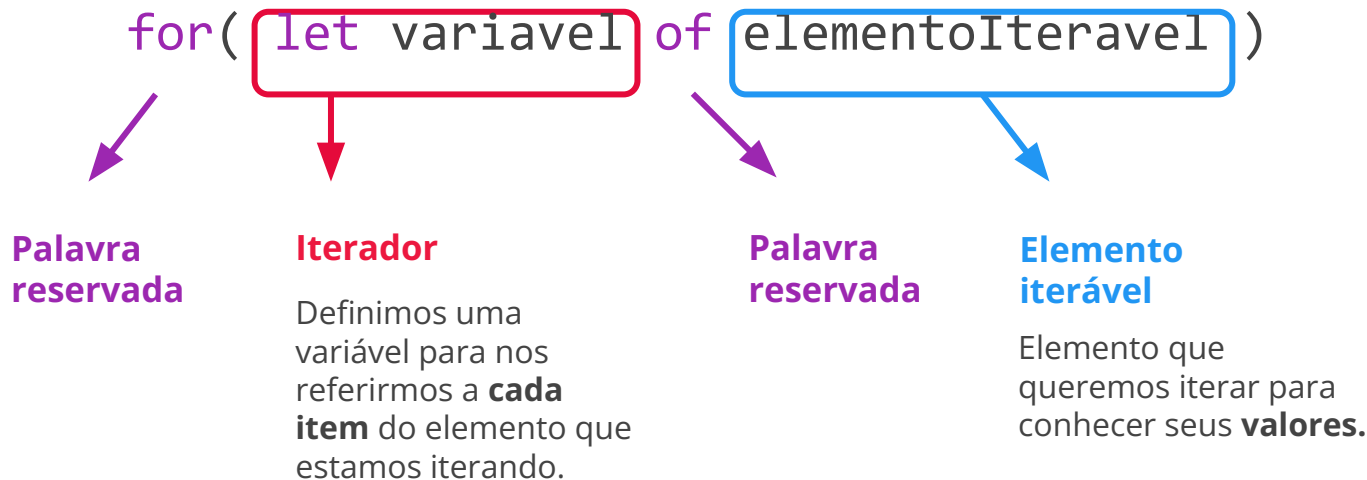
```
var pessoa = {  
  nome: 'Guilherme',  
  idade: 23  
};  
  
for (let dado in pessoa) {  
  console.log(dado, pessoa[dado]);  
};
```



# 2 | for/of

# Estrutura do for/of

O loop **for ... of** nos permite **iterar** sobre cada um dos **valores** de um elemento iterável, por exemplo, um array.



# {código}

```
var musicos = ['Vinicius','Tom','João'];
```

```
for (let musico of musicos) {  
    console.log(musico);  
};
```

Declaramos a variável `músicos` e armazenamos um array com 3 elementos.

# {código}

```
var musicos = ['Vinicius','Tom','João'];
```

```
for (let musico of musicos) {  
    console.log(musico);  
};
```

Declaramos a estrutura  
do **for ... of**.

# {código}

```
var musicos = ['Vinicius','Tom','João'];
```

```
for (let musico of musicos) {  
    console.log(musico);  
};
```

Declaramos a variável que vai representar cada elemento do array durante a iteração.

A mesma pode ser declarada usando var ou let e pode ter o nome que desejarmos.



# {código}

```
var musicos = ['Vinicius','Tom','João'];
```

```
for (let musico of musicos) {  
    console.log(musico);  
};
```

Chamamos o array **musicos** para iterar sobre ele.

# {código}

```
var musicos = ['Vinicius','Tom','João'];
```

```
for (let musico of musicos) {
```

```
    console.log(musico);
```

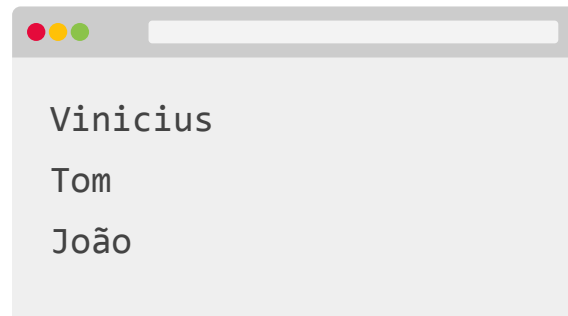
```
};
```

Imprimimos no console  
cada músico.

# {código}







Assim será impresso no console o valor de cada elemento do array.

```
var musicos = ['Vinicius','Tom','João'];  
  
for (let musico of musicos) {  
    console.log(musico);  
};
```



# Comparando **sentenças**

Para conseguir um objetivo pontual e escolher a melhor ferramenta para fazê-lo, é recomendável **sempre** pensar no **contexto** e nas **características** da ferramenta que pretendemos utilizar.

