



Javascript



Introdução

- Uma linguagem leve, de alto nível;
- Amplamente utilizada por desenvolvedores WEB;
- Interpretada;
- Pode ser incorporada em páginas HTML para adicionar interatividade;
- baseada em objetos.

HTML, CSS e Javascript

- **HTML** é a linguagem de marcação usada para estruturar e dar significado ao conteúdo web;
- **CSS** é uma linguagem de regras de estilo usada para aplicar estilo ao conteúdo HTML;
- **JavaScript** é a linguagem de programação que permite a criação de conteúdo que se atualiza dinamicamente.

Jogador

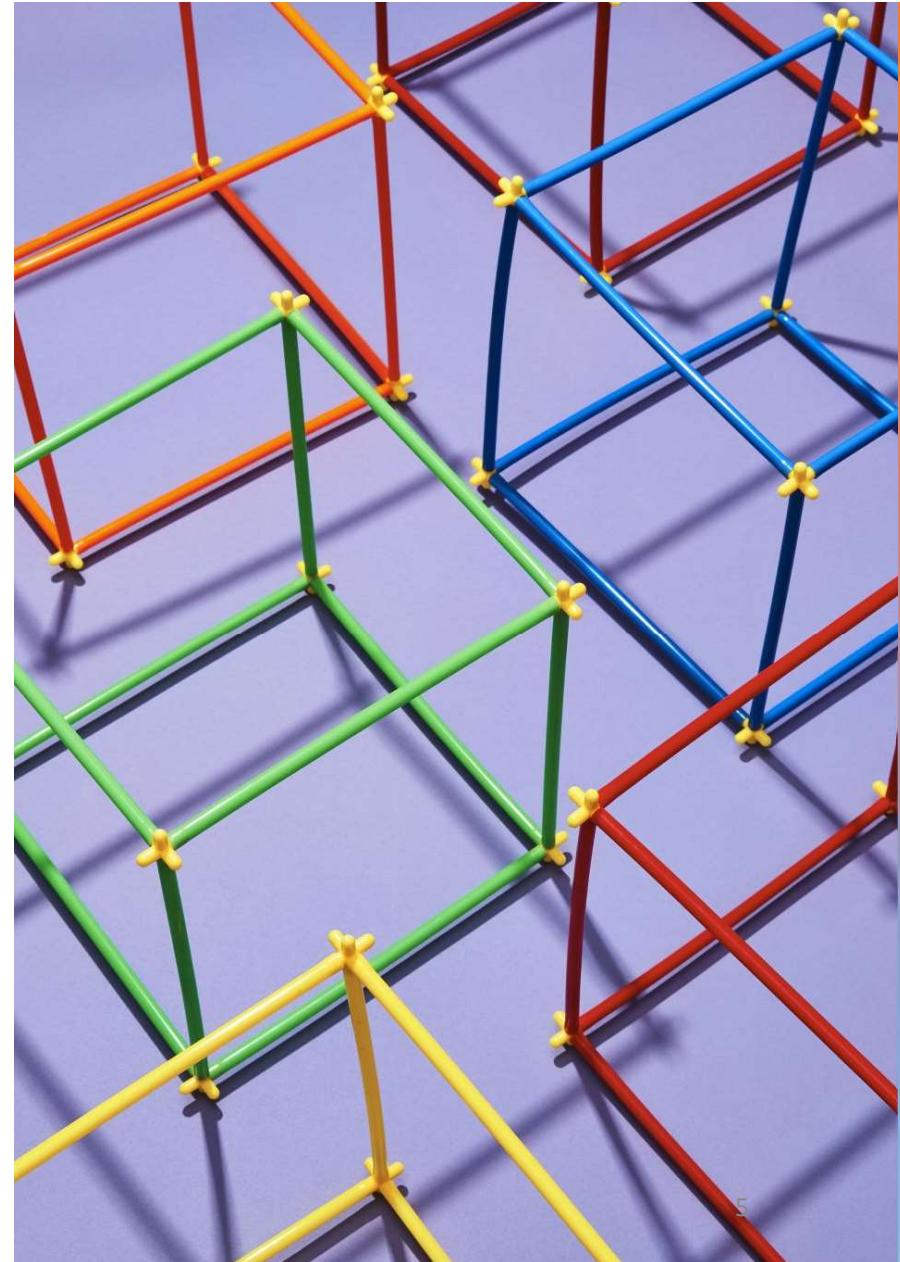
Javascript em ação

```
.roteiro-comprar{  
    border-radius: 10px;  
    color: white;  
    padding: 8px;  
    border:none;  
    background-color: #rgb(143, 214, 35);  
    display: block;  
    margin: auto;|  
}
```

```
<body>  
    <button class="roteiro-comprar" id="botao">Jogador</button>  
  
<script>  
    const botao = document.querySelector("button");  
  
    botao.addEventListener("click", atualizarNome);  
  
    function atualizarNome() {  
        var nome = prompt("Insira um novo nome");  
        botao.textContent = "Seja bem vindo Sr: " + nome;  
    }  
</script>  
</body>  
</html>|
```

