

Rômulo Francisco de Souza Maia

# Linguagens de script para web

**Dados Internacionais de Catalogação na Publicação (CIP)**  
**(Jeane Passos de Souza - CRB 8ª/6189)**

---

Maia, Rômulo Francisco de Souza

Linguagens de script para web / Rômulo Francisco de Souza Maia.  
– São Paulo : Editora Senac São Paulo, 2021. (Série Universitária)

Bibliografia.

e-ISBN 978-65-5536-616-7 (ePub/2021)  
e-ISBN 978-65-5536-617-4 (PDF/2021)

1. JavaScript (Linguagem de programação) 2. Websites – Criação  
3. Websites – Desenvolvimento I. Título. II. Série.

20-1262t

CDD – 005.133  
BISAC COM051260

---

**Índice para catálogo sistemático**

**1. JavaScript : Linguagem de programação 005.133**

# **LINGUAGENS DE SCRIPT PARA WEB**

Rômulo Francisco de Souza Maia





## Administração Regional do Senac no Estado de São Paulo

### Presidente do Conselho Regional

Abram Szajman

### Diretor do Departamento Regional

Luiz Francisco de A. Salgado

### Superintendente Universitário e de Desenvolvimento

Luiz Carlos Dourado

## Editora Senac São Paulo

### Conselho Editorial

Luiz Francisco de A. Salgado  
Luiz Carlos Dourado  
Darcio Sayad Maia  
Lucila Mara Sbrana Sciotti  
Jeane Passos de Souza

### Gerente/Publisher

Jeane Passos de Souza ([jpassos@sp.senac.br](mailto:jpassos@sp.senac.br))

### Coordenação Editorial/Prospecção

Luís Américo Tousi Botelho ([luis.tbotelho@sp.senac.br](mailto:luis.tbotelho@sp.senac.br))  
Dolores Crisci Manzano ([dolores.cmanzano@sp.senac.br](mailto:dolores.cmanzano@sp.senac.br))

### Administrativo

[grupoedsadministrativo@sp.senac.br](mailto:grupoedsadministrativo@sp.senac.br)

### Comercial

[comercial@editorasenacsp.com.br](mailto:comercial@editorasenacsp.com.br)

### Acompanhamento Pedagógico

Mônica Rodrigues dos Santos

### Designer Educacional

Hágara Rosa da Cunha Araujo

### Revisão Técnica

Gustavo Moreira Calixto

### Preparação e Revisão de Texto

Daniele Lippert

### Projeto Gráfico

Alexandre Lemes da Silva  
Emilia Corrêa Abreu

### Capa

Antonio Carlos De Angelis

### Editoração Eletrônica

Michel Iuiti Navarro Moreno  
Sidney Foot Gomes

### Ilustrações

Michel Iuiti Navarro Moreno  
Sidney Foot Gomes

### Imagens

Adobe Stock Photos

### E-pub

Ricardo Diana

Proibida a reprodução sem autorização expressa.  
Todos os direitos desta edição reservados à

### Editora Senac São Paulo

Rua 24 de Maio, 208 – 3º andar  
Centro – CEP 01041-000 – São Paulo – SP  
Caixa Postal 1120 – CEP 01032-970 – São Paulo – SP  
Tel. (11) 2187-4450 – Fax (11) 2187-4486  
E-mail: [editora@sp.senac.br](mailto:editora@sp.senac.br)  
Home page: <http://www.livrariasenac.com.br>

© Editora Senac São Paulo, 2021

# Sumário

## Capítulo 1

### Introdução às linguagens de script para web, 7

- 1 Web server-side vs client-side, 8
  - 2 JavaScript, 9
  - 3 ECMAScript, 10
  - 4 JS flavors e transpiladores, 11
  - 5 Principais frameworks, 13
  - 6 Node.js, 17
  - 7 CDN e NPM, 17
  - 8 WebAssembly, EMScripten e ASM, 18
- Considerações finais, 20  
Referências, 20

## Capítulo 2

### Sintaxe básica do JavaScript, 23

- 1 Usando o developer tools e console do browser, 23
  - 2 Tipagem dinâmica, 27
  - 3 Declaração de variáveis e constantes, 28
  - 4 Números, 33
  - 5 Strings, 34
  - 6 Booleanos, 37
  - 7 Vetores, 39
  - 8 Objetos, 42
- Considerações finais, 44  
Referências, 44

