

# Introdução

# 01

---

Entender conceitos fundamentais do JS e instalar todas as dependências

# O que é JavaScript?

- É uma linguagem de programação de alto nível;
- Originalmente chamada de LiveScript;
- Porém recebeu o nome de JavaScript por causa da grande fama de Java;
- JavaScript = JS = Vanilla JavaScript = ECMAScript;
- Criada para deixar as páginas web vivas;



# O que é JavaScript?

- Just-in-time compiled;
- Orientada a objetos;
- Criada em 1995;



# Onde JavaScript é utilizada?

- Interagir com HTML e CSS (DOM);
- Calcular, manipular e validar dados;
- Pode ser utilizada como linguagem server-side (Node.js);
- Linguagem base de grandes frameworks (React, Angular, Vue);



# Fazendo download de um editor

- <https://code.visualstudio.com/>



# Como executar o JS em um navegador

- Criar o arquivo index.html e adicionar JavaScript;



# Executando JS direto do console

- Digitar código JavaScript no console do Chrome



# Outra alternativa para rodar JS

- <https://jsfiddle.net/>





# Como eu aconselho você a seguir o curso

- Use a abordagem de: editor de texto + navegador;
- Salve todas as aulas em arquivos separados para consultar posteriormente;
- Cada aula crie um exemplo a mais com as suas ideias para praticar;



# Introdução

Conclusão da unidade

# 01

---

# Tipos de dados e Operadores

# 02

---

Conhecer os tipos de dados em JS e  
também os operadores da linguagem

# O que são tipos de dados?

- A classificação/categoria de um dado;
- Dados são por exemplo: 13, 'olá', True;
- **Os tipos de dados existentes no JavaScript são:**
- Number (Aritimético, Special Numbers);
- String;
- Boolean (Comparações, operadores lógicos);
- Empty Values (null, undefined);



# Numbers

- Obviamente este tipo trata de números;
- `console.log(typeof 13);`
- `console.log(typeof 1.8);`
- `console.log(typeof -5);`



# Numbers: aritmética

- Operação mais feita com os números em JS;
- E o resultado da operação aritmética produz um novo Number;
- `console.log(2 + 2);`
- `console.log(2 * 4 - 3);`
- `console.log(8 / 4);`
- Funciona com a mesma da matemática;
- `console.log(5 + (2 * 4));`



# Numbers: aritmética operadores

- + -> soma;
- - -> subtração;
- / -> divisão;
- \* -> multiplicação;
- % -> resto;



# Numbers: Special Numbers

- Considerados números, mas não funcionam como números;
- Infinity;
- -Infinity;
- NaN (Not A Number);





# Strings

- String = texto;
- `console.log(typeof 'Isso é uma String');`
- `console.log(typeof "Este texto aqui também");`
- `console.log(typeof `E este também`);` (template literals)



# Strings: detalhes mais técnicos

- A \ pode dar um 'escape' na String, e isso permite 'efeitos especiais';
- Por exemplo: \n pula uma linha
- `console.log("Essa é uma String \n De duas linhas");`
- Para inserir uma ' ou " devemos iniciar a String com a aspa inversa que desejamos inserir;
- O template literals serve para computar valores também, ex:
- `console.log(`A soma de 2 + 2 é ${2+2}`);`
- Concatenação é um processo de 'somar' Strings, veja:
- `console.log("salada " + "de" + " fruta");`

JS



# Booleans

- Serve para guardar um valor de uma comparação, por exemplo;
- Os únicos valores possíveis são:
- True (verdadeiro);
- False (falso);
- `console.log(5 > 2);`
- `console.log(3 > 10);`



# Booleans: comparações

- Maior que: >
- Menor que: <
- Maior ou igual: >=
- Menor ou igual: <=
- Igual: ==
- Diferente: !=
- Idêntico: ===



# Booleans: operadores lógicos

- Por meio de uma comparação resultam em um Boolean
- && - and -> para ser true, os dois 'lados' da comparação precisam ser true
- || - or -> para ser true, basta um dos 'lados' da comparação ser true;
- ! - not -> inverter os valores (true vira false);

JS



# Booleans: operadores lógicos

- && - and
- || - or
- ! - not

A	B	A AND B	A OR B	NOT A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False



# Booleans: operadores lógicos exemplos

- `console.log(true && true)`
- `console.log(true && false);`
- `console.log(false || false)`
- `console.log(!true);`



# Booleans: operador ternário

- Faz um comparativo em apenas uma linha de código;
- `console.log(true ? 1 : 2);`
- `console.log(false ? 'falso' : 'verdadeiro');`
- Não é muito utilizado e pode deixar o código complicado de ler;

JS





# Empty Values

- Temos duas palavras reservadas da linguagem que servem para estes casos:
- `undefined` e `null`;
- Sempre que você se deparar com estas palavras, o JS basicamente quer dizer que os valores não existem;

JS



# Conversão de tipo automática

- O JavaScript em determinadas operações, converte silenciosamente o tipo do resultado final da operação, veja:
- `console.log(5 * null) // 0`
- `console.log("5" - 3) // 2`
- `console.log("5" + 1) // 51`
- `console.log("dois" * "três"); // NaN`



# Tipos de dados e Operadores

Conclusão da unidade

# 02

---

# Seção de exercícios

# 03

---

Exercícios sobre tipos de dados

# Exercício 01

- Escreva três valores em string em um arquivo e exiba no navegador com o `console.log()`;
- Com aspas duplas, simples e template literals;



## Exercício 02

- Escreva três valores em number em um arquivo e exiba no navegador com o `console.log()`;
- Com números inteiros, números com casa decimal e por aritmética;



## Exercício 03

- Escreva três comparações com boolean;
- Uma com maior, menor ou igual e diferente;



## Exercício 04

- Escreva três comparações com operadores lógicos;
- Com and, or e not;





## Exercício 05

- Faça uma operação que emita NaN no console do navegador;



# Seção de exercícios

# 03

---

Conclusão da unidade

# Estruturas de Programação

## 04

---

Vamos começar a programar, utilizando as técnicas mais comuns como: estrutura de controle, repetição, funções e etc.

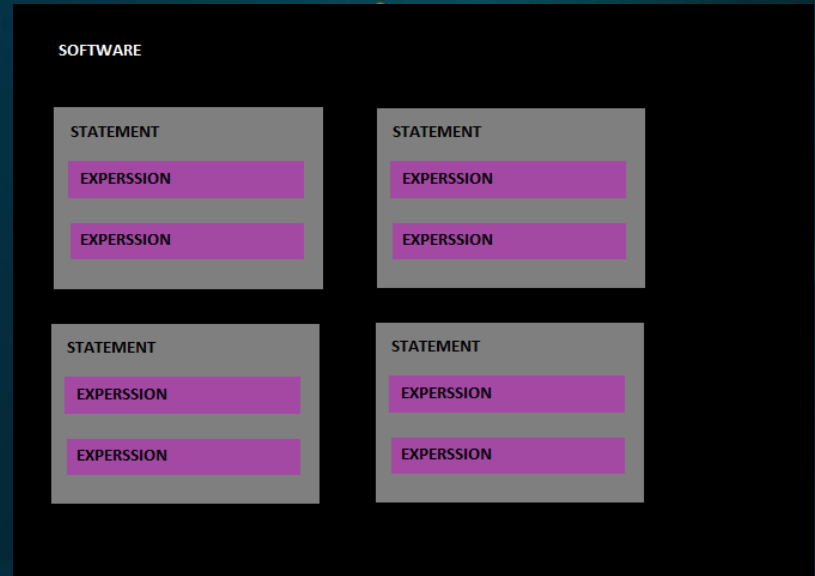
# O que é um programa/software?

- É um conjunto de declarações (statements);
- Statements são conjuntos de expressões (expressions);
- Expression é todo fragmento de código que produz um valor;



# O que é um programa/software?

- Software tem um 'objetivo';
- Statements 'guiam' o software para seu objetivo;
- Expressions são os valores que os statements esperam para guiar o software;



# O que é um programa/software?

- Nós já vimos statements e expressions de forma simples;
- Statement: `console.log(1 > 2);`
- Expression: `console.log('batata');`
- Porém ainda não é o suficiente para criar um software;
- E é essa junção dos conceitos nosso objetivo nesta seção! =)



# Como salvar valores na memória

- Salvamos os valores em variáveis (isso é um statement);
- `let laranjas = 5;`
- E depois podemos criar expressões com os valores salvos;
- `console.log(laranjas * laranjas);`
- `console.log(`Tem ${laranjas} na sexta`);`



# Como salvar valores na memória

- Podemos também mudar o valor que foi salvo anteriormente;
- poer
- `console.log(laranjas);`
- Um `let` pode definir várias variáveis também;
- `let quadrado = 4, triangulo = 3, retangulo = 4;`
- `console.log(quadrado * (triangulo + retangulo));`





# Outras maneiras de salvar valor

- Podemos definir valores com var e const;
- `var nome = 'Pedro';`
- `console.log(nome);`
- `const sobrenome = 'Soares';`
- `console.log(sobrenome);`



# Outras maneiras de salvar valor

- var é uma forma mais antiga, você deve optar por let ou const;
- const vem de constante, ou seja, o seu valor não pode ser alterado;
- Mais a frente no curso, veremos em detalhes diferenças de var, let e const;



# Convenção no nome das variáveis

- Não pode começar com um número (let 2teste);
- Mas pode terminar com número (let teste9 = 'testando');
- Pode ter \$ ou \_, mas não outros caracteres especiais (let \$nome, \_nome);
- Mas não pode ter pontuação ou outros especiais (let @teste)
- Podemos iniciar com letra maiúscula (let Nome = 'Matheus');
- Ou usar camelCase (let meuPrimeiroNome = 'Matheus');



# Nomes de variáveis reservados

- Alguns nomes não podem ser utilizados para iniciar variáveis, veja:
- break case catch class const continue debugger default delete do else  
enum export extends false finally for function if implements import  
interface in instanceof let new package private protected public  
return static super switch this throw true try typeof var void while  
with yield
- Ex: let break = 'parar';
- Porém podemos usar elas + alguma palavra/digito:
- Ex: let breakMatheus = 1;

# Como funciona o ambiente

- Quando qualquer programa é iniciado, um ambiente é criado;
- E este ambiente não inicia vazio
- Ele contém **funções** nativas da linguagem;
- **Funções** são blocos de código, que nos retornam um valor ou ação;
- O ambiente de JavaScript, neste curso, poderíamos entender como o navegador



# A estrutura de uma função

- Antes de mais nada: funções são chamadas durante o programa;
- Você também pode ouvir: executar, chamar, invocar uma função;
- E nós chamamos a função utilizando o nome dela + abrindo e fechando parênteses;

```
nomeDaFuncao();
```



# A estrutura de uma função

- Também podemos inserir parâmetros, em algumas funções são obrigatórios;
- Basicamente são valores que podem mudar o resultado da função;

```
nomeDaFuncao(parametro1, parametro2);
```



# Função built-in: prompt()

- Recebe um input do usuário do sistema e pode guardar este valor;
- Ex: `let idade = prompt('Qual sua idade?');`
- `console.log(idade);`
- Esta função é pouco utilizada.





