Fundamentos

Bootcamp Desenvolvedor Fullstack

Raphael Gomide

2020

Fundamentos

Bootcamp Desenvolvedor Fullstack

Raphael Gomide

Sumário

Capítulo 1. Ecossistema JavaScript

Neste primeiro capítulo serão vistas as principais ferramentas para o desenvolvimento com JavaScript que serão utilizadas e demonstradas tanto no decorrer desta apostila quanto nas videoaulas da disciplina.

Visual Studio Code

O Visual Studio Code é um dos principais editores de código utilizados atualmente. É open source, gratuito e mantido por colaboradores da Microsoft. Quanto ao JavaScript, possui excelentes funcionalidades que podem inclusive ser estendidas através de seus plugins e extensões.

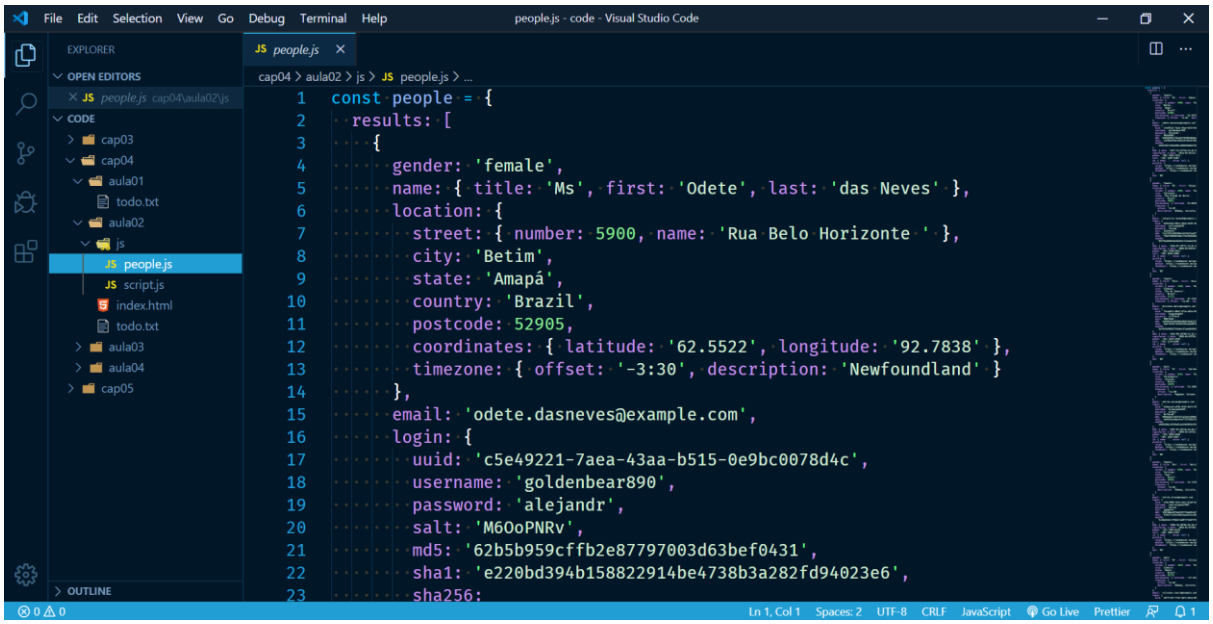
Dentre suas diversas funcionalidades destacam-se o excelente suporte ao Git, a portabilidade – com versões para Windows, Linux e MacOS – e a presença de um terminal de comandos embutido na ferramenta.

Esta será, portanto, a ferramenta de edição de código HTML, CSS e JavaScript utilizada no decorrer de toda a disciplina.

Para mais detalhes sobre o Visual Studio Code verifique as videoaulas deste capítulo, onde são realizadas a instalação do programa, configuração inicial e também a instalação de algumas extensões.

A Figura abaixo mostra um exemplo de tela do Visual Studio Code no Windows.

Figura 1 – Visual Studio Code.



Node.js

O Node.js pode ser denominado com um ambiente de execução (runtime) de código JavaScript com suporte a Windows, Linux e MacOS. Dentre suas diversas funcionalidades, destacam-se:

- Criação e execução de scripts tanto em computadores pessoais quanto servidores.
- Servidor web e/ou API com o apoio de bibliotecas como o express.js.
- Conexão e integração com bancos de dados.

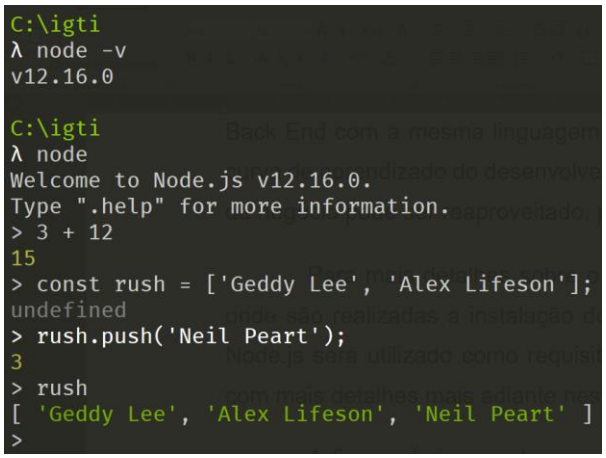
Graças ao Node.js é possível executar código JavaScript com acesso aos arquivos e recursos do computador e servidores – como bancos de dados por exemplo – ao contrário da web onde esse acesso direto é proibido por questões de segurança. Assim, pode-se concluir que o JavaScript, através do Node.js, pode atuar no Back End assim como outras linguagens de programação como o Java, PHP e C#.

Esta possibilidade, portanto, de se trabalhar tanto no Front End quanto no Back End com a mesma linguagem de programação — JavaScript — faz com que a curva de aprendizado do desenvolvedor seja cada vez menor. Muito código de regras de negócio pode ser reaproveitado, por exemplo.

Para mais detalhes sobre o Node.js verifique as videoaulas deste capítulo, onde são realizadas a instalação do programa e testes iniciais. Nesta disciplina, o Node.js será utilizado como requisito da biblioteca **live-server**, que será explicada detalhadamente mais adiante nesta apostila.

A Figura abaixo mostra um teste para verificar se o Node.js está instalado corretamente (comando "node -v", que verifica a versão instalada do Node.js) e também exemplos de algumas instruções JavaScript no ambiente de execução de comandos do Node.js, mais conhecido como REPL (*Read-Eval-Print-Loop*), onde os comandos são interpretados indefinidamente até que o usuário force o cancelamento (Ctrl + C no Windows).

Figura 2 – Exemplos[1] de instruções no modo REPL do Node.js.



[1] Nesta imagem, foi utilizada a ferramenta Cmder (Windows) como terminal de comandos, pois é bem mais flexível que o prompt de comando nativo e tem suporte para comandos do Unix/Linux.

NPM

A sigla NPM significa *Node Package Manager*, que é o gerenciador de pacotes oficial do Node.js. Seu repositório — o npmjs.com — é um dos maiores repositórios de bibliotecas/pacotes de código-fonte de toda a internet. Lá é possível encontrar diversas bibliotecas — tanto para Front End quanto Back End — que facilitam muito o dia a dia do desenvolvedor JavaScript e podem, claro, aumentar sua produtividade.

Além disso, o **npm** é uma **ferramenta** de linha de comando vinculada ao Node.js para a manutenção de dependências de projeto através de suporte à instalação e remoção de pacotes, por exemplo. Para esta disciplina será necessária a instalação de apenas um pacote — o live-server — e, para isso, basta executar o seguinte comando (considerando que o Node.js já esteja devidamente instalado):

```
npm install -g live-server
```

O comando acima instala o pacote de forma global, ou seja, que pode ser executado a partir de qualquer diretório do computador. Isso é possível graças ao parâmetro **-g**. Caso -g não seja utilizado, a instalação é feita localmente, ou seja, apenas no diretório onde o comando foi executado.

Mais detalhes sobre o Node.js e o npm podem ser vistos nas videoaulas deste capítulo.

Biblioteca live-server

A biblioteca live-server será utilizada no decorrer da disciplina para que seja possível a criação de um servidor web local para desenvolvimento. Isso facilita muito o trabalho do desenvolvedor, pois esta ferramenta tem uma funcionalidade conhecida como *live-reloading*, onde qualquer alteração no código-fonte faz com que o site seja recarregado automaticamente. Isso aumenta muito a produtividade no desenvolvimento em geral.

Além da biblioteca hospedada no NPM, a ferramenta live-server também existe como uma extensão do Visual Studio Code. Essa instalação será demonstrada durante as videoaulas deste capítulo.

Para inicializar um servidor web manualmente, basta acessar o diretório raiz do site no terminal de comandos e digitar "live-server", considerando que tanto o Node.js quanto a própria biblioteca já estejam devidamente instalados em seu computador. A Figura abaixo ilustra melhor esse procedimento.

Figura 3 – Exemplo de execução manual de live-server.

Noções de HTML

HTML (Hyper Text Markup Language) é considerada uma **linguagem de marcação** (e não de programação), utilizada para definir a estruturação de conteúdo na web. É composta de diversos **elementos**, que por sua vez são decompostos em:

Tags

Tags são representadas pelos símbolos <></> para **delimitar o início e fim do elemento**. Cada tag possui um identificador único, como por exemplo , <h2></h2>, <a> e <img>. Nem todas as tags possuem fechamento, como por exemplo <img>. Uma dica para saber se determinada tag possui ou não o fechamento é verificar se ela depende de **conteúdo**. Em caso afirmativo, ela deve de fato possuir fechamento.

Atributos:

Representam características da tag. Em geral, os principais atributos utilizados atualmente são **class** e **id**, que são utilizados pelo CSS e pelo JavaScript para identificação de um ou mais elementos com as mesmas características. Um exemplo de **atributo**, portanto, poderia ser o seguinte: `<p class='content'>Parágrafo</p>`.

Valores de atributos

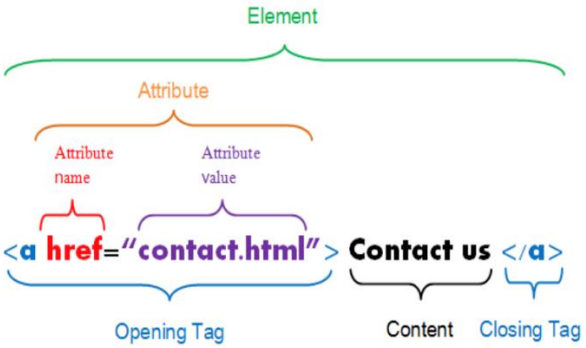
Representam valores que os atributos podem possuir. O valor está separado do atributo através do símbolo **=** e seu conteúdo está delimitado por aspas simples **ou** duplas. Um exemplo de **valor de atributo**, portanto, poderia ser o seguinte:

```
<p class='content'>Parágrafo</p>.
```

Conteúdo

Representa o texto delimitado pela tag. Sempre que uma tag possui conteúdo, ela deve possuir um fechamento. Nem todas as tags possuem conteúdo, como por exemplo,
 (quebra de linha) e <input> (entrada de dados). Um exemplo de conteúdo, portanto, poderia ser o seguinte: `<p class='content'>Parágrafo</p>`. Mais detalhes podem ser vistos na Figura abaixo.

Figura 4 – Estrutura de um elemento HTML.



Fonte: shawnsanto.com.

Principais elementos

A seguir são listados alguns dos principais elementos do HTML e suas aplicações:

- <p> → Parágrafos.
- <h1> a <h6> → Títulos.
- → Trechos de texto a serem destacados.
- → Divisões da página.
- <img> → Definição de imagens.
- <table> <tr> <td> → Definição de tabelas, linhas e colunas.
-   → Listas e itens de lista.
- e → Ênfase no texto.
- <a> → Âncoras (links).

É muito importante salientar que, apesar dos navegadores interpretarem e renderizarem elementos HTML de formas semelhantes, como por exemplo, elementos <h1> a <h6> com diferentes tamanhos de fonte e texto em negrito, o que realmente importa no HTML é a semântica, ou seja, o significado do elemento. Cores, tamanhos, fontes, estilos, posicionamento etc., são responsabilidade do CSS, que será visto em breve.

Caminho absoluto x Caminho relativo

A definição de localização de arquivos no site é muito utilizada em imagens e links, por exemplo.

O **caminho absoluto** define o local físico do recurso em disco (arquivo, imagem etc.) e, por isso, só funciona no ambiente de desenvolvimento do site/página HTML. Essa é uma das grandes causas da famigerada frase “Na minha máquina funciona!”. Portanto, **evite** a utilização de caminhos absolutos.

Já o **caminho relativo** define o local do recurso (arquivo, imagem etc.) **em relação** aos demais arquivos do projeto. Os principais símbolos utilizados são:

- Pasta local → `'./'`
- Pasta pai/mãe → `'../'`

Em resumo, **utilize sempre caminhos relativos**. Assim, as referências aos recursos de seu projeto funcionarão em **qualquer ambiente**.

Para mais detalhes sobre caminho absoluto, o caminho relativo e também sobre o HTML em geral, verifique as videoaulas do capítulo 1.

Noções de CSS

O termo CSS significa Cascading Style Sheets, ou seja, folhas de estilo em cascata. Esse princípio é semelhante ao da herança em orientação a objetos, ou seja, a definição de CSS pode ser feita para um elemento e reaplicada automaticamente a elementos semelhantes conforme a regra definida.

O CSS é utilizado para a estilização do conteúdo HTML. Permite a alteração de cores, estilos de texto, posicionamento de elementos etc. Possui, portanto, foco no conteúdo visual.

Assim como o HTML, o CSS é uma linguagem de marcação e é escrita de forma declarativa.

A Figura abaixo ilustra a sintaxe básica do CSS, onde podem ser vistos os seguintes agrupamentos:

- **Seletor**, que podem ser elementos, classes (`.classe`) ou id's (`#id`).
- **Declaração**, que são instruções separadas por **;** (ponto-e-vírgula).
- **Propriedade**, que são características do seletor.
- **Valores**, que alteram as propriedades.