	FOR IN	FOR OF
Aplicar para	<b>Propriedades</b> enumeráveis	Elementos <b>iteráveis</b>
Uso com <b>objetos</b> ?		
Uso com <b>arrays</b> ?		
Uso com <b>strings</b> ?		

# Objeto Date

“

O Javascript, igual a  
muitas linguagens de  
programação, nos  
oferece um objeto para  
**gerar datas e  
trabalhar com elas.**

”



# Iniciando **Date**

A primeira coisa que temos que fazer para começar a trabalhar com o objeto **Date** é criar uma **instância** do mesmo.

Se não escrevermos argumentos, o objeto Date é criado com a hora e a data do momento.

```
let minhaData = new Date();
```

**Variável**

Criar uma variável para armazenar o objeto date.

**Palavra reservada**

**Método construtor**

Método que **devolve** um objeto date de JavaScript.

# .getDate()

Este método retorna o número do dia do mês de uma data.  
Devolverá um número entre 1 e 31.

```
{}  
  let diaDaMinhaData = minhaData.getDate();  
  console.log(diaDaMinhaData);  
  // Retorna o número do dia da data → ex: 22
```



# .getMonth()

Este método retorna o número do mês de uma data.  
Devolverá um número entre 0 (janeiro) e 11 (dezembro).

```
{}  
    let mesDaMinhaData = minhaData.getMonth();  
    console.log(mesDaMinhaData);  
    // Retorna o número do mês da minhaData → ex: 5 (Junho)
```

# .getDay()

Este método retorna o dia da semana de uma data.  
Devolverá um número entre 0 (Domingo) e 6 (Sábado).

```
{}
```

```
let diaSemanaDaMinhaData = minhaData.getDay();  
console.log(diaSemanaDaMinhaData);  
// Retorna o número do dia da semana da minhaData  
// → ex: 2 (Terça-feira)
```

# .getFullYear()

Este método retorna o ano completo (4 dígitos) de uma data.  
Devolverá um número entre 1000 e 9999.

```
{}  
  let anoDaMinhaData = minhaData.getFullYear();  
  console.log(anoDaMinhaData);  
  // Retorna o número do ano atual → ex: 2019
```

# Datas dinâmicas

O objeto Date nos permite criar uma determinada data. Quando instanciamos o nosso objecto, podemos lhe passar 3 parâmetros que representam, por ordem, o ano completo, o mês e o dia.

```
{ } let minhaDataDeAniversario = new Date(1995,11,22);
```

Agora, a variável **minhaDataDeAniversario** é um objeto do tipo **Date** com uma **data específica**, e podemos implementar os **métodos** vistos anteriormente.

```
{ } let minhaDataDeAniversario = new Date(1995,11,22);
```

# Date

## Documentação

A classe Date e seus métodos tem muito mais coisa para ser vista, que não cabem em um curso. [Acesse a documentação](#) e entenda mais sobre isso!

# Desestruturação

“

Nos permite **extrair**  
dados de **arrays** e  
**objetos literais** de uma  
maneira mais simples e  
fácil de implementar.

”



# Utilizando Desestruturação

Para extrair dados de um **array**, é necessário criar uma variável e atribuir-lhe um elemento do array usando o operador de **índice**.

```
{}  
let cores = ['Roxo', 'Azul', 'Amarelo'];  
let azul = cores[1];
```

Para extrair dados de um **objeto**, é necessário criar uma variável e atribuir-lhe uma **propriedade** específica para esse objeto.

```
{}  
let carro = {marca: 'Ford', ano: 1998};  
let marcaCarro = carro.marca;
```



# Desestruturando arrays

Para desestruturar um **array**, declaramos uma variável (podemos usar `var`, `let` ou `const`), e entre colchetes, escrevemos o nome que queremos. Podemos declarar mais de uma variável, separando cada uma com uma vírgula `,`.

Em seguida, adequamos essa estrutura ao array a partir do qual queremos extrair os dados.

```
{}
```

```
let cores = ['Roxo', 'Azul', 'Amarelo'];  
let [roxo, azul, amarelo] = cores;
```

# Desestruturando arrays

A partir de um **array** previamente definido, cada dado é transferido para as variáveis que definimos.

O Javascript irá atribuir para cada variável os dados extraídos da estrutura que escolhermos, **respeitando a ordem original**.

```
{  
  {}  
} let array = ['Roxo', 'Azul', 'Amarelo'];
```



```
{  
  {}  
} let [cor1, cor2, cor3] = array;
```



# Desestruturando arrays

Se quisermos pular um valor, podemos deixar vazio o nome da variável que corresponderia a essa posição.

```
{ } let array = ['Roxo', 'Azul', 'Amarelo'];
```

```
{ } let [cor1, , cor3] = array;
```

  
Espaço vazio

# Desestruturando objetos

Para desestruturar um **objeto literal**, criamos uma variável (podemos usar var, let ou const), e entre chaves, declaramos o nome ou nomes das propriedades que queremos extrair.