# Função built-in: alert()

- Exibe uma mensagem na tela do usuário;
- Ex: `alert('Veja esta mensagem!');`
- Esta função é pouco utilizada.



# Função built-in: Math.x()

- Função utilizada para expressões/cálculos matemáticos;
- Ex: `let maiorNumero = Math.max(1,5,2,3);`
- `console.log(maiorNumero);`
- Esta função é muito utilizada.



# Função built-in: console.log()

- Exibe uma mensagem na tela, que é o argumento que passamos;
- Ex: `console.log('exibindo esta mensagem!');`
- Esta função é muito utilizada.
- Veremos como fazer as nossas funções mais a frente;

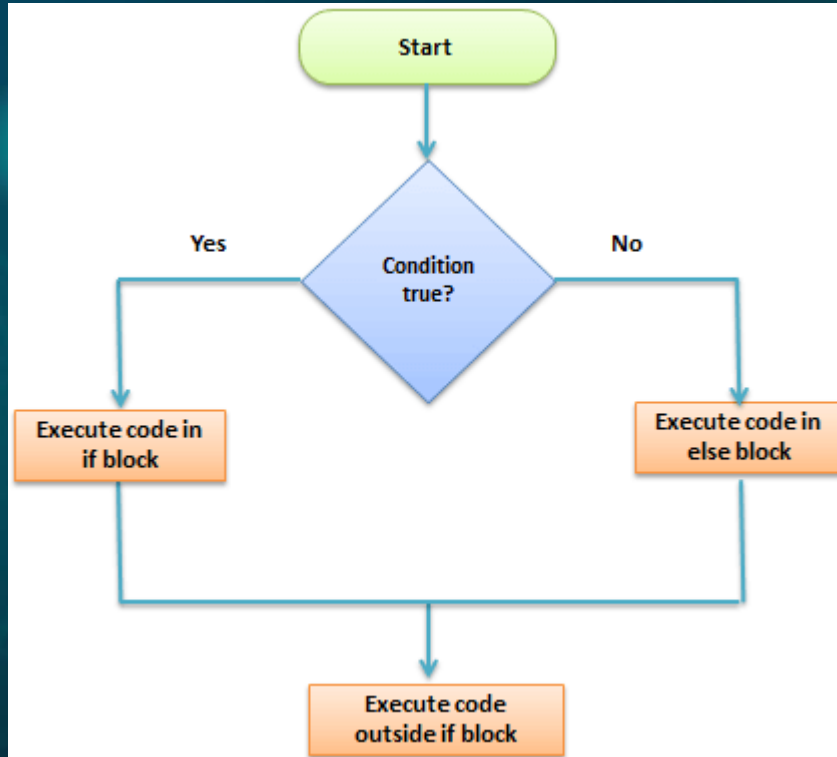


# O que são estruturas de controle?

- Um programa é executado de cima para baixo;
- Com as estruturas podemos modelar o fluxo do programa;
- Ou seja, dependendo dos valores de statements e expressions, ele tomará um caminho diferente;



# O que são estruturas de controle?



# Estrutura condicional: if

- O programa vai executar um bloco de código, SE algo acontecer;
- Onde algo é a condição imposta por um statement;
- Que resultar em um boolean (true or false);

```
let idade = 19;  
  
if(idade > 18) {  
    console.log('Pode entrar');  
}
```



# Estrutura condicional: else

- Podemos executar um outro bloco de código, caso a instrução do if não seja atendida;
- Ou seja, SE NÃO EXECUTAR o bloco if, EXECUTE o bloco else;

```
let nome = 'Pedro'  
  
if(nome == 'João') {  
    console.log('Seu nome é João');  
} else {  
    console.log('O seu nome não é João');  
}
```



# Estrutura condicional: else if

- Ainda podemos encadear mais condições com o else if;
- Ou seja, antes de executar um else, ou até mesmo sem ele, podemos verificar mais uma condição;

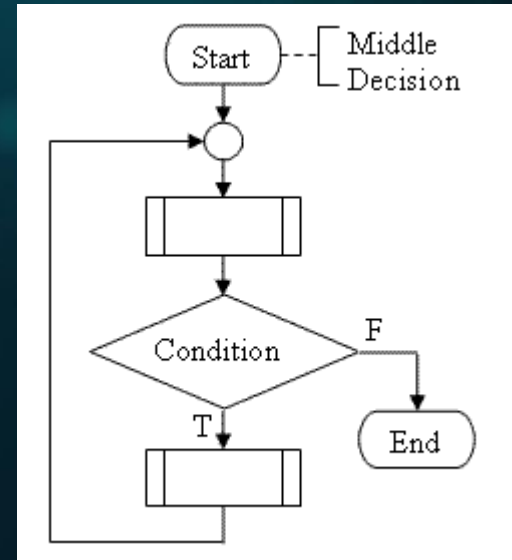
```
let a = 5;  
let b = 3  
  
if(a + b == 3) {  
    console.log('O resultado é 3');  
} else if(a == 4) {  
    console.log('O valor de a é 4');  
} else if(b == 3) {  
    console.log('O valor de b é 3')  
} else {  
    console.log('Nenhuma das condições acima!');  
}
```





# O que são estruturas de repetição

- A ideia é repetir uma ação até atingir uma condição;
- Ao invés de repetirmos o mesmo código várias vezes, criamos um statement que fará uma checagem em cada loop;
- Também chamada de loop;



# Estrutura de repetição: while

- Fará uma ação, até que a condição seja atingida
- Precisamos realmente 'definir um fim' para o loop, para não termos o problema de loop infinito;

```
let x = 10

while(x > 0) {
  console.log(x);
  x = x - 1;
}
```



# Estrutura de repetição: do while

- Semelhante ao while, porém a estrutura muda um pouco
- Este método quase não é utilizado!

```
let y = 0;  
  
do {  
  console.log(y);  
  y = y + 1;  
} while(y < 5);
```



# Estrutura de repetição: for

- Semelhante ao while, porém amplamente utilizado
- Na minha opinião: é a estrutura de repetição mais fácil de compreender, aconselho a preferir pelo for

```
for(let numero = 2; numero < 100; numero = numero * 2) {  
  console.log("O numero é: " + numero);  
}
```



# Precisamos falar sobre indentação

- Serve para organizar o código e deixar mais legível;
- Não há uma regra, porém cada bloco aninhado deve ser indentado uma vez;
- Pessoas vão te agradecer no futuro por indentar o código!
- Para a execução do código não afeta em nada.

```
let x = 2;
let y = 0;

if(x > 2) {
  y = x + 2;
  if(y == 4) {
    console.log('Y é 4');
  }
}
```



# Forçando a saída de um loop: break

- As vezes precisamos parar o loop antes que complete todo o seu ciclo;
- Para isso utilizamos o **break**;

```
for(let i = 5; i < 20; i = i + 1) {  
  if(i % 10 == 0) {  
    console.log('Saiu do loop');  
    break;  
  }  
  console.log('Prosseguindo o loop');  
}
```



# Pulando uma execução do loop: continue

- Dependendo da nossa lógica, podemos pular o resto da execução do loop;
- Para isso utilizamos a palavra **continue**;

```
let x = 0

while(x < 11) {
  x = x + 1;
  if(x % 2 == 0) {
    continue;
  }
  console.log(x);
  x = x + 1;
}
```



# Incrementando a variável: forma fácil

- Podemos incrementar uma variável de forma mais fácil: `x += 1;`
- Ou até mesmo `x++;`

```
for(let numero = 0; numero < 10; numero += 1) {  
    console.log("O numero é: " + numero);  
}
```





# Estrutura condicional: switch

- Quando há a necessidade de vários ifs, podemos utilizar o switch/case;
- Para sair de um case podemos utilizar o break;
- Podemos inserir uma expressão default, para caso nenhum valor for correspondido;
- **Obs:** muitos programadores optam por usar vários ifs!



# Estrutura condicional: switch

```
switch(nome) {  
  case "João":  
    console.log("O seu nome é João");  
    break;  
  case "Marcos":  
    console.log("O seu nome é Marcos");  
    break;  
  default:  
    console.log("O seu nome não é João nem Marcos!");  
    break;  
}
```

JS



# Um pouco mais sobre declaração de variáveis

- `let carrovermelhogrande;` (difícil de ler)
- `let carro_vermelho_grande;`
- `let CarroVermelhoGrande;`
- `let carroVermelhoGrande;` (mais utilizado)



# Comentários no JavaScript

- Utilizado para explicar o que acontece no código;
- Ou também para testar se algum código está afetando o sistema;
- Completamente ignorado pelo interpretador de código;
- Uma linha e multi-linha;

```
// Comentário de uma linha
```

```
/*  
    Comentário  
    de várias  
    linhas  
*/
```



# Estruturas de Programação

# 04

---

Conclusão da unidade

# Seção de exercícios

# 05

---

Exercícios sobre estruturas de fluxos, de  
repetição e tipos de dado

## Exercício 06

- Armazene em valores em variáveis com cada um dos tipos de dados vistos;
- String, Number e Boolean;



## Exercício 07

- Cria uma estrutura if que verifica a entrada na balada, se tiver mais de 18 anos pode entrar;
- Armazenar a idade em uma variável com let;
- Insira uma instrução console.log("Pode entrar"), caso a pessoa tenha mais que 18 anos;





## Exercício 08

- Crie uma constante com o seu nome como valor;
- Depois uma estrutura if que verifica se o seu nome é o que está na constante;
- Se estiver, emita uma mensagem de saudação com `console.log()`;

JS



## Exercício 09

- A função `Math.pow()` exibe a potencia de um número;
- Teste a função com `console.log()` e as seguintes bases: 2, 3, 18 e o expoente deve ser 2;
- Recebe 2 argumentos, base e expoente;
- Ex: `Math.pow(5,3);`



## Exercício 10

- Armazene a velocidade de um carro em uma variável, com o número que desejar;
- Faça uma estrutura if/else que verifica se ele está acima da velocidade;
- 80 é a velocidade máxima permitida;
- Se estiver acima ou abaixo exiba mensagens com console.log



# Exercício 11

- Faça uma estrutura if/else para verificar se um usuário pode dirigir;
- Armazene em variáveis algumas informações sobre o usuário: idade, se tem CNH
- Se a idade for maior que 18 e não possuir CNH, exiba uma mensagem;
- Se a idade for maior que 18 e tem CNH, exiba uma mensagem;
- Se não tiver 18 nem CNH, exiba outra mensagem;



## Exercício 12

- Escreva um loop while que exibe números de 0 a 10 no console;

JS



## Exercício 13

- Escreva um loop for que exibe números de 100 a 50 no console;

JS



## Exercício 14

- Escreva um loop for ou while que exiba qual número é par e qual número é ímpar,
- O contador deve iniciar em 0 e ir até 50;



## Exercício 15 - Desafio

- Verifique se o número é primo!
- Um número primo, é um número natural, maior que 1 e apenas divisível por si próprio e por 1;





# Seção de exercícios

# 05

---

Conclusão da unidade

# Funções

# 06

---

Nesta seção vamos aprender a criar nossas funções e também mais detalhes sobre este assunto.