Modelo de Objeto de Documento (DOM)

- O DOM (Document Object Model) é a representação de dados dos objetos que compõem a estrutura e o conteúdo de um documento na Web;
- O Document Object Model (DOM) é uma interface de programação para os documentos HTML e XML;
- Representa a página de forma que os programas possam alterar a estrutura do documento, alterar o estilo e conteúdo;
- O DOM representa o documento com nós e objetos, dessa forma, as linguagens de programação podem se conectar à página.



DOM

- A representação DOM deste documento é uma árvore com **document** como raiz;
- O objeto **document** possui um filho, **html**, que possui dois filhos, **head** e **body**.
- O objeto **head** possui um filho, **title**, que possui um filho, um nó de texto com o conteúdo "**Residencia T18**".
- O objeto **body** possui dois filhos, **h1** e **p**, cada um dos quais possui um filho, um nó de texto com o conteúdo "**Olá Mundo!**" e "**Este é um parágrafo.**", respectivamente.

```
1 <html>
2 <head>
3   |   <title>Residencia T18</title>
4 </head>
5 <body>
6   |   <h1>Olá Mundo!</h1>
7   |   <p>Este é um parágrafo</p>
8 </body>
9 </html>
```

Manipulação do DOM

- Você pode manipular o DOM usando JavaScript. Por exemplo, você pode alterar o conteúdo do elemento h1 assim:

```
document.querySelector('h1').textContent = 'Diga ae, DOM!';
```

- Este código seleciona o elemento h1 e altera seu conteúdo de texto para “Diga ae, DOM!”.

```
let novoParagrafo = document.createElement('p');
novoParagrafo.textContent = 'Esse parágrafo foi criado com javascript!';
//document.querySelector('body').appendChild(novoParagrafo);
document.body.appendChild(novoParagrafo);
```

Manipulação do DOM

Criar novos elementos e adicionar no DOM

DOM – Declarações finais

- O DOM não é uma linguagem de programação, mas sem ela, a linguagem JavaScript não teria nenhum modelo ou noção de páginas da web, documentos HTML, documentos XML e suas partes componentes;
- Cada elemento de um documento - o documento como um todo, o cabeçalho, as tabelas do documento, os cabeçalhos da tabela, o texto nas células da tabela - faz parte do modelo de objeto do documento desse documento, para que todos possam ser acessados e manipulados usando o método DOM e uma linguagem de script como JavaScript.

DOM – Declarações finais

- API (página HTML ou XML) = DOM + JS (linguagem de script)
- O DOM foi projetado para ser independente de qualquer linguagem de programação específica, disponibilizando a representação estrutural do documento a partir de uma única API consistente.

método addEventListener()

- **Eventos** são ações que acontecem quando o usuário ou o navegador manipula uma página;
- Podem fazer com que elementos da página da Web mudem de forma dinâmica;
- Quando o navegador, por exemplo, termina de carregar um documento, ocorreu um evento **load**;
- Quando um usuário clica num elemento, o evento **click** é acionado;
- O manipulador de evento `addEventListener()` pode ser anexado ao elemento HTML específico para o qual você deseja monitorar eventos.

método addEventListener()

```
addEventListener(type, listener)
addEventListener(type, listener, options)
addEventListener(type, listener, useCapture)
```

```
const botao = document.querySelector("button");

botao.addEventListener("click", atualizarNome);

function atualizarNome() {
    var nome = prompt("Insira um novo nome");
    botao.textContent = "Seja bem vindo Sr: " + nome;
}
```

Estrutura de dados do Javascript

- As variáveis em JavaScript não estão diretamente associadas a nenhum tipo de valor específico;
- Qualquer variável pode receber (e reatribuir) valores de todos os tipos;

```
let foo = 42; // foo agora é um número
foo = "bar"; // foo agora é uma string
foo = true; // foo agora é um booleano
```

Estrutura de dados do Javascript

- É uma linguagem de tipagem fraca, o que significa que permite a conversão implícita de tipo quando uma operação envolve tipos incompatíveis, em vez de gerar erros de tipo.

```
const foo = 42; //foo é um número
const result = foo + "1"; // JavaScript coage foo para uma string, então ela pode
ser concatenada com o outro operando
console.log(result); // 421
```



Tipos de dados

- Primitivos:
 - Number
 - String
 - Boolean
 - Null
 - Undefined
 - Symbol
 - BigInt
- Objetos
 - Coleções de propriedades