Figura 5 – Sintaxe de declaração de CSS.

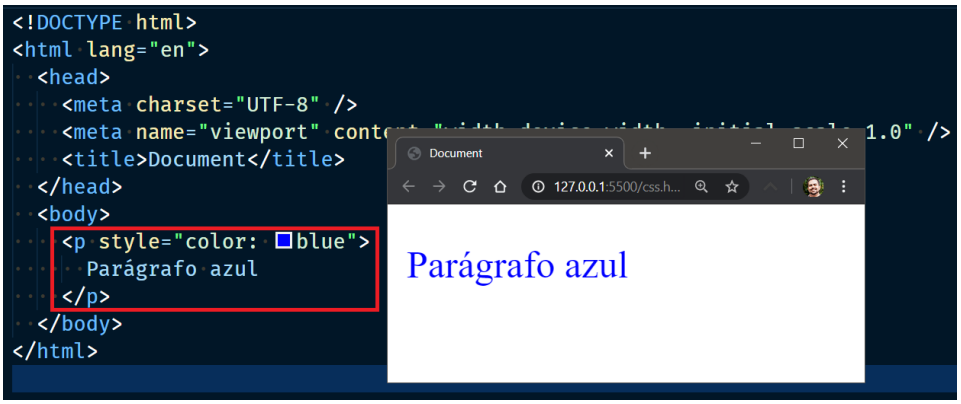
Fonte: howtocodeschool.com.

O CSS deve estar de alguma forma vinculado ao arquivo HTML e pode ser definido, em regra, de três formas:

Atributo style

Esta técnica é conhecida como *inline CSS* e é utilizada através do **atributo style** em tags. Em regra, não é recomendada. A Figura abaixo mostra um exemplo.

Figura 6 – Exemplo de utilização de CSS com o atributo *style*.



Tag style

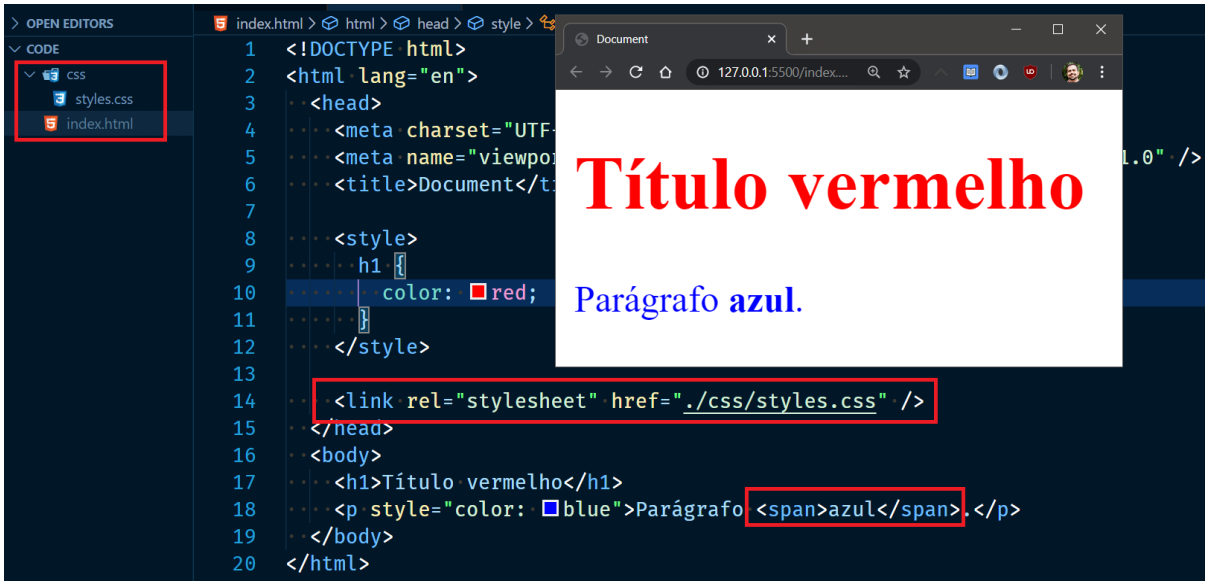
É possível também a utilização da **tag <style>**, que fica geralmente posicionada ao final de **<head>**. A organização do código HTML melhora um pouco, mas ainda assim esta técnica não é tão recomendada. A Figura abaixo mostra um exemplo.

Figura 7 – Exemplo de utilização de CSS com a tag <style>.

Arquivo externo

Em regra, essa é a técnica mais recomendada. Com a separação de HTML e CSS em arquivos é possível, inclusive, que dois profissionais atuem ao mesmo tempo no projeto sem risco de conflitos, um cuidando do HTML e outro cuidando do CSS. Para referenciar um arquivo CSS, utilize a tag **<link>** com os atributos **rel** e **href** em **<head>**. A Figura abaixo ilustra um exemplo.

Figura 8 – Exemplo de utilização de CSS com arquivo externo.



CSS Reset

Essa técnica é utilizada para remover qualquer estilização padrão que os navegadores aplicam ao HTML. Isso ajuda a padronizar o site para os diversos navegadores existentes.

Uma das técnicas mais utilizadas é a inclusão do arquivo `reset.css` antes de qualquer outro arquivo CSS. Este arquivo foi concebido por Eric Meyer e pode ser obtido aqui.

Para mais detalhes sobre CSS, verifique as videoaulas do capítulo 1.

Capítulo 2. Introdução ao JavaScript

O JavaScript é uma linguagem de programação que tem evoluído bastante nos últimos anos, o que tem contribuído muito para a sua popularidade.

Em relação à linguagem de programação Java, JavaScript só tem a semelhança no nome e em poucas características de sintaxe. Acredita-se que, devido à popularidade do Java nos anos 90 — época em que o JavaScript surgiu — foi sugerido um nome semelhante para a linguagem. Entretanto, o nome formal do JavaScript é ECMAScript.

O JavaScript é mais comumente utilizado como uma linguagem de Front End, atuando na interação do usuário com sites feitos com HTML e CSS. Entretanto, esta não é a única forma de aplicação da linguagem. Destacam-se, também, as seguintes:

- Back End, com Node.js.
- Aplicações desktop, com Electron.
- Dispositivos embarcados, com Espruino.
- Internet das Coisas (IoT), com Johnny Five.
- Machine learning, com TensorFlow.js.

Integração com HTML

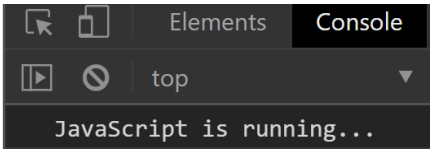
Existem algumas maneiras de se integrar o JavaScript com HTML. A mais comum e recomendada é a criação de um arquivo externo e vinculação do mesmo na tag **<script>**, utilizando caminho relativo como valor do atributo **src**. A Figura abaixo mostra melhor essa forma de integração.

Figura 9 – Integração de JavaScript com HTML.

```
index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>Document</title>
7    </head>
8    <body>
9      <h1>Teste de integração com JavaScript</h1>
10
11     <script src="./js/script.js"></script>
12   </body>
13 </html>
```

```
JS script.js
js > JS script.js
1  console.log('JavaScript is running...');
```

Teste de integração com JavaScript



JavaScript básico

A seção a seguir trata especificamente da sintaxe da linguagem JavaScript.

Principais tipos e valores

Atualmente o JavaScript possui oito tipos de dados. Serão destacados abaixo somente os tipos principais, com exemplos de possíveis valores:

- Number → 1, -3, 8.56
- String → "Teste", "3.14", 'Aspas simples'
- Boolean → true, false
- Null → null
- Undefined → undefined

- Object → [1, 3, 5], [6, 'sete', true], {id: 2, nome: 'Raphael'}

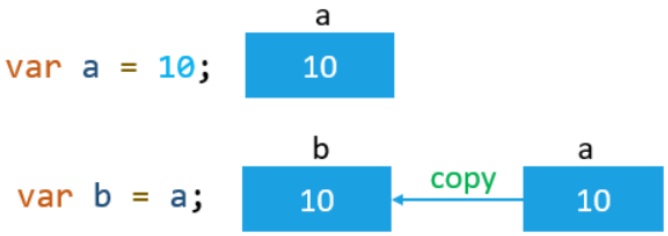
Algumas considerações importantes devem ser feitas sobre os tipos listados acima. O tipo **Number**, diferentemente de outras linguagens, pode representar tanto valores inteiros quanto valores decimais. Valores decimais são escritos com o símbolo de ponto (.) para representar separar a parte inteira da parte decimal.

O tipo **String** também aceita o símbolo de crase (`) como delimitador, além das aspas simples e duplas. Entretanto, essa é uma funcionalidade do JavaScript moderno, que será vista posteriormente.

Valores **null** são explicitamente definidos pelo programador. Já os valores **undefined** são retornados pelo JavaScript em declarações de variáveis sem valor, por exemplo.

Todos os tipos listados nos tópicos acima são considerados **tipos primitivos**, ou seja, a atribuição de variáveis com esses valores a outras variáveis é feita via **cópia dos dados em memória**. A Figura abaixo ilustra melhor esse conceito.

Figura 10 – Valores primitivos.

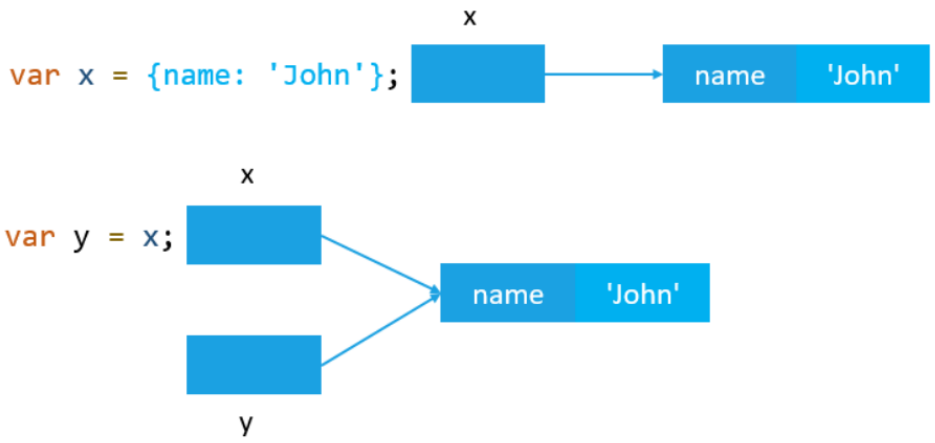


Fonte: javascripttutorial.net.

Já o tipo **Object**, representado tanto por **Objetos**, delimitados por chaves {}, quanto por **Arrays** (vetores), delimitados por colchetes [], são considerados tipos de **referência**. Em linhas gerais, eles se situam em uma área diferente de memória e são **referenciados**, ou seja, uma atribuição de valor desse tipo a uma outra variável **não configura cópia**. O que acontece é que ambas as variáveis passam a **apontar para o endereço de memória** onde o valor se encontra. Isso é algo muito importante

a ser aprendido, pois é uma grande fonte de bugs em aplicações. A Figura abaixo ilustra melhor esse conceito.

Figura 11 – Valores de referência.



Fonte: javascripttutorial.net.

É interessante também mostrar alguns exemplos de objetos e vetores, conforme pode ser visto na Figura abaixo:

Figura 12 – Exemplos de arrays e objetos.

```
[1, 2, 3]                                    // Array de Number
['1', '2', '3']                              // Array de String
{id: 1, name: 'Alex Lifeson', band: 'Rush'}   // Objeto
[{id: 1}, {id: 2}, {id: 3}]                  // Array de objetos
```

Variáveis

Variáveis são criadas para guardar valores em memória. Isso é muito comum principalmente quando é necessário reutilizar o valor posteriormente. JavaScript é considerada uma linguagem de programação com **tipagem fraca**, pois é possível atribuir valores de diversos tipos às variáveis sem distinção. A palavra-chave para a

criação de variáveis é, por padrão, **var**. A Figura abaixo mostra um exemplo de declaração e utilização de variável.

Figura 13 – Declaração e reutilização de variável.

```
var x = 100; // Declaração
undefined
x + 200; // Utilização
300
x // O valor original foi mantido
100
x = x + 200; // Reatribuição de valor
300
x // O valor foi modificado
300
```

Operadores

Assim como as demais linguagens de programação, o JavaScript também possui operadores. Os principais operadores são exibidos em forma de código-fonte nas figuras abaixo.

Uma observação muito importante é que, quanto aos operadores de igualdade, sejam utilizados somente *===* e *!==* , ao invés de *==* e *!==* pois os primeiros testam tanto o **valor** quanto o **tipo**, para retornarem **true**. Isso pode garantir mais estabilidade e menos bugs no código.

Figura 14 – Operadores aritméticos.

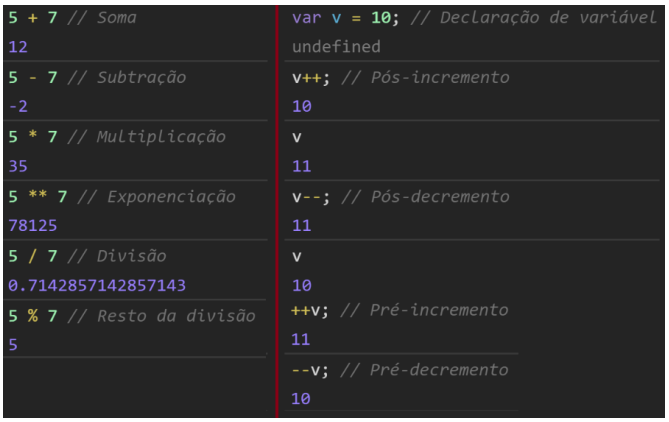


Figura 15 – Operadores de atribuição.