# O que são funções?

- Funções são estruturas de códigos menores que são reaproveitadas durante a execução/construção de um programa;
- Principal objetivo: poupar repetição de código;
- Podem ser consideradas 'subprogramas';



# Definindo uma função

- Uma função tem uma estrutura um pouco mais complexa;
- Devemos declarar a função sempre com a palavra function;
- Uma função deve ter um nome;
- Pode conter argumentos/parâmetros, definidos entre ( );
- O corpo da função é definido entre { };
- Geralmente uma função retorna um valor;
- É possível declarar funções em variáveis;



# Definindo uma função

```
function escreverNoConsole() {  
    console.log("Escrevendo no console!");  
}  
  
escreverNoConsole();  
  
const textoNoConsole = function() {  
    console.log("Texto no console!");  
}  
  
textoNoConsole();  
  
const textPorParametro = function(texto) {  
    console.log(texto);  
}  
  
textPorParametro("Testando por parâmetro!");
```



# Mais sobre funções

```
const soma = function(a, b) {  
  return a + b;  
}  
  
console.log(soma(3, 5));  
  
const saudacao = function(nome) {  
  if(nome == "Matheus") {  
    return "Olá Matheus";  
  }  
}  
  
console.log(saudacao("Matheus"));
```



# Escopo de uma função

- O que acontece dentro de uma função fica separado do escopo global;
- O escopo global seria todo o arquivo de JavaScript;

```
let n = 10;

const numero = function() {
  let n = 25;
  console.log(n);
}

console.log(n);
```



# Mais sobre escopo

- Atualmente com **let** e **const**, qualquer bloco de código pode separar seu escopo (um if por exemplo);
- Isso é muito bom pois separa os contextos, com **var** isso não acontecia
- Ou seja, podemos ter escopos diferentes não só com funções;

```
let x = 10;

if(true) {
  let x = 20;
  console.log(x); // escopo if
}

console.log(x) // escopo global
```





# Escopo aninhado (Nested Scopes)

- Por causa da possibilidade de criar um escopo, podemos ter mais níveis de escopo;

```
let y = 5;

const multiplicar = function(n) {

  let y = n * 2;

  console.log(y) // escopo função

  if(y == 10) {

    let y = 55;

    console.log(y) // escopo if dentro da função

  }

}

multiplicar(y);

console.log(y) // escopo global
```



# Arrow Functions

- Uma outra forma de escrever funções;
- Bem utilizada nos frameworks modernos;
- Porém não deve substituir as functions por completo (veremos mais tarde os detalhes);

```
const parOuImpar = (n) => {  
  return n % 2;  
};  
  
console.log(parOuImpar(3));
```



# Mais sobre Arrow Functions

- Se só tem um parâmetro podemos remover os parênteses do argumento e o return;
- Se a expressão for pequena, pode até ser feita em uma linha sem prejudicar a leitura do código;

```
const raizQuadrada = (x) => {  
  return x * x;  
}  
  
const raizQuadrada2 = n => n * n;  
  
console.log(raizQuadrada(2));  
console.log(raizQuadrada2(4));
```



# Argumentos opcionais

- Podemos chamar uma função em JS sem o número igual de parâmetros determinados;

```
function nomeComIdade(nome, idade) {  
    if(idade === undefined) {  
        console.log("Seu nome é " + nome);  
    } else {  
        console.log("Seu nome é " + nome + " e você tem " + idade + " anos");  
    }  
}  
  
nomeComIdade("João");  
nomeComIdade("João", 42);
```

# Argumentos com valor default

- Podemos pre-determinar um valor para um argumento;

```
function repetirFrase(frase, n=2) {  
  for(let x = 1; x <= n; x++) {  
    console.log(frase + " " + x);  
  }  
}  
  
repetirFrase("Testando", 5);  
repetirFrase("Só duas vezes");
```



# Closure

- Uma função que se lembra do ambiente em que ela foi criada;

```
function armazenarSoma(x) {  
  return y => x + y;  
}
```

```
let soma1 = armazenarSoma(3);  
console.log(soma1(5)); // 8  
let soma2 = armazenarSoma(5);  
console.log(soma2(10)); // 15
```



# Recursion

- Uma funcionalidade que permite uma função se chamar novamente;
- Isso pode ser um problema caso a função se chame **muitas** vezes;

```
function retornarNumeroPar(n) {  
  if(n % 2 == 0) {  
    console.log("n agora é par: " + n);  
  } else {  
    console.log(n);  
    retornarNumeroPar(n - 1);  
  }  
}  
  
retornarNumeroPar(33);
```



# Funções 06

Conclusão da unidade



# Seção de exercícios

# 07

---

Exercícios sobre funções

# Exercício 01

- Escreva uma função que imprime “Hello World!” no console;



## Exercício 02

- Escreva uma função que recebe um parâmetro de idade;
- E imprima esta mensagem no console com template literals dizendo “Você tem x anos”;



## Exercício 03

- Escreva uma função que some dois números e retorne eles;
- Depois imprima este retorno;



## Exercício 04

- Escreva uma função que retorne um número aleatório;
- O número máximo retornado deve ser passado via parâmetro;
- Dica: utilize `Math.random()`;



## Exercício 05

- Escreva uma função que recebe a idade de uma pessoa;
- Se ela tem mais de 18 anos ela pode entrar na auto escola, imprima uma mensagem informando isso;
- Se ela tem menos, ela não pode, imprima outra mensagem com este aviso;
- Execute a função nos dois casos;

JS



## Exercício 06

- Escreva uma função que detecta o tipo de dado passado;
- Verifique se é um: number, boolean ou string
- E retorne uma mensagem para cada tipo;
- Execute uma função para cada caso;



## Exercício 07

- Escreva uma função que receba um número negativo e retorne um número positivo;
- Dica: utilize a função `Math.abs`





## Exercício 08

- Escreva uma função que recebe uma string;
- Se o texto conter mais de 10 caracteres imprima “Texto muito longo”;
- Se conter menos, imprima “Texto dentro do limite”;



## Exercício 09

- Escreva uma função que receba dois números, o primeiro é a base e o segundo a potência;
- Depois faça essa operação e retorne o resultado;
- Por exemplo:  $3, 2 = 9$



## Exercício 10

- Escreva uma função que recebe um número, e o decrementa de 1 em 1 com um loop;
- Além disso imprima somente os números pares no console;



# Seção de exercícios

# 07

---

Conclusão da unidade

# Objetos e Arrays

# 08

---

Vamos aprender sobre as estruturas de dados: objetos e arrays

# Arrays

- Possibilidade de adicionar um conjunto de valores a uma variável;
- O array deve ser escrito entre colchetes, separando os valores por vírgulas, veja:

```
let numeros = [1, 3, 5, 8, 12];  
  
let informacoes = ["Matheus", 12, true, "Teste", 2];
```



# Mais sobre Arrays

- Podemos acessar um array pelo seu índice, por exemplo: `arr[1]`
- Lembrando que o primeiro índice sempre é 0, ou seja, para acessar o primeiro elemento: `arr[0]`

```
let numeros = [1, 3, 5, 8, 12];  
  
let informacoes = ["Matheus", 12, true, "Teste", 2];
```



# Propriedades

- Propriedades são informações que podem ser verificadas de um valor;
- Quase todos os valores de JavaScript tem propriedades, menos null e undefined
- Podemos acessar as propriedades de duas maneiras:

```
let numeros = [1, 3, 5, 8, 12];  
  
console.log(numeros.length);  
console.log(numeros['length']);  
console.log(numeros[3]);  
  
let nome = 'Matheus';  
  
console.log(nome.length); // número de caracteres da string
```





# Métodos

- Métodos são propriedades que funcionam como funções;
- Strings e arrays contêm métodos;

```
let marca = 'Nike';  
  
console.log(typeof marca.toUpperCase);  
console.log(marca.toUpperCase());
```



# Objetos

- Uma coleção de propriedades, parecidos com arrays
- Podemos acessar estas propriedades

```
let pessoa = {  
  nome: "Matheus",  
  profissao: "Programador",  
  idade: 28,  
}  
  
console.log(pessoa.nome);  
console.log(pessoa.idade);
```



# Objetos: deletando e criando propriedades

- Podemos adicionar e deletar propriedades ao longo do nosso programa;

```
let carro = {  
  marca: "VW",  
  portas: 4,  
  eletrico: false,  
  motor: 1.0  
}  
  
console.log(carro.portas);  
  
delete carro.portas;  
  
console.log(carro.portas);  
  
carro.tetoSolar = true;  
  
console.log(carro.tetoSolar);
```

# Objetos: mais sobre objetos

- Podemos copiar todas as propriedades de um obj para outro;

```
let objetoA = {  
  prop1: 'teste',  
  prop2: 'testando',  
}  
  
let objetoB = {  
  prop3: 'propriedade'  
}  
  
Object.assign(objetoA, objetoB);  
  
console.log(objetoA);
```



# Objetos: mais sobre objetos

- E também podemos verificar quais as chaves cada objeto possui;

```
let objetoA = {  
  prop1: 'teste',  
  prop2: 'testando',  
}  
  
console.log(Object.keys(objetoA));
```



# Mutação (Mutability)

- Um objeto pode herdar todas as características do outro, virando uma referência ao mesmo;

```
let objetoA = {  
  pontos: 10  
};  
  
objetoB = objetoA;  
  
let objetoC = {  
  pontos: 10  
};  
  
console.log(objetoA == objetoB); // true  
console.log(objetoA == objetoC); // false
```



# Mutação (Mutability)

- Além disso a mutação cria uma ligação entre os objetos (binding), veja:

```
objetoA.pontos = 20;  
  
console.log(objetoB.pontos); // 20  
console.log(objetoC.pontos); // 10
```



# Mutação (Mutability)

- Além disso a mutação cria uma ligação entre os objetos (binding), veja:

```
objetoA.pontos = 20;  
  
console.log(objetoB.pontos); // 20  
console.log(objetoC.pontos); // 10
```





# Loops em Arrays

- Uma técnica muito utilizada na programação é o loop nos arrays;
- Como é uma lista com muitos valores, muitas vezes precisamos ver cada um deles;

```
let numeros = [1,5,10,15,20,25];  
  
for(let i = 0; i <= numeros.length; i++) {  
  console.log(numeros[i]);  
}
```



# Métodos de array: push e pop

- Métodos prontos e muito úteis que ajudam a manipular arrays;
- push: adiciona um ou mais elementos ao fim de um array;
- pop: remove um elemento no fim do array;

```
let numeros = [1,5,10,15,20,25];
```

```
numeros.push(30);
```

```
console.log(numeros);
```

```
numeros.push(35,40,45);
```

```
console.log(numeros);
```



# Métodos de array: push e pop

```
let pessoas = ["Matheus", "João", "Ricardo"];  
let pessoaRemovida = pessoas.pop();  
console.log(pessoas);  
console.log(pessoaRemovida);
```



