

Pós-Graduação Lato Sensu
Curso de Especialização em Tecnologia Java

Frameworks Web

JAVASCRIPT

Prof. Esp. Hugo Baker Goveia



1. O que é JavaScript?

É uma linguagem leve, multiplataforma (procedural, funcional e orientada a objetos), com tipagem de dados dinâmica.

Também conhecida como JS, uma linguagem de script para páginas web.

É uma linguagem de programação de alto nível, interpretada, ou seja, ela passa por um interpretador para em seguida ser entendida pela máquina e poder executar suas funcionalidades. Hoje em dia ela é baseada em um padrão chamado ECMAScript que surgiu em 1995 com o Netscape.

Originalmente desenvolvida para adicionar interatividade às páginas web, JavaScript se tornou uma das linguagens mais populares e versáteis do mundo.

Hoje é uma linguagem FULL-STACK, ou seja, podemos usar ela tanto no Front-end, quanto no Back-end.

Pontos Chave do JavaScript:

Interatividade Web: JavaScript é amplamente utilizado para criar interatividade em páginas web. Com ele, é possível construir elementos dinâmicos como menus suspensos, sliders de imagem, validações de formulários, etc.

Execução no Navegador: JavaScript é executado diretamente nos navegadores web. Isso permite que os desenvolvedores adicionem funcionalidades interativas sem a necessidade de comunicação com o servidor.

Ambiente de Execução: Além do navegador, JavaScript pode ser executado em outros ambientes, como servidores (usando Node.js), aplicativos móveis (usando frameworks como React Native) e até mesmo em sistemas de automação de testes.

Versatilidade: JavaScript pode ser usado tanto no lado do cliente (Front-end) quanto no lado do servidor (Back-end), tornando-se uma linguagem de programação completa para desenvolvimento web.

Comunicação com Servidores: JavaScript pode enviar e receber dados de servidores web, permitindo a criação de aplicações web assíncronas. Isso é feito através de APIs como Fetch API.

Manipulação do DOM: JavaScript pode acessar e manipular o Document Object Model (DOM) de uma página web, permitindo a alteração da estrutura, estilo e conteúdo de um documento HTML.

Frameworks e Bibliotecas: A comunidade de JavaScript é grande e ativa, resultando em uma vasta quantidade de frameworks (como Angular e Vue.js) e bibliotecas (como jQuery e Lodash) que facilitam e ampliam as possibilidades de desenvolvimento com JavaScript.

Exemplo:

```

1  <!DOCTYPE html>
2  <html lang="pt-BR">
3  <head>
4      <meta charset="UTF-8">
5      <title>Exemplo usando JavaScript</title>
6      <script>
7          function alterarTexto() {
8              document.getElementById("mensagem").innerHTML = "Olá, Alunos da turma de Java da UTFPR!";
9          }
10     </script>
11 </head>
12 <body>
13     <h1 id="mensagem">Clique no botão para alterar este texto</h1>
14     <button onClick="alterarTexto()">Clique aqui</button>
15 </body>
16 </html>
17
18

```

Acima o documento HTML estrutura a página com um cabeçalho <h1> e um botão <button>.

Implementamos uma função JavaScript chamada “**alterarTexto**” que foi definida para alterar o conteúdo do elemento <h1> com id “**mensagem**” quando o botão é clicado.

O atributo **onclick** no botão é usado para chamar a função JavaScript, demonstrando como o JavaScript pode ser usado para criar interatividade em uma página web.

Esse exemplo ilustra a essência do que o JavaScript pode fazer na web, transformando uma página estática em uma interface dinâmica e interativa.

2. Surgimento do JavaScript

A história do JavaScript se entrelaça com a evolução da web. Desde sua criação até se tornar uma das linguagens de programação mais populares do mundo, o JavaScript passou por várias fases importantes.

Seu criador foi **Brendan Eich**, um programador da Netscape Communications, que criou o JavaScript em 1995.

A ideia era adicionar uma linguagem de script ao navegador Netscape Navigator para tornar as páginas web mais dinâmicas e interativas. Inicialmente, foi chamada de Mocha, depois LiveScript, e finalmente JavaScript.

Sendo assim o JavaScript foi lançado em setembro de 1995 como parte do Netscape Navigator 2.0.

O nome “JavaScript” foi escolhido por razões de marketing, devido à popularidade da linguagem Java na época, apesar de JavaScript e Java serem linguagens bastante diferentes.

Em 1996, a Netscape submeteu JavaScript à ECMA (European Computer Manufacturers Association) para padronização, resultando na especificação ECMAScript.

A primeira edição do ECMAScript (ES1) foi lançada em 1997.

JavaScript rapidamente se tornou a linguagem padrão para scripts em navegadores web, sendo suportada por outros navegadores como Internet Explorer e Opera.

A terceira edição do ECMAScript (ES3) foi lançada em 1999, introduzindo várias melhorias, incluindo suporte para regex, melhor tratamento de exceções e novos métodos para arrays.

Em meados dos anos 2000, a técnica **Ajax (Asynchronous JavaScript and XML)** permitiu a atualização assíncrona de páginas web, sem necessidade de recarregar a página inteira. Isso levou à criação de aplicações web mais dinâmicas e interativas, como Google Maps e Gmail.

A sexta edição do ECMAScript, conhecida como **ES6** ou ECMAScript 2015, foi lançada em 2015. Esta versão trouxe mudanças significativas e novos recursos como classes, módulos, arrow functions, template strings, let e const, e muito mais.

A introdução de módulos ajudou a estruturar o código de maneira mais organizada e reutilizável.

Em 2009 surge o Node.js, que permitiu o JavaScript ser executado no lado do servidor, ampliando ainda mais suas aplicações.

A última década viu o surgimento de frameworks e bibliotecas poderosas como Angular, React, e Vue.js, que revolucionaram o desenvolvimento frontend.

Desde a ES6, a linguagem continua a evoluir com versões anuais (ES7, ES8, etc.), introduzindo melhorias incrementais e novos recursos baseados nas necessidades e feedback da comunidade.

JavaScript é atualmente uma linguagem de programação extremamente popular no mundo, com uma comunidade ativa e um ecossistema robusto.

Além de desenvolvimento web, JavaScript é usado em áreas como desenvolvimento de aplicativos móveis, desenvolvimento de jogos, automação de testes e até mesmo em Internet das Coisas (IoT).

3. Características do JavaScript

LINGUAGEM INTERPRETADA

É uma **linguagem interpretada**, isso significa que o código passa por um interpretador para em seguida ser entendido pela máquina e executar suas funcionalidades.

Sendo assim o JavaScript é interpretado diretamente pelo navegador ou pelo ambiente de execução (como Node.js) sem a necessidade de compilação prévia.

A sua execução é imediata, ou seja, as mudanças no código podem ser rapidamente testadas e vistas em ação, o que acelera o desenvolvimento.

TIPAGEM DINÂMICA

Possui **tipagem dinâmica**, ou seja, as variáveis em JavaScript podem armazenar diferentes tipos de dados ao longo da execução do programa, sem a necessidade de declarar explicitamente o

tipo. Isso proporciona grande flexibilidade, mas também exige cuidados adicionais para evitar erros de tipo.

Se não é preciso expressar a tipagem de dados porque ela é dinâmica, então eu posso ter uma variável com um tipo de dados “number” e depois eu posso usar a mesma variável e atribuir uma “string” e assim por diante.

Também é possível guardar nessa variável uma função, pois em JavaScript uma função é como se fosse um objeto. Portanto as funções são objetos e são tratadas como variáveis, podendo ser passadas como parâmetro também.

Demonstração de tipagem dinâmica:

```
1  var x = 5;
2  x = 'Cinco';
3  x = 5.7;
4  x = function() { console.log('Sou aluno da UTFPR'); }
5
```

ORIENTADO A OBJETO

O JavaScript é basicamente todo **orientado a objeto**, mas não de forma explícita. Por exemplo o **objeto window** é o navegador, já no NodeJs como não temos o navegador, usamos o objeto global, chamado de “**global**”.

O JavaScript é baseado em prototipagem, pois utiliza um modelo baseado em protótipos em vez de classes. Objetos podem herdar diretamente de outros objetos.

Permitindo a criação de objetos dinâmicos que podem ser modificados em tempo de execução.

FUNCIONAL

As funções em JavaScript podem ser atribuídas a variáveis, passadas como argumentos e retornadas por outras funções.

JavaScript também suporta closures, que são funções que lembram o escopo em que foram criadas.

BASEADO EM EVENTOS

Entregando maior interatividade o JavaScript pode responder a eventos como cliques, movimentos do mouse, e entradas de teclado, tornando a web interativa e dinâmica.

LINGUAGEM MULTI-PARADIGMA

O JavaScript suporta estilos de programação **imperativos, funcionais e orientados a objetos**.

Com isso nós desenvolvedores podemos escolher o paradigma que melhor se adapta ao problema que precisamos resolver.