Igualamos esta estrutura ao objeto do qual queremos extrair os dados.

```
{}
```

```
let pessoa = {nome: 'Laura', idade: 31, faltas: 3};  
let {nome, idade} = pessoa;
```

# Desestruturando objetos

A partir de um **objeto** previamente definido, cada propriedade ou método é transferido para uma ou mais variáveis que definimos.

O Javascript irá atribuir para cada variável, **o valor da propriedade que escolhemos.**

```
{ } let pessoa = { nome: 'Laura', idade: 31, faltas: 3};
```



```
{ } let { nome, faltas } = pessoa;
```



# Desestruturando objetos

É possível que, em alguns casos, tenhamos que mudar o nome da variável que estamos criando.

Nesse caso, após extrairmos a propriedade que desejamos, colocamos dois pontos `:` seguidos do novo nome.

```
{ } let pessoa = { nome: 'Laura', idade: 31, faltas: 3 };
```

```
{ } let { nome, faltas: totalFaltas } = pessoa;
```

Novo nome

“

A desestruturação não modifica o **array** ou **objeto literal** de origem.

Seu único objetivo é copiar os **valores** de forma mais prática e rápida. ”



# Spread Operator

## Rest Parameter



# Índice

1. Spread Operator
2. Rest Parameter

# 1 | Spread Operator

“ Este operador permite **expandir** cada um dos dados de um **elemento iterável** dentro de outro elemento. ”



# Uso e sintaxe

O operador **spread** pode ser usado em qualquer elemento iterável.  
Permite-nos copiar e mover dados de um lugar para outro de forma eficiente.



# Spread em arrays

Implementando este operador, podemos **copiar** todos os dados de um array em um **novo array**.

{}

```
let clubesUm = ['Boca', 'Palmeiras', 'Barcelona'];  
let clubesDois = ['River', 'Santos', 'Inter Milan'];  
let todosOsClubes = [...clubesUm, ...clubesDois];
```

Nós podemos também **adicionar** todos os dados de um **array** em um **array existente**.

{}

```
let partes = ['aniversário', 'para'];  
let frase = ['Feliz', ...partes, 'você'];
```

# Spread em objetos

Implementando este operador, podemos **copiar** todas as propriedades de um **objeto** para outro **objeto existente**.

```
{}  
let carro = {marca:'Ferrari', kms:0, ano:2019};  
let pilotoUm = {nome:'Vettel', idade:32, ...carro};  
let pilotoDois = {nome:'Leclerc', idade:21, ...carro};
```

Tanto `pilotoUm` como `pilotoDois` agora tem todas as propriedades que definimos no objeto `carro` sem ter que defini-las em cada um deles manualmente.

# Spread em funções

Implementando este operador, podemos passar um array para uma **função** como **argumento**. O operador `...` expandirá os dados para que a função os tome como argumentos separados.

Para exemplificar, vamos usar o método do JavaScript `Math.min()`, que recebe N quantidade de argumentos e retorna o menor.

```
{ } let notas = [9.3, 8.5, 3.2, 7, 10];  
Math.min(...notas); // Devolve 3.2
```

# 2 | Rest Parameter



“

Usado como o último **parâmetro** de uma função, nos permite **capturar** cada um dos **argumentos** adicionais passados para essa função.

”



# O parâmetro rest

O parâmetro rest é escrito da mesma forma que o **operador spread** `...` .

A diferença é que ele é usado durante a definição da função e não durante a sua execução.

O parâmetro rest irá **gerar** um **array** com todos os argumentos **adicionais** passados para a função.

```
{}  
function minhaFuncao(param1, param2, ...outros) {  
    return outros;  
}  
minhaFuncao('a', 'b', 'c', 'd', 'e');  
// retornará ['c', 'd', 'e']
```

# 0 parâmetro rest

Implementando o rest parameter, podemos definir uma função que aceite qualquer número de argumentos.

```
{  
  function somar(...numeros) {  
    // Sabendo que os números agora são um array utilizamos  
    // o método reduce para obter a somatória  
    return numeros.reduce((acum, num) => acum += num);  
  }  
  
  somar(1, 4); // devolve 5  
  somar(13, 6, 8, 12, 23, 37); // devolve 99  
}
```

# 0 parâmetro rest

Como o **parâmetro rest** captura todos os argumentos restantes, **sempre deve ser o último parâmetro da função**, caso contrário, receberemos um erro.

```
{  
  function somar(...numeros, outroParametro) {  
    // Utilizamos o método reduce para obter a soma  
    return numeros.reduce((acum, num) => acum += num);  
  }  
}
```

SyntaxError: parameter after rest parameter