## Capítulo 3

### JavaScript funcional, 45

- 1 Funções, sintaxe básica e arrow-function, 46
  - 2 Argumentos: valor e referência, rest, default values, 48
  - 3 Hoisting, 50
  - 4 Contexto de variáveis locais, 51
  - 5 Closures, 53
  - 6 Métodos de vetores (map, reduce, filter, forEach), 53
  - 7 Generators e yield, 56
  - 8 Funções assíncronas: async e await, 58
- Considerações finais, 58  
Referências, 58

## Capítulo 4

### Orientação a objetos em JavaScript, 61

- 1 Função construtora (keywords new e this), 62
  - 2 Prototype, 62
  - 3 Classes, 63
  - 4 JSON, 68
  - 5 Spread operator, 70
  - 6 For in e for of, 72
  - 7 Map e set, 73
  - 8 TypeScript, 75
- Considerações finais, 76  
Referências, 77

## **Capítulo 5** **Document Object Model (DOM), 79**

- 1** Ligando o JavaScript ao HTML, 80
- 2** Modelo de documento por objetos (DOM – Document Object Model), 81
- 3** Selecionando elementos: getElement e querySelector, 84
- 4** Navegando, 85
- 5** Manipulando elementos, 86
- 6** Considerações finais, 89
- 7** Referências, 90

## **Capítulo 6** **Eventos e interação, 91**

- 1** Eventos no JavaScript, 92
  - 2** Adicionando e removendo listeners, 93
  - 3** Bubble e capture, 95
  - 4** Prevent default e cancelamento, 97
  - 5** Eventos de mouse, 99
  - 6** Eventos de teclado, 100
  - 7** Load, scrolling, change e outros eventos, 101
  - 8** Custom events, 103
  - 9** Interagindo com o usuário, 104
  - 10** Timers: timeout, interval, requestAnimationFrame, 106
- Considerações finais, 109
- Referências, 110

## **Capítulo 7** **Comunicação com o back-end, 111**

- 1** Coletando e manipulando dados de formulários, 112
  - 2** Requisições assíncronas: XHR e fetch, 115
  - 3** Promises, 122
  - 4** Tratando o resultado – do JSON para o DOM, 123
- Considerações finais, 126
- Referências, 127

## **Capítulo 8** **Tópicos avançados, 129**

- 1** Bibliotecas: uso, criação e distribuição (CDN e NPM), 130
  - 2** Componentes web, 137
  - 3** Shadow DOM, 137
  - 4** Storage, 139
  - 5** Websockets e webworkers, 142
  - 6** Recursos multimídia, 144
- Considerações finais, 145
- Referências, 145

## **Sobre o autor, 147**

## Capítulo 1

# Introdução às linguagens de script para web

Nos últimos anos, o setor de tecnologia tem apresentado grande desenvolvimento, tanto de hardware como de software. O setor de serviços de informática tem sido um dos focos desse crescimento, principalmente as atividades para a web. À medida que evoluímos, novas oportunidades de trabalho aparecem e outras são extintas. Nesse ritmo, o usuário não percebe a quantidade de interações tecnológicas para as páginas web que o desenvolvedor precisa conhecer para oferecer um serviço de qualidade e segurança.

As páginas web de hoje exigem um esforço muito grande da comunidade para garantir que sejam compatíveis com todos os navegadores da web e equipamentos. As linguagens scripts para a web existem para

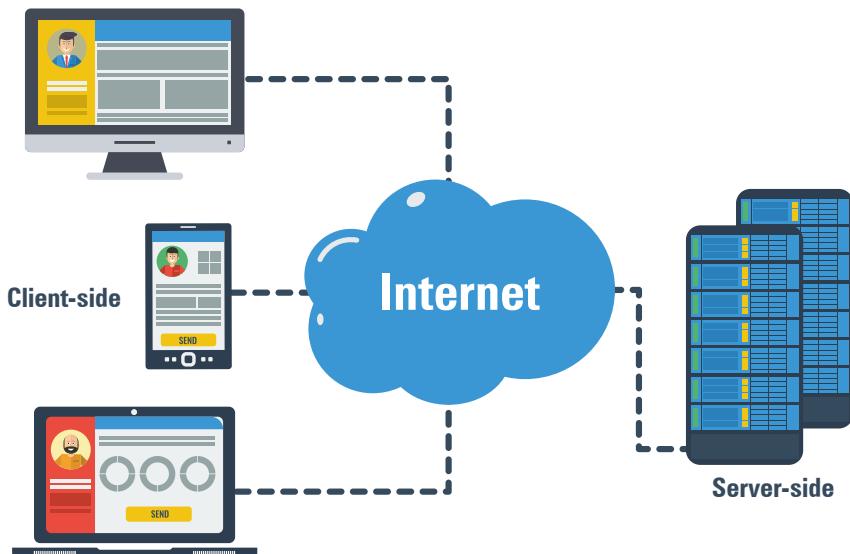
oferecer ao usuário uma execução especial de suas tarefas no server-side (ou do lado de uma estação de trabalho). Portanto, exploraremos o uso do JavaScript, que é uma linguagem de script dinâmica para o desenvolvimento de aplicações web e executadas no seu navegador de internet.