4. Prototype em JavaScript

Em JavaScript, o protótipo (prototype) é um mecanismo pelo qual objetos podem herdar características (propriedades e métodos) de outros objetos. Isso é fundamental para a implementação do modelo de herança.

Funcionamento do Prototype

Cada função em JavaScript tem uma propriedade especial chamada prototype. Quando essa função é usada como um construtor (usando o operador new), os objetos criados terão acesso ao prototype dessa função. Isso permite que todos os objetos criados por um determinado construtor compartilhem as mesmas propriedades e métodos.

O Prototype é parecido com uma classe e criamos ele através de uma função, como no exemplo abaixo:

```
function stark(nome, idade, corDoCabelo) {
  this.nome = nome;
  this.idade = idade;
  this.corDoCabelo = corDoCabelo;

  this.sobrenome = 'Stark';
}

var ned = new stark('ned', 40, 'preto');
var sansa = new stark('sansa', 13, 'cobre');
```

Esses dois objetos acima “ned” e “sansa” herdam o prototype de “stark”.

5. Closures

É uma função que tem acesso ao escopo pai mesmo depois desse escopo ter sido destruído. Na closure encapsulamos algo, mas mantemos o acesso a ela.

No exemplo abaixo, por causa do closure é possível continuar incrementando o conteúdo da variável “valor” mesmo ela tendo sido mantida dentro da “function”, e a variável “valor” sendo mantida dentro da function é uma forma de proteger essa variável logo mais abaixo dela ser atribuído algum valor a ela, **ex: valor = 5001** e ser modificada a variável “valor”.

```
19 function incrementar() {
20   var valor = 0;
21
22   return function() {
23     return ++valor;
24   }
25 }
26
27 var fn = incrementar ();
28 console.log(fn()); // => vai imprimir 1
29 console.log(fn()); // => vai imprimir 2
30 console.log(fn()); // => vai imprimir 3
31
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\Bloco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"

1
2
3

[Done] exited with code=0 in 0.143 seconds

Outro exemplo de closure:

```

34 function saudacao(nome) {
35     const mensagem = 'Olá';
36
37     function saudar() {
38         console.log(`${mensagem}, ${nome}`);
39     }
40
41     return saudar;
42 }
43
44 const saudacaoAlunos = saudacao('Alunos da UTFPR');
45 saudacaoAlunos(); // Saída: Olá, Alunos da UTFPR

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\8loco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"

Olá, Alunos da UTFPR

[Done] exited with code=0 in 0.136 seconds

Neste exemplo, a função “saudar” é um closure. Ela “lembra” das variáveis mensagem e nome, mesmo depois que a função saudacao já tenha sido executada e retornada. Quando chamamos saudacaoAlunos, ele ainda consegue acessar e usar mensagem e nome.

Exemplo de Closure sendo usado para criar função fábrica:

```

51 function criarSaudacao(saudacao) {
52     return function(nome) {
53         console.log(`${saudacao}, ${nome}`);
54     };
55 }
56
57 const saudacaoOi = criarSaudacao('Oi');
58 saudacaoOi('Maria'); // Saída: Oi, Maria
59
60 const saudacaoBomDia = criarSaudacao('Bom dia');
61 saudacaoBomDia('Paulo'); // Saída: Bom dia, Paulo

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\8loco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"

Oi, Maria

Bom dia, Paulo

[Done] exited with code=0 in 0.14 seconds

Dessa forma, no exemplo acima, usamos o closure para gerar outras funções com um comportamento específico.

Exemplo de closure sendo usada para manipular variáveis privadas:

```

64
65 function criarBanco() {
66     let saldo = 0;
67
68     return {
69         depositar: function(valor) {
70             saldo += valor;
71             console.log('Depositado: R${valor}. Saldo atual: R${saldo}');
72         },
73         sacar: function(valor) {
74             if (valor <= saldo) {
75                 saldo -= valor;
76                 console.log('Sacado: R${valor}. Saldo atual: R${saldo}');
77             } else {
78                 console.log('Saldo insuficiente.');

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE



[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\BLoco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"



Depositado: R$100. Saldo atual: R$100



Sacado: R$50. Saldo atual: R$50



Saldo atual: R$50



[Done] exited with code=0 in 0.141 seconds


```

No exemplo acima, saldo é uma variável privada que só pode ser acessada e modificada através dos métodos “depositar”, “sacar” e “verSaldo” retornados pelo closure.

6. JavaScript e Paradigmas

Um **paradigma de programação** é um estilo ou "modo" de escrever programas. Linguagens de programação que suportam múltiplos paradigmas permitem aos desenvolvedores escolherem diferentes estilos de programação dependendo do problema que estão tentando resolver. JavaScript é considerada uma linguagem multiparadigma porque suporta diversos paradigmas de programação.

IMPERATIVO

O paradigma imperativo foca em descrever como um programa opera, utilizando declarações que mudam o estado do programa. Ele envolve comandos que dizem ao computador o que fazer passo a passo.


```

94 let total = 0;
95 for (let i = 0; i < 10; i++) {
96     total += i;
97 }
98 console.log(total); // Saída: 45
99
100

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"

45

[Done] exited with code=0 in 0.132 seconds

No exemplo acima, estamos descrevendo explicitamente o que o programa deve fazer para calcular a soma dos primeiros 10 números.

ESTRUTURADO

A programação estruturada é um subconjunto do paradigma imperativo que utiliza estruturas de controle, como loops (for, while), condicionais (if, else). Digamos que a ideia principal é dividir o programa em blocos que podem ser facilmente compreendidos e mantidos, colocando dentro de funções por exemplo.

```

101
102 function calcularFatorial(n) {
103     if (n ≤ 1) return 1;
104     return n * calcularFatorial(n - 1);
105 }
106
107 console.log(calcularFatorial(5)); // Saída: 120
108

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"

120

[Done] exited with code=0 in 0.142 seconds

No exemplo acima, usamos uma função recursiva e uma estrutura de controle (if), para calcular o fatorial de um número.

BASEADO EM OBJETOS

A programação baseada em objetos (também chamada de orientação a objetos) organiza o código em torno de objetos, que são instâncias de classes. JavaScript implementa orientação a objetos através de protótipos.

```

111 class Pessoa {
112     constructor(nome, idade) {
113         this.nome = nome;
114         this.idade = idade;
115     }
116
117     cumprimentar() {
118         console.log(`Olá, meu nome é ${this.nome} e eu tenho ${this.idade} anos.`);
119     }
120 }
121
122 const pessoa = new Pessoa('Fulano', 25);
123 pessoa.cumprimentar(); // Saída: Olá, meu nome é João e eu tenho 25 anos.
124
125

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\BLoco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"

Olá, meu nome é Fulano e eu tenho 25 anos.

[Done] exited with code=0 in 0.141 seconds

No exemplo acima, “Pessoa” é uma classe que define um modelo para objetos pessoas. Cada instância da classe “Pessoa” tem suas próprias propriedades (nome e idade) e métodos (cumprimentar).

FUNCIONAL

O paradigma funcional é o processo de construir software através de composição de funções puras, evitando compartilhamento de estados, dados mutáveis e efeitos colaterais. É declarativa ao invés de Imperativa.

```

127 const numeros = [1, 2, 3, 4, 5];
128 const quadrados = numeros.map(num => num * num);
129 console.log(quadrados); // Saída: [1, 4, 9, 16, 25]

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\BLoco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"

[1, 4, 9, 16, 25]

[Done] exited with code=0 in 0.151 seconds

No exemplo acima, usamos a função “map” para aplicar uma transformação (elevar ao quadrado) a cada elemento do array “números”.

VANTAGENS DE UMA LINGUAGEM MULTIPARADIGMA

- **Flexibilidade:** Os desenvolvedores podem escolher o paradigma que melhor se adequa ao problema que estão tentando resolver.
- **Reutilização de Código:** Diferentes abordagens e padrões de projeto podem ser aplicados, facilitando a reutilização de código.
- **Facilidade de Manutenção:** A capacidade de escolher diferentes paradigmas permite escrever código mais limpo e modular, facilitando a manutenção e evolução do software.

JavaScript ser uma linguagem multiparadigma oferece uma grande flexibilidade aos desenvolvedores, permitindo-lhes escolher o estilo de programação mais adequado para cada situação. Com suporte para paradigmas imperativo, estruturado, orientado a objetos e funcional, JavaScript se adapta bem a uma variedade de cenários de desenvolvimento.

7. O que é ES6

ES6, também conhecido como **ECMAScript 2015** ou **ECMAScript 6**, é a sexta edição da especificação ECMAScript, que é a base do JavaScript. Publicada em junho de 2015 pela Ecma International, ES6 trouxe uma série de melhorias significativas e novas funcionalidades para a linguagem JavaScript, tornando-a mais poderosa e eficiente para o desenvolvimento de aplicações modernas.

PRINCIPAIS CARACTERÍSTICAS E FUNCIONALIDADES INTRODUZIDAS PELO ES6

LET E CONST

Não é uma boa prática usar o “var” para definir variáveis. Ao invés usamos “let” e “const”.

Outra questão importante sobre o “var” é que ele não respeita o escopo de bloco, por exemplo se eu declarar uma variável usando o var dentro de um “for”, quando sair do for ela deveria deixar de existir, mas se usar o var para declarar essa variável, eu posso imprimir o valor dela por exemplo depois que o for acabou.

A não ser que essa variável seja declarada com o “var” dentro de uma função, daí o escopo dela ficará restrito para aquela função apenas.

Se usamos o “let” por exemplo, e tentamos acessar a variável declarada dentro do “for”, ele retorna uma “exception”.

Outro ponto do “var” por exemplo, é que ele permite eu redeclarar a variável novamente dentro de um mesmo escopo e não dá erro, se eu usar o “let” ele daria erro se eu tentasse redeclarar ela novamente. Se puder escolher, sempre usar o “LET” ao invés do “VAR”.

- **let:** Permite a declaração de variáveis com escopo de bloco, o que evita problemas de escopo que ocorrem com “var”, conforme mencionado acima. Pois o escopo de bloco só é acessível dentro de onde a variável está declarada.