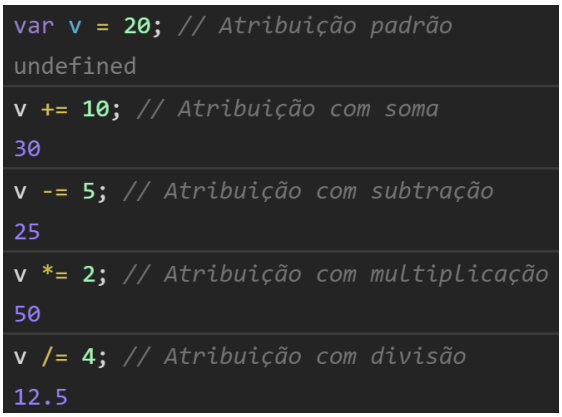


Figura 16 – Operadores de comparação.

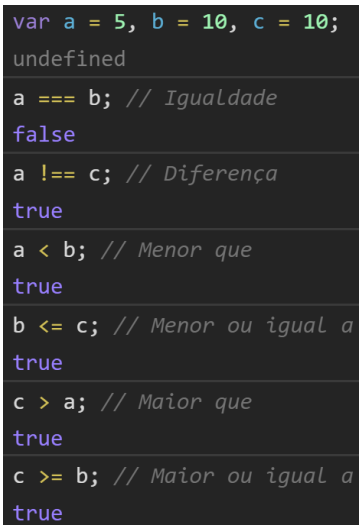
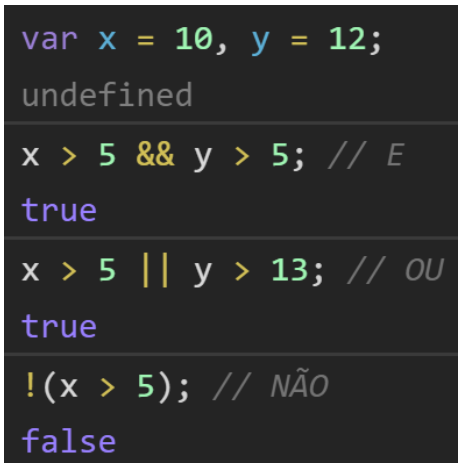


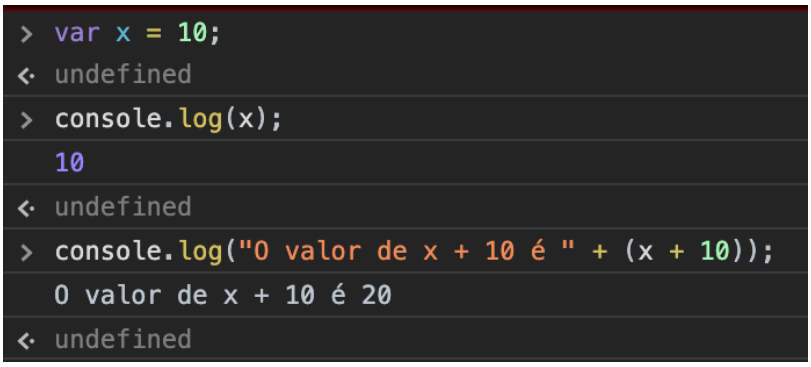
Figura 17 – Operadores lógicos.



Console

O comando `console` do JavaScript é um dos mais utilizados para investigação e depuração de erros (debug). A utilização mais comum é feita com `console.log()`. A Figura abaixo mostra alguns exemplos.

Figura 18 – Exemplos com o comando `console.log`.

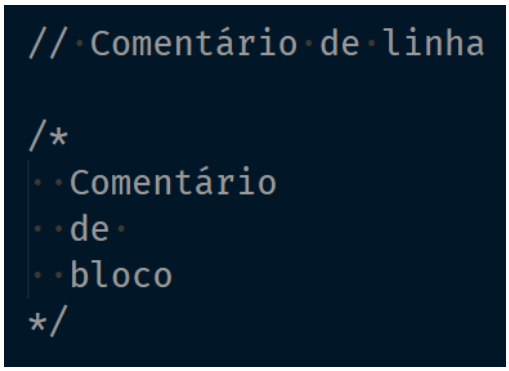


Comentários de código

A utilização de comentários de código pode ser interessante em casos de algoritmos complexos, por exemplo. Comentários são ignorados pelo interpretador JavaScript e não são, portanto, executados. Entretanto, o ideal é que a própria modularização feita pelo programador — através de funções, por exemplo —

documente o código. No JavaScript há duas formas de comentários: comentário de linha, com `//`, e comentário de bloco, com `*/ */`. A Figura abaixo mostra alguns exemplos de comentários de código.

Figura 19 – Exemplos de comentários de código.



Comandos de bloco

Em JavaScript, os blocos são definidos por chaves (`{}`) e delimitam a execução de uma ou mais instruções. Nesta seção serão estudados três dos principais tipos de comandos de bloco, sendo:

- Estruturas de decisão.
- Estruturas de repetição.
- Funções.

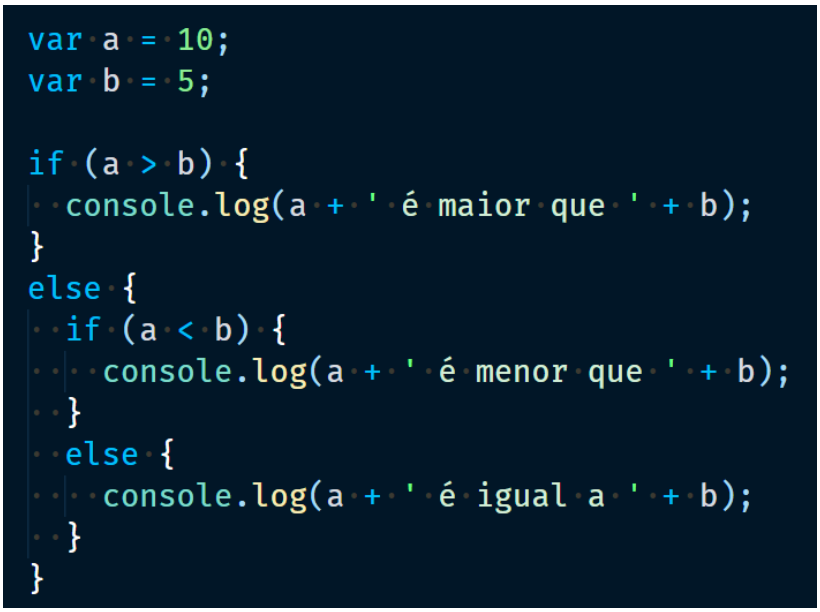
Estruturas de decisão

São utilizadas para definir **se** determinado bloco de código será executado ou não. O teste é feito através de **proposições lógicas** – afirmativas que podem ser avaliadas como verdadeiras **ou** como falsas. Essas afirmativas geralmente estão acompanhadas de operadores de comparação e operadores lógicos. As estruturas de decisão mais comuns no JavaScript são:

- `if/else`
- `switch`
- Operador ternário (? :)

A Figura abaixo mostra um exemplo de **if/else** — que é a estrutura de decisão mais comum e mais utilizada. É importante salientar que o comando **else** é opcional e não testa nenhuma proposição. O bloco **if** executa o que é **verdadeiro** e o bloco **else** executa o que é **falso**. Assim, pode-se afirmar que **apenas um dos blocos é executado**.

Figura 20 – Comando if/else.



```
var a = 10;
var b = 5;

if (a > b) {
  console.log(a + ' é maior que ' + b);
}
else {
  if (a < b) {
    console.log(a + ' é menor que ' + b);
  }
  else {
    console.log(a + ' é igual a ' + b);
  }
}
```

A Figura abaixo mostra um exemplo de código com o comando **switch** — que é uma estrutura de decisão utilizada para **testes de igualdade** com várias possibilidades de valores, o que a torna mais **elegante** que o if/else nesses casos. Cada possiblidade de valor é definida com a palavra-chave **case** e é geralmente seguida do comando **break**, que interrompe a execução, evitando que os demais casos sejam testados desnecessariamente. A palavra-chave **default** geralmente fica ao final do bloco, é opcional e representa o **else**, ou seja, quando nenhum dos casos explicitados atende a proposição.

Figura 21 – Exemplo de utilização do comando switch.

```
const dayOfWeek = 5;

switch (dayOfWeek) {
  case 1:
    console.log('Segunda-feira');
    break;
  case 2:
    console.log('Terça-feira');
    break;
  case 3:
    console.log('Quarta-feira');
    break;
  case 4:
    console.log('Quinta-feira');
    break;
  case 5:
    console.log('Sexta-feira');
    break;
  case 6:
    console.log('Sábado');
    break;
  case 7:
    console.log('Domingo');
    break;

  default:
    console.log('Dia inválido!');
}
```

A Figura abaixo mostra um exemplo de código com o operador ternário (`? :`), que é uma estrutura de decisão utilizada para **decisões simples** com uma escrita mais sucinta e elegante que o if/else. O código a ser executado para a proposição **verdadeira** é precedido do símbolo **"?"** (interrogação) e o código a ser executado para a proposição **falsa** é precedido do símbolo **":"** (dois pontos).

Figura 22 – Exemplo de código com operador ternário.

```
// Total vendido
var totalSales = 1000;

/**
 * Regra do bônus
 * Vendas maiores ou iguais a 600 reais ==> 10%
 * Vendas menores que 600 reais ==>  5%
 */
var bonus = totalSales >= 600 ? totalSales * 0.1 : totalSales * 0.05;

// Transformando o código acima para if/else
if (totalSales >= 600) {
  bonus = totalSales * 0.1;
} else {
  bonus = totalSales * 0.05;
}

console.log(bonus);
```

Estruturas de repetição

As estruturas de repetição são muito utilizadas para executar tarefas semelhantes repetidamente. É importante salientar que em algum momento elas devem ser interrompidas, caso contrário há o loop infinito, que pode travar a aplicação. No JavaScript, as estruturas de repetição mais conhecidas são:

- `while`.
- `do... while`.
- `for`.

A Figura abaixo exemplifica melhor os três comandos e mostra que, na grande maioria dos casos, o **for** é a **melhor** estrutura a ser utilizada por possuir um código mais bem organizado que as demais.

Figura 23 – Exemplos com while, do... while e for.

```
/**
 * Realizando o somatório de números de 1 a 100 com while, do... while e
 */

// Para guardar a soma
var sum = 0;

// Índice/sentinela/contador - variável de controle
var i = 1;

// No while, o teste da proposição é feito no início do bloco
while (i <= 100) {
  sum += i;
  i++; // Sem esse incremento, há o loop infinito
}

// Reiniciando
i = 1;
sum = 0;

/**
 * No do... while o teste da * proposição é feito no final do bloco.
 * Assim, é garantido que o bloco é executado pelo menos uma vez
 */
do {
  sum += i;
  i++; // Sem esse incremento, há o loop infinito
} while (i <= 100);

// Reiniciando
sum = 0;

/**
 * O for é mais organizado pois, em uma linha, inicializa a variável de
 * controle, define a proposição e o incremento. Assim, a lógica fica
 * isolada e mais fácil de ser entendida
 */
for (i = 0; i <= 100; i++) {
  sum += i;
}
```

Funções

Funções são blocos de código utilizados para **isolar** determinado **comportamento**. Isso faz com que esse bloco possa ser **reutilizado** em outros trechos do código. Além disso, funções permitem que uma **regra** possa ser **centralizada**, o que facilita a posterior manutenção.

Em JavaScript, funções são criadas com a palavra-chave **function**, possuem um identificador textual e podem possuir **0...n parâmetros de entrada**, também conhecidos como **argumentos**. Funções podem retornar **0 ou 1 valor**, utilizando a palavra-chave **return**.

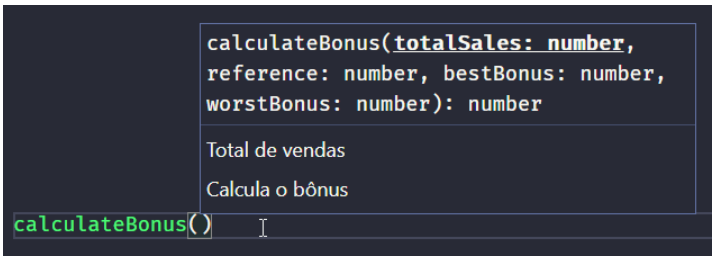
A Figura abaixo mostra um exemplo de função, utilizando o exemplo de cálculo de bônus utilizado na Figura sobre o operador ternário.

Figura 24 – Exemplo de função.

```
/**
 * Calcula o bônus
 * @param {Number} totalSales Total de vendas
 * @param {Number} reference Valor de referência
 * @param {Number} bestBonus Bônus "bom" (0.xx)
 * @param {Number} worstBonus Bônus "ruim" (0.xx)
 */
function calculateBonus(totalSales, reference, bestBonus, worstBonus) {
  var bonus =
      totalSales >= reference ? totalSales * bestBonus : totalSales * worstBonus;

  return bonus;
}

calculateBonus(1000, 600, 0.1, 0.05); // 100 reais
calculateBonus(500, 600, 0.1, 0.05); // 25 reais
```

O Visual Studio Code fornece por padrão uma estrutura de documentação de funções. Basta escrever a função, levar o cursor para a linha acima da mesma e digitar /** + espaço. Assim, um bloco semelhante ao da Figura será criado automaticamente. Isso é interessante pois sua função fica documentada durante a utilização, conforme Figura abaixo.

Figura 25 – Documentação de funções com o Visual Studio Code.