Declaração de variáveis

```
//variavel do tipo string  
let nome="João";  
//varivel do tipo number  
let idade=20;  
//variavel do tipo boolean  
let condicao=true;  
//variavel do tipo object  
let objeto = {chave: "valor"};  
//variavel do tipo funcao  
let funcao = function(){};  
//variavel do tipo object  
let listaDeNumeros = [1,2,3,4,5];
```

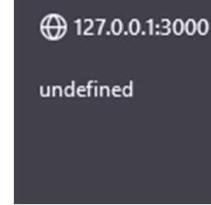
- São utilizadas as palavras: **var**, **let** ou **const** para declarar variáveis.

var

- < ES6. Escopo global quando declarada fora da função e escopo local, quando declarada dentro da função.
- Podem ser declaradas novamente e depois atualizadas;
- Hoisting de var
 - Mecanismo que faz com que declarações de variáveis e de funções sejam movidas para o topo antes que o código seja executado;

```
<script>
  alert(saudacao);
  var saudacao = "Olá todo mundo!";
</script>
```

```
<!--//Interpretação-->
<script>
  var saudacao;
  alert(saudacao); //saudacao vai ser undefined
  saudacao = "Olá todo mundo!";
</script>
```



let

- Forma preferida de declaração de variáveis > EC6;
- Tem escopo de bloco;
- Pode ser declarado mas não atualizada novamente;
- Hoisting de let
 - Diferente de uma variável var, a variável let não é inicializada, se for tentando usar uma variável let antes da sua declaração, uma reference error é disparada, que é quando uma variável que não existe está sendo atualmente sendo referenciada;

```
alert(saudacao2);
let saudacao2 = "Olá todo mundo 2!";
```

```
const numeroIP = "127.0.0.1";
numeroIP = "192.168.0.13"; //erro
```

const

- Tem escopo de bloco;
- Não pode ser atualizada e nem declarada novamente;
- var e let podem ser declaradas sem ser inicializadas, const precisa da inicialização durante a declaração.

Expressões

- Combinações de valores, variáveis e operadores que produzem um resultado;
- Dois tipos de expressões: aquelas que atribuem um valor a uma variável;
 - $X = 10$;
- Aquelas que possuem um valor;
 - $5 + 2$;

```
<script>
    let somaDeDoisNumeros = 1 + 2;
    alert(somaDeDoisNumeros);

    let subtracaoDeDoisNumeros = 1 - 2;
    alert(subtracaoDeDoisNumeros);

    let multiplicacaoDeDoisNumeros = 1 * 2;
    alert(multiplicacaoDeDoisNumeros);

    let divisaoDeDoisNumeros = 1 / 2;
    alert(divisaoDeDoisNumeros);

    let somaTotalDasVariaveis = somaDeDoisNumeros +
        subtracaoDeDoisNumeros + multiplicacaoDeDoisNumeros
        + divisaoDeDoisNumeros;
    alert("soma: "+somaTotalDasVariaveis);
</script>
```

Operadores de atribuição

- Um operador de atribuição atribui um valor ao operando à sua esquerda baseado no valor do operando à direita;
- O operador de atribuição básico é o igual ($=$), que atribui o valor do operando à direita ao operando à esquerda. Isto é, $x = y$ atribui o valor de y a x

Nome	Operador encurtado	Significado
Atribuição	$x = y$	$x = y$
Atribuição de adição	$x += y$	$x = x + y$
Atribuição de subtração	$x -= y$	$x = x - y$
Atribuição de multiplicação	$x *= y$	$x = x * y$
Atribuição de divisão	$x /= y$	$x = x / y$
Atribuição de resto	$x %= y$	$x = x \% y$
Atribuição exponencial	$x **= y$	$x = x ** y$
Atribuição bit-a-bit por deslocamento á esquerda	$x <<= y$	$x = x << y$
Atribuição bit-a-bit por deslocamento á direita	$x >>= y$	$x = x >> y$
Atribuição de bit-a-bit deslocamento á direita não assinado	$x >>>= y$	$x = x >>> y$
Atribuição AND bit-a-bit	$x &= y$	$x = x \& y$
Atribuição XOR bit-a-bit	$x ^= y$	$x = x ^ y$
Atribuição OR bit-a-bit	$x = y$	$x = x y$

Expressões primárias

- This: é utilizado para se referir ao objeto atual. Em geral, o this se refere ao objeto chamado em um método;
 - **this.NomeDApropriedade;**

```
function valide(obj, minimo, maximo) {  
    if (obj.valor < minimo || obj.valor > maximo) alert("Valor inválido!");  
}
```

HTML

```
<b>Informe um número entre 18 e 99:</b>  
<input type="text" name="idade" size="3" onChange="valide(this, 18, 99);"/>
```