```

132
133 let x = 10;
134 if (x === 10) {
135     let x = 20; // x dentro do bloco
136     console.log(x); // 20
137 }
138 console.log(x); // 10
139
140

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\Bloco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"

20

10

[Done] exited with code=0 in 0.147 seconds

- **const:** Declara constantes, valores que não podem ser reatribuídos.

```

142 const y = 30;
143 // y = 40; // Isso causará um erro
144 console.log(y); // 30
145
146

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"

30

[Done] exited with code=0 in 0.144 seconds

Observando o exemplo acima, se eu retirar o comentário da linha “143” que contém uma instrução que muda o valor de “y”, dará erro, pois “y” é uma constante. Observe no exemplo abaixo.

```

141
142 const y = 30;
143 y = 40; // Isso causará um erro
144 console.log(y); // 30
145
146

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"

d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js:2

y = 40; // Isso causará um erro

^

TypeError: Assignment to constant variable.

at Object.<anonymous> (d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js:2:3)

at Module._compile (node:internal/modules/cjs/loader:1097:14)

at Object.Module._extensions..js (node:internal/modules/cjs/loader:1149:10)

at Module.load (node:internal/modules/cjs/loader:975:32)

at Function.Module._load (node:internal/modules/cjs/loader:822:12)

at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)

at node:internal/main/run_main_module:17:47

Node.js v17.2.0

[Done] exited with code=1 in 0.138 seconds

ARROW FUNCTIONS

Funções de seta (arrow functions) são uma sintaxe mais concisa para escrever funções e não vinculam seu próprio this.

São consideradas funções anônimas.

```

147
148 const add = (a, b) => a + b;
149 console.log(add(2, 3)); // 5
150

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"

5

[Done] exited with code=0 in 0.151 seconds

No exemplo acima, não foi preciso colocar o “return”, pois quanto omitimos as “chaves”, o retorno é implícito.

TEMPLATE LITERALS (TEMPLATE STRINGS)

Permitem a criação de “strings multilinha” e a “interpolação” de variáveis de forma mais fácil. Para evitar o uso tradicional da concatenação com o uso de “+” e acesso de variáveis que fica bastante verboso no código, existe o “Template String” que são strings que suportam expressões embutidas e facilitam atividades de concatenação e definição de textos com múltiplas linhas, até

facilitando também a questão de quebra de linha com “\n”, no string template não precisamos nos preocupar com isso, bastando usar um “enter” no lugar de concatenar o “\n”.

Abaixo seguem dois exemplos:

```
152
153 const person = 'Jorge';
154 const greeting = `Olá, ${person}!`;
155 console.log(greeting); // Olá, Jorge!
156
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\Bloco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"

Olá, Jorge!

[Done] exited with code=0 in 0.138 seconds

```
158
159 const cachorro = 'Fofão';
160 const saudacaoCanina = `Olá, meu nome é:
161 ${cachorro}!`;
162 console.log(saudacaoCanina);
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\Bloco02_javascript-typescript\exemplo01-apostila\tempCodeRunnerFile.js"

Olá, meu nome é:
Fofão!

[Done] exited with code=0 in 0.143 seconds

DESESTRUTURAÇÃO

Através da desestruturação é possível entregar o valor para uma função sem especificar parâmetro por parâmetro, pois o Javascript entende automaticamente qual campo é qual. Permitindo extrair dados de arrays ou objetos em variáveis distintas também.

```
1 const fruits = ['banana', 'morango', 'manga'];
2 const [firstFruit, secondFruit] = fruits;
3
4 console.log(fruits);
5 console.log(firstFruit);
6 console.log(secondFruit);
7
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\Bloco02_javascript-typescript\exemplo02-apostila\tempCodeRunnerFile.js"

['banana', 'morango', 'manga']
banana
morango

[Done] exited with code=0 in 0.165 seconds

No exemplo acima, estamos aplicando a desestruturação retirando elementos de um vetor. Nesse caso o JavaScript entende que o primeiro elemento do array “fruits” deve ser colocado na primeira constante, que é chamada de “firstFruit”. E o segundo elemento deve ser colocado na próxima constante que é chamada de “secondFruit”. Isso é um exemplo de desestruturação.

```

1  const fruits = ['banana', 'morango', 'manga'];
2  const [ , secondFruit] = fruits;
3
4  console.log(fruits);
5  console.log(secondFruit);
6
7

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\8Loco02_javascript-typescript\exemplo02-apostila\tempCodeRunnerFile.js"

['banana', 'morango', 'manga']
morango

[Done] exited with code=0 in 0.137 seconds

No exemplo acima, estamos desestruturando um array, pegando apenas o segundo valor “morango” e colocando na variável “secondFruit”. Para isso eu não devo colocar nada como primeiro item da desestruturação e deixo apenas a vírgula para isso, conforme demonstrada na linha 2 do exemplo acima.

```

1  const pessoa = {
2    nome: 'Sérgio',
3    idade: 60
4  }
5
6  const { nome } = pessoa;
7
8  console.log(pessoa);
9  console.log(nome);
10

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\8Loco02_javascript-typescript\exemplo02-apostila\tempCodeRunnerFile.js"

{ nome: 'Sérgio', idade: 60 }
Sérgio

[Done] exited with code=0 in 0.181 seconds

No exemplo acima nós estamos dizendo que é para criar uma constante “nome” e colocar dentro dela o que está no atributo “nome” que está no objeto “pessoa” fazendo assim a desestruturação desse objeto.

E se eu quisesse desestruturar o objeto “pessoa” pegando também o atributo “idade” eu devo fazer da seguinte forma: **const { nome, idade } = pessoa;**

```

1  const pessoa = {
2    nome: 'Sérgio',
3    idade: 60,
4    contato: {
5      email: 'sergio@teste.com'
6    }
7  }
8
9  const { nome, contato:{ email } } = pessoa;
10
11 console.log(pessoa);
12 console.log(nome);
13 console.log(email);
14

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\8Loco02_javascript-typescript\exemplo02-apostila\tempCodeRunnerFile.js"

{ nome: 'Sérgio', idade: 60, contato: { email: 'sergio@teste.com' } }
Sérgio
sergio@teste.com

[Done] exited with code=0 in 0.139 seconds

No exemplo acima, para pegar a propriedade “email” que está dentro de “contato” eu tenho de usar o “contato: { email }”

```
1  const pessoa = {
2    nome: 'Sérgio',
3    idade: 60,
4    contato: {
5      email: 'sergio@teste.com'
6    }
7  }
8
9  const { nome: batata } = pessoa;
10
11 console.log(pessoa);
12 console.log(batata);
13
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\8Loco02_javascript-typescript\exemplo02-apostila\tempCodeRunnerFile.js"
{ nome: 'Sérgio', idade: 60, contato: { email: 'sergio@teste.com' } }
Sérgio

[Done] exited with code=0 in 0.14 seconds

No exemplo acima nós estamos desestruturando dizendo que a propriedade “nome” ficará guardada dentro da variável “batata”.

```
1  const pessoa = {
2    nome: 'Sérgio',
3    idade: 60,
4    contato: {
5      email: 'sergio@teste.com'
6    }
7  }
8
9  const { cpf } = pessoa;
10
11 console.log(pessoa);
12 console.log(cpf);
13
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\8Loco02_javascript-typescript\exemplo02-apostila\tempCodeRunnerFile.js"
{ nome: 'Sérgio', idade: 60, contato: { email: 'sergio@teste.com' } }
undefined

[Done] exited with code=0 in 0.149 seconds

Se eu tento desestruturar um objeto tentando acessar uma propriedade que não existe no objeto, como por exemplo no objeto “pessoa” acima, se eu tento desestruturar uma propriedade “cpf” que não existe, ele retorna “undefined”.