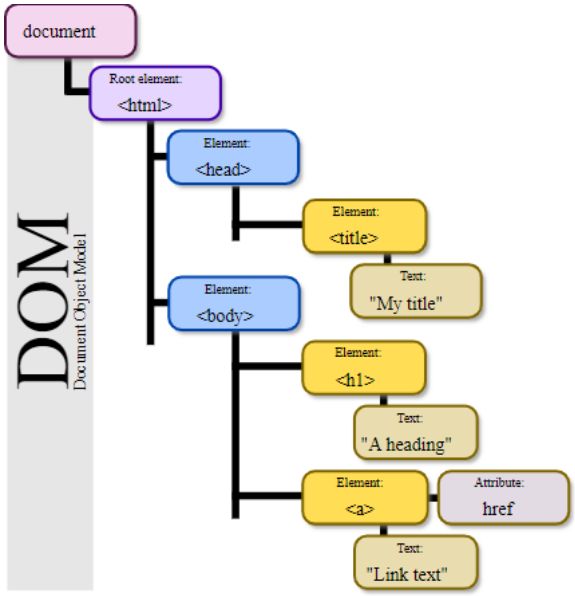
Para mais detalhes e exemplos sobre o JavaScript básico, verifique as videoaulas do capítulo 2.

Capítulo 3. DOM, Formulários e Eventos

No ecossistema de Front End, o JavaScript situa-se em um ambiente junto ao navegador e, por isso, tem acesso a uma API conhecida como DOM (*Document Object Model*), que representa os elementos da página através de uma espécie de árvore.

Através do DOM, o JavaScript consegue facilmente manipular HTML e CSS através do DOM e, com isso, realizar diversas interações com o usuário. A Figura abaixo ilustra um exemplo de DOM.

Figura 26 – Representação do DOM.



Fonte: Wikipédia.

Existem diversos comandos no JavaScript para obtenção de dados a partir do DOM. Os dois mais utilizados são:

- `querySelector.`
- `querySelectorAll`.

O comando querySelector

Este comando é utilizado para se obter **um valor** do DOM. No parâmetro, podem ser utilizados elementos (`'p'`), classes (`'.destaque'`) ou id's (`'#table'`). A Figura abaixo ilustra um exemplo de utilização de querySelector. Percebe-se que **textContent** é utilizado para extrair o **conteúdo** do elemento.

Figura 27 – Exemplo de utilização de querySelector.

```
<!-- DOM de exemplo -->
<div id="div">
  <h1>Título</h1>
  <p id="paragraph-1">Parágrafo 1</p>
  <p id="paragraph-2">Parágrafo 2</p>
  <ul>
    <li class="item">Item 1</li>
    <li class="item">Item 2</li>
    <li class="item">Item 3</li>
  </ul>
</div>

// Utilização de querySelector

// Obtendo o primeiro parágrafo
var p1 = document.querySelector('p');
console.log(p1.textContent); // "Parágrafo 1"

// Obtendo o segundo parágrafo através de id
var p2 = document.querySelector('#paragraph-2');
console.log(p2.textContent); // "Parágrafo 2"
```

O comando querySelectorAll

O querySelectorAll, diferentemente do comando anterior, é utilizado para se obter n valores do DOM, ou seja, ele retorna todos os elementos que satisfazem ao critério de busca. Como esse retorno é do tipo NodeList (específico do DOM), é comum a conversão desse valor para array através de Array.from. A Figura abaixo mostra um exemplo.

Figura 28 – Exemplo com querySelectorAll.

```
<!-- DOM de exemplo -->
<div id="div">
  <h1>Título</h1>
  <p id="paragraph-1">Parágrafo 1</p>
  <p id="paragraph-2">Parágrafo 2</p>
  <ul>
    <li class="item">Item 1</li>
    <li class="item">Item 2</li>
    <li class="item">Item 3</li>
  </ul>
</div>

// Utilizando querySelectorAll
var items = document.querySelectorAll('.item');
console.log(item);

var itemsArray = Array.from(items);
console.log(itemsArray);
```

```
// Retorno de NodeList e de Array
▶ NodeList(3) [li.item, li.item, li.item]
▶ (3) [li.item, li.item, li.item]
```

A propriedade textContent

Essa propriedade é utilizada para se alterar o **conteúdo** de elementos. A Figura abaixo ilustra um exemplo.

Figura 29 – Modificando o DOM com textContent.

```
<!-- DOM de exemplo -->
<div id="div">
  <h1>Título</h1>
  <p id="paragraph-1">Parágrafo 1</p>
  <p id="paragraph-2">Parágrafo 2</p>
  <ul>
    <li class="item">Item 1</li>
    <li class="item">Item 2</li>
    <li class="item">Item 3</li>
  </ul>
</div>

var items = document.querySelectorAll('.item');
console.log(items);

var itemsArray = Array.from(items);
console.log(itemsArray);

for (var i = 0; i < itemsArray.length; i++) {
  var currentItem = itemsArray[i];
  currentItem.textContent = 'Valor ' + (i + 1);
}
```

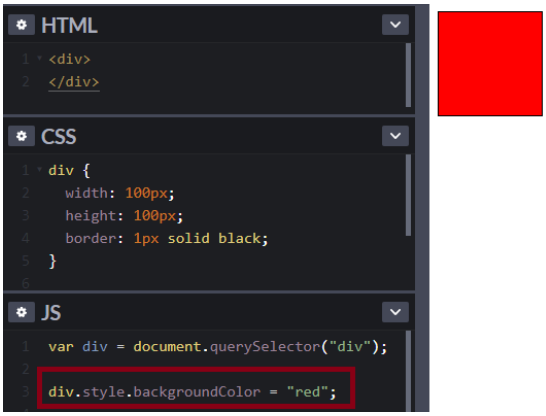
```
<!-- DOM modificado para -->
<ul>
  <li class="item">Valor 1</li>
  <li class="item">Valor 2</li>
  <li class="item">Valor 3</li>
</ul>
```

Manipulando o CSS com JavaScript

Quanto à manipulação de CSS, uma **prática desaconselhada** é a utilização propriedade **`style`**, conforme Figura abaixo. Isso ocorre porque há uma mistura entre

JavaScript e CSS. A recomendação é que as tecnologias sejam, em regra, utilizadas isoladamente.

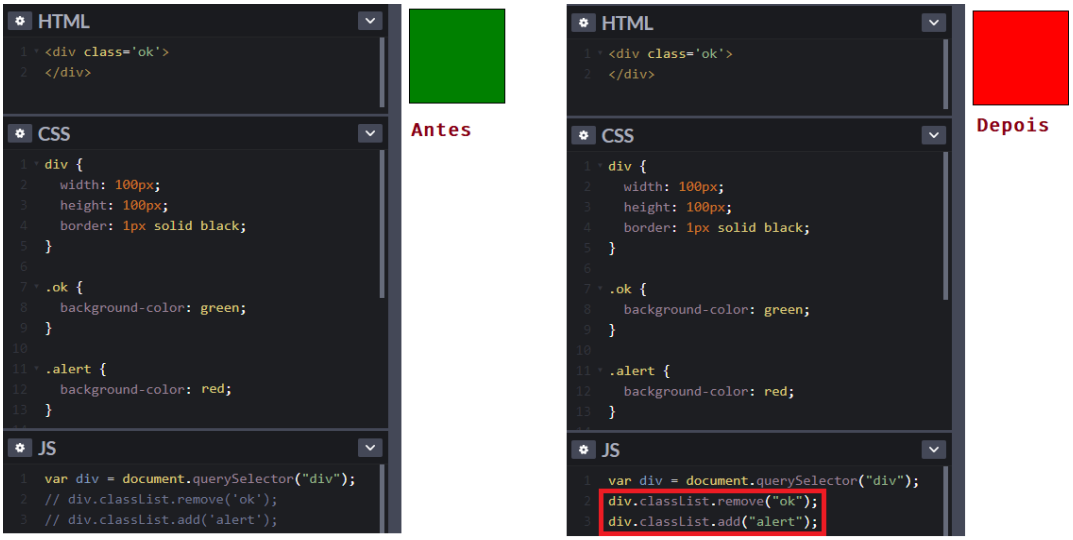
Figura 30 – Utilização da propriedade style.



Fonte: codepen.io.

Para que esse isolamento seja possível, há a opção de **adição e/ou remoção de classes** através das funções **`classList.add`** e **`classList.remove`**, respectivamente. Assim, todo o CSS continua concentrado em seu respectivo arquivo. A Figura abaixo mostra um exemplo de utilização desses comandos.

Figura 31 – Exemplo de utilização de `classList.add` e `classList.remove`.

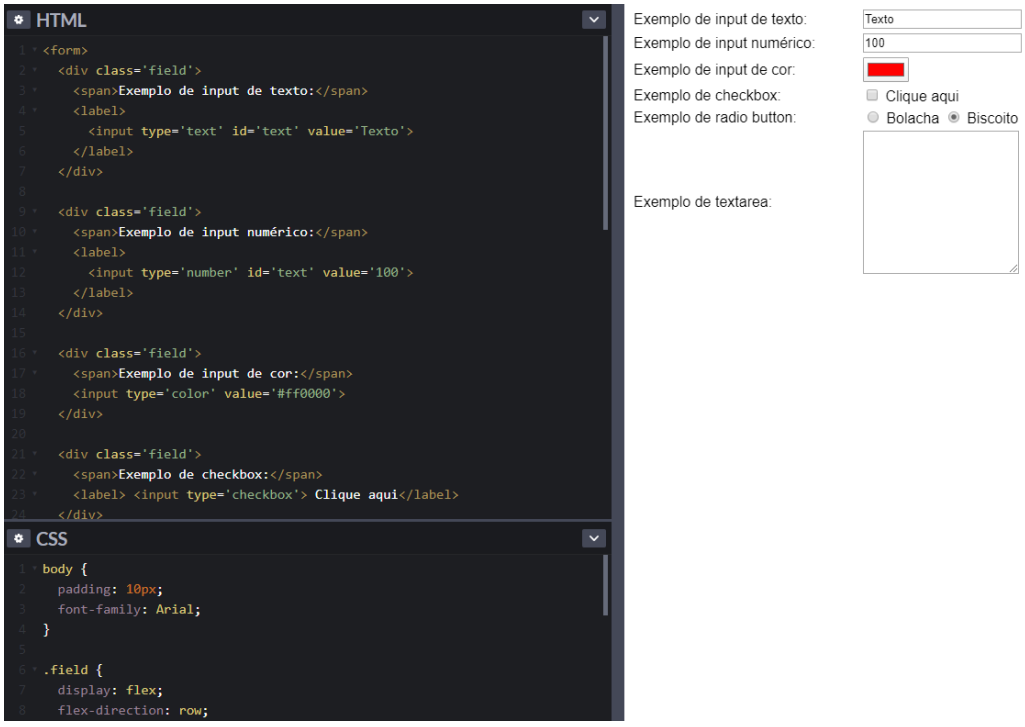


Fonte: codepen.io.

Formulários HTML

A interação do usuário com o sistema se dá em grande parte através de formulários. Dentre os elementos disponíveis, destacam-se **input**, **textarea** e **button**. Mais detalhes de como manipular formulários com JavaScript serão vistos posteriormente tanto nesta apostila quanto nas videoaulas. A Figura abaixo ilustra um trecho de exemplo de escrita de formulário. Para mais detalhes, acesse o link da fonte.

Figura 32 – Exemplo de formulário HTML.



Fonte: codepen.io.

Manipulação de eventos

Eventos, de forma semelhante ao **se** das **estruturas de decisão** e ao **enquanto** das **estruturas de repetição**, representam o **quando**. A Figura abaixo representa os diversos tipos de eventos existentes no ecossistema web.

Figura 33 – Eventos com JavaScript no Front End.

Evento	Ativação	Categoria
onload	Após o carregamento	document / window / body
onchange	Conteúdo do elemento alterado	form
onfocus	Elemento recebe foco	form
onblur	Elemento perde o foco	form
onselect	Elemento é selecionado	form
onsubmit	Dados do formulário são enviados ao servidor	form
onkeydown	Tecla pressionada	teclado
onkeypress	Tecla pressionada e solta	teclado
onkeyup	Tecla solta	teclado
onclick	Clique do mouse no elemento	mouse
ondblclick	Clique duplo do mouse no elemento	mouse
onmousemove	Mouse se moveu sobre o elemento	mouse
onmouseout	Mouse saiu do elemento	mouse
onmouseover	Mouse passou sobre o elemento	mouse
onmouseup	Botão do mouse solto sobre o elemento	mouse

Dentre os eventos listados na Figura anterior, esses são os mais comuns:

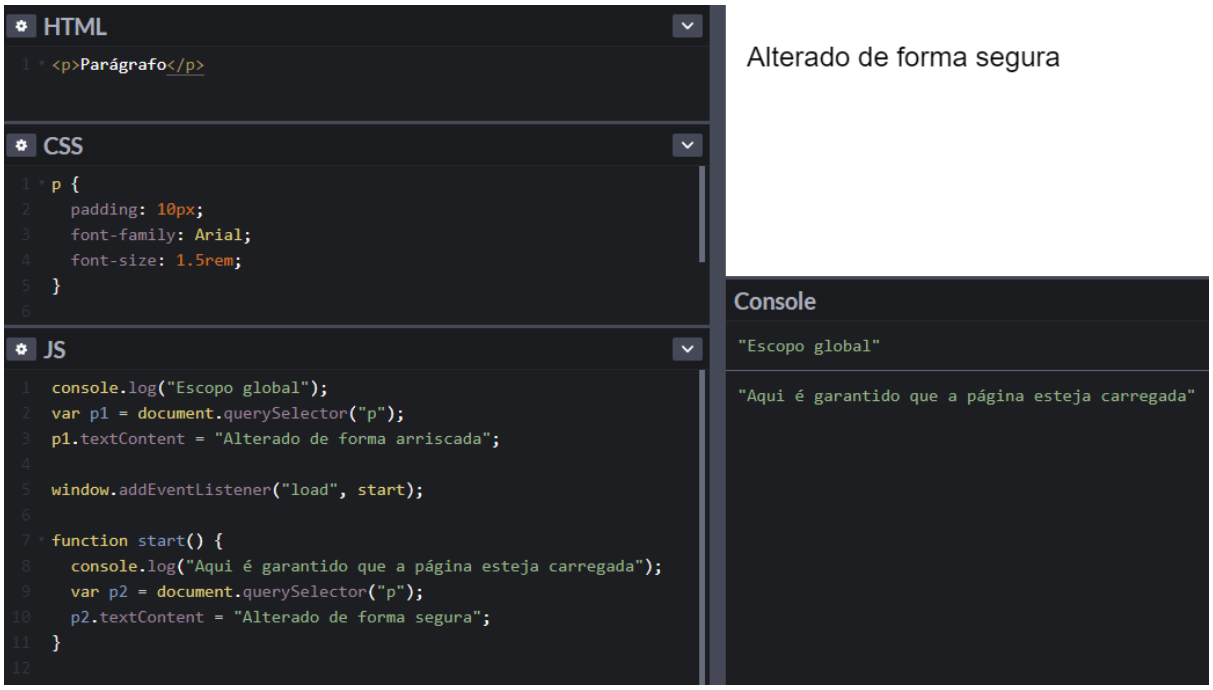
- Eventos de página.
- Eventos de formulário.
- Eventos de mouse.
- Eventos de teclado.