Operadores de comparação

Operador	Descrição	Exemplos que retornam verdadeiro
Igual (<code>==</code>)	Retorna verdadeiro caso os operandos sejam iguais.	<code>3 == var1 "3" == var1 3 == '3'</code>
Não igual (<code>!=</code>)	Retorna verdadeiro caso os operandos não sejam iguais.	<code>var1 != 4 var2 != "3"</code>
Estritamente igual (<code>===</code>)	Retorna verdadeiro caso os operandos sejam iguais e do mesmo tipo. Veja também Object.is e igualdade em JS (en-US) .	<code>3 === var1</code>
Estritamente não igual (<code>!==</code>)	Retorna verdadeiro caso os operandos não sejam iguais e/ou não sejam do mesmo tipo.	<code>var1 !== "3" 3 !== '3'</code>
Maior que (<code>></code>)	Retorna verdadeiro caso o operando da esquerda seja maior que o da direita.	<code>var2 > var1 "12" > 2</code>
Maior que ou igual (<code>>=</code>)	Retorna verdadeiro caso o operando da esquerda seja maior ou igual ao da direita.	<code>var2 >= var1 var1 >= 3</code>
Menor que (<code><</code>)	Retorna verdadeiro caso o operando da esquerda seja menor que o da direita.	<code>var1 < var2 "12" < "2"</code>
Menor que ou igual (<code><=</code>)	Retorna verdadeiro caso o operando da esquerda seja menor ou igual ao da direita.	<code>var1 <= var2 var2 <= 5</code>

Operadores aritméticos

Operador	Descrição	Exemplo
Módulo (%)	Operador binário. Retorna o inteiro restante da divisão dos dois operandos.	12 % 5 retorna 2.
Incremento (++)	Operador unário. Adiciona um ao seu operando. Se usado como operador prefixado (<code>++x</code>), retorna o valor de seu operando após a adição. Se usado como operador pósfixado (<code>x++</code>), retorna o valor de seu operando antes da adição.	Se <code>x</code> é 3, então <code>++x</code> define <code>x</code> como 4 e retorna 4, enquanto <code>x++</code> retorna 3 e, somente então, define <code>x</code> como 4.
Decremento (--)	Operador unário. Subtrai um de seu operando. O valor de retorno é análogo àquele do operador de incremento.	Se <code>x</code> é 3, então <code>--x</code> define <code>x</code> como 2 e retorna 2, enquanto <code>x--</code> retorna 3 e, somente então, define <code>x</code> como 2.
Negação (-)	Operador unário. Retorna a negação de seu operando.	Se <code>x</code> é 3, então <code>-x</code> retorna -3.
Adição (+)	Operador unário. Tenta converter o operando em um número, sempre que possível.	<code>+"3"</code> retorna 3. <code>.+true</code> retorna 1.
Operador de exponenciação (**)	Calcula a base elevada á potência do expoente, que é, base <code>expoente</code>	<code>2 ** 3</code> retorna 8. <code>10 ** -1</code> retorna 0.1

Operadores lógicos

Operador	Utilização	Descrição
AND lógico (&&)	<code>expr1 && expr2</code>	(E lógico) - Retorna <code>expr1</code> caso possa ser convertido para falso; senão, retorna <code>expr2</code> . Assim, quando utilizado com valores booleanos, <code>&&</code> retorna verdadeiro caso ambos operandos sejam verdadeiros; caso contrário, retorna falso.
OU lógico ()	<code>expr1 expr2</code>	(OU lógico) - Retorna <code>expr1</code> caso possa ser convertido para verdadeiro; senão, retorna <code>expr2</code> . Assim, quando utilizado com valores booleanos, <code> </code> retorna verdadeiro caso ambos os operandos sejam verdadeiro; se ambos forem falsos, retorna falso.
NOT lógico (!)	<code>!expr</code>	(Negação lógica) Retorna falso caso o único operando possa ser convertido para verdadeiro; senão, retorna verdadeiro.

Operadores de string

```
console.log("minha " + "string"); // exibe a string "minha string".
```

Estruturas de Controle

– Condição IF

- Código executado se uma condição for verdadeira;

```
if (condição) {  
    // Bloco de código.  
}
```



Estruturas de Controle – Condição Else

- Bloco de código é executado, caso a condição IF seja falsa;

```
if (condicao) {  
    // bloco de código  
} else {  
    // caso o if seja falso, esse bloco será  
    // executado  
}
```

Estruturas de Controle – Condição else if

- O else if para especificar uma nova condição, caso as primeiras condições sejam falsas;

```
if (numero > 100) {  
    alert("O numero " + numero + " é maior que 100");  
} else if(numero > 10) {  
    alert("O numero " + numero + " é maior que 10");  
} else {  
    alert("O numero " + numero + " é menor ou igual a 10");  
}
```