```

1  const pessoa = {
2    nome: 'Manoel',
3    idade: 60
4  }
5
6  const pessoa2 = {
7    nome: 'Roberto',
8    idade: 62
9  }
10
11  const pessoa3 = {
12    nome: 'Adriano',
13    idade: 55
14  }
15
16  const amigos = [ pessoa, pessoa2, pessoa3 ];
17
18  const [ , {nome}] = amigos;
19
20  console.log(pessoa);
21  console.log(pessoa2);
22  console.log(pessoa3);
23  console.log(nome);
24

```

Comments

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\Bloco02_javascript-typescript\exemplo02-apostila\index7.js"

```

{ nome: 'Manoel', idade: 60 }
{ nome: 'Roberto', idade: 62 }
{ nome: 'Adriano', idade: 55 }
Roberto

```

[Done] exited with code=0 in 0.15 seconds

No exemplo acima, estou recuperando através da desestruturação do array apenas a propriedade “nome” do segundo objeto array de “amigos”.

```

1  function criarUsuario (nome, idade, email) {
2    return {
3      nome,
4      idade,
5      contato: { email }
6    }
7  }
8
9  const { nome } = criarUsuario('Fabio', 45, 'fabio@teste.com');
10
11  console.log(nome);
12

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\Bloco02_javascript-typescript\exemplo02-apostila\index8.js"

```

Fabio

```

[Done] exited with code=0 in 0.14 seconds

Se observarmos o exemplo acima, perceberemos que o retorno da função “criarUsuario” é um objeto. E objetos são passíveis de desestruturação, sendo assim, podemos fazer como consta na linha “9” desestruturando o objeto retornado pegando o “nome”. Com isso, podemos dizer que estamos desestruturando uma função.

DEFINIR VALORES PADRÃO

É possível definir um valor padrão para os parâmetros de funções, conforme no exemplo abaixo podemos observar que se não passamos o valor para a variável “b”, ele assume que é para usar

o valor 1 na multiplicação, caso contrário se for passado um valor para “b” ele multiplica pelo valor informado no parâmetro.

```
1 function multiplique(a, b = 1) {  
2   return a * b;  
3 }  
4 console.log(multiplique(5)); // 5  
5 console.log(multiplique(5, 2)); // 10  
6
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo02-apostila\index9.js"

5
10

[Done] exited with code=0 in 0.145 seconds

SPREAD OPERATOR

Expande elementos de um array ou objeto. Ele basicamente é usado para pegar todas as propriedades de um objeto e depois sobrescrever uma propriedade específica por outra nova que podemos passar.

```
1 const partes = ['ombro', 'joelho', 'orelha'];  
2  
3 const corpo = ['cabeça', partes[0], partes[1], partes[2], 'pés'];  
4 console.log(corpo);  
5  
6 const corpoComSpread = ['cabeça', ...partes, 'pés'];  
7 console.log(corpoComSpread);  
8
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo02-apostila\index10.js"

['cabeça', 'ombro', 'joelho', 'orelha', 'pés']
['cabeça', 'ombro', 'joelho', 'orelha', 'pés']

[Done] exited with code=0 in 0.138 seconds

No exemplo acima, nós temos a variável “corpo” que foi gerada acessando posição por posição do array “partes” na linha “3”.

Já na linha “6” podemos observar o uso do “spread operator” que atua desestruturando o “partes” e inserindo como item a item no novo array chamado “corpoComSpread”.

```

1 // SEM SPREAD OPERATOR
2 function createUser (name, age, contact1, contact2, contact3) {
3     return {
4         name,
5         age,
6         contacts: [contact1, contact2, contact3]
7     }
8 }
9
10 // COM SPREAD OPERATOR
11 function createUserSpread (name, age, ...contacts) {
12     return {
13         name,
14         age,
15         contacts
16     }
17 }
18
19 const usuarioA = createUser('Fabio', 45, {email: 'fabio@teste.com'}, {fone: '1234-1234'}, {whatsapp: '4321-4321'});
20
21 const usuarioB = createUserSpread('Fabio', 45, {email: 'fabio@teste.com'}, {fone: '1234-1234'}, {whatsapp: '4321-4321'});
22
23 console.log(usuarioA);
24 console.log(usuarioB);
25

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

```

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo02-apostila\index11.js"
{
  name: 'Fabio',
  age: 45,
  contacts: [
    { email: 'fabio@teste.com' },
    { fone: '1234-1234' },
    { whatsapp: '4321-4321' }
  ]
}
{
  name: 'Fabio',
  age: 45,
  contacts: [
    { email: 'fabio@teste.com' },
    { fone: '1234-1234' },
    { whatsapp: '4321-4321' }
  ]
}

[Done] exited with code=0 in 0.145 seconds

```

No exemplo acima, usamos o spread operator na passagem de parâmetro de uma função. Com isso podemos observar que foi possível evitar a declaração de diversas variáveis no parâmetro da função.

```

1 const usuario = {
2     nome: 'hugobaker',
3     email: 'hugobaker@teste.com'
4 };
5
6 const novoUsuario = {...usuario, regra: 'admin'};
7
8 console.log(usuario);
9 console.log(novoUsuario);
10

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

```

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo02-apostila\index12.js"
{ nome: 'hugobaker', email: 'hugobaker@teste.com' }
{ nome: 'hugobaker', email: 'hugobaker@teste.com', regra: 'admin' }

[Done] exited with code=0 in 0.135 seconds

```

No exemplo acima foi usado o Spread Operator para **clonar um objeto**.

Da forma que foi feito acima é como se o Spread Operator (...) retirasse as chaves do objeto “usuário”, trazendo apenas o conteúdo dos atributos do objeto. Com isso é possível adicionar os atributos no objeto “novoUsuario” que estará apontando para outro endereço de memória diferente do endereço de memória que está o objeto “usuário”.

Dessa forma podemos dizer que é feito o clone do objeto, em outro objeto em memória.

```

1  const partes = ['ombro', 'joelho'];
2  const corpo = ['cabeça', 'pé'];
3
4  const corpoCompleto = ['cabeça', ...partes, 'pé'];
5  console.log(corpoCompleto);
6
7  // abaixo, tentativa de juntar errada
8  const corpo2 = ['cabeça', partes, 'pé'];
9  console.log(corpo2);
10

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\8Loco02_javascript-typescript\exemplo02-apostila\index13.js"

['cabeça', 'ombro', 'joelho', 'pé']

['cabeça', ['ombro', 'joelho'], 'pé']

[Done] exited with code=0 in 0.349 seconds

Observe o uso do Spread Operator com arrays no exemplo acima. Para juntar os arrays “partes” e “corpo” nós usamos o spread operator na linha “4” e imprimimos o resultado no console na linha “5” e deu certo, como resultado temos um array contendo dois novos itens, ou seja, contendo cada item do array “partes” dentro agora do array “corpo”.

Para testar outra maneira, na linha “8” criei o array “corpo2” juntando os itens do array “partes” e do array “corpo”, mas simplesmente incluindo o array “partes” sem o spread operator. E como resultado podemos ver na linha “9” ao imprimir o array “corpo2” que temos um array, dentro de outro array, diferente do que gostaríamos que acontecesse nesse caso.

```

1  const ingredientesSemSpread = ['ovo', 'presunto'];
2  const temperos = ['sal', 'pimenta'];
3
4  ingredientesSemSpread.push(temperos);
5  console.log(ingredientesSemSpread);
6
7  const ingredientesComSpread = ['ovo', 'presunto'];
8  ingredientesComSpread.push(...temperos);
9  console.log(ingredientesComSpread);
10

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\8Loco02_javascript-typescript\exemplo02-apostila\index14.js"

['ovo', 'presunto', ['sal', 'pimenta']]

['ovo', 'presunto', 'sal', 'pimenta']

[Done] exited with code=0 in 0.143 seconds

No exemplo acima nós aplicamos o spread operator em uma função, no caso a função “push” que inclui novos itens no array. E para exemplificar usei o array “ingredientesSemSpread” para mostrar o uso do push incluindo o array “temperos” na linha “4” e é possível observar que temos como resultado um array dentro de outro array se não usarmos o spread operator.

Com o novo exemplo na linha “8” ao aplicarmos o spread operator temos como resultado o array “ingredientesComSpread” imprimindo o array com os dois novos itens incluídos corretamente.

REST OPERATOR

É muito parecido com o spread operator, só que basicamente o REST OPERATOR é a operação inversa do SPREAD.