Eventos de página

O evento de página mais utilizado é o load (carregamento). Ele é disparado quando a página HTML terminou de ser renderizada pelo navegador. É um bom local para iniciar a manipulação do DOM, caso necessário. Para a manipulação de eventos com JavaScript é utilizada a função **addEventListener**.

A Figura abaixo ilustra um exemplo com uma função (start) vinculada ao evento **load** da página.

Figura 34 – Exemplo com evento 'load'.



Fonte: codepen.io.

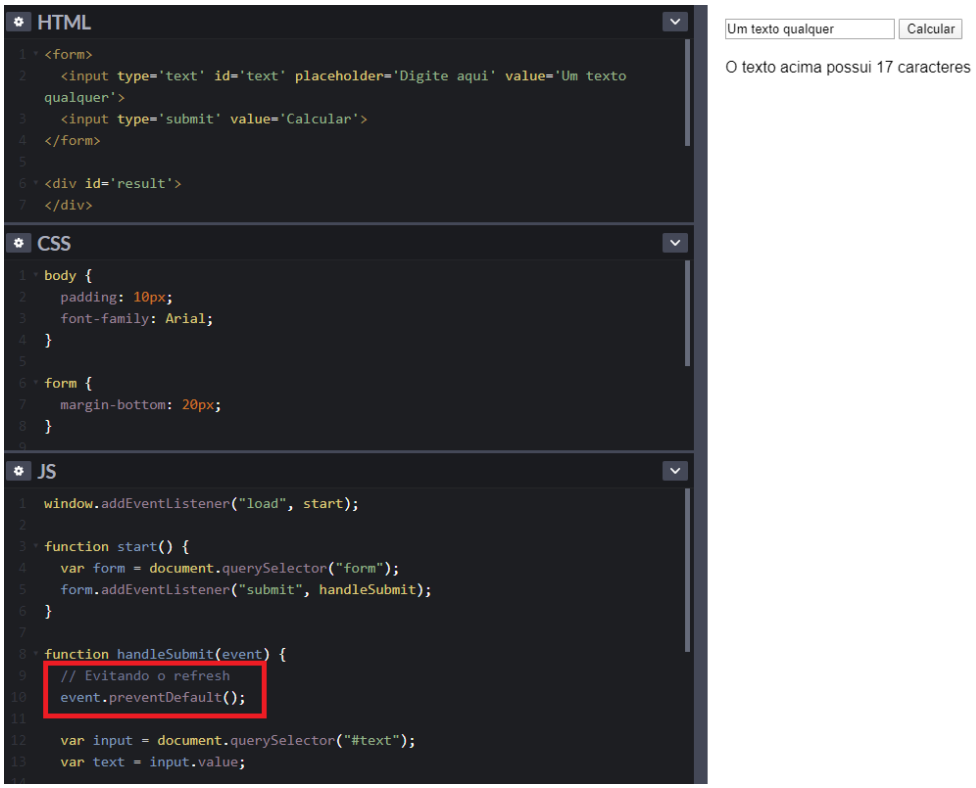
Eventos de formulário

O principal evento vinculado ao formulário é o **submit**. O comportamento padrão é que todos os dados vinculados ao formulário — através dos inputs, textareas etc., — sejam enviados ao servidor e, com isso, há o recarregamento da página (**refresh**).

Entretanto, a tendência atual é que as aplicações web sejam SPA's (Single Page Applications). Assim, não é mais comum que exista o refresh, pois as aplicações se comportam como aplicações desktop. Pode-se dizer que os dados são trabalhados em "tempo real" e que, caso exista necessidade de requisições ao servidor, elas são feitas de forma assíncrona, sem a necessidade do recarregamento da página. Mais detalhes sobre requisições HTTP serão vistos mais à frente na apostila e também nas videoaulas.

A Figura abaixo mostra como isso pode ser implementado, graças à função **preventDefault** presente no objeto de evento. Esse objeto de evento é comumente nomeado como **event** e é vinculado à função atrelada ao evento. Para mais detalhes sobre esta implementação, acesse o link da fonte.

Figura 35 – Exemplo com preventDefault.

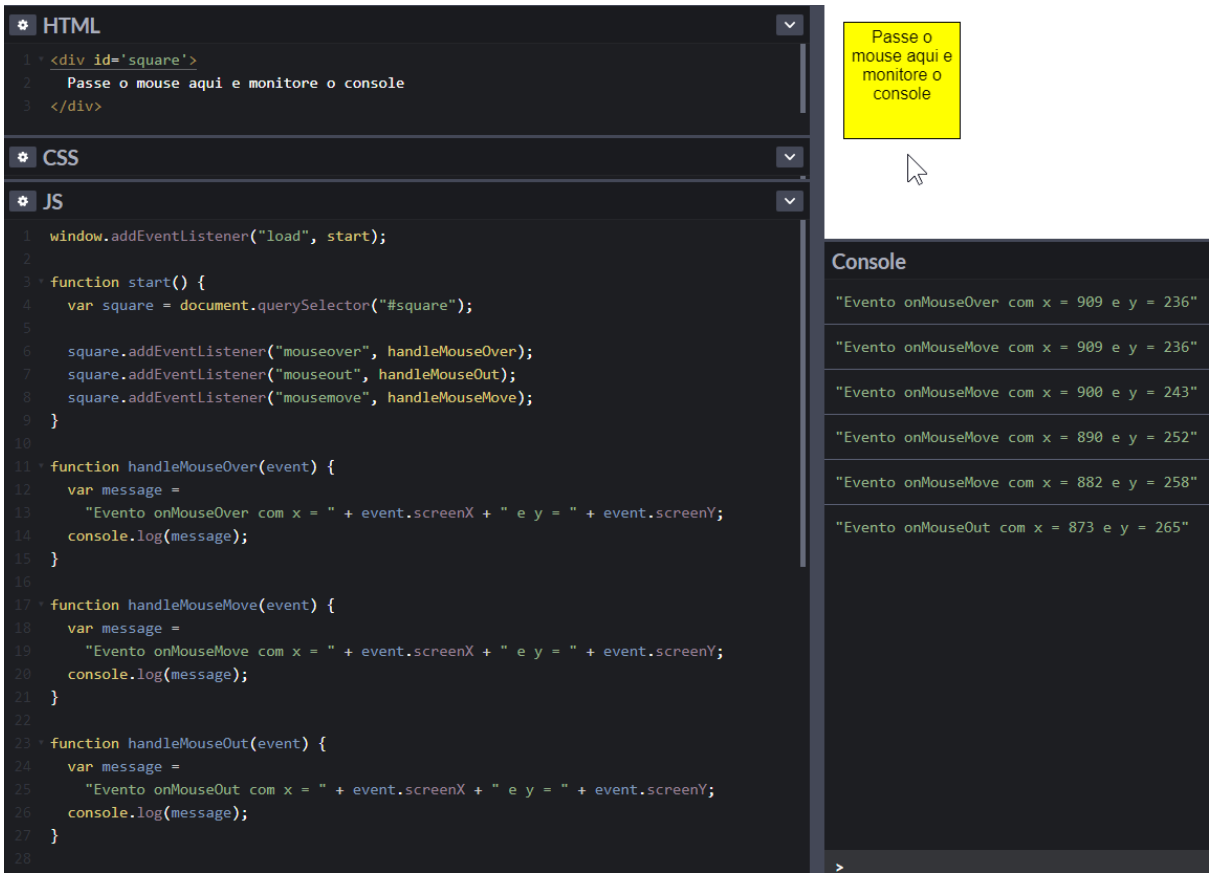


Fonte: codepen.io.

Eventos de mouse

Os eventos de mouse mais utilizados são **click**, **doubleclick**, **mouseover**, **mousemove** e **mouseout**. A Figura abaixo mostra alguns eventos de mouse. Foi feita uma implementação para monitorar um quadrado (representado por um elemento). Mais detalhes podem ser vistos no link da fonte.

Figura 36 – Exemplos de eventos de mouse.

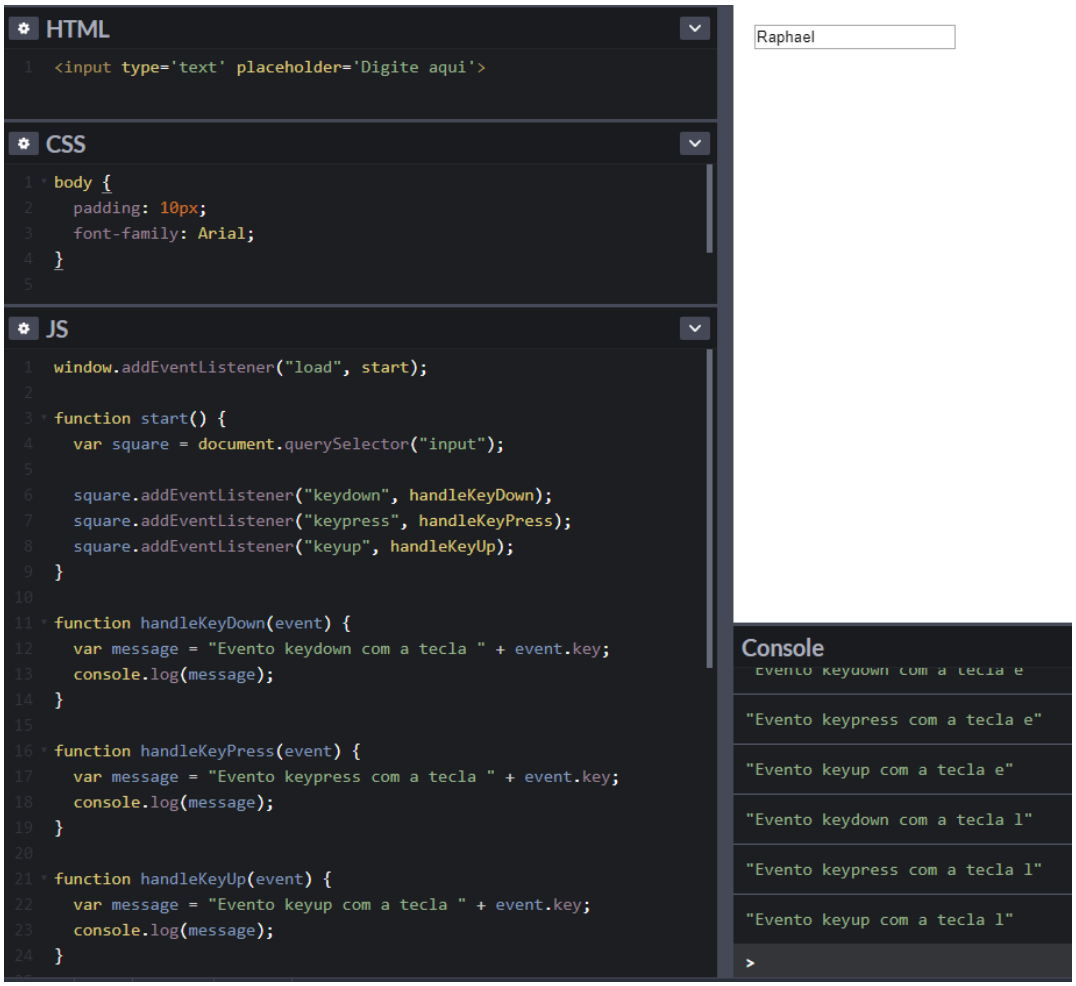


Fonte: codepen.io.

Eventos de teclado

Os eventos de teclado são geralmente aplicados a inputs e os mais utilizados são **keydown**, **keypress**, **keyup, change, focus e blur**. A Figura abaixo mostra alguns eventos de teclado. Mais detalhes podem ser vistos no link da fonte.

Figura 37 – Exemplos de eventos de teclado.



Fonte: codepen.io.

Para mais detalhes sobre eventos, incluindo a resolução de um desafio, verifique as videoaulas deste capítulo.

Capítulo 4. JavaScript moderno

Este capítulo demonstra alguns recursos muito interessantes do JavaScript que não necessariamente funcionam em todos os navegadores e são, por isso, considerados parte do JavaScript moderno. A adoção do JavaScript moderno é gradual e lenta nos navegadores porque há risco de problemas, o que pode afetar a World Wide Web como um todo. A entidade responsável por isso é a TC39. Para simplificar, o termo ES6+ (ECMAScript 6 e superiores) será utilizado para se referir ao JavaScript moderno daqui em diante.

Esta questão de código moderno não funcionar em navegadores pode ser resolvida com o apoio de transpiladores, como o Babel. Essas ferramentas convertem o código moderno em código compatível durante o processo de build da aplicação. Isso garante uma boa experiência de desenvolvimento e ao mesmo tempo garante uma aplicação que funcione na grande maioria dos navegadores.

De qualquer forma, todos os exemplos abaixo serão executados no Google Chrome, que tem um excelente suporte ao JavaScript moderno sem a necessidade de utilização de transpiladores, pelo menos por enquanto.