Estruturas de Controle – Condição Switch

- Seleciona um de muitos blocos de códigos se sua condição for verdadeira;

```
switch(cor) {  
    case 'red':  
        document.body.style.backgroundColor = 'red';  
        break;  
    case 'blue':  
        document.body.style.backgroundColor = 'blue';  
        break;  
    case 'green':  
        document.body.style.backgroundColor = 'green';  
        break;  
    default:  
        document.body.style.backgroundColor = 'white';  
}
```

Loops

- Os loops são utilizados para repetir uma execução de bloco de código;
 - for
 - while

```
let i = 0;
while (i < contador) {
    var numeroAleatorio = Math.floor(Math.random() * 20) + 1;
    numbersDiv.innerHTML += '<p>' + numeroAleatorio + '</p>';
    i++;
}
```

```
for (var i = 0; i < contador; i++) {
    var numeroAleatorio = Math.floor(Math.random() * 20) + 1;
    numbersDiv.innerHTML += '<p>' + numeroAleatorio + '</p>';
    alert(numbersDiv.innerHTML);
}
```

Operador condicional (ternário)

```
condicao ? valor1 : valor2
```

```
var status = idade >= 18 ? "adulta" : "menor de idade";
```

Funções

- Blocos de construção fundamentais no JavaScript;
- Um conjunto de instruções que realiza um trabalho;
- Declarando uma função:
 - Nome da função;
 - Lista de Argumentos para a função, entre parênteses e separados por vírgula;
 - Conjunto de instruções entre chaves {};

```
// Função para calcular a soma de todos os elementos de um array.  
function calculaSoma(array) {  
    var soma = 0;  
    for (var i = 0; i < array.length; i++) {  
        soma += array[i];  
    }  
    return soma;  
}
```

Funções

- Parâmetros primitivos são passados por **valor**, ou seja se a função alterar o valor do parâmetro, esta mudança não reflete globalmente ou na função chamada;
- Parâmetros não primitivos (objetos, tal como Array ou um objeto definido por usuário) são passados por referência, se a função alterar as propriedades do objeto, essa mudança é visível fora da função;

```
function minhaFuncao(objeto) {  
    objeto.make = "Toyota";  
}  
  
var meucarro = { make: "Honda", model: "Accord", year: 1998 };  
var x, y;  
  
x = meucarro.make; // x recebe o valor "Honda"  
  
minhaFuncao(meucarro);  
y = meucarro.make; // y recebe o valor "Toyota"  
// (a propriedade make foi alterada pela função)
```

Expressão de função

```
var square = function (numero) {  
    return numero * numero;  
};  
  
var x = square(4); //x recebe o valor 16
```

```
var factorial = function fac(n) {  
    return n < 2 ? 1 : n * fac(n - 1);  
};  
  
console.log(factorial(3));
```

- Funções também podem ser criadas por uma expressão de função;
- Tal função pode ser anônima, ou seja ela não tem que ter um nome;

```
const materials = ['Hydrogen', 'Helium', 'Lithium', 'Beryllium'];

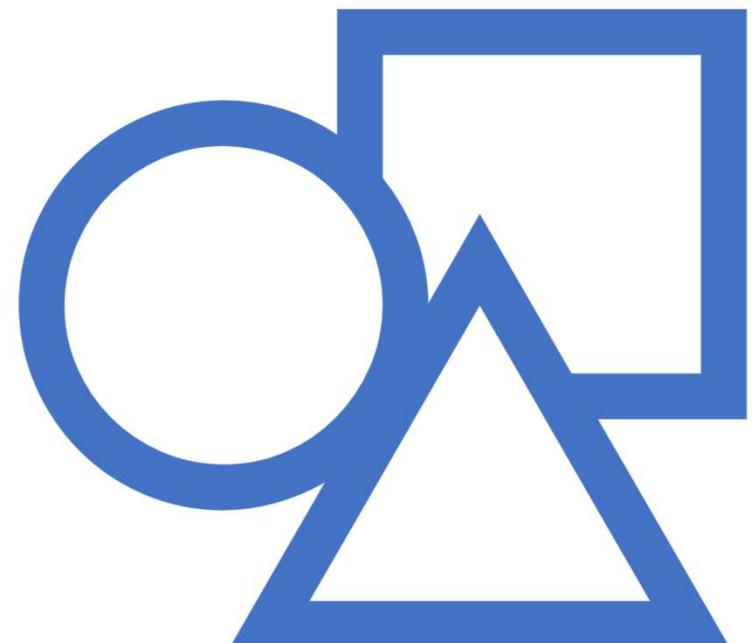
console.log(materials.map((material) => material.length));
// Expected output: Array [8, 6, 7, 9]
```

Funções de seta

- Uma **expressão arrow function** possui uma sintaxe mais curta quando comparada a uma expressão de função;

Objetos

- Um objeto é uma coleção de propriedades;
- Uma propriedade é uma associação entre chave e um valor;
- Um objeto é uma entidade independente, com propriedades e tipo;
- Por exemplo, considere uma xícara como um objeto;
 - Ela tem propriedades. Uma xícara tem uma cor, uma forma, peso, um material de composição, etc.
 - Da mesma forma, objetos em JavaScript podem ter propriedades, que definem suas características.