# Métodos de array: shift e unshift

- shift: remove o primeiro elemento de um array;
- unshift : adiciona um ou mais elementos no início de um array;

```
let frutas = ["Maçã", "Melão", "Uva"];  
  
let frutaRemovida = frutas.shift();  
  
console.log(frutas);  
  
console.log(frutaRemovida);
```



# Métodos de array: shift e unshift

```
let nums = [1,2];  
  
nums.unshift(0);  
  
console.log(nums);  
  
arr.unshift(-2,-1);  
  
console.log(nums);
```



# Métodos de array: indexOf e lastIndexOf

- indexOf: encontra o índice de um determinado elemento;
- lastIndexOf: encontrar o último índice de um elemento;

```
let numeros = [0, 1, 2, 1, 0];  
  
console.log(numeros.indexOf(1));  
console.log(numeros.lastIndexOf(1));
```



# Métodos de array: slice

- Retorna um array a partir de outro array;
- O array retornado é determinado pelos parâmetros que são os índices de início e fim do novo array;

```
let numeros = [0,1,2,3,4,5];  
console.log(numeros.slice(2,3));  
console.log(numeros.slice(3));
```



# Métodos de array: forEach

- Itera cada elemento do array;

```
const nums = [1, 2, 3, 4, 5, 6];  
  
nums.forEach(num => {  
  console.log(num);  
});
```





# Métodos de array: includes

- Verifica se o array tem um determinado elemento;

```
let carros = ["BMW", "Fiat", "Audi"];  
console.log(carros.includes("Fiat"));
```



# Métodos de array: reverse

- Inverte um array;

```
let arr = [1,2,3,4,5,6,7,8,9,10];  
console.log(arr.reverse());
```



# Métodos de string: trim

- Remove tudo que não é string

```
let texto = ' \n teste \n ';  
console.log(texto.trim()); // teste
```



# Métodos de string: padStart

- Inserir caracteres antes da string;

```
let milAoContrario = '1';  
console.log(milAoContrario.padStart(4, "0"));
```



# Métodos de string: split

- Divide uma string por um separador, e retorna um array

```
let frase = "O rato roeu a roupa do rei de roma";  
let palavras = frase.split(" ");  
console.log(palavras);
```



# Métodos de string: join

- Junta elementos em um array em uma frase, por meio de um separador;

```
let fraseMontada = palavras.join('<->');  
console.log(fraseMontada);
```



# Métodos de string: repeat

- Repete uma string de acordo com um parâmetro number

```
console.log('teste'.repeat(5));
```



# Rest operator

- Uma forma de uma função receber indefinidos parâmetros;
- O operador rest vai virar um array;
- O parâmetro é definido por: ...nome

```
function imprimirNumeros(...args) {  
  for(let i = 0; i < args.length; i++) {  
    console.log(args[i]);  
  }  
}
```





# Destructuring com objetos

- Podemos definir variáveis com propriedades do objeto com uma notação diferente, chamada destructuring;

```
const person = {  
  name: 'Jhon',  
  lastname: 'Doe'  
}  
  
const {name: fname, lastname: lname} = person;  
  
console.log(fname);  
console.log(lname);
```



# Destructuring com array

- Podemos definir variáveis com os valores de um array utilizando o destructuring;

```
let nomes = ['Matheus', 'João', 'Pedro'];  
  
let [nomeA, nomeB, nomeC] = nomes;  
  
console.log(nomeA);  
console.log(nomeB);  
console.log(nomeC);
```



# JSON

- JSON = JavaScript Object Notation;
- Utilizado para comunicação entre serviços, ex: back end <-> front end
- Basicamente um tipo de dado padronizado, que lembra muito os objetos do JavaScript;

```
{  
  "name": "Matheus",  
  "age": 29,  
  "position": "Developer",  
  "languages": ["PHP", "JavaScript", "Python"]  
}
```



# Mais sobre JSON

- Rigoroso na estrutura, ou seja, você deve seguir o padrão imposto por este formato;
- Apenas aspas duplas;
- Não aceita comentários;

```
{  
  "name": "Matheus",  
  "age": 29,  
  "position": "Developer",  
  "languages": ["PHP", "JavaScript", "Python"]  
}
```



# JSON para String

- Podemos converter o JSON para uma string de forma fácil;
- Ou também uma string para JSON;

```
const car = {  
  "brand": "BMW",  
  "wheels": 4,  
  "doors": 2,  
  "type": "Sedan"  
}  
  
let jsonString = JSON.stringify(car);  
  
console.log(jsonString);  
  
let stringToJson = JSON.parse(jsonString);  
  
console.log(stringToJson);
```



# Objetos e Arrays

# 08

---

Conclusão da unidade

# Seção de exercícios

# 09

---

Exercícios sobre arrays e objetos

# Exercício 01

- Crie um array com 5 itens;
- Acesse o item 1,3,4;
- Obs: arrays começam no índice 0;





## Exercício 02

- Crie um array com 2 elementos e outro com 4;
- Imprima o número de elementos de cada um na tela;

JS



## Exercício 03

- Crie um objeto onibus;
- Com as propriedades rodas = 8;
- Limite de passageiros = 40;
- Portas = 2;
- Imprima todas as propriedades no console;



## Exercício 04

- Adicione a propriedade janelas no ônibus, com o valor de 20;
- Delete a propriedade rodas;
- Imprima a propriedade janelas no console;



## Exercício 05

- Crie um array com 5 nomes, incluindo o seu;
- Verifique se o seu nome existe no array;
- Se existir imprima alguma mensagem no console;



## Exercício 06

- Crie dois arrays, um com mais de 5 elementos e outro com menos;
- Faça uma função que verifica o número de elementos;
- Se possuir menos que 5, imprima “Poucos elementos” no console;
- Se tiver mais, imprima “Muitos elementos”;



## Exercício 07

- Crie um array com 5 elementos;
- Faça uma iteração entre todos eles e imprima no console o valor;



## Exercício 08

- Crie um JSON com 3 propriedades;
- Atribua ele a uma variável;
- Acesse as propriedades imprimindo no console;



## Exercício 09

- Crie um array a partir de uma frase;
- Imprima cada palavra do array no console por meio de um for;

JS





## Exercício 10 - Desafio calculadora

- Crie um objeto calculadora;
- Que tem os seguintes métodos: somar, subtrair, multiplicar e dividir;
- Os métodos só devem aceitar dois parâmetros;
- Utilize cada um dos métodos e imprima os valores no console;



# Seção de exercícios

# 09

---

Conclusão da unidade

# Conceitos de orientação a objetos

# 10

---

Vamos dar um passo a mais nos objetos,  
aprendendo conceitos do paradigma de  
orientação

# O que é orientação a objetos?

- Uma forma de programar, que utiliza os objetos como o seu principal princípio;
- Além de utilizar conceitos e técnicas que envolvem objetos;
- A maioria dos softwares, na parte de back-end, são desenvolvidos em cima desse paradigma;
- Grandes frameworks se aproveitam desta técnica: Laravel, Django e etc.



# Métodos

- Propriedades que servem como funções;
- Ou seja, as ações dos objetos;
- Invocamos os métodos da mesma maneira que funções;

```
let cachorro = {  
  latir: function() {  
    console.log("Au au");  
  }  
}  
  
cachorro.latir();
```



# Mais sobre Métodos

- Normalmente os métodos interagem com os objetos;
- Até mudando os valores das suas propriedades para corresponder a lógica do programa desenvolvido;

```
let pessoa = {  
  nome: '',  
  setNome: function(novoNome) {  
    this.nome = novoNome;  
  },  
  getNome: function() {  
    return this.nome;  
  }  
}  
  
pessoa.setNome("Matheus");  
  
console.log(pessoa.getNome());
```



# Prototypes

- Um objeto fallback de outro objeto;
- Quando um objeto recebe uma requisição de uma propriedade que não tem, ela é procura no prototype deste objeto;
- O prototype de um objeto criado do zero é o Object, que tem os métodos nativos da linguagem;

```
let pessoa = {  
  maos: 2  
}  
  
console.log(Object.getPrototypeOf(pessoa));  
console.log(Object.getPrototypeOf(pessoa) == Object.prototype);
```



# Mais sobre Prototypes

- Quando criamos um objeto a partir de um outro, o base será o prototype;
- Ele herdará tanto os métodos e propriedades de Object (o prototype do objeto base);
- Quanto os do objeto base para este novo;

```
let pessoa = {  
  maos: 2  
}  
  
let pessoaNova = Object.create(pessoa);  
  
console.log(pessoaNova.maos);  
console.log(Object.getPrototypeOf(pessoaNova) == pessoa);
```





# Classes

- O prototype do JavaScript pode ser chamado de classe;
- Pois nas outras linguagens uma Class é um molde de um objeto;
- Ou seja, podemos criar diversos objetos em cima de um prototype;

```
let cachorro = {  
  raca: 'SRD',  
}  
  
let pastorAlemão = Object.create(cachorro);  
pastorAlemão.raca = 'Pastor Alemão';  
console.log(pastorAlemão.raca);
```



# Classes: construtor por função

- Construtores são formas de instanciar uma classe em uma linguagem de programação;
- Instanciar = criar um objeto novo;
- No construtor já podemos definir propriedades;

```
function criarCachorro(raca) {  
  let cachorro = Object.create({});  
  cachorro.raca = raca;  
  return cachorro;  
}  
  
let doberman = criarCachorro('Doberman');  
  
console.log(doberman.raca);
```



# Classes: construtor por new

- Em muitas linguagens temos a possibilidade de instanciar um objeto com new, no JS também;

```
function Cachorro(raca) {  
  this.raca = raca;  
}  
  
let husky = new Cachorro('Husky');  
  
console.log(husky.raca);
```



# Classes: construtor com método

- Além de propriedades, podemos criar a classe base já com métodos;

- Basta definir no prototype o método desejado;

```
function Cachorro(raca) {  
  this.raca = raca;  
}  
  
Cachorro.prototype.uivar = function() {  
  console.log('Auuuuuuuu');  
}  
  
let pug = new Cachorro('Pug');  
  
console.log(pug.raca);  
pug.uivar();
```



# Construtor na classe (ES6)

- Com a versão do ES6, uma possibilidade de criar uma classe (objeto) com construtor foi adicionada;
- Então não precisamos mais criar por meio de uma função, veja:

```
class Cachorro {  
  constructor(raca) {  
    this.raca = raca;  
  }  
}  
  
let labrador = new Cachorro('Labrador');  
console.log(labrador.raca);
```



# Mais sobre classes

- Não podemos adicionar propriedades na classe, só via prototype;
- A classe só aceita métodos:

```
class Cachorro {  
  
    constructor(raca) {  
        this.raca = raca;  
    }  
  
    latir() {  
        console.log("Au au");  
    }  
  
}  
  
Cachorro.prototype.patas = 4;  
  
let poodle = new Cachorro('Poodle');
```



# Override nas propriedades do Prototype

- Sempre que adicionamos uma propriedade a um objeto, é criada uma idêntica no Prototype;
- Podemos substituir a do prototype;

```
class Cachorro {  
    constructor(raca) {  
        this.raca = raca;  
    }  
}  
  
let poodle = new Cachorro('Poodle');  
  
Cachorro.prototype.raca = 'SRD';  
  
console.log(poodle.raca);  
console.log(Cachorro.prototype.raca);
```