Outro aspecto fundamental abordado neste capítulo é que, a cada ano, têm surgido novas versões para atender às melhorias tecnológicas do mercado, proporcionando aos desenvolvedores vários recursos interessantes e necessidade muito grande de aprendizado contínuo.

## 1 Web server-side vs client-side

Para que o desenvolvedor disponibilize os conteúdos dos sites na web, existe uma variedade de linguagens de computadores disponíveis. Essas linguagens de desenvolvimento web dividem-se em client-side e server-side, conforme ilustra a figura 1.

Figura 1 – Cliente/servidor



O client-side representa as linguagens de programação que são executadas no navegador web do lado cliente, isto é, o browser do cliente processa toda a programação do site no lado do usuário. Quando falamos de processar no client-side, falamos da linguagem JavaScript, HTML e CSS. O desenvolvedor escolhe processar do lado cliente principalmente quando pretende diminuir a largura de banda na comunicação com o servidor web, proporcionando uma resposta rápida da aplicação ao usuário. Contudo, existe uma desvantagem: o usuário pode desabilitar essa funcionalidade no seu navegador, provocando o não funcionamento do site.

Já o server-side representa as linguagens de programação que o servidor web entende, ou seja, os sites são executados do lado servidor e enviam a resposta para o lado cliente. Algumas linguagens, como PHP, Java, C#, Python e outras, inclusive o próprio JavaScript (NodeJS), são executadas no servidor.



### NA PRÁTICA

No mercado de trabalho, é muito comum o termo front-end, para se referir às linguagens do lado cliente, e back-end, para as linguagens do lado servidor. Os desenvolvedores experientes em ambas as linguagens são chamados de full-stack.

Agora que já sabemos as diferenças entre os lados do cliente e do servidor, vamos tratar de uma importante linguagem de script para web: o JavaScript.

## 2 JavaScript

O JavaScript é uma linguagem de programação leve, interpretada e orientada a objetos com funções de primeira classe, ou seja, as funções são consideradas como valores que são manipulados (FLANAGAN,

2012). O JS é conhecido como a linguagem de scripting para páginas web, mas também é utilizado em muitos ambientes fora dos navegadores. Foi criado por Brendan Eich, funcionário da Netscape Corporation, em 1995, com o objetivo de tornar as consultas web através do navegador mais dinâmicas do lado cliente, além de oferecer recursos do lado servidor. Em agosto de 1996, foi introduzido no Internet Explorer 3.0, sob o nome de Jscript (MOZILLA, [s. d.]).

Em 1996, para não perder a concorrência, a Netscape resolveu normatizar a linguagem e decidiu, então, procurar a associação europeia de fabricantes de computadores (ECMA – European Computer Manufacturers Association), fundada em 1961, para a padronização de sistemas de informação. Nesse período, descobriu-se que a fabricante americana Sun Microsystems já havia patenteado o nome JavaScript, o que, após a padronização, provocou o surgimento do nome ECMAScript (MOZILLA, [s. d.]).



#### PARA SABER MAIS

A diferença entre o JavaScript e o Java Applets é que o primeiro depende do seu browser para ser executado, enquanto o segundo cria aplicações compiladas que são executadas em uma máquina virtual instalada no client-side, o Java Runtime Environment. O Java foi desenvolvido pela Sun Microsystems, que hoje está sob responsabilidade da Oracle.