Objetos e propriedades

- Um objeto em JavaScript tem propriedades associadas a ele;
- Uma propriedade de um objeto pode ser explicada como uma variável que é ligada ao objeto;
- Propriedades de objetos são basicamente as mesmas que variáveis normais em JavaScript, exceto pelo fato de estarem ligadas a objetos;
- As propriedades de um objeto definem as características do objeto.

```
var aviao = {  
    Modelo: "Boeing 747",  
    Capacidade: 350,  
    Alcance: "13500 KM",  
    VelocidadeMax: "988 km/h",  
    Motores: 4  
};
```

```
var p51 = new Object();  
p51.modelo = "P51 Mustang";  
p51.capacidade = 1;  
p51.alcance = "1.600 km";  
p51.velocidadeMax = "703 km/h";  
p51.motores = 1;
```

```
// função construtora de Aviao
function Aviao(modelo, capacidade, alcance, velocidadeMax, motores) {
    this.modelo = modelo;
    this.capacidade = capacidade;
    this.alcance = alcance;
    this.velocidadeMax = velocidadeMax;
    this.motores = motores;
}

// Cria 3 objetos Aviao usando a função construtora
var boing747 = new Aviao("Boeing 747", 350, "13500 KM", "988 km/h", 4);
var spitfire = new Aviao("Supermarine Spitfire", 1, "1,135 km", "594 km/h", 1);
var mustang = new Aviao("P-51 Mustang", 1, "2,656 km", "703 km/h", 1);
```

Função construtora

- Criar o tipo de objeto escrevendo uma função construtora;
- Criar uma instância do objeto com new.

Definindo métodos

- Um *método* é uma função associada a um objeto;
- Ou: um método é uma propriedade de um objeto que é uma função;

```
// Adiciona o metodo voar ao prototype do objeto Aviao
function voar() {
    return this.modelo + " está voando a uma velocidade máxima de: " + this.velocidadeMax;
};

this.voar = voar;
```

```
// funcao construtora de Aviao
function Aviao(modelo, capacidade, alcance, velocidadeMax, motores) {
    this.modelo = modelo;
    this.capacidade = capacidade;
    this.alcance = alcance;
    this.velocidadeMax = velocidadeMax;
    this.motores = motores;
    this.voar = voar;
}
```

Arrays

- Coleção de itens relacionados que podem ser armazenados na mesma variável;
- O objeto Array do JavaScript é um objeto global usado na construção de 'arrays';

```
let cacasWW2 = [ "Supermarine Spitfire",
                  "P-51 Mustang",
                  "Messerschmitt Bf 109",
                  "Focke-Wulf Fw 190",
                  "Yak-3",
                  "A6M Zero"];
```

Acessando os itens do array

Array[0], array[1], array[2]...

```
function geraListaHTML(array) {  
    let html = "<ol>";  
    for (let i = 0; i < array.length; i++) {  
        html += "<li>" + array[i] + "</li>";  
    }  
    html += "</ol>";  
    return html;  
}
```

Operações com um Array

- Iterar um Array;

```
frutas.forEach(function (item, indice, array) {  
  console.log(item, indice);  
});
```

- Adicionar um item ao final do Array;

```
|  
|  cacasWW2.push("P-38 Lightning");
```

- Remover um item do final do Array;

```
cacاسWW2.pop();
```

Operações com um Array

- Remover do início do Array;

```
cacasWW2.shift()
```

- Adicionar no inicio do Array;

```
cacasWW2.unshift("P47 Thunderbolt");
```

Operações com um Array

- Procurar o índice de um item no Array;

```
let indiceP47 = cacasWW2.indexOf("P47 Thunderbolt");
```

- Remover item/itens de um Array;

```
|  
cacasWW2.splice(indiceP47, 1);
```

- Copiar um Array;

```
let cacas = cacasWW2.slice();
```

Array Methods Visualised (part 1)

The slide displays ten examples of array methods, each showing the original array, the method call, and the resulting array. The arrays are represented by brackets containing colored circles (orange, purple, red, green). An arrow points from the original array to the result. A small note at the bottom right indicates that some methods update the original array.