# Symbols

- Propriedades únicas, que não podem ser alteradas e nem criadas duas vezes;
- Podemos utilizar como uma constante, só que para propriedade de objeto;

```
class Cachorro {  
    constructor(raca) {  
        this.raca = raca;  
    }  
}  
  
let patas = Symbol();  
  
Cachorro.prototype[patas] = 4;  
  
let golden = new Cachorro('Golden Retriever');  
  
console.log(golden.prototype[patas]);
```





# Getters e setters

- Get: serve para resgatar o valor de uma propriedade;
- Set: serve para alterar o valor de uma propriedade;

```
class Cachorro {  
  constructor(raca) {  
    this.raca = raca;  
  }  
  
  get verRaca() {  
    return 'A raça é ' + this.raca;  
  }  
  set novaRaca(value) {  
    this.raca = value;  
  }  
}  
  
let golden = new Cachorro('Pastor Alemão');  
console.log(golden.verRaca);  
golden.novaRaca = 'Golden';  
console.log(golden.verRaca);
```



# Herança (inheritance)

- Uma classe pode herdar propriedades de outra classe por herança;
- Para isso utilizamos extends:

```
class Mamifero {  
  constructor(patas) {  
    this.patas = patas;  
  }  
}  
  
class Cachorro extends Mamifero {  
  constructor(patas, raca) {  
    super(patas, patas);  
    this.raca = raca;  
  }  
}  
  
let pug = new Cachorro(4, 'Pug');  
console.log(pug);
```



# Instanceof operator

- Podemos verificar quem é o pai do objeto utilizando o instanceof;

```
class Mamifero {  
  constructor(patas) {  
    this.patas = patas;  
  }  
}  
  
class Cachorro extends Mamifero {  
  constructor(patas, raca) {  
    super(patas, patas);  
    this.raca = raca;  
  }  
}  
  
console.log(new Cachorro instanceof Mamifero);
```



# Conceitos de orientação a objetos

# 10

---

Conclusão da unidade

# Seção de exercícios

# 11

---

Exercícios sobre conceitos de OO

# Exercício 01

- Crie uma classe que simula uma conta no banco (utilize a forma que preferir);
- Deve conter a propriedade saldo;
- E os métodos deposito e saque;
- Teste os métodos;



## Exercício 02

- Cria uma classe que simula um carrinho de compras de um e-commerce;
- Propriedades itens, quantidade total, valor total;
- Crie os métodos para adicionar e remover itens;



## Exercício 03

- Crie um objeto que simula um endereço de um cliente;
- Propriedades: Rua, Bairro, Cidade e Estado;
- No construtor o endereço já deve ser definido por completo;
- Crie métodos para atualizar todas as propriedades;





## Exercício 04

- Crie uma classe que simule um carro;
- Propriedades: marca, cor, gasolina restante;
- Crie um método de dirigir o carro, que vá diminuindo a gasolina gradativamente;
- E um de abastecer para aumentar a gasolina quando necessário;



## Exercício 05

- Crie uma classe conta bancaria;
- Com as propriedades de saldo na conta corrente, saldo na conta poupança e juros da poupança;
- Crie os métodos de depósito e saque, também um método para transferir dinheiro da poupança para a corrente;
- Além disso crie uma conta especial que herda da conta normal;
- Na conta especial os juros são dobrados da conta normal;



# Seção de exercícios

# 11

---

Conclusão da unidade

# Debugs e tratamento de erros

# 12

---

Vamos aprender a lidar com erros no JavaScript, tratar e também entender as mensagens de erro

# O que é bug e debug?

- **Bug:** Problema que ocorre no código, muitas vezes por erro do programador, que impede o funcionamento correto do mesmo;
- **Debug:** O ato de resolver os bugs encontrados no código ou também a forma que é feita a visualização de valores de variáveis ou fluxo do código;



# Strict mode

- Deixa o JavaScript mais rigoroso na hora de se programar;
- Deve ser declarado no topo de arquivos ou funções;
- Colocar o strict ajuda você a codificar de forma correta e não vai impedir/limitar nada no seu software ou programar;
- Veja o exemplo de uma variável declarada sem let/const/var:

```
"use strict"  
  
opa = 'teste';
```

```
✖ ▶ Uncaught ReferenceError: opa is not defined  
  at window.onload ((index):34)
```



# Método de debug: console.log

- Um método bastante utilizada para debug, é o console.log;
- Função que estávamos utilizando para mostrar os valores no console;

```
let a = 1;
let b = 2;

if(a == 1) {
    a = b + 2;
}

console.log(a);

for(i = 0; i < b; i++) {
    a = a + 2;
    console.log(a);
}

if(a > 10) {
    a == 25;
}

console.log(a);
```



# Método de debug: debugger

- Funcionalidade que para o código quando atingir a linha debugger;

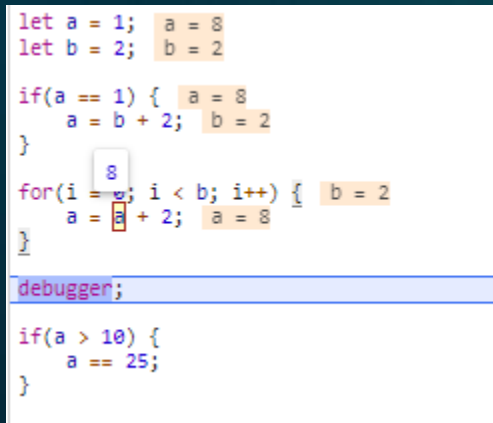
```
let a = 1;
let b = 2;

if(a == 1) {
    a = b + 2;
}

for(i = 0; i < b; i++) {
    a = a + 2;
}

debugger;

if(a > 10) {
    a == 25;
}
```



```
let a = 1;
let b = 2;

if(a == 1) {
    a = b + 2;
}

for(i = 0; i < b; i++) {
    a = a + 2;
}

debugger;

if(a > 10) {
    a == 25;
}
```





# Tratamento de input por função

- Não podemos controlar os dados que o usuário enviar, então para o bom funcionamento do software, devemos tratar eles;
- Veja um exemplo de tratamento de number:

```
function checarNumero(valor) {  
  let resultado = Number(valor);  
  if(Number.isNaN(resultado)) {  
    return null;  
  } else {  
    return resultado;  
  }  
}  
  
console.log(checarNumero(5));  
console.log(checarNumero('teste'));
```



# Exceptions

- Podemos criar erros no programa, caso alguma condição não seja atendida;
- Porém as exceptions abortam o programa, só geram o erro;

```
let a = 1;  
  
if(a != 2) {  
    throw new Error("O valor de a não pode ser 1");  
}
```

✖ ▶ Uncaught Error: O valor de a não pode ser 1  
at window.onload ((index):36)



# Try e catch

- O bloco try catch, vai tentar executar um código, caso não consiga ele pode retornar o erro que esse código gerou;
- Muito útil para debug;

```
try {  
  let c = a + b;  
} catch(error) {  
  console.log("Algo deu errado: " + error);  
}
```

```
Algo deu errado: ReferenceError: a is not defined
```



# Finally

- Finally é executada independente do resultado do try/catch;
- E pode existir com try e catch ou apenas try;

```
try {  
    let c = a + b;  
} catch(error) {  
    console.log("Algo deu errado: " + error);  
} finally {  
    console.log("Executou")  
}
```



# Assertions

- Verificações no programa, que são utilizadas para assegurar que tudo ocorra da forma esperada;

```
function iterarArray(arr) {  
  if(arr.length == 0) {  
    throw new Error("O array precisa ter elementos");  
  } else {  
    for(let i = 0; i < arr.length; i++) {  
      console.log(arr[i]);  
    }  
  }  
}
```



# Debugs e tratamento de erros

# 12

---

Conclusão da unidade

# Expressões regulares

# 13

---

Vamos aprender a criar expressões  
regulares com JavaScript

# O que são expressões regulares?

- Uma forma de encontrar padrões em uma string;
- Por exemplo: validar se só tem números;
- Em vez de criarmos funções complexas, podemos encontrar dados em texto por meio das expressões regulares;
- Também chamadas de regex;





# Criando uma expressão regular

- As expressões regulares no JS são um tipo de objeto;
- Podemos instancia-lo de duas formas;
- New RegExp e `/.../`;

```
let reg1 = new RegExp("test");  
let reg2 = /test/;
```



# Verificando padrões com regex

- Depois da definição do padrão por meio da regex;
- Utilizamos o método `test()` para verificar se o padrão é retornado;
- Com isso vamos receber de resposta `true` or `false`, veja:

```
console.log(/testando/.test("ttttestandooooo"));  
console.log(/testando/.test("asdtestasd"));|
```



# Conjunto de caracteres

- Podemos definir um conjunto de caracteres para encontrar por regex;
- Basta por entre [ ];
- Com um traço podemos definir um intervalo também, veja:

```
console.log(/[123]/.test("Existe 123 aqui?"));  
console.log(/[0-9]/.test("0 número 8 está presente aqui?"));
```



# Caracteres especiais

- \d - Qualquer dígito de caractere
- \w - Um caractere alfanumérico (“teste”)
- \s - Qualquer caractere de espaço em branco
- \D - Caracteres que não são dígitos
- \W - Caractere não-alfanumérico
- \S - Caractere que não seja espaço em branco
- . - Qualquer caractere, menos nova linha



# Regex com caracteres especiais

- Podemos utilizar os caracteres especiais de várias formas, veja:

```
let ano = /\d\d\d\d/;  
console.log(ano.test("05"));  
console.log(ano.test("2019"));  
console.log(ano.test("opa"));  
  
let palavraTresLetras = /\w\w\w/;  
console.log(palavraTresLetras.test("dia"));  
console.log(palavraTresLetras.test("ano"));  
console.log(palavraTresLetras.test("oi"));  
console.log(palavraTresLetras.test("teste"));
```



# Operador not ^

- Podemos escrever um set que aceitam tudo, menos alguns caracteres com o not;
- Lembrando que combinações serão aceitas, o negado é apenas se bate com o set;
- Mais adiante aprenderemos negar de outras formas;

```
let palavrasSemAeB = /^[^123]/;  
  
console.log(palavrasSemAeB.test("1112"));  
console.log(palavrasSemAeB.test("14"));  
console.log(palavrasSemAeB.test("1"));
```



# Operador plus +

- Quando um + está após alguma expressão, este elemento pode se repetir mais de uma vez;

```
let muitosOuPoucosDigitos = /\d+/;  
  
console.log(muitosOuPoucosDigitos.test("123"));  
console.log(muitosOuPoucosDigitos.test("123456789"));  
console.log(muitosOuPoucosDigitos.test(""));
```



# Operador ?

- Faz com que o dígito anterior seja opcional;

```
let opcional = /Prova\s?\d?/;  
  
console.log(opcional.test("Prova"));  
console.log(opcional.test("Prova 1"));  
console.log(opcional.test("Prova 2"));
```