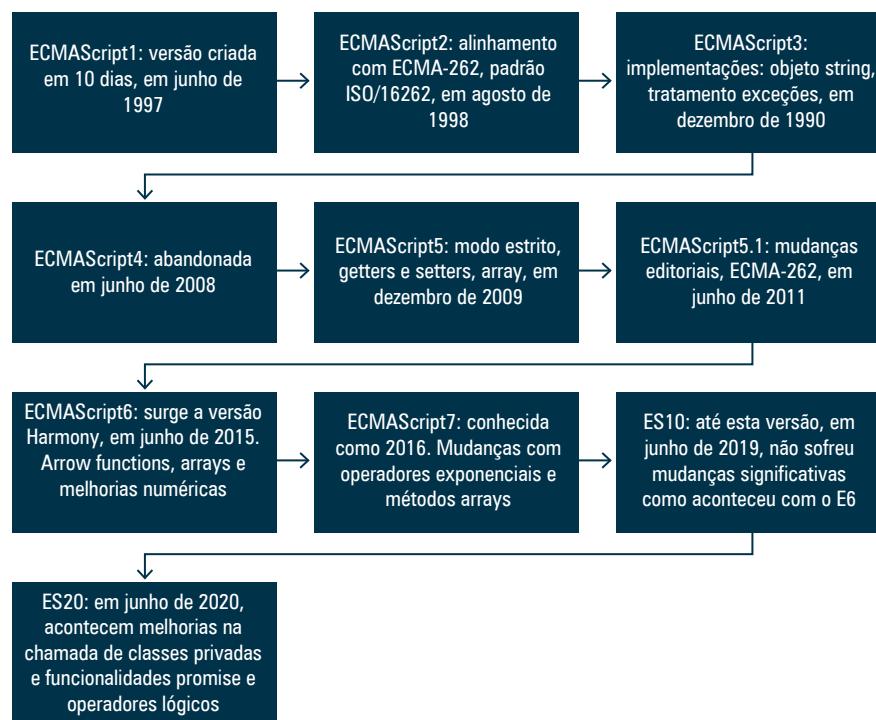
## 3 ECMAScript

Todos os navegadores web adotaram a padronização ECMAScript a partir de 2012. Os navegadores mais antigos ofereciam suporte para a versão ECMAScript 3. Em 2015, a ECMA International apresentou ao mercado a versão 6 do ECMAScript (ou ES6). A partir dessa data, anualmente, novas versões são lançadas. Toda a documentação oferece instruções da última versão, chamada de ECMAScript 2018, que tem a

vantagem de ser mais fácil e próxima das outras linguagens orientadas a objetos, como Java e Python. (MOZILLA, [s. d.]).

Na figura 2, podemos acompanhar a evolução do ECMAScript desde que foi criado, em 1997, até a versão de junho de 2020.

**Figura 2 – Evolução do ECMAScript**



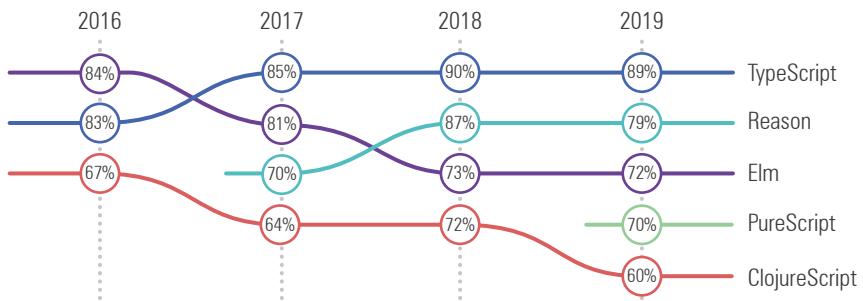
Fonte: adaptado da Mozilla [s. d.].

## 4 JS flavors e transpiladores

A linguagem JavaScript tem se tornado mais popular para o desenvolvimento de aplicações web. No entanto, mesmo que o padrão ECMAScript tenha tido sucesso, estão surgindo várias contribuições, como o TypeScript, ClojureScript e ELM. Essas versões passaram a ser

chamadas de flavors do JS. Mas nem todas têm conquistado a popularidade no mercado de trabalho. Acompanhe, no gráfico 1, a satisfação das linguagens de programação que compilam para o JavaScript.

**Gráfico 1 – Satisfação das linguagens de programação que compilam para o JavaScript**



Fonte: adaptado de State of JS (2019).

Essas contribuições precisam passar pelo processo de transpilação, em que as linguagens consideradas de alto nível são recebidas como entrada e convertidas para JavaScript antes de serem interpretadas. Em resumo, transpilação é o processo de conversão de um tipo de linguagem-fonte para outra linguagem-fonte.