- [Orange, Orange, Orange].push(Green) → [Orange, Orange, Orange, Green]
- [Orange, Orange, Orange].unshift(Green) → [Green, Orange, Orange, Orange]
- [Orange, Purple, Red, Green].pop() → [Orange, Purple, Red]
- [Orange, Purple, Red, Green].shift() → [Purple, Red, Green]
- [Orange, Green, Green, Orange].filter(Green) → [Green, Green]
- [Orange, Orange, Orange].map((Orange) => Purple) → [Purple, Purple, Purple]
- [Orange, Purple, Red, Green].join("-") → "Orange-Purple-Red-Green"
- [Orange, Purple].concat([Red, Green]) → [Orange, Purple, Red, Green]
- [Orange, Purple, [Red, Green]].flat() → [Orange, Purple, Red, Green]
- [Orange, Purple, Red, Green].slice(1, 3) → [Purple, Red]

Method updates original array

Métodos do Array

Array Methods Visualised



Array Methods Visualised (part 2)

[● ● ●].splice(1, 0, ●)	→ [● ● ● ●]
[0 1 2 3].indexOf(0)	→ 2
[● ● ● ●].includes(0)	→ True
[0 1 2 3].at(2)	→ ●
[● ● ● ●].sort()	→ [● ● ● ●]
[● ● ● ●].reverse()	→ [● ● ● ●]
[● ● ● ●].fill(1, ●)	→ [● ● ● ●]
Array.from("0,1,2")	→ ["0", "1", "2"]
Array.isArray("[0,1,2,3]")	→ False
Array.of([0,1,2])	→ [0,1,2]

Métodos do Array



David Mráz
@davidm_ai

Classes e protótipos

- Classes em JavaScript são introduzidas no ECMAScript 2015;
- São simplificações da linguagem para as heranças baseadas nos protótipos;
- A sintaxe para classes **não** introduz um novo modelo de herança de orientação a objetos em JavaScript;
- Classes em JavaScript provêm uma maneira mais simples e clara de criar objetos e lidar com herança.

Classes

```
<script>
  class Aviao {
    constructor(nome, modelo, topSpeed, alcance, motores, fabricante, preco) {
      this.nome = nome;
      this.modelo = modelo;
      this.topSpeed = topSpeed;
      this.alcance = alcance;
      this.motores = motores;
      this.fabricante = fabricante;
      this.preco = preco;
    }

    display() {
      return this.nome + ", " + this.modelo + ", " + this.topSpeed + ", "
      + this.alcance + ", " + this.motores + ", " + this.fabricante + ", "
      + this.preco;
    }
  }

  let p51 = new Aviao("P-51 Mustang", "Mk II", 703, 1646, 1, "North American Aviation", 50000);
  alert(p51.display());

</script>
```

Métodos Protótipos

```
get modelo() {  
    return this._modelo;  
}
```

```
alert(p51.modelo)
```

Métodos estáticos

- A palavra-chave static define um método estático de uma classe;
- Métodos estáticos são chamados sem a instanciação da sua classe e não podem ser chamados quando a classe é instanciada;
- Métodos estáticos são geralmente usados para criar funções de utilidades por uma aplicação.

Exemplo de um método estático

```
<script>
  class Aviao {

    static count = 0;

    constructor(nome, modelo, topSpeed, alcance, motores, fabricante, preco) {
      this.nome = nome;
      this._modelo = modelo;
      this.topSpeed = topSpeed;
      this.alcance = alcance;
      this.motores = motores;
      this.fabricante = fabricante;
      this.preco = preco;
      Aviao.count++;
    }

    display() {
      return this.nome + ", " + this.modelo + ", " + this.topSpeed + ", "
      + this.alcance + ", " + this.motores + ", " + this.fabricante + ", "
      + this.preco;
    }

    static getAvioesDoHangar(){
      return Aviao.count;
    }

    get modelo() {
      return this._modelo;
    }
  }

  let p51 = new Aviao("P-51 Mustang", "Mk II", 703, 1646, 1, "North American Aviation", 50000);

  alert(Aviao.getAvioesDoHangar());
</script>
```

Map

- O objeto Map contém pares de chave-valor e lembra a ordem original da inserção das chaves;
- Qualquer valor (objetos e valores primitivos) podem ser usados como chave ou valor.

```
1 const map1 = new Map();
2
3 map1.set('a', 1);
4 map1.set('b', 2);
5 map1.set('c', 3);
6
7 console.log(map1.get('a'));
8 // Expected output: 1
9
10 map1.set('a', 97);
11
12 console.log(map1.get('a'));
13 // Expected output: 97
14
15 console.log(map1.size);
16 // Expected output: 3
17
18 map1.delete('b');
19
20 console.log(map1.size);
21 // Expected output: 2
22
```

Convertendo object em Map

- O método Object.entries() retorna uma array dos próprios pares [key, value] enumeráveis de um dado objeto;

```
//converte o objeto Aviao p51 em um Map
let p51Map = new Map(Object.entries(p51));
```