# Ocorrência precisa

- Inserir o número de ocorrência entre { };

```
let telefone = /\d{4,5}-\d{4}/;  
  
console.log(telefone.test("4004-5050"));  
console.log(telefone.test("99999-8080"));  
console.log(telefone.test("999-999"));  
console.log(telefone.test("9999-9"));
```



# Método exec

- O método exec nos retorna um objeto com algumas informações sobre a regex;
- Se nada for encontrado, retorna null;

```
let teste = /\d+/.exec("0 número 100");  
  
console.log(teste);  
console.log(teste.index);
```



# Método match

- O método match funciona de forma similar ao exec;

```
let teste = "0 número 100".match(/\d+/);  
  
console.log(teste);  
console.log(teste.index);
```



# Choice pattern

- Podemos colocar uma instrução na regex que funciona como um || (or) das condicionais;

```
let frutas = /\d+ (bananas|maçãs|laranjas)/;  
  
console.log(frutas.test("10 bananas"));  
console.log(frutas.test("25 batatas"));  
console.log(frutas.test("8 laranjas"));
```



# Na prática: validando um domínio

- Da seguinte forma, podemos validar um domínio:

```
let validarDominio = /www.\w+\.\com|com.br/;  
  
console.log(validarDominio.test("www.google.com"));  
console.log(validarDominio.test("www.teste"));  
console.log(validarDominio.test("teste.com"));  
console.log(validarDominio.test("www.horadecodar.com.br"));
```



# Na prática: validando e-mail

- Da seguinte forma, podemos validar um e-mail:

```
let validarEmail = /\w+@\w+\.\w+;/;

console.log(validarEmail.test("teste@email.com"));
console.log(validarEmail.test("email@email"));
console.log(validarEmail.test("ronaldo@yahoo.com.br"));
console.log(validarEmail.test("email.com"));
```



# Na prática: validando data de nascimento

- Da seguinte forma, podemos validar uma data de nascimento:

```
let validarDataDeNasc = /^[0-9]{2}[/][0-9]{2}[/][0-9]{4}/;  
  
console.log(validarDataDeNasc.test('25/12/2015'));  
console.log(validarDataDeNasc.test('25/12/15'));  
console.log(validarDataDeNasc.test('2/2/2015'));  
console.log(validarDataDeNasc.test('30/02/1999'));
```



# Expressões regulares

# 13

---

Conclusão da unidade



# Seção de exercícios

# 14

---

Exercícios sobre expressões regulares

# Exercício 01

- Crie uma regex que aceite só letras maiúsculas;
- Depois teste;



## Exercício 02

- Crie uma regex que só aceite strings terminadas com ID;
- Depois teste;



## Exercício 03

- Crie uma regex que aceite a seguinte expressão “Marca: nomeDaMarca”;
- Onde nomeDaMarca pode variar para Nike, Adidas, Puma, Asics;
- Depois teste;



## Exercício 04

- Crie uma regex que valide endereços de IP;
- Ex: 127.0.0.1
- Depois teste;



## Exercício 05

- Crie uma regex que valide nome de usuários no sistema;
- Aceita letras de a-z, \_ e - , números de 0-9, mínimo de 3 caracteres e máximo de 16;
- Depois teste;



# Seção de exercícios

# 14

---

Conclusão da unidade

# Programação Assíncrona

# 15

---

Vamos aprender como programar de forma  
assíncrona no JS



# O que é programação assíncrona?

- Até agora programamos de uma maneira que uma ação acontecia após a outra;
- A programação assíncrona trabalha nesta questão, ações podem ser executadas ao tempo todo sem uma 'fila';
- **Um exemplo:** usuário está no checkout, manda salvar seu endereço na conta, mas pode prosseguir para a finalização sem recarregar a página, pois adicionar endereço ocorre de forma assíncrona;



# Callbacks

- Uma das vertentes a prog. Assíncrona é fazer ações que aconteçam depois de um tempo por meio de callbacks;
- Callback é uma função que faz uma ação após algum acontecimento no código;
- Podemos realizar um callback com a função setTimeout, veja:

```
console.log("Ainda não chamou o callback");

setTimeout(function() {
  console.log("Chamou o callback");
}, 2000);

console.log("Ainda não chamou o callback");
```



# Promises

- As promises são ações assíncronas que podem produzir um valor em algum momento no código;
- Uma forma de dizer a linguagem que um valor pode estar presente em um futuro do código;
- O objeto das promises é Promise, resolve é o método que resolve uma Promise, o then é o que faz ela poder ser executada em um ponto futuro;

```
let promessa = Promise.resolve(4 + 8);  
  
console.log('Algum código');  
  
promessa.then((value) => console.log(`A soma é ${value}`));
```



# Falha nas Promises

- Uma Promise pode falhar, podemos reter esse erro com um método chamado catch;
- Com ele podemos exibir o erro no console, por exemplo, e fazer o debug no código;

```
let promiseErrada = Promise.resolve(new Error("Algo deu errado"));

promiseErrada
  .then((value) => console.log(value))
  .catch(reason => console.log("Erro: " + reason));
```



# Rejeitando Promises

- Além do resolve, há o método reject;
- Que é quando determinada lógica não satisfaz nosso programa, então podemos ir para outra com o reject, em vez do resolve;
- Resolve e reject terminam a Promise, ou seja, não podemos chamar mais o then, por exemplo:

```
function verificarAlgo(num) {  
  return new Promise((resolve, reject) => {  
    if(num == 2) {  
      resolve(console.log("O número é 2"));  
    } else {  
      reject(new Error("Falhou"));  
    }  
  })  
}  
  
verificarAlgo(3);  
verificarAlgo(2);
```



# Resolvendo várias Promises

- Com o método `all`, podemos resolver várias promessas de uma vez só;
- Ou seja passamos elas por array e quando a última for resolvida, receberemos a resposta;

```
const p1 = new Promise(function(resolve, reject) {  
  setTimeout(function() {  
    resolve(100);  
  }, 2500);  
})  
  
const p2 = Promise.resolve(5);  
  
const p3 = new Promise(function(resolve, reject) {  
  resolve(10);  
});  
  
Promise.all([p1,p2,p3]).then((values) => console.log(values));
```



# Async functions

- Podemos criar funções assíncronas com a palavra reservada `async`;
- Elas retornam uma Promise;
- Se retornar algo, a promessa é resolvida, se der alguma exception a promessa é rejeitada;

```
async function somar(a,b) {  
    return a + b;  
}  
  
somar(2,2).then(function(value) {  
    console.log(value);  
});
```



# Await

- Nas async functions, podemos determinar uma instrução await;
- Que vai esperar uma promise ser resolvida, para apresentar os resultados;

```
function somaComDelay(a,b) {  
    return new Promise(resolve => {  
        setTimeout(function() {  
            resolve(a + b);  
        }, 2000);  
    });  
}  
  
async function soma(a,b,c,d) {  
    let x = somaComDelay(a,b);  
    let y = somaComDelay(c,d)  
  
    return await x + await y;  
}  
  
soma(2,4,6,8).then(v => console.log(v));
```





# Generators

- Generators funcionam semelhantes as Promises;
- Onde ações podem ser pausadas e continuadas depois;
- Caracterizados pelo function\* e yield, veja:
- O yield pode salvar o estado da variável;

```
function* genTest() {  
  let id = 0;  
  while(true) {  
    yield id++;  
  }  
}  
  
let criarId = genTest();  
  
console.log(criarId.next().value);  
console.log(criarId.next().value);  
console.log(criarId.next().value);
```



# Programação Assíncrona

# 15

---

Conclusão da unidade

# JavaScript no navegador

# 16

---

Vamos aprender como o JS interage com o browser e como aplicar ele em uma página web

# A web: Protocolos

- Um protocolo é uma forma de comunicação entre computadores através da rede;
- O **HTTP**, serve para nós solicitarmos arquivos, imagens e etc (Hyper Text Transfer Protocol);
- Entramos em sites graças ao protocolo HTTP;
- **SMTP** = protocolos para enviar e-mail;
- **TCP** = protocolo de transferência;



# A web: URL

- Cada arquivo que carrega no navegador é nomeado por uma URL;
- A URL (Uniform Resource Locator) pode ser dividida em três partes: protocolo, servidor e arquivo;
- <http://www.horadecodar.com.br/index.html>
- http = protocolo;
- [www.horadecodar.com.br](http://www.horadecodar.com.br) = servidor (DNS para um IP);
- index.html = arquivo;



# A web: HTML

- HTML (HyperText Markup Language) é uma linguagem de marcação;
- Responsável pelos textos e elementos que vemos na tela ao acessar uma página web;
- As tags do HTML criam: títulos, parágrafos, imagens, listas e etc.
- Uma tag é definida por `<p></p>`;
- Obs: algumas tags não possuem fechamento;



## A web: HTML parte 2

- Todo documento HTML tem sempre duas partes importantes: head e body
- O head serve para configurações do documento, como links de folha de estilo ou até de scripts de JavaScript;
- Tags de configuração: meta, link
- O body para os elementos (tags), ou seja, a parte visual;
- Tags do body: p, h1, div, ul, li



# A web: HTML e o JS

- Podemos utilizar JavaScript no HTML por meio da tag script;
- Ou linkar um arquivo .js por meio de script com o atributo src (forma mais utilizada);
- Algumas tags HTML aceitam atributos que executam JS, mas não é muito utilizado;
- Lembrando que sempre que linkamos um arquivo externo, estamos chamando ele via protocolo HTTP;





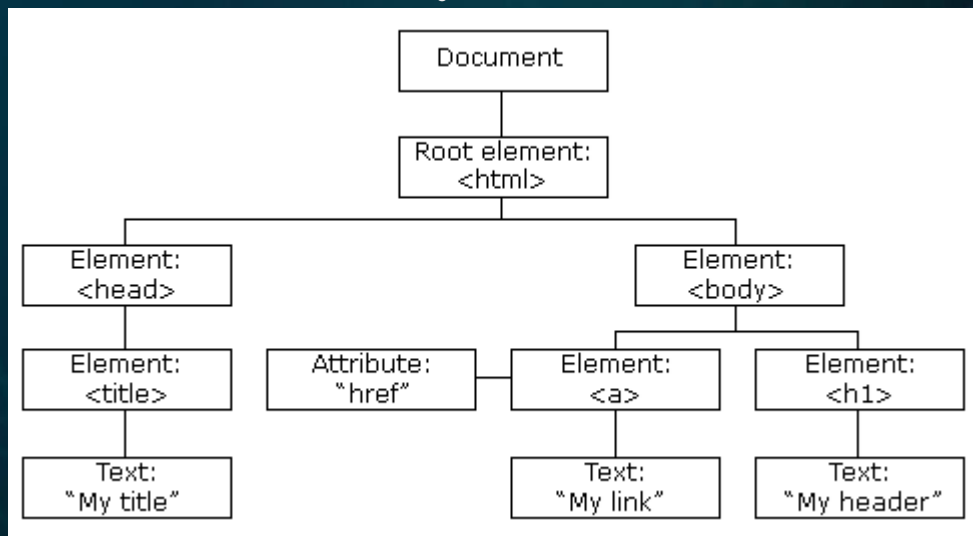
# A estrutura HTML e o DOM

- Quando uma página web é requisitada, ela recebe um texto que é transformado em uma estrutura HTML;
- As tags vão se aninhando uma às outras, formando uma estrutura em que elementos ficam dentro dos outros;
- Essa estrutura tem uma representação exatamente igual para o JS chamada de DOM;
- Que significa Document Object Model;
- E por meio do JS podemos acessar cada um destes elementos do HTML através do DOM;

```
<!DOCTYPE html>
<html>
<head>
  <title>Título da página</title>
  <meta charset="UTF-8">
</head>
<body>
  <div>
    <h1>Título principal</h1>
    <p>Algum texto</p>
  </div>
  <div>
    <h2>Lista de coisas a fazer:</h2>
    <ul>
      <li>Coisa 1</li>
      <li>Coisa 2</li>
      <li>Coisa 3</li>
    </ul>
  </div>
</body>
</html>
```