Em outras palavras, a evolução do ECMAScript ofereceu várias mudanças até hoje. Todas foram bem recebidas pela comunidade de desenvolvedores, mas geraram um trabalho enorme para atualizar os aplicativos web para os novos recursos. O Babel veio para resolver essa conversão do JavaScript novo para o mais antigo, facilitando a vida dos desenvolvedores, uma vez que permite que o código seja escrito uma única vez e executado por navegadores抗igos.

O Vanilla é um tipo de técnica ou estilo que os desenvolvedores utilizam para a codificação em JavaScript. Possui utilitários próprios e pode ser usado em qualquer versão, sem qualquer biblioteca. Leve e de rápida execução, é utilizado pela Google, YouTube, Facebook e outras grandes

corporações. Talvez seu nome tenha relação com o sabor baunilha, por ser mais comum que os outros sabores.



### **PARA SABER MAIS**

Em desuso, temos o ActionScript da Adobe Flash Player, que iniciou no mercado como Flash 2, depois se tornou Flash 3 e, finalmente, Flash MX. O padrão do ActionScript seguiu o mesmo padrão do JavaScript. O CoffeeScript foi outra linguagem com o objetivo de ser leve, intuitiva e limpa, e se compila ao JavaScript.

## **5 Principais frameworks**

O JavaScript foi inicialmente usado apenas para o client-side. No entanto, atualmente, também é usado como linguagem de programação do server-side. Para resumir em apenas uma frase simples: o JavaScript é a linguagem da web. Desse modo, falaremos sobre os principais frameworks dessa poderosa linguagem de programação.

Os frameworks são mais adaptáveis para o design de sites e, portanto, detêm a preferência dos desenvolvedores. Frameworks JavaScript são um tipo de ferramenta que torna o trabalho mais fácil e suave, possibilitando também a reutilização do código. A capacidade de resposta é outra razão pela qual os frameworks JavaScript são bastante populares quando se trata de usar uma linguagem de máquina de alto nível. Vamos conhecer agora os principais frameworks JavaScript.

### **5.1 React**

O React.js é uma biblioteca JavaScript de front-end de código aberto que foi criada por uma equipe de desenvolvedores do Facebook liderada por Jordan Walke, em 2011, e se tornou open source em junho de 2013.

O protótipo inicial foi apelidado de FaxJs e foi experimentado pela primeira vez no feed de notícias do Facebook.

O React proporcionou uma verdadeira inovação na modelagem de aplicativos da web que vemos hoje, pois introduziu um estilo de programação baseado em componentes, funcional e declarativo para a criação de interfaces de usuário interativas, principalmente para aplicativos da web de página única. O React oferece uma renderização extremamente rápida, utilizando o virtual DOM, que renderiza apenas os componentes que foram alterados em vez de renderizar a página inteira. Outro recurso importante do React é o uso da sintaxe JSX, que é uma extensão de sintaxe para JavaScript; juntos, eles contribuem com a interface.

## 5.2 Vue

Embora desenvolvido no ano de 2016, esse framework JavaScript já fez seu caminho no mercado e provou seu valor ao oferecer vários recursos. A utilização do Vue facilita a integração entre as bibliotecas do JavaScript e seus projetos, sendo um dos recursos mais atraentes para a criação de SPA (Single Page Application ou aplicação de página única). É uma plataforma muito confiável para o desenvolvimento de plataforma cruzada.

O Vue é considerado uma forma moderna de se estruturar o JavaScript por fornecer ótimos recursos para seu desenvolvimento com HTML, além de um substituto para as bibliotecas JQuery. Permite reutilizar códigos criados dentro de blocos de script ou instâncias através de seus componentes. Por exemplo:

- O DOM virtual absorve todas as alterações destinadas ao DOM presentes na forma de estruturas de dados JavaScript, que são comparadas com a estrutura de dados original.

- O Data Binding, que representa a relação entre o código JavaScript e o HTML, deixou de trabalhar em um único sentido. Isso significa que valores atribuídos dentro do código JavaScript são enviados ao HTML e valores dentro do código HTML são enviados para o JavaScript.

## 5.3 Angular

O Angular é um framework de código aberto do Google usado para construir aplicativos da web de página única do client-side. Foi criado em 2010 com o nome AngularJS (ou Angular 1). O Angular 1 foi amplamente aclamado, mas o surgimento do React fez com que essa primeira versão fosse esquecida. Assim, o Angular 1 foi completamente reescrito e o Angular 2 (ou apenas Angular) foi lançado em seu lugar, em 2016.

Nessa nova versão, o Angular apresentou grandes mudanças: a maior delas foi a