Entendendo var x const x let

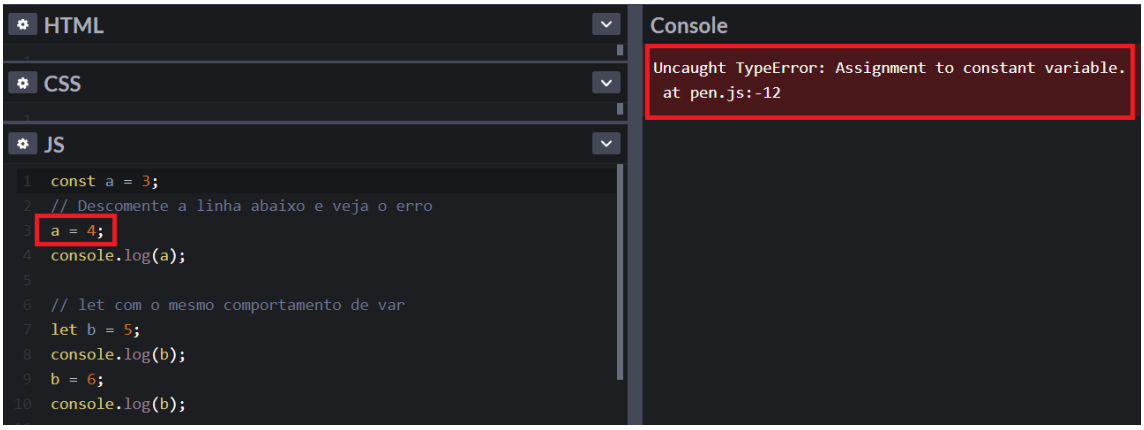
As palavras-chave **const** e **let** foram introduzidas em ES6+ em substituição ao **var**. Assim, recomenda-se a **substituição total de var** por **const** ou **let**.

Utiliza-se **const** para variáveis que **não serão** ou **não podem ser reatribuídas** posteriormente. Recomenda-se, inclusive, que o desenvolvedor sempre utilize **const** e troque para **let** caso julgue necessário posteriormente.

Já o comportamento de **let** é semelhante ao de **var**, ou seja, permite **reatribuições**. Entretanto, **let** possui um escopo mais reduzido que **var**, o que pode garantir mais estabilidade e menos bugs no seu código.

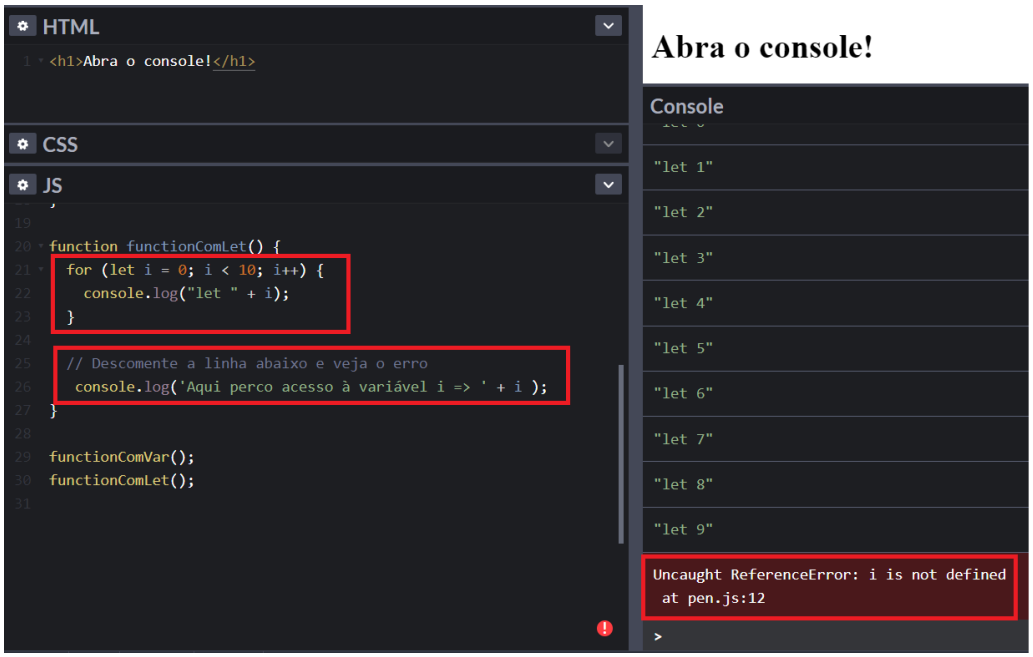
As figuras abaixo ilustram exemplos com **const** e **let**, demonstrando suas importâncias perante a utilização de **var**. Para mais detalhes, acesse o link da fonte.

Figura 38 – Exemplo de utilização de const.



Fonte: codepen.io.

Figura 39 – Exemplos de utilização de let.



Fonte: codepen.io.

Sobre a Figura acima, percebe-se que foi criada a variável **i** com **let** dentro de um comando **for**. Como o **escopo** de **let** é **reduzido**, a variável **i deixa de existir**

ao final do **for**. Isso pode garantir **mais estabilidade** e **menos bugs** no código. Com **var**, teríamos acesso a **i** após o bloco do **for**.

Arrow functions

Arrow functions representam uma forma mais moderna de se criar funções. Elas permitem uma **escrita mais direta** e **declarativa** e, quando o corpo da função possui somente uma instrução, uma escrita bem **mais resumida**.

Além disso, o escopo interno de **arrow functions** (utilização da palavra-chave **this**) é melhor **deduzível** e isso **facilita** o desenvolvimento com **React**, por exemplo. Esta questão foge do escopo desta disciplina, mas para quem se interessar, é possível ler mais a respeito aqui.

A Figura abaixo mostra alguns exemplos de escrita de funções com **arrow functions** onde, ao invés de se utilizar a palavra-chave **function**, é declarada uma variável com **const**.

Figura 40 – Exemplo de utilização de arrow functions.

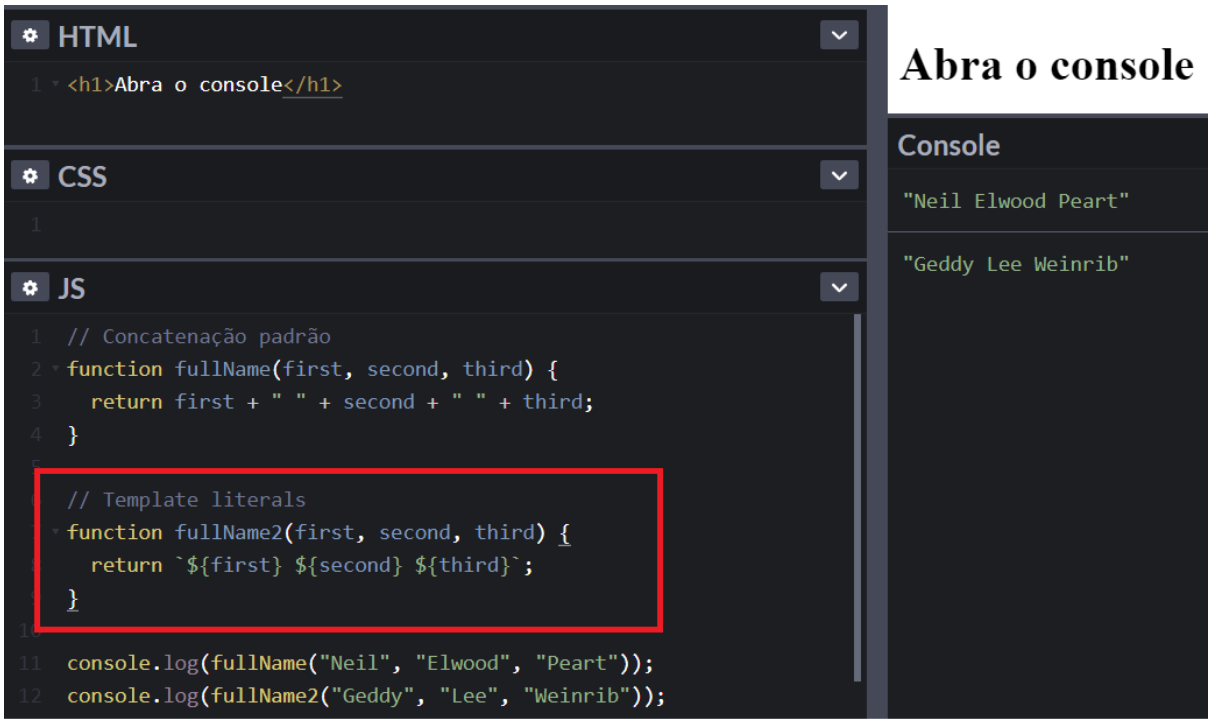
```
// Function padrão
function sum(a, b) {
  return a + b;
}

// Arrow function "completa"
const sum2 = (a, b) => {
  return a + b;
}

// Quando há apenas uma instrução,
// pode-se reduzir ainda mais a escrita,
// removendo as chaves do bloco e também
// a palavra chave return
const sum3 = (a, b) => a + b;
```

Fonte: codepen.io.

Template literals

A utilização de **template literals** é uma forma mais moderna de se escrever **Strings**, utilizando o operador ` (crase). Isso evita diversas concatenações de **Strings** com **+**, por exemplo. Utilizamos **${}** para expressões JavaScript dentro das **Strings** delimitadas com `. A Figura abaixo demonstra um exemplo.

Figura 41 – Exemplo com template literals.

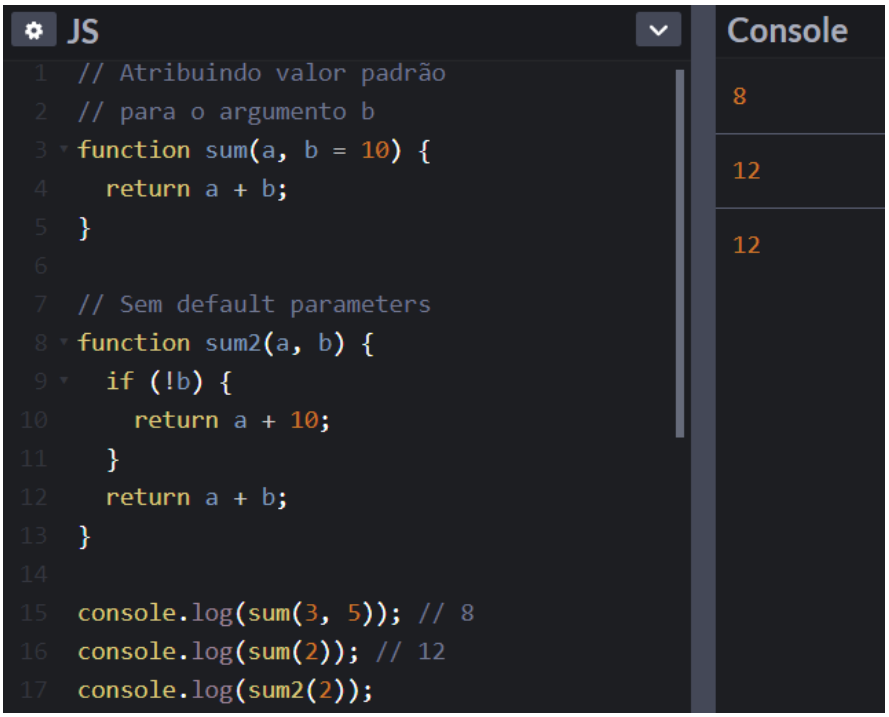


Fonte: codepen.io.

Default parameters

Em ES6+, as funções possuem um valor padrão para argumentos caso nenhum tenha sido fornecido. Na prática, isso pode evitar a utilização de **if/else** na implementação de funções, por exemplo deixando o código mais elegante e declarativo. A Figura abaixo demonstra um exemplo:

Figura 42 – Exemplo de utilização de default parameters.



Fonte: codepen.io.

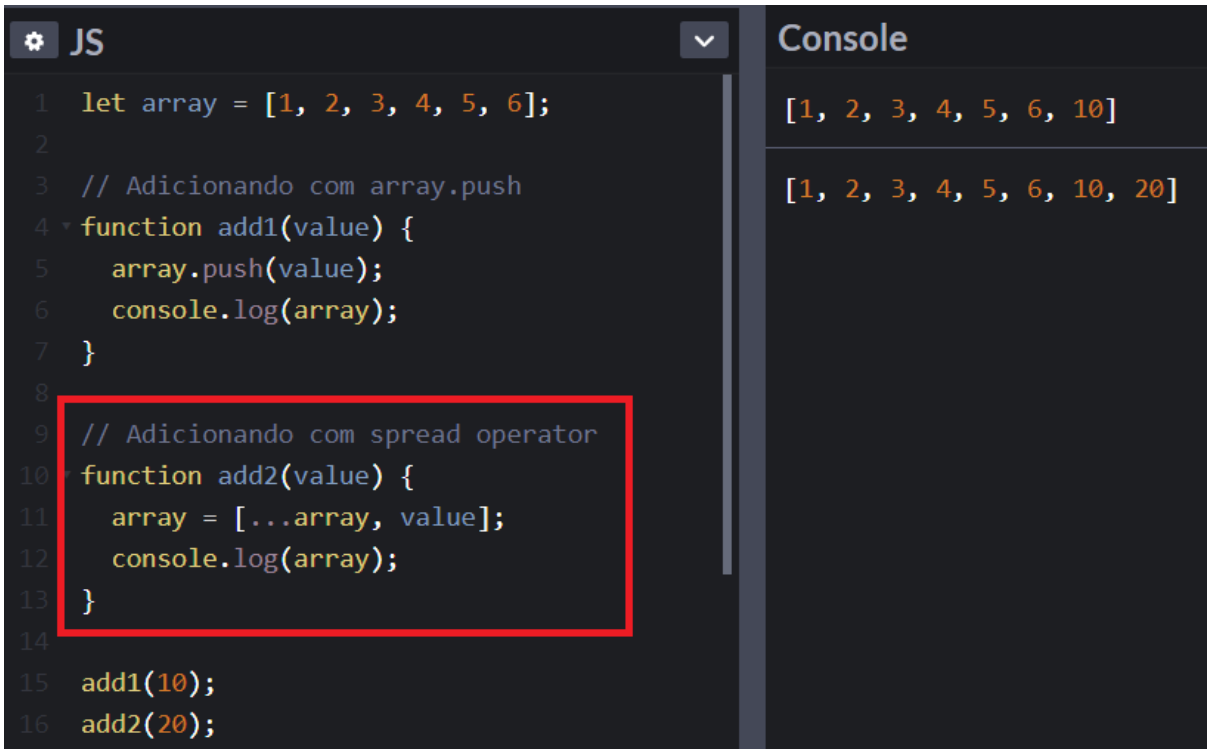
Operador ... (spread/rest)

O operador ... (três pontos em sequência) possui duas aplicações em JavaScript, que serão vistas com mais detalhes em seguida.