E ambas as operações são feitas usando os três pontinhos “...”

```

1 function calcular(subtrairItem, multiplicarItem, ...itensSomar) {
2   let total = 0;
3   for (const item of itensSomar) {
4     total += item;
5   }
6   return (total - subtrairItem) * multiplicarItem;
7 }
8
9 console.log(calcular(1, 2, 3, 4));
10
11 console.log(calcular(1, 2, 3, 4, 5, 6, 7, 8));
12
13

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\Bloco02_javascript-typescript\exemplo02-apostila\index15.js"

12
64

[Done] exited with code=0 in 0.18 seconds

No exemplo acima o rest operator junta todos os itens passados como parâmetros após o parâmetro “multiplicarItem” colocando em um array de “itensSomar”, agrupando cada um deles como um novo item desse array.

Podemos observar na linha “9” que foram passados dois números a somar “3” e “4”. Já no exemplo da linha “11” foram passados 6 itens para somar “3”, “4”, “5”, “6”, “7” e “8”.

```

1 const frutas = ['banana', 'morango', 'manga', 'kiwi', 'maracujá'];
2 const [primeiroItem, segundoItem, ...outrosItens] = frutas;
3
4 console.log(primeiroItem);
5 console.log(segundoItem);
6 console.log(outrosItens);

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\Bloco02_javascript-typescript\exemplo02-apostila\index16.js"

banana
morango
['manga', 'kiwi', 'maracujá']

[Done] exited with code=0 in 0.131 seconds

No exemplo acima nós estamos pegando o primeiro elemento “banana” e guardando na variável “primeiroItem”, depois pegamos o segundo elemento “morando” e guardamos no “segundoItem”, até aqui tivemos o processo de desestruturação, que pega dois primeiros itens do array de frutas.

Depois é feito um agrupamento usando o **rest operator**, onde pegamos os demais itens do array de “frutas” e guardamos dentro do array “outrosItens”.

É importante ressaltar que o **rest operator** deve ser o último item a ser aplicado na relação de variáveis, pois ele representa o RESTO DOS ELEMENTOS.

CLASSES

A partir da versão ES6, temos o suporte de classes. Similar a java, mas por debaixo dos panos são tratadas como funções construtoras do ES5, para se tornar compatível e rodar com a maioria dos navegadores.

```

1  class Pessoa { // assinatura da classe
2
3      constructor(nome) { // construtor da classe
4          this.nome = nome;
5      }
6
7      saudacao() { // método saudacao da classe
8          console.log('Olá, meu nome é ${this.nome}');
9      }
10
11 }
12 const hugo = new Pessoa("Hugo");
13 hugo.saudacao();
14

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo02-apostila\index17.js"
Olá, meu nome é Hugo

[Done] exited with code=0 in 0.147 seconds

Acima podemos observar a implementação de um exemplo de classe. Onde temos um construtor e um método “saudação”, mas que vale observar que o método é definido como sendo uma função, mas sem a palavra **function**.

Os atributos são referenciados obrigatoriamente por meio do **this**.

O **método construtor** é executado toda vez que instanciamos nossa classe.

É importante ressaltar que em JavaScript não temos o conceito de private, protected como temos em Java, todos os atributos são públicos.

MODULES

Permitem a importação e exportação de partes de código, facilitando a organização e reutilização de código.

```

// no arquivo calculadora.js
export function somar(a, b) {
    return a + b;
}

// em outro arquivo
import { somar } from './calculadora';
console.log(somar(2, 3)); // 5

```

No exemplo acima a função somar é exportada de dentro de um arquivo chamado “calculadora.js” e depois em outro arquivo de nosso projeto podemos fazer uso da função somar ao realizar um “import” dessa função, com isso ela fica disponível para ser usada dentro desse outro arquivo.

PROMISES

Uma maneira de trabalhar com operações assíncronas de forma mais legível e gerenciável.

```

1  const promise = new Promise((resolve, reject) => {
2    setTimeout(() => resolve("Feito!"), 3000);
3  });
4
5  promise.then(result => {
6    console.log(result); // Feito!
7  });
8
9

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo02-apostila\index19.js"

Feito!

[Done] exited with code=0 in 3.169 seconds

Para exemplificar o uso do Promise, temos o exemplo acima aplicando um “delay” para aguardar a resolução do que foi pedido. Nesse caso é feito um retorno da string “Feito!” para essa Promise. Depois aplicamos o “then” que é executado após a resolução dessa Promise, imprimindo o que é fruto do resultado dela.

Ao executar esse código vocês poderão observar que o “then” só acontece após o tempo definido no setTimeout acabar. É uma simulação de que o Promise nos permite trabalhar de forma assíncrona.

Uma Promise representa um valor que pode estar disponível agora, depois ou nunca. Essa Promise pode ser **resolvida(resolve)** ou **rejeitada(reject)**.

Assincronismo: Um código que é executado sem bloquear a execução principal do programa é denominado um código assíncrono.

E assim a execução do programa não fica “travado” esperando aquela ação acontecer, ele continua sendo executado em paralelo com o código assíncrono, como se fossem as threads em Java.

```

1  import fetch from 'node-fetch'; 92.5k (gzipped: 25.2k)
2
3  const resposta = fetch('http://api.postmon.com.br/v1/cep/80230901');
4
5  resposta.then(data => data.json())
6    .then(data => { console.log(data) })
7    .catch(erro => { console.log(erro) });
8

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\B loco02_javascript-typescript\exemplo02-apostila\index20.js"

```

{
  bairro: 'Centro',
  cidade: 'Curitiba',
  logradouro: 'Avenida Sete de Setembro, 3165',
  estado_info: { area_km2: '199.307,985', codigo_ibge: '41', nome: 'Paraná' },
  cep: '80230901',
  cidade_info: { area_km2: '435,036', codigo_ibge: '4106902' },
  estado: 'PR'
}

```

[Done] exited with code=0 in 0.382 seconds

No exemplo acima, após o fetch retornar na variável “resposta” os dados do endereço da sede da UTFPR, o “then” é acionado, mas ele só é executado quando chegar o retorno.

O **catch** é usado para tratar o erro da função, se observarmos no final é adicionado o **catch**, que faz a tratativa do erro, pois retornando algum erro o fluxo cai no **catch**.

Status de uma promise

Uma promise pode ter o status de **pendente(pending)**, que é uma promise que ainda está sendo resolvida.

Pode ter o status **resolvida(fulfilled)**, que é quando ela é executada com sucesso.

Pode também ter o status **rejeitada(rejected)**, que é quando ela foi executada e houve um erro na sua execução.

Dentro do Javascript temos o objeto global chamado Promise que recebe uma função de callback que recebe dois parâmetros **resolve** e **reject**.

Entendendo o recebimento de valor de uma Promise com “Then” e “Catch”

Normalmente encapsulamos as **promises** dentro de uma função para que elas não sejam resolvidas ao mesmo tempo ou de forma desordenada que atrapalhe o fluxo do nosso código.

Dessa forma a **promise** do exemplo abaixo irá ser executada somente quando invocarmos o método **handleClick**.

```
1 function handleClick() {
2   return new Promise((resolve, reject) => {
3     setTimeout(() => {
4       resolve(123)
5     }, 5000);
6   });
7 }
8
```

Ao aplicar no exemplo acima o método “then”, ele só é executado quando a nossa promise for resolvida. Observemos no código abaixo:

```
1 function handleClick() {
2   return new Promise((resolve, reject) => {
3     setTimeout(() => {
4       resolve(123)
5     }, 5000);
6   });
7 }
8
9 const result = handleClick()
10 .then( res => {
11   console.log( 123 === res );
12   console.log( 'finalizou' );
13   if (res === 123) {
14     return 'Oba! deu certo...';
15   } else return 'Xiii, falhou...';
16 })
17 .catch( err => {
18   console.log('Tivemos o erro abaixo:');
19   console.log( err );
20 });
```

É importante notarmos que o valor de retorno que temos dentro do “then”, é o valor de retorno do “resolve” do nosso **promise**.

Ele checa se o “res === 123”(res igual a 123) é verdadeiro, se for ele retorna “Oba! Deu certo...” e caso contrário ele retorna “Xiii, falhou...”

No exemplo acima, temos o retorno do **then** sendo definido pelo tipo de retorno do resolve no nosso promise.

E na sequência temos o **catch** que é executado quando a promise retorna rejeitado, daí nós temos ali uma instrução que captura o erro e imprime no console por exemplo.

OBJETOS LITERAIS APRIMORADOS (Enhanced Object Literals)

A notação literal é a maneira mais simples(e recomendável) de se criar objetos em JavaScript. Um objeto literal é composto por um par de chaves “{ }”, que envolve uma ou mais propriedades. Cada propriedade segue o formato “nome: valor” e devem ser separadas por vírgula.

Exemplo:

```

1  var album = {
2      title: "Metallica (Black Album)",
3      released: 1991,
4      showInfo: function() {
5          console.log(`Título do álbum: ${this.title} - Lançado em: ${this.released}`);
6      }
7  };
8
9  console.log(album);
10 album.showInfo();
11

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\Bloco02_javascript-typescript\exemplo02-apostila\index22.js"

```

{
  title: 'Metallica (Black Album)',
  released: 1991,
  showInfo: [Function: showInfo]
}
Título do álbum: Metallica (Black Album) - Lançado em: 1991
[Done] exited with code=0 in 0.126 seconds

```

Quando declaramos um objeto literal nós temos apenas uma única instância, ou seja, não conseguimos gerar várias outras instâncias dele porque não temos aí nessa ação a aplicação do conceito de classes.