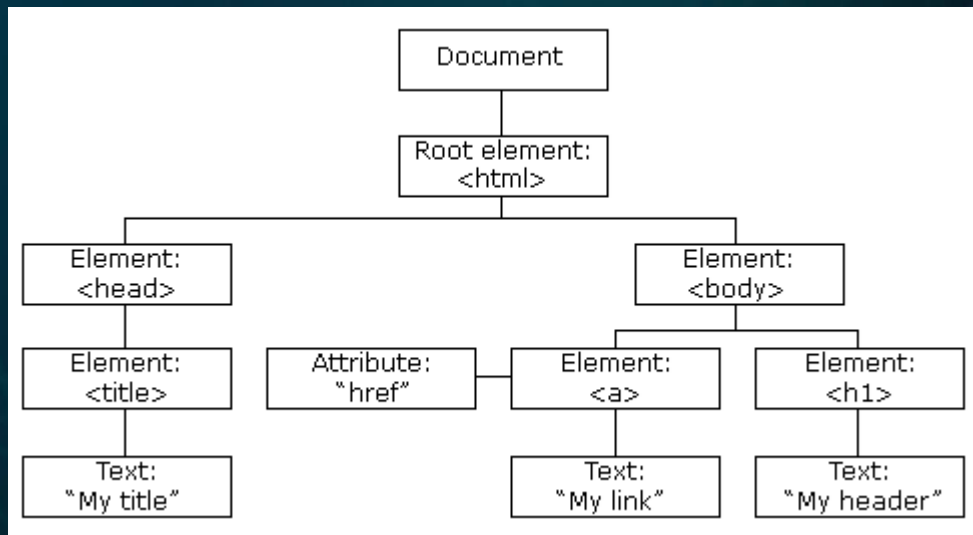
# O DOM

- Podemos mudar completamente uma página HTML através do DOM;
- É possível alterar: elementos, atributos, CSS;
- Além de alterar podemos adicionar e também remover;
- Além de ser possível criar eventos na página, como animações;



# A árvore do DOM

- O documento HTML seria a árvore completa;
- A raiz da árvore é o Document (só a uma raiz);
- Cada elemento da árvore chamamos de node (nó);
- Conteúdos como texto são chamados de leaf nodes (as folhas);



# Movendo-se através do DOM

- Podemos acessar todos os elementos a partir de document.body;
- A partir dele vamos entrando nos childNodes;
- E depois acessando as propriedades que nos interessam;

```
console.log(document.body);  
  
console.log(document.body.childNodes);  
  
console.log(document.body.childNodes[1]);  
  
console.log(document.body.childNodes[1].childNodes[1]);  
  
console.log(document.body.childNodes[1].childNodes[1].textContent);
```

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Título da página</title>  
    <meta charset="UTF-8">  
  </head>  
  <body>  
    <div>  
      <h1>Título principal</h1>  
      <p>Algum texto</p>  
    </div>  
    <div>  
      <h2>Lista de coisas a fazer:</h2>  
      <ul>  
        <li>Coisa 1</li>  
        <li>Coisa 2</li>  
        <li>Coisa 3</li>  
      </ul>  
    </div>  
  </body>  
</html>
```

# Encontrando elementos

- Podemos encontrar elementos de uma forma mais fácil do que se movendo pelo DOM;
- Como no HTML temos tags, classes e ids, o JavaScript nos possibilita utilizar estas características para encontrar o que desejamos;
- Há alguns métodos para isto: **getElementsByName**, **getElementById**, **getElementsByClassName**, e também o **querySelector**;
- Vamos ver cada um deles em detalhes;



# Encontrando elementos: por tag

- Utilizamos o seguinte método para encontrar elementos pela tag:
- `document.getElementsByTagName('h1');`

```
console.log(document.getElementsByTagName('h1'));
```



# Encontrando elementos: por id

- Utilizamos o seguinte método para encontrar elementos por id:
- `document.getElementById('algum-id');`

```
console.log(document.getElementById('_algum-id_'));
```



# Encontrando elementos: por classe

- Utilizamos o seguinte método para encontrar elementos por classe:
- `document.getElementsByClassName('alguma-classe');`

```
console.log(document.getElementsByClassName('teste'));
```





# Encontrando elementos: por query

- Um método que se assemelha a grande funcionalidade do jQuery;
- Nos permitindo encontrar elementos pelo seus seletores de CSS;
- O que facilita muito nossa vida, veja:

```
console.log(document.querySelector('#algum-id'));  
console.log(document.querySelector('.teste'));  
console.log(document.querySelector('h1'));
```



# Alterando o HTML

- Podemos mudar quase tudo através da DOM;
- Adicionar, remover e clonar elementos;
- Podemos utilizar métodos como: **insertBefore**, **appendChild** e **replaceChild**;
- Vamos ver em detalhes como utilizar cada um destes;



# Alterando a DOM com insertBefore

- Insere um nó antes do nó de referência do método;
- Precisamos de um elemento para ser adicionado;
- O elemento que terá o outro elemento adicionado antes dele;
- E o elemento pai destes dois;

```
let span = document.createElement("span");  
let el = document.querySelector("h1");  
let pai = el.parentNode;  
  
pai.insertBefore(span, el);
```



# Alterando a DOM com appendChild

- Adiciona um nó após todos os elementos do elemento pai especificado;

```
let p = document.createElement("p");  
let el2 = document.querySelector("h1");  
let pai2 = el2.parentNode;  
  
pai2.appendChild(p);
```



# Alterando a DOM com replaceChild

- Repõe um antigo elemento no DOM, com um novo;

```
let newHeading = document.createElement('h1')
newHeading.textContent = 'Novo Texto';

let oldHeading = document.getElementById("old-heading");

let pai = oldHeading.parentNode;

pai.replaceChild(newHeading, oldHeading);
```



# Criando nós de texto

- Podemos criar um nó de texto puro e inserir em um elemento;

```
let texto = document.createTextNode("Este é o texto");  
  
let p = document.getElementById("p-sem-texto");  
  
p.appendChild(texto);
```



# Criando nós de elemento

- Podemos criar um nó de elemento com o createElement, e depois inserir no HTML;

```
let span = document.createElement("span");  
let el = document.querySelector("h1");  
let pai = el.parentNode;  
  
pai.insertBefore(span, el);
```



# Modificando e lendo atributos

- Podemos resgatar o valor de um atributo ou trocar com o JS;
- Por exemplo o href de um a ou o src de um elemento img;
- `getAttribute` pega o valor;
- `setAttribute` atualiza o valor;

```
let url = document.getElementById('link');  
console.log(url.getAttribute('href'));  
  
url.setAttribute("href", "www.horadecodar.com.br");  
console.log(url.getAttribute('href'));
```





# Verificando altura e largura do elemento

- Com o JS também é possível verificar propriedades do elemento como altura e largura, veja:

```
let elTeste = document.getElementById('titulo');  
  
console.log(elTeste.offsetWidth);  
console.log(elTeste.offsetHeight);  
  
console.log(elTeste.clientWidth); // desconsidera bordas  
console.log(elTeste.clientHeight); // desconsidera bordas
```



# Posição do elemento na tela

- Também é possível checar a posição do elemento na tela;
- `getBoundingClientRect` nos dá as posições de top, left, right, bottom do elemento e outras informações;

```
let elTeste = document.getElementById('titulo');  
console.log(elTeste.getBoundingClientRect());
```



# Estilizando com JS

- Podemos também mudar as propriedades de estilo dos elementos do HTML com o JS
- Acessando a propriedade style podemos fazer as modificações
- O estilo é manipulado direto na tag, ou seja, substitui o do CSS (na maioria das vezes)

```
let p = document.getElementById('algum-p');  
console.log(p.style);  
p.style.color = 'red';
```



# Selecionar vários elementos com query

- Para selecionar diversos elementos com query, podemos utilizar o método `querySelectorAll`
- Em vez de um, podemos pegar todos os elementos semelhantes com o seletor do CSS

```
let todosPs = document.querySelectorAll('p');  
  
console.log(todosPs);  
  
console.log(count(todosPs));
```



# JavaScript no navegador

# 16

---

Conclusão da unidade

# Eventos com JavaScript

# 17

---

Vamos aprender a utilizar eventos no navegador com a linguagem JS

# O que são eventos?

- Ações condicionadas a algum tipo de resposta feita pelo usuário;
- Por exemplo: clicks, apertar alguma tecla ou até movimento do mouse;
- Podemos atrelar lógica a essas ações do usuário;
- Por meio de **handlers**;



# Como acionar um evento

- Devemos atrelar o evento a um elemento, por exemplo um botão;
- Depois inserir um listener e o tipo de evento como argumento;
- Aí o elemento responderá por este evento e, obviamente, os outros da página não;

```
let btn = document.querySelector("#btn");  
  
btn.addEventListener("click", function() {  
  console.log("Clicou");  
});
```





# Removendo eventos

- Da mesma forma que podemos adicionar eventos, podemos remover quando acharmos necessário;
- Para isso utilizamos o método `removeEventListener`, onde passamos o evento e a função que o evento está escutando;

```
let btn = document.querySelector("#btn");

function msg() {
  console.log("Clicou");
}

btn.addEventListener("click", msg);

setTimeout(function() {
  btn.removeEventListener("click", msg);
}, 3000);
```



# O objeto do evento

- Quando criamos eventos, temos a opção de utilizar um argumento opcional, que é chamado de objeto do evento;
- Ele contém propriedades que podem ser utilizadas a nosso favor;
- O objeto é criado pelo JavaScript automaticamente;

```
function msg(e) {  
    console.log(e);  
}  
  
btn.addEventListener("click", msg);
```



# Propagação

- Quando não definimos um elemento muito bem (seletor brando) ou um elemento que está dentro de outro tem um evento;
- Pode acontecer a propagação, ou seja, o outro elemento ativar um evento também, aí teremos uma duplicação;
- Por isso temos um método que para esta propagação e resolve este problema, o **stopPropagation**;

```
let p = document.querySelector('p');  
let btn = document.querySelector('button');  
  
p.addEventListener("click", function() {  
  console.log("click 1");  
});  
  
btn.addEventListener("click", function() {  
  console.log("click 2");  
});
```

# Ações default

- Muitos elementos/teclas já tem ações pre-definidas, como clicar num link nos leva a outra página;
- Podemos para este evento default, e criar uma lógica diferente para o elemento em questão;