Spread operator (...)

Esse operador é muito utilizado em Arrays e Objetos e tem a função de **espalhar** os valores atuais. A ideia é que os valores sejam retirados do array para depois se unirem a outros valores ou um outro array, por exemplo. Essa utilização deixa a escrita mais declarativa. A Figura abaixo demonstra um exemplo.

Figura 43 – Exemplo com spread operator.



Fonte: codepen.io.

Analisando a Figura acima pode-se perceber que, na função **add2** foi criado um **array** delimitado por **[]**, contendo **o próprio array** com os **seus valores espalhados** juntamente com o valor que "chegou" na função.

Rest operator (...)

O operador **...** atua como **rest operator** em **argumentos de funções**. Assim, **o argumento passa a ser um array**. Isso permite construções interessantes como somas infinitas (soma com 0...n parâmetros). A Figura abaixo ilustra um exemplo.

Figura 44 – Exemplo com rest operator.

Fonte: codepen.io.

Array e object destructuring

Esta sintaxe facilita a criação de variáveis a partir de arrays com [], e objetos com {}. A Figura abaixo demonstra alguns exemplos. Para mais detalhes, acesse o link da fonte.

Figura 45 – Object e array destructuring.

```
const object = {
  id: 1,
  name: 'Neil Peart',
  band: 'Rush',
  instrument: 'Drums'
};

const array = [1, 2, 3, 4, 5];

// Método "comum" para criação de variáveis a partir de um objeto
// const id = object.id;
// const name = object.name;
// const band = object.band;
// const instrument = object.instrument;

// Método "comum" para criação de variáveis a partir de um array
// const um = array[0];
// const dois = array[1];
// const tres = array[2];
// const quatro = array[3];
// const cinco = array[4];

// Utilizando object destructuring
const { id, name, band, instrument } = object;

// Utilizando array destructuring
const [ um, dois, tres, quatro, cinco ] = array;
```

Fonte: codepen.io.

Array methods

Os métodos a seguir não são, na verdade, exclusividade do ES6+. Eles já existiam em versões anteriores do ECMAScript. Entretanto, eles se tornaram mais populares e, portanto, muito mais utilizados com o advento das **arrow functions**. Para cada método a ser apresentado, é possível perceber que o **mesmo recebe uma função como argumento**. Esta função é executada **para cada item** a ser **iterado** do array. É **comum** a denominação do parâmetro da função interna como **“item”** ou então o **nome do array no singular**.

Array.map()

A função **map** pode ser utilizada para uma **transformação de dados** do array. **Array.map** é imutável, ou seja, cria um novo array. A função interna deve retornar um novo **item** transformado de alguma forma, conforme a lógica.

Array.filter()

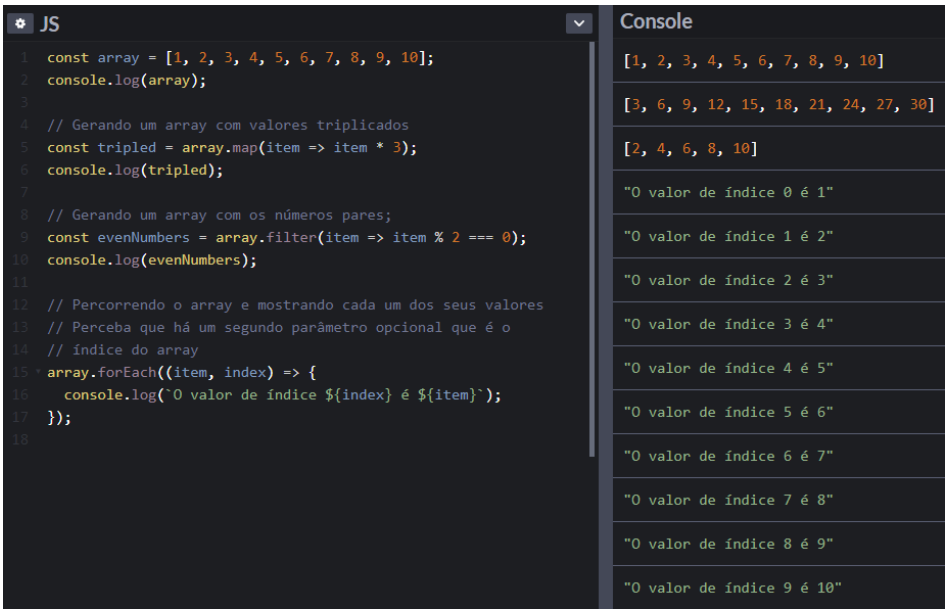
A função **filter** pode ser utilizada para **eliminar itens** do array. **Array.filter** é imutável, ou seja, cria um novo array. A função interna deve retornar **true** ou **false** conforme a lógica.

Array.forEach()

A função **forEach** pode ser utilizada para **percorrer todos os elementos** do array, aplicando alguma lógica.

A Figura abaixo demonstra exemplos com **Array.map**, **Array.filter** e **Array.forEach**. Para mais detalhes, acesse o link da fonte.

Figura 46 – Exemplos com Array.map, Array.filter e Array.forEach.



Fonte: codepen.io.

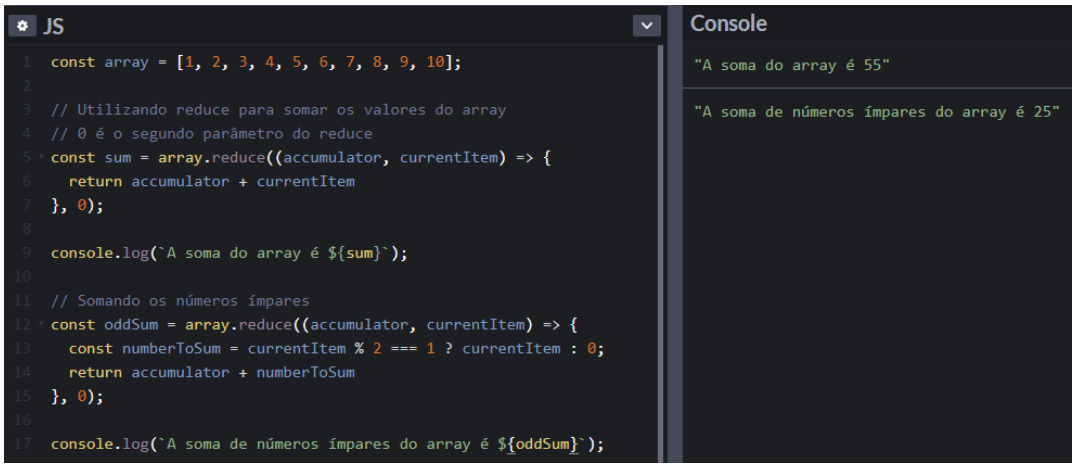
Array.reduce()

A função **reduce** pode ser utilizada para **percorrer todos os elementos** do array, aplicando alguma **lógica de cálculo iterativa**, ou seja, o resultado final é **um valor** e não um array.

A função interna do **reduce** recebe, por padrão, **dois parâmetros**: o **primeiro** é o **acumulador** — valor acumulado a cada iteração — e o **segundo** é o **item atual**. Além disso, o **reduce** aceita como **segundo parâmetro**, após a função, um **valor inicial**.

A Figura abaixo demonstra alguns exemplos com reduce. Para mais detalhes, acesse o link da fonte.

Figura 47 – Exemplos com Array.reduce.



Fonte: codepen.io.

Array.find()

A função **find** percorre o **array** em **busca do primeiro elemento** que satisfaça a condição da função e **retorna esse elemento**. Caso não encontre nenhum elemento, retorna **undefined**.

Array.some()

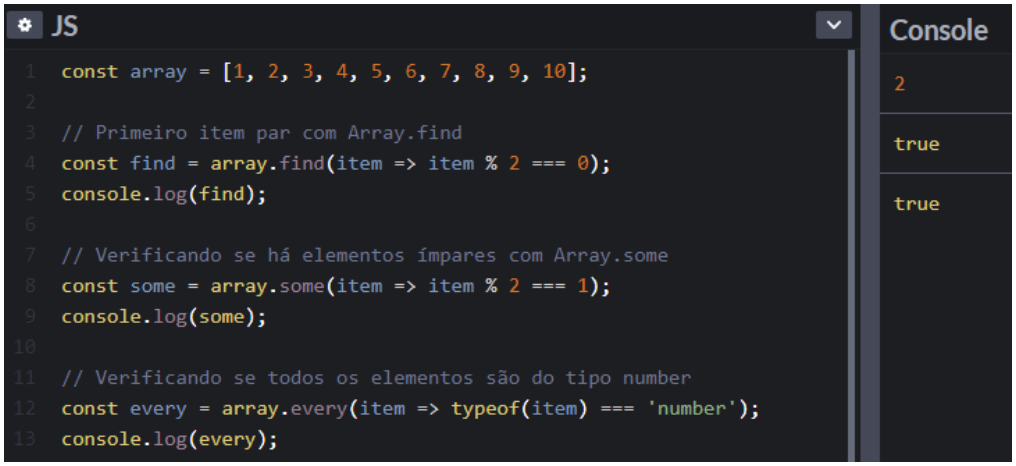
A função **some** percorre o **array** em **busca do primeiro elemento** que satisfaça a condição da função e **retorna true ou false**.

Array.every()

A função **every** percorre o **array** e verifica se **todos os elementos** satisfazem à condição da função, **retornando true ou false**.

A Figura abaixo demonstra exemplos com **Array.find**, **Array.some** e **Array.every**. Para mais detalhes, acesse o link da fonte.

Figura 48 – Exemplos com Array.find, Array.some e Array.every.



Fonte: codepen.io.

Array.sort()

A função **sort** percorre o **array** e aplica o algoritmo de ordenação informado na função do parâmetro.

A função **sort** é **mutável**, ou seja, altera o próprio **array**.

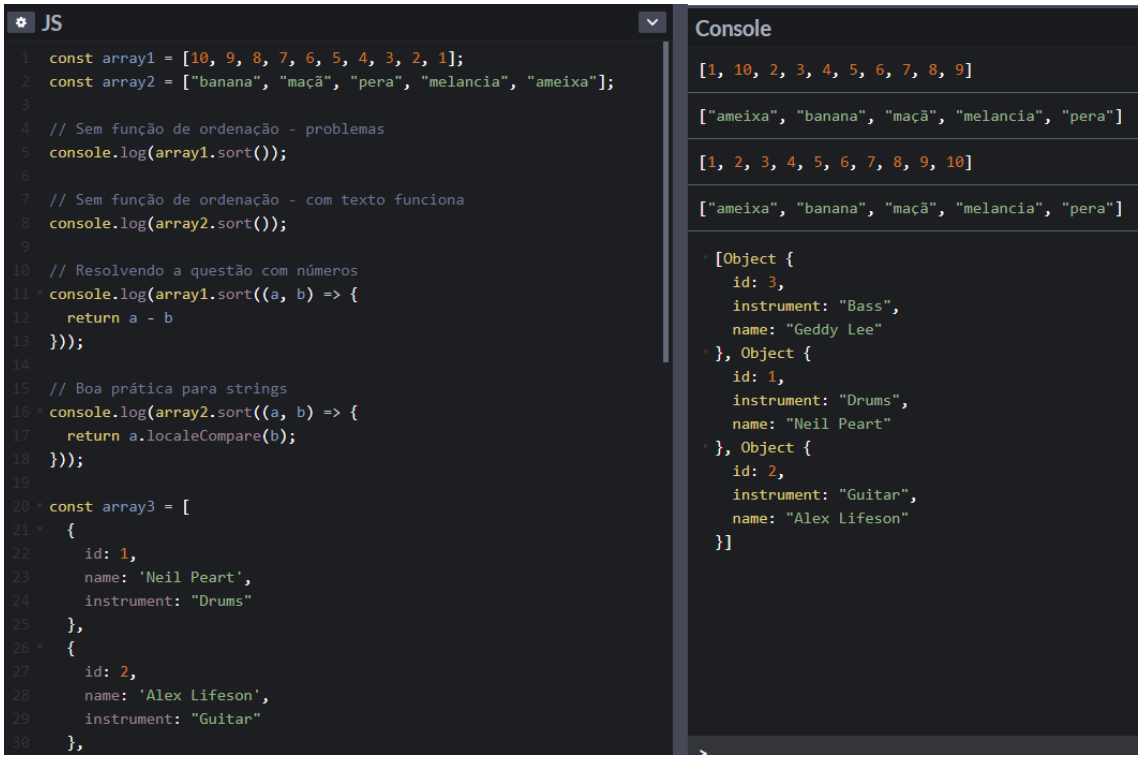
Se não há função de ordenação no parâmetro, **sort** converte os valores para **Strings** e faz a ordenação a partir dos caracteres. Isso gera problemas com **arrays** com números, por exemplo, pois “80” < “9” mas na verdade 80 > 9. Por isso é uma boa prática sempre informar a função de ordenação.