Após ES6

Entendido o que é objeto literal, é preciso saber que após a versão ES6 foram adicionadas várias melhorias e conveniências à sintaxe de criação de objetos literais em JavaScript.

ES6 representa um avanço significativo para o JavaScript, trazendo uma série de recursos que tornam a linguagem mais moderna e poderosa.

Esses recursos não só melhoram a legibilidade e a manutenção do código, mas também introduzem novas formas de lidar com problemas comuns no desenvolvimento de software.

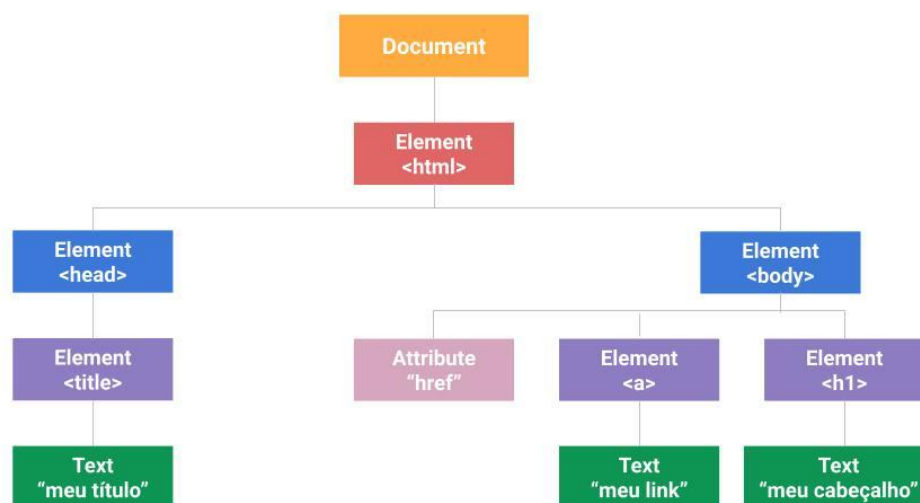
8. O que podemos fazer com JavaScript?

Atualmente podemos fazer uma aplicação completa tanto o Frontend quando Backend com Javascript. Ele é bem versátil e amplamente usado no desenvolvimento web.

DESENVOLVIMENTO FRONT-END

JavaScript é a linguagem de programação essencial para o desenvolvimento Front-end, permitindo a criação de interfaces de usuário interativas e dinâmicas.

- Permite **injetar lógica** em páginas escritas em HTML e adicionar interatividade nas páginas HTML
- O JavaScript pode **ler e escrever HTML** dentro de uma página, alterando o conteúdo do documento exibido, no caso ele é usado para manipular o DOM(Document Object Model)



Quando o navegador interpreta o HTML ele gera um objeto na memória do navegador para cada “tag” do código HTML e o JavaScript manipula isso, o DOM (que é a árvore de objetos)

Então ao manipular o DOM ele manipula os objetos que estão na memória do browser.

- Ele pode escutar um **evento** ocorrer para disparar uma ação em resposta (Listener)
- O JavaScript pode ser usado para **validar informações** antes que elas sejam enviadas ao servidor
- Permite **comunicação assíncrona** com o servidor
- O JavaScript pode ser usado para desenvolver **aplicações completas na web**.

DESENVOLVIMENTO BACK-END

JavaScript também pode ser usado no lado do servidor, principalmente com o ambiente de execução Node.js. Isso permite criar aplicações de servidor escaláveis e de alto desempenho.

9. Variáveis e Tipos de Dados em JavaScript

Em JavaScript, como em outras linguagens de programação, as variáveis são usadas para armazenar dados que podem ser manipulados e referenciados ao longo da execução do programa. Existem três palavras-chave principais para declarar variáveis, conforme já visto ao longo deste material: `var`, `let`, e `const`.

var: Tem escopo de função e pode ser redeclarada e reatribuída.

let: Tem escopo de bloco e pode ser reatribuída, mas não redeclarada no mesmo escopo.

const: Tem escopo de bloco e não pode ser reatribuída nem redeclarada.

TIPOS DE DADOS

JavaScript possui vários tipos de dados primitivos e um tipo complexo (que é o objeto).

- **Número (Number):** Representa tanto números inteiros quanto de ponto flutuante.

```
1 let idade = 30;  
2 let preco = 99.99;  
3
```

- **String:** Representa uma sequência de caracteres.

```
let nome = "João";  
let mensagem = 'Olá, Mundo!';
```

- **Boolean:** Representa um valor lógico: `true` ou `false`.

```
let isActive = true;  
let isCompleted = false;
```

- **Undefined:** Representa uma variável que foi declarada, mas ainda não recebeu um valor.

```
let indefinido;  
console.log(indefinido); // undefined
```

- **Null:** Representa a ausência intencional de valor.

Ex: `let vazio = null;`

- **Object:** Representa uma coleção de propriedades, que são pares de chave e valor.

```
let pessoa = {  
  nome: "João",  
  idade: 30  
};  
  
console.log(pessoa.nome); // João
```

10. NULL vs UNDEFINED

Em JavaScript, `null` e `undefined` são dois valores especiais que são frequentemente confundidos. Ambos representam a ausência de valor, mas são utilizados em contextos diferentes. Entender as distinções entre esses dois valores é fundamental para escrever código JavaScript robusto e evitar bugs.

UNDEFINED

O valor **undefined** indica que uma variável foi declarada, mas não foi inicializada com nenhum valor. Isso também pode ocorrer quando se tenta acessar uma propriedade ou variável que não existe.

Portanto **undefined** é um tipo de dados propriamente dito.

Situações onde encontramos **undefined**:

- Variáveis não inicializadas

```
let x;
console.log(x); // undefined
```

- Propriedades não existentes

```
let obj = {};
console.log(obj.prop); // undefined
```

- Valores de retorno de funções sem retorno explícito

```
function foo() {}
console.log(foo()); // undefined
```

- Parâmetros de funções que não foram passados

```
function bar(param) {
  console.log(param);
}
bar(); // undefined
```

- Propriedade não definida em um objeto

```
let person = { name: "John" };
console.log(person.age); // undefined
```

Quando usar undefined?

JavaScript automaticamente atribui **undefined** a variáveis não inicializadas e propriedades inexistentes. Geralmente, você não deve atribuir **undefined** a uma variável; em vez disso, deixe que o JavaScript faça isso automaticamente.

NULL

O valor **null** é um valor especial que representa a ausência intencional de qualquer valor de objeto. Em outras palavras, **null** é usado quando queremos intencionalmente indicar que uma variável não deve ter valor.

Portanto o **null** é considerado um **valor** especial de objeto.

Quando usar null?

Use **null** quando quiser explicitamente indicar que uma variável ou propriedade não deve ter valor.

11. ARRAYS

Em JavaScript, um array é uma estrutura de dados que permite armazenar uma coleção de elementos, que podem ser de qualquer tipo (números, strings, objetos, etc.). Arrays são dinâmicos e podem crescer ou diminuir conforme necessário.

Declaração e Inicialização de Arrays

Você pode criar um array em JavaScript usando a sintaxe de colchetes ([]) ou o construtor Array.

Sintaxe de colchetes: `let frutas = ['maçã', 'banana', 'laranja'];`

Construtor Array: `let numeros = new Array(1, 2, 3, 4, 5);`

Acessando Elementos de um Array

Os elementos de um array são acessados usando índices, que começam em 0.

```
let frutas = ['maçã', 'banana', 'laranja'];  
console.log(frutas[0]); // maçã  
console.log(frutas[1]); // banana
```

Métodos de Arrays

Os arrays em JavaScript vêm com uma variedade de métodos úteis para manipulação de dados.