```
let a = document.querySelector('a');  
  
a.addEventListener("click", function(e) {  
  e.preventDefault();  
  console.log("cliqueu mas não mudou de link");  
});
```



# Eventos de tecla (key event)

- Sempre que uma tecla é pressionada, são gerados dois eventos: keyup e keydown;
- Podemos realizar ações nestes eventos também;
- Keyup é quando soltamos a tecla;
- Keydown é quando apertamos;

```
window.addEventListener("keydown", function(e) {  
  
    if(e.key == "v") {  
        console.log("Apertou a letra v");  
    }  
  
});
```



# Outros eventos de mouse

- No mouse temos também eventos como mousedown e mouseup, semelhante aos das teclas;
- Dblclick para ativar com dois cliques;

```
btn = document.querySelector("button");  
  
btn.addEventListener("dblclick", function() {  
    console.log("Ativou com double click");  
});
```



# Movimento do mouse

- Podemos ativar eventos com a movimentação do mouse também;
- O nome dele é mousemove;
- Através desse evento podemos detectar a posição do ponteiro do mouse na tela;

```
window.addEventListener('mousemove', function(e) {  
  console.log(e.x);  
  console.log(e.y);  
});
```



# Eventos por scroll

- Podemos atrelar evento ao scroll da tela também, pelo evento scroll;
- Por exemplo: podemos criar um elemento assim que o scroll atingir uma posição x;

```
window.addEventListener('scroll', function(e) {  
    if(window.pageYOffset > 100) {  
        console.log("Chegou na posição");  
    }  
});
```





# Eventos por foco

- Quando focamos em um elemento ou saímos dele, podemos também atrelar um evento a esta ação;
- Focus para quando um elemento recebe foco e blur quando ele perde;

```
let input = document.querySelector("input");  
  
input.addEventListener("focus", function() {  
    console.log("foco no input");  
});  
  
input.addEventListener("blur", function() {  
    console.log("perdeu o foco");  
});
```



# Evento de carregamento

- Podemos atrelar um evento quando a página carrega, pelo evento load;
- E antes do usuário fechar a página pelo evento beforeunload;

```
window.addEventListener("beforeunload", function(e) {  
  });
```



# Debounce

- Um evento que dispara múltiplas vezes pode ser um problema para a o computador do cliente;
- Por isso podemos fazer um debounce, que é um suavizador de evento, para não chamar o mesmo tantas vezes;

```
window.addEventListener("mousemove", function() {  
  
    clearTimeout(timeout);  
    timeout = setTimeout(() => console.log("Debounce!"), 500);  
  
});
```



# Eventos com JavaScript

# 17

---

Conclusão da unidade

# Node.js 18

---

Aprender o que é Node.js e as suas possibilidades para desenvolvimento web

# O que é Node?

- Não é uma linguagem de programação;
- Uma ferramenta construída sobre o motor JavaScript do Google Chrome para criar aplicações rápidas e escaláveis;
- O problema: cada conexão de um cliente aloca um espaço na memória;
- Com Node: cada conexão é um evento executado na engine no Node;
- Ou seja, o Node suporta muito mais conexões do que PHP e Java em uma máquina igual, por exemplo;



# Instalando o Node

- Site oficial: <https://nodejs.org/en/>
- No windows é um executável, que funciona com a maioria dos outros programas para instalar (next, next, next);
- No Linux: `sudo apt install nodejs` e `sudo apt install npm`
- Depois teste com: `node -v`



# Testando o Node

- Podemos criar um arquivo JavaScript e executar com o node;
- Node test.js

```
let msg = "Hello World";  
console.log(msg);
```





# O comando node

- Podemos executar instruções de JavaScript rodando o comando node;
- Podemos fazer operações matemáticas, utilizar funções do JavaScript e muito mais;

```
$ node
Welcome to Node.js v13.2.0.
Type ".help" for more information.
> 1 + 1
2
> console.log("teste");
teste
undefined
> |
```



# O npm

- Ferramenta utilizada para instalar módulos de JavaScript;
- Que podem fazer desde operações matemáticas até comunicação HTTP;
- Instalamos um pacote pelo comando: **npm install pacote;**
- Há um curso completo de npm na Udemy feito por mim :)
- Vem junto com o Node;



# Iniciando um projeto

- Na última aula vimos que o node reclamou da falta do package.json;
- O certo sempre é ter um arquivo desse ao iniciar um novo projeto;
- Podemos criar a partir do template, utilizando o comando: npm init;
- Aí teremos um projeto com o package.json;
- Este arquivo serve para configurações e salva nossas dependências;



# Módulo de file system

- Um dos módulos mais utilizados;
- Ele serve para trabalhar com arquivos e diretórios;
- Já é incluso no node, ou seja, não precisamos instalar

```
let {readFile} = require('fs');

readFile("arquivo.txt", "utf8", (error, text) => {
  if(error) {
    throw error;
  } else {
    console.log(text);
  }
});
```



# File system: escrevendo

- Além de ler arquivo, podemos também alterar o mesmo;
- Chamamos isso de escrever no arquivo;

```
writeFile("arquivo.txt", "Inserir este texto", (error) => {  
  if(error) {  
    console.log(error);  
  } else {  
    console.log("Escreveu no arquivo");  
  }  
});
```



# Módulo HTTP

- Outro módulo muito utilizado, para fazer comunicação via HTTP;
- O módulo já vem com o Node;
- Pode criar um servidor que serve páginas web para nós;

```
const {createServer} = require("http");

let server = createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(`
    <h1>Hello World!</h1>
    <p>Primeira página web com Node.js</p>`);
  response.end();
});

server.listen(8000);

console.log("Listening! (port 8000)");
```



# O que é uma API?

- Application Programming Interface;
- Uma maneira de se comunicar via HTTP a um servidor e fazer operações como: visualizar, deletar, criar e atualizar dados;
- Grandes aplicações como Facebook e Instagram utilizam APIs;
- Grande parte dos softwares desenvolvidos em Node.js tem uma API;
- E utilizam um framework chamado Express para isso;



# O que é REST e RESTful?

- Quando falamos em API estamos condicionados a falar de REST e RESTful também;
- REST = Representational State Transfer, princípios e regras que permitem criar uma aplicação com interfaces bem definidas;
- RESTful = Capacidade de aplicar os princípios do REST;





# REST e os verbos HTTP

- Sempre que criamos uma API devemos utilizar os verbos HTTP corretos para cada URL da nossa aplicação (veremos isso na prática);
- Verbo GET: para solicitar dados;
- Verbo POST: para inserir dados;
- Verbo DELETE: para deletar dados;
- Verbo PUT: para atualizar dados;



# Express

- Um framework web, muito utilizado para criar aplicações em Node.js;
- Utiliza em seu core o módulo HTTP para criar as comunicações;
- Além de outros módulos;
- Podemos criar uma API com o Express facilmente;



# Instalando o Express

- O Express precisa ser instalado;
- Então vamos instalar pelo npm e criar um projeto novo;
- `npm init`;
- `npm install express`;
- O Express será instalado e adicionado as dependências do nosso projeto;



# O que são Rotas?

- Nos sites e nas APIs temos o que chamamos de rotas;
- Que é basicamente a URL que queremos acessar;
- Porém quando tratamos de uma API chamamos as URLs de rota;
- No Express vamos criar as nossas rotas para serem acessadas via HTTP e responderem baseadas numa lógica que inserirmos;
- E cada rota vai receber um método HTTP, que deve coincidir com o proposto pelo REST, conforme vimos anteriormente;



# Testando o Express

- Vamos criar nossa primeira aplicação e primeira rota;
- Além disso fazer os testes;

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Primeira rota com Express!');
});

app.listen(3000, function () {
  console.log('O Express está funcionando na porta 3000');
});
```



# Instalando o Postman

- Postman é o software mais utilizado para testar APIs;
- Quando ainda não é uma interface gráfica, podemos testar por meio dele as rotas de nossa API;
- <https://www.postman.com/downloads/>

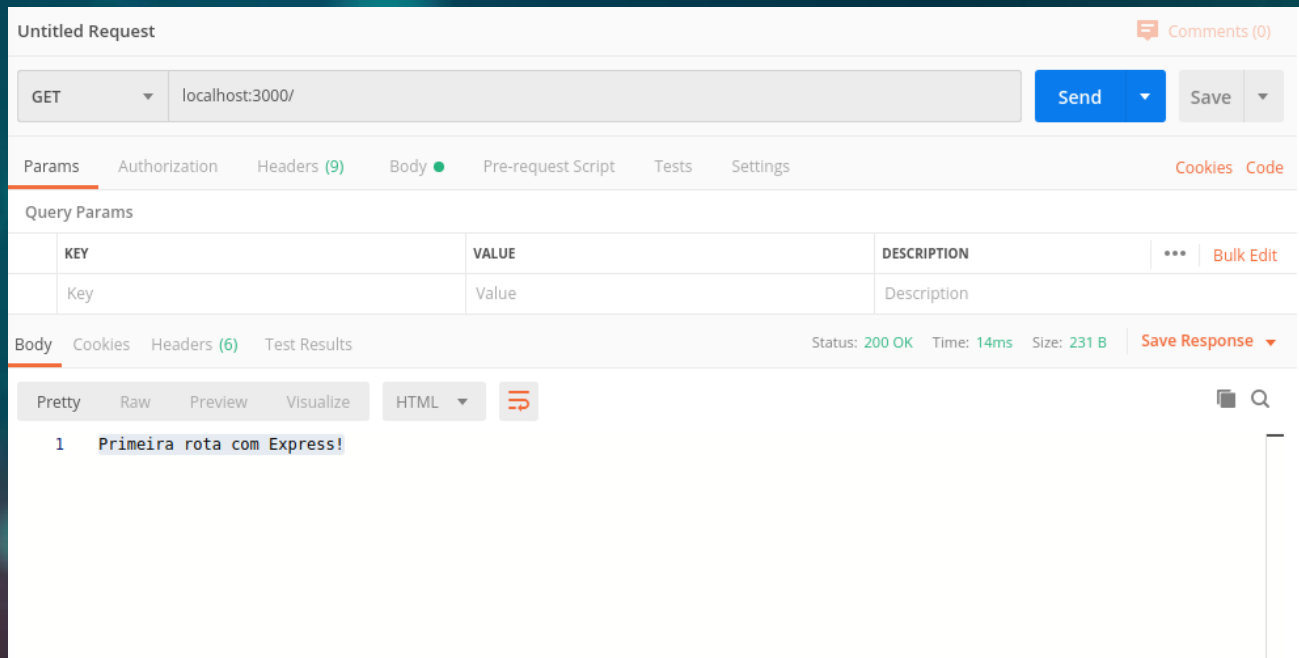


POSTMAN



# Testando rota com Postman

- Apenas precisamos definir o verbo correto (GET) e a rota;
- Assim receberemos a resposta que codamos no Express;



# Node.js 18

---

Conclusão da unidade