A função de ordenação a ser implementada recebe dois parâmetros e, em regra, exige um retorno de:

- Número negativo → Primeiro elemento é “menor” que o segundo.
- Zero (0) → Elementos equivalentes.
- Número positivo → Primeiro elemento é “maior” que o segundo.

A Figura abaixo demonstra alguns exemplos com **Array.sort**.

Figura 49 – Exemplos com Array.sort.



Fonte: codepen.io.

Para mais detalhes sobre métodos de Arrays, verifique as videoaulas do capítulo 4.

Capítulo 5. Programação assíncrona com JavaScript

O funcionamento do processamento de instruções do JavaScript, em geral, se dá através da call stack, que é uma pilha lógica onde as instruções são executadas uma a uma sequencialmente.

Entretanto, existem algumas operações no ecossistema JavaScript que podem ser lentas, como por exemplo as requisições HTTP. Essas são lentas porque parte do processamento depende do computador no destino (servidor) e do tráfego de rede. Assim, não é possível prever **quando** a resposta estará pronta, mas é possível determinar o que fazer com a resposta **quando** ela "chega".

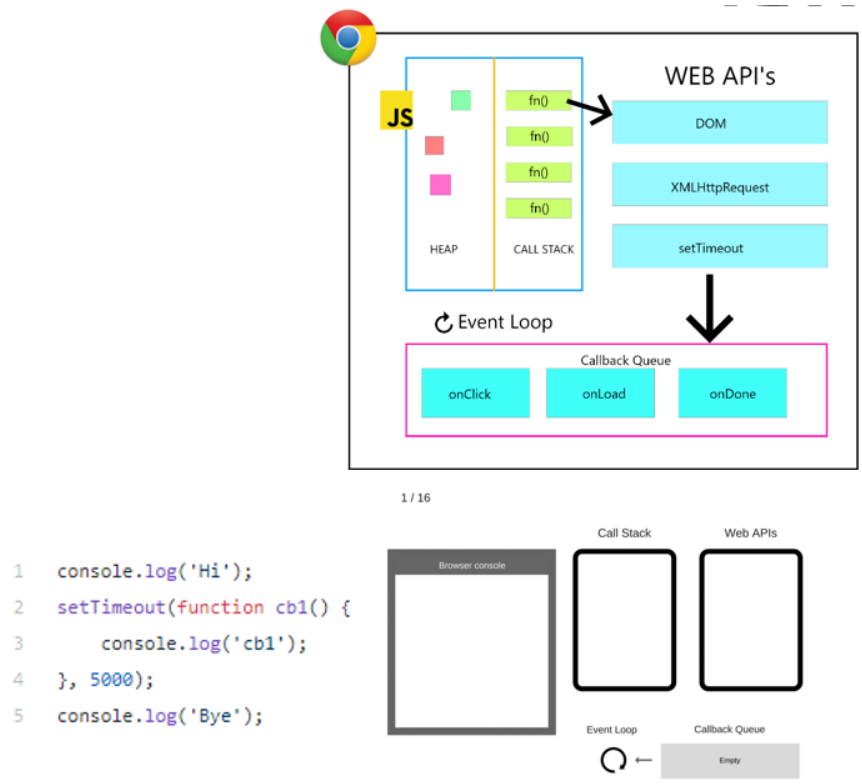
Para esses casos o JavaScript, por padrão, **não espera a execução**; o que faz total sentido, pois caso contrário a aplicação poderia travar, o que prejudicaria muito a experiência do usuário. Para isso, o JavaScript utiliza internamente um recurso conhecido como **event loop**. O event loop está presente internamente nos diversos ecossistemas onde o JavaScript pode atuar (Node.js, web etc.).

No caso da web, onde há DOM, instruções consideradas potencialmente lentas situam-se em uma camada lógica conhecida como Web API's, cuja responsabilidade de execução é do navegador. Cada instrução desse tipo entra em uma fila denominada callback queue, que executa as instruções uma a uma sempre que houver disponibilidade, ou seja, o event loop sempre sinaliza. Isso é feito monitorando a **call stack**, que é a pilha de execução de instruções padrão. Com isso, a execução do fluxo principal do JavaScript não para e essas instruções são retornadas e os dados processados sempre que possível.

Entretanto, cabe ao desenvolvedor implementar um recurso para capturar o momento em que essas instruções finalizam e por fim trabalhar com os dados retornados. Em linha gerais, esse recurso é conhecido como callback, que é representado por uma função passada por parâmetro para esse tipo de instrução.

A Figura abaixo demonstra o funcionamento básico do event loop.

Figura 50 – Funcionamento do event loop.



Fonte: sessionstack.com.

Analisando a Figura acima é possível perceber que os eventos do DOM (que já foram vistos anteriormente), requisições HTTP e manipulação de tempo (setTimeout/setInterval) são exemplos de instruções que são executadas com o apoio do event loop e permitem a programação assíncrona com JavaScript. Quando o assunto é programação assíncrona, lembre-se sempre do **quando**, que é a palavra-chave nesses casos.

Para mais detalhes sobre o event loop, verifique as videoaulas do capítulo 5 e também este artigo, que inclui, inclusive, uma animação bastante interessante sobre o funcionamento do event loop.

As funções *setTimeout* e *setInterval*

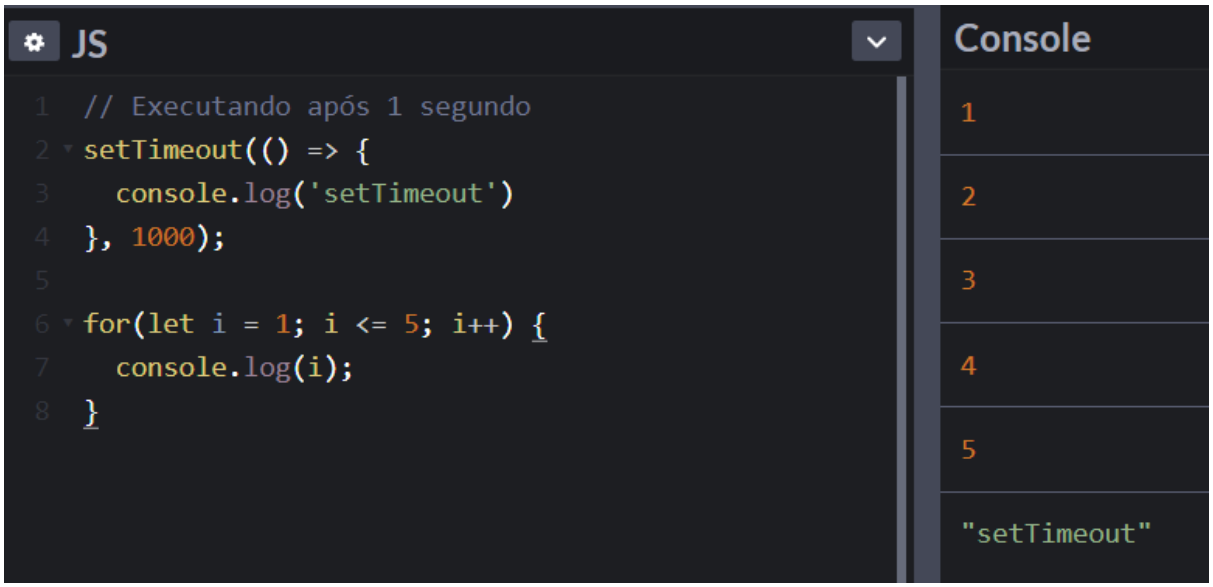
Essas funções são utilizadas para manipulação de tempo no JavaScript. Alguns exemplos serão demonstrados a seguir.

setTimeout

Essa função é utilizada para **postergar** a execução de uma função **após x milissegundos**, aproveitando-se do event loop. A função interna ao **setTimeout** (**callback**) é geralmente expressa como **arrow function** devido a uma melhor legibilidade do código.

A Figura abaixo demonstra um exemplo de implementação com setTimeout. Para mais detalhes acesse o link da fonte.

Figura 51 – Exemplo com setTimout.



Fonte: codepen.io.

Analisando a Figura acima é possível perceber que, mesmo que o comando **setTimeout** tenha sido **invocado primeiro**, ele foi de fato executado **após** as instruções do comando **for** devido, claro, à sua postergação de 1 segundo (1000ms). É importante frisar também que **não é garantido** que a execução da função de

setTimeout seja feita **exatamente após 1000ms**, pois isso depende da **disponibilidade** do **event loop**.

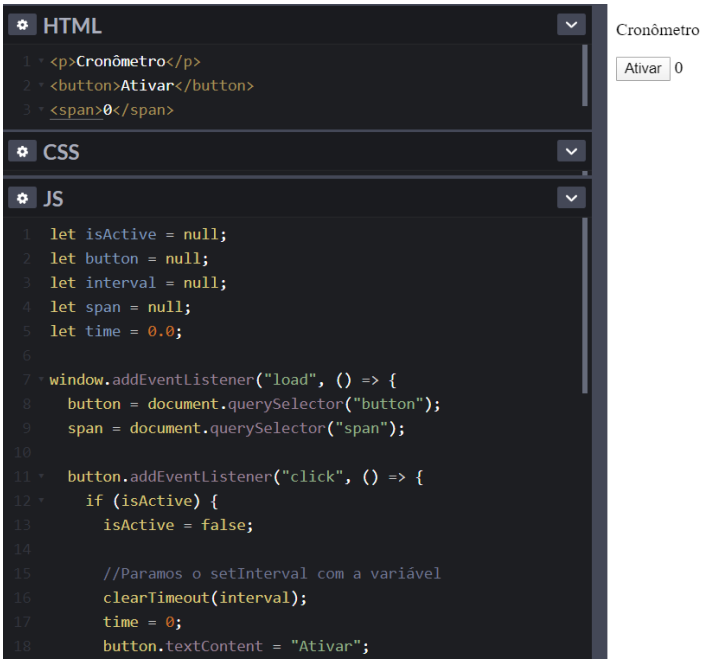
Para mais detalhes sobre o comando setTimeout, verifique as videoaulas do capítulo 5.

setInterval

Com **setInterval**, a execução do callback é **repetida a cada x milissegundos**. Portanto, é importante que a referência seja guardada em uma variável para uma posterior paralização através da função **clearInterval**. Caso isso não aconteça, podem ocorrer bugs de vazamento de memória, cuja detecção é geralmente difícil.

A Figura abaixo demonstra um exemplo de implementação com setInterval. Para mais detalhes, acesse o link da fonte.

Figura 52 – Exemplo com setInterval.



Fonte: codepen.io.

Para mais detalhes sobre o comando setInterval, verifique as videoaulas do capítulo 5.

Requisições HTTP com fetch, Promises e Async/Await

Como visto anteriormente, as requisições HTTP também fazem parte de um conjunto de instruções que não são executadas imediatamente. Nesse caso, há processamento por parte do servidor e não é possível prever **quando** o resultado será retornado. Entretanto, é possível **"escutar"** esse retorno e realizar algum processamento **quando** isso acontecer.

O comando mais utilizado para requisições HTTP atualmente é o **fetch.** Esse comando é um tanto quanto peculiar pois utiliza o conceito de **Promises**.

Uma **promise** é uma estrutura utilizada pelo desenvolvedor para **executar** instruções que necessitam do **event loop** e ao mesmo tempo **definir** o que será feito quando a instrução for **resolvida** (a instrução foi executada corretamente) e/ou rejeitada (ocorreu algum erro na instrução, como falha na conexão com a internet). Assim, promises representam **promessas** que podem ser **cumpridas/resolvidas** ou **rejeitadas** no futuro.

As instruções a serem executadas quando a **promise** é **resolvida** são inseridas em forma de callback em **then**, e as instruções a serem executadas quando a **promise** é **rejeitada** são inseridas em forma de callback em **catch**.

É importante salientar que nós, desenvolvedores, podemos criar nossas próprias promises. Entretanto, o mais comum é que somente utilizemos promises a partir de APIs conhecidas, como o fetch, por exemplo.

A Figura abaixo demonstra a criação de uma promise simulando uma operação de divisão, que normalmente é resolvida a não ser que o divisor seja 0. Para mais detalhes, acesse o link da fonte.

Figura 53 – Exemplo de criação de Promise.

Fonte: codepen.io.

Utilizando o comando fetch

O comando **fetch** realiza uma **requisição HTTP** a partir da **URL** passada no **parâmetro** e retorna, em caso de sucesso, uma **promise** com dados binários. Em geral, esses dados estão no formato **JSON** e é possível realizar a conversão através do método **json()**, que também retorna uma outra promise que, em caso de sucesso, retorna finalmente os dados propriamente ditos.

A Figura abaixo demonstra um exemplo com fetch sendo resolvido e rejeitado.

Figura 54 – Exemplos com fetch.

Fonte: codepen.io.

Analisando a Figura acima é possível perceber que foi forçado o erro na segunda execução de **fetch** com alteração na URL. Além disso, a segunda **promise** foi finalizada antes da primeira, ou seja, **não é garantida a ordem de execução**. Isso é programação assíncrona!

Utilizando async/await

Em ES6+ existe uma forma de escrita de **promises** mais elegante, conhecida como **async/await**. Nela, decoramos o cabeçalho da função com a palavra-chave **async** e, em cada instrução que será executada de forma assíncrona, utilizamos **await**. Isso torna a escrita muito mais sucinta, declarativa e elegante.

A Figura abaixo demonstra um exemplo com **async/await**.

Figura 55 – Exemplo com async/await.

Fonte: codepen.io.

Para mais detalhes sobre promises e async/await verifique as videoaulas do capítulo 5, que também inclui um desafio que utiliza esses conceitos na prática.

Referências

MOZILLA DEVELOPER NETWORK WEB DOCS. Home. 2020. Disponível em: <https://developer.mozilla.org/en-US/>. Acesso em: 08 mai. 2020.