- **toString()**

Converte o array em uma string de valores separados por vírgulas.

```
let frutas = ['maçã', 'banana', 'laranja'];  
console.log(frutas.toString()); // maçã,banana,laranja
```

- **join(separator)**

Une todos os elementos de um array em uma string, separados por um especificador opcional.

```
let frutas = ['maçã', 'banana', 'laranja'];  
console.log(frutas.join(' - ')); // maçã - banana - laranja
```

- **push(element)**

Adiciona um ou mais elementos ao final do array.

```
let frutas = ['maçã', 'banana'];  
frutas.push('laranja');  
console.log(frutas); // ['maçã', 'banana', 'laranja']
```

- **unshift(element)**

Adiciona um ou mais elementos ao início do array.

```
let frutas = ['maçã', 'banana'];
frutas.unshift('laranja');
console.log(frutas); // ['laranja', 'maçã', 'banana']
```

- **pop()**

Remove o último elemento do array e o retorna.

```
let frutas = ['maçã', 'banana', 'laranja'];
let ultimaFruta = frutas.pop();
console.log(ultimaFruta); // laranja
console.log(frutas); // ['maçã', 'banana']
```

- **shift()**

Remove o primeiro elemento do array e o retorna.

```
let frutas = ['maçã', 'banana', 'laranja'];
let primeiraFruta = frutas.shift();
console.log(primeiraFruta); // maçã
console.log(frutas); // ['banana', 'laranja']
```

- **slice(start, end)**

Retorna uma cópia superficial de uma porção do array em um novo array, sem modificar o array original.

```
let frutas = ['maçã', 'banana', 'laranja', 'uva'];
let citrus = frutas.slice(1, 3);
console.log(citrus); // ['banana', 'laranja']
console.log(frutas); // ['maçã', 'banana', 'laranja', 'uva']
```

- **array.from**

Cria um novo array a partir de um objeto semelhante a um array ou de um objeto iterável.

```
let str = 'hello';
let arr = Array.from(str);
console.log(arr); // ['h', 'e', 'l', 'l', 'o']
```

- **array.of**

Cria um novo array com um número variável de elementos.

Ex: const array = Array.of(6)

Ele resulta em um array com seis posições, mas vazias e eu consigo iterar sobre elas, no contrário, se eu usar **const array = new array(6)** ele também cria o array, mas eu não consigo iterar sobre as posições dele caso seja necessário.

```
let arr = Array.of(1, 2, 3);
console.log(arr); // [1, 2, 3]
```

No exemplo acima é criado um array com os elementos passados por parâmetro.

- **forEach**

Executa uma função fornecida uma vez para cada elemento do array.

```
let frutas = ['maçã', 'banana', 'laranja'];
frutas.forEach(function(item, index) {
  console.log(index, item);
});
// 0 'maçã'
// 1 'banana'
// 2 'laranja'
```

No exemplo abaixo temos uma iteração para cada item do array fruits e usando uma arrow function nós podemos acessar o item específico.

```
105 const fruits = ['banana', 'morango', 'manga'];
106
107 function eatAllFruits() {
108   fruits.forEach((fruit, index) => {
109     console.log(`hummm, acabei de comer um(a) ${fruit}`);
110     console.log(`ele(a) era o ${index + 1} item da minha lista`);
111   })
112 }
113
114 eatAllFruits();
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

```
[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\BLoco02_javascript-typescript\exemplo02-apostila\index.24.js"
hummm, acabei de comer um(a) banana
ele(a) era o 1 item da minha lista
hummm, acabei de comer um(a) morango
ele(a) era o 2 item da minha lista
hummm, acabei de comer um(a) manga
ele(a) era o 3 item da minha lista

[Done] exited with code=0 in 0.202 seconds
```

- **map**

Cria um novo array com os resultados da chamada de uma função para cada elemento do array.

```
117 const fruits = ['banana', 'morango', 'manga'];
118 const capitalizedFruits = fruits.map(fruit => {
119   | return fruit.toUpperCase();
120 });
121
122 console.log(fruits);
123 console.log(capitalizedFruits);
124
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

```
[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\BLoco02_javascript-typescript\exemplo02-apostila\index.24.js"
[ 'banana', 'morango', 'manga' ]
[ 'BANANA', 'MORANGO', 'MANGA' ]

[Done] exited with code=0 in 0.185 seconds
```

O método map serve para de certa forma estar iterando sobre os itens do array. A diferença dele para o forEach é que durante a iteração ele retorna o conteúdo editado, para isso basta usar o **return** para cada item e ele irá retornar o item editado.

Então o map retorna o valor alterado e o array **“fruits”** não fica alterado, ou seja, sem efeitos colaterais, pois o retorno foi guardado na nova variável **“capitalizedFruits”**.

- **filter**

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

```

126 const alunos = [
127   { nome: 'Lucas', media: 8 },
128   { nome: 'Mario', media: 2 },
129   { nome: 'Jean', media: 10 },
130   { nome: 'Rogerio', media: 6 },
131   { nome: 'Marcos', media: 5 },
132 ];
133
134 const recuperacao = alunos.filter(aluno => {
135   return aluno.media < 6
136 });
137
138 console.log(recuperacao);
139

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\8loco02_javascript-typescript\exemplo02-apostila\index.24.js"

[{ nome: 'Mario', media: 2 }, { nome: 'Marcos', media: 5 }]

[Done] exited with code=0 in 0.181 seconds

No exemplo acima eu filtro apenas para os alunos que tem média abaixo de “6”

- **reduce**

Aplica uma função contra um acumulador e cada elemento do array (da esquerda para a direita) para reduzi-lo a um único valor.

```

126 const alunos = [
127   { nome: 'Lucas', media: 8 },
128   { nome: 'Mario', media: 2 },
129   { nome: 'Jean', media: 10 },
130   { nome: 'Rogerio', media: 6 },
131   { nome: 'Marcos', media: 5 },
132 ];
133
134 const recuperacao = alunos.filter(aluno => {
135   return aluno.media < 6
136 });
137
138 console.log(recuperacao);
139
140 const pontosObtidos = alunos.reduce(
141   (total, aluno) => {
142     return total + aluno.media
143   },
144   0
145 );
146
147 console.log(pontosObtidos);
148

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\8loco02_javascript-typescript\exemplo02-apostila\index.24.js"

[{ nome: 'Mario', media: 2 }, { nome: 'Marcos', media: 5 }]

31

[Done] exited with code=0 in 0.158 seconds

No exemplo acima a partir da linha “140” implementamos o reduce para obter a quantidade de pontos que todos os alunos fizeram.

O primeiro parâmetro passado dentro do `reduce` é uma função de call-back e o segundo parâmetro é o valor inicial que queremos reduzir, ou seja, o valor que iniciará a “soma”, no caso usamos o valor “0”. Que esse valor é colocado no “total” inicialmente e depois vamos somando à ele o valor de cada nota.

Para não retornar um “undefined”, nós temos que ter um retorno nessa função.

O “aluno” contém o item do array referente ao aluno que está sendo iterado e com isso conseguimos acessar o valor da média.

- **find**

Retorna o primeiro valor encontrado no array que satisfaz a função de teste fornecida. Caso contrário, retorna undefined.

```
let numeros = [1, 2, 3, 4];
let encontrado = numeros.find(function(num) {
  return num > 2;
});
console.log(encontrado); // 3
```

Uma diferença entre o **filter** e o **find** é que o `filter` retorna um array com o resultado, e o `find` retorna diretamente o elemento do array.

No exemplo acima o valor “4” também é maior que “2”, mas ele retorna somente o “3” porque é o primeiro item que atende a essa condição “num > 2”.

Se fosse o `filter` por exemplo, ele retornaria “3” e “4”.

- **some e every**

some verifica se pelo menos um elemento no array passa no teste implementado pela função fornecida.

every verifica se todos os elementos no array passam no teste implementado pela função fornecida.

```
174 let numeros = [1, 2, 3, 4, 5];
175
176 let algumMaiorQueTres = numeros.some(function(num) {
177   return num > 3;
178 });
179 console.log(algunMaiorQueTres); // true
180
181 let todosMaioresQueTres = numeros.every(function(num) {
182   return num > 3;
183 });
184 console.log(todosMaioresQueTres); // false
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

```
[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\02_loco02_javascript-typescript\exemplo02-apostila\index.24.js"
true
false
```