Iteração

Um Map é iterável, então ele pode ser diretamente iterável;

```
function createTable(map) {
    let table = document.createElement('table');

    for (let [key, value] of map) {

        let row = document.createElement('tr');
        let keyCell = document.createElement('td');
        let valueCell = document.createElement('td');
        keyCell.textContent = key;
        valueCell.textContent = value;
        row.appendChild(keyCell);
        row.appendChild(valueCell);
        table.appendChild(row);

    }
    return table;
}
```

Propriedade localStorage

- O objeto localStorage permite salvar pares chave/valor no navegador;
- O objeto localStorage armazena dados sem data de expiração;
- Os dados não são excluídos quando o navegador é fechado e ficam disponíveis para sessões futuras.

```
localStorage.corDefinida = "#a4509b";
localStorage["corDefinida"] = "#a4509b";
localStorage.setItem("corDefinida", "#a4509b");
```

```
var currentColor = localStorage.getItem("bgcolor");
var currentFont = localStorage.getItem("font");
var currentImage = localStorage.getItem("image");
```

Exercício0

- Crie um gerenciador permanente de tarefas no navegador;

- Crie um formulário com o campo tarefa e um botão de adicionar tarefa;
- Crie um elemento `` com o atributo `id="listaTarefas"`;
- Crie uma classe Tarefa que recebe como parâmetro do construtor, o nome da tarefa;
- Crie uma função `adicionaTarefaDOM` que recebe um objeto Tarefa e adiciona em `listaTarefas` do HTML.
- Crie uma função `adicionaTarefaNoStorage` que recebe um objeto Tarefa, recupera as tarefas do `localStorage` (se houver) e adiciona a nova tarefa. Considere armazenar as tarefas no `localStorage` como um array [] de objetos JSON.
- Associe um evento ao botão de adicionar que ao ser clicado ele deve pegar o valor do campo tarefa, criar um objeto Tarefa, chamar a função `adicionaTarefaDOM` e `adicionaTarefaNoStorage` e por fim limpar o campo tarefa do formulário;

Exercício (Continuação ...)

- Considere utilizar o método JSON.parse para criar um objeto JSON e alimentá-lo com as tarefas vindas do localStorage;
 - https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse
- Considere utilizar o método JSON.stringify() para gravar as tarefas no localStorage;
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify
- Crie uma função carregaTarefasDoStorage, carrega todas as tarefas armazenadas no localStorage as adiciona no DOM;
- Chame essa função no final do seu script

Exercício 1

- Criar uma classe que represente um Computador;
 - Crie pelo menos 4 propriedades;
 - Crie pelo menos 2 métodos;
- Crie 3 objetos da classe Computador;
- Crie uma função que receba um objeto Computador e retorne um map desse objeto;
- Crie uma função que receba um map como parâmetro e altere o arquivo html (utilizando o DOM) para criar uma lista desordenada de cada entrada do map. Cada linha da lista deve ter o seguinte formato:
 - **Chave:valor**
- Utilize os 3 objetos do Tipo Computador nas suas funções e visualize o resultado na página HTML

Exercício 2

- Crie um formulário com todos os campos para representar cada propriedade da classe Avião (s.48)
- Associe um evento ao botão submit. Esse evento deve chamar uma função que pegue todos os valores dos campos do formulário e crie um objeto Avião.
- Crie um elemento div utilizando o DOM;
- Adiciona um conteúdo nesse div chamando o método display do objeto Avião;

Declarações de Manipulação de Erro

- É possível chamar uma exceção usando a declaração throw e manipulá-la usando a declaração try...catch;
- A declaração try...catch coloca um bloco de declarações em try, e especifica uma ou mais respostas para uma exceção lançada;

```
document.getElementById('showPlaneButton').addEventListener('click', function() {
    let indice = document.getElementById('indexInput').value;

    try{
        if (indice < 0 || indice >= cacasWW2.length) {
            throw new Error('Índice inválido');
        }
        document.getElementById('output').textContent = cacasWW2[indice];
    } catch (error) {
        alert("Erro: " + error.message);
    }
});
```

Referências

- JavaScript
 - <https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript>
- Expressões e operadores
 - https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Guide/Expressions_and_Operators
- Tomando decisões no seu código — condicionais
 - https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/Building_blocks/conditionals
- Coleções chaveadas: Maps, Sets, WeakMaps, WeakSets
 - https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Data_structures#coleções_chaveadas_maps_sets_weakmaps_weaksets
- HTML DOM Documents
 - https://www.w3schools.com/jsref/dom_obj_document.asp
- Api WebStorage
 - https://developer.mozilla.org/pt-BR/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API
- Element.addEventListener()
 - <https://developer.mozilla.org/pt-BR/docs/Web/API/EventTarget/addEventListener>