```
[Done] exited with code=0 in 0.177 seconds
```

No exemplo acima o “some” está verificando se tem algum número maior que “3” e se tiver ele retorna true. Nesse caso, dada a relação que temos no array “números”, podemos observar que temos o “4” e o “5”, portanto retorna true, pois encontrando apenas 1 já satisfaz essa condição.

Também na linha “181” temos o uso do “every” que valida se “TODOS” os elementos iterados atendem à condição de serem maior que “3”. E considerando os itens do array “números” temos o retorno de “false” porque os itens “1” e “2” desse array não atendem a essa condição.

12. DATE

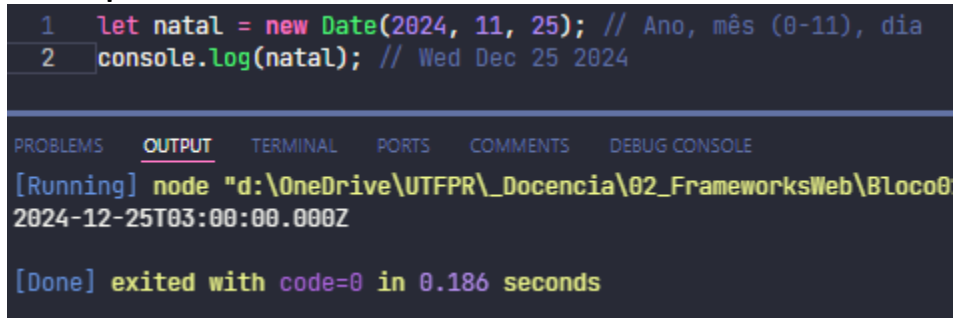
Em JavaScript, a manipulação de datas é feita principalmente através do objeto Date. Este objeto permite criar, formatar, e manipular datas e horários de maneira bastante flexível.

Exemplos

- **var d = new Date();** cria um objeto do tipo data e já coloca a data e hora atual preenchida nesse objeto.
- **var d = new Date(“07/01/2016”);** cria um objeto do tipo data colocando a data especificada no parâmetro.
- **var d = new Date(“07/01/2016 05:00:30”);** também é válido para colocar as horas na nova data.
- **var d = new Date(1467827595397);** podemos também criar usando os milissegundos desde que se passaram o primeiro de janeiro de 1970. Que dá 06 de julho de 2016.

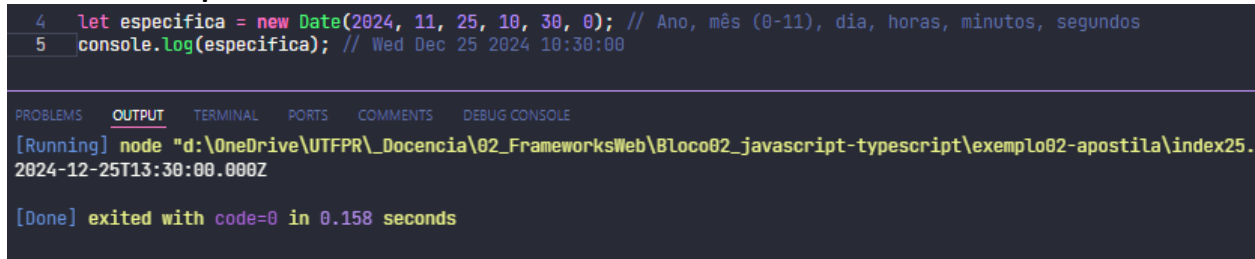
- **Data específica**

```
1 let natal = new Date(2024, 11, 25); // Ano, mês (0-11), dia
2 console.log(natal); // Wed Dec 25 2024
```



- **Data e Hora específica**

```
4 let especifica = new Date(2024, 11, 25, 10, 30, 0); // Ano, mês (0-11), dia, horas, minutos, segundos
5 console.log(especifica); // Wed Dec 25 2024 10:30:00
```



Nesse exemplo podemos observar que é acrescentado 3 horas devido ao fuso horário não ter sido tratado. É importante ficar atento a essas condições na hora de exibir informações com datas para os usuários.

MÉTODOS DO OBJETO DATE

O objeto **Date** oferece vários métodos para acessar e modificar diferentes partes de uma data. Observe abaixo o código com exemplos:

```

7
8   let agora = new Date();
9
10  console.log(agora.getFullYear()); // Ano atual
11  console.log(agora.getMonth()); // Mês atual (0-11)
12  console.log(agora.getDate()); // Dia do mês (1-31)
13  console.log(agora.getDay()); // Dia da semana (0-6, sendo 0 domingo)
14  console.log(agora.getHours()); // Hora (0-23)
15  console.log(agora.getMinutes()); // Minutos (0-59)
16  console.log(agora.getSeconds()); // Segundos (0-59)
17  console.log(agora.getMilliseconds()); // Milissegundos (0-999)
18

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

```

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\Bloco02_javascript
2024
4
26
0
19
37
15
401

[Done] exited with code=0 in 0.179 seconds

```

REDEFININDO INFORMAÇÕES DE UMA DATA

```

21  let data = new Date();
22
23  data.setFullYear(2025);
24  data.setMonth(0); // Janeiro (0)
25  data.setDate(15);
26  data.setHours(10);
27  data.setMinutes(30);
28  data.setSeconds(45);
29
30  console.log(data); // Data ajustada

```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

```

[Running] node "d:\OneDrive\UTFPR\Docencia\02_Fram
2025-01-15T13:30:45.436Z

[Done] exited with code=0 in 0.278 seconds

```

Caso fosse ajustar a hora, sem usar bibliotecas externas, poderia aplicar o comando => `data.setHours(data.getHours() - 3);`

CONVERTENDO DATAS PARA STRINGS

- `toString()`

```
let agora = new Date();
console.log(agora.toString()); // Ex: "Tue May 24 2024"
```

- `getTimeString()`

```
let agora = new Date();
console.log(agora.getTimeString()); // Ex: "12:34:56 GMT-0300 (Brazil Standard Time)"
```

- `toISOString()`

```
let agora = new Date();
console.log(agora.toISOString()); // Ex: "2024-05-24T12:34:56.789Z"
```

- `toLocaleDateString()`

```
let agora = new Date();
console.log(agora.toLocaleDateString('pt-BR')); // Ex: "24/05/2024"
```

- `toLocaleTimeString()`

```
let agora = new Date();
console.log(agora.toLocaleTimeString('pt-BR')); // Ex: "12:34:56"
```

13. Manipulando Strings

JavaScript fornece diversos métodos para trabalhar com strings, permitindo operações como pesquisa, extração, substituição, conversão de maiúsculas/minúsculas, entre outras.

Comprimento de uma String

O comprimento de uma string pode ser obtido usando a propriedade `length`.

```
let texto = "Olá, mundo!";
console.log(texto.length); // 12
```

Pesquisa em Strings

Para encontrar a posição de um substring dentro de uma string, usamos os métodos `indexOf` e `lastIndexOf`.

Onde o **indexOf** retorna qual posição a palavra informada está, lembrando que ele só mostra a primeira vez que a palavra aparece na string. Caso ele não encontre, será retornado "-1".

E o **lastIndexOf** retorna qual a última vez que essa palavra aparece e retorna “-1” quando esse elemento não é encontrado.

Se observar no exemplo abaixo, quando usamos o **lastIndexOf** procurando pela letra “o” ele retorna o valor 10, que é onde essa letra aparece pela última vez.

```
let texto = "Olá, mundo!";
console.log(texto.indexOf("mundo")); // 5
console.log(texto.lastIndexOf("o")); // 10
```

Extração de Substrings

Podemos extrair partes de uma string usando **slice** e **substring**.

No exemplo abaixo podemos ver que no **slice** passamos a posição inicial “0” e depois “5” como quantidade de caracteres a serem extraídos. Já no segundo exemplo de uso do **slice** é informado um número negativo para dizer quantos caracteres queremos extrair da string considerando de **traz para frente**.

Para o **substring** também informamos a posição inicial e a quantidade de caracteres a serem extraídos da string.

```
let texto = "Olá, mundo!";

// slice
console.log(texto.slice(0, 5)); // "Olá, "
console.log(texto.slice(-6)); // "mundo!"

// substring
console.log(texto.substring(0, 5)); // "Olá, "
```

Substituição de Substrings

O método **replace** é utilizado para substituir parte de uma string por outra.

```
let texto = "Olá, mundo!";
let novoTexto = texto.replace("mundo", "JavaScript");
console.log(novoTexto); // "Olá, JavaScript!"
```

Alterando Maiúsculas e Minúsculas

Podemos converter uma string para maiúsculas ou minúsculas usando **toUpperCase**(para maiúsculas) e **toLowerCase**(para minúsculas).

```
let texto = "Olá, Mundo!";
console.log(texto.toUpperCase()); // "OLÁ, MUNDO!"
console.log(texto.toLowerCase()); // "olá, mundo!"
```

Concatenando Strings

Strings podem ser concatenadas usando o operador “+” ou o método “concat”.

No exemplo abaixo temos dois exemplos usando cada uma das formas de concatenação de strings.

```
let saudacao = "Olá";
let nome = "Mundo";

// Usando o operador +
let texto = saudacao + ", " + nome + "!";
console.log(texto); // "Olá, Mundo!"

// Usando concat
let texto2 = saudacao.concat(", ", nome, "!");
console.log(texto2); // "Olá, Mundo!"
```

Dividindo Strings

O método split divide uma string em um array de substrings com base em um delimitador.

No exemplo abaixo podemos observar que a string foi dividida usando um espaço como delimitador, ou seja, foi ele o alvo utilizado para entender onde era para dividir a string.

```
105 let texto = "Olá, mundo!";
106 let palavras = texto.split(" ");
107 console.log(palavras); // ["Olá,", "mundo!"]
108
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\Bloco08
['Olá,', 'mundo!']

[Done] exited with code=0 in 0.539 seconds

Acessando Caracteres

Podemos acessar caracteres específicos de uma string usando charAt ou indexação.

```
112
113 let texto = "Olá, mundo!";
114
115 // Usando charAt
116 console.log(texto.charAt(2)); // "á"
117
118 // Usando indexação
119 console.log(texto[2]); // "á"
120
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS DEBUG CONSOLE

[Running] node "d:\OneDrive\UTFPR\Docencia\02_FrameworksWeb\Bloco08
á
á

[Done] exited with code=0 in 0.166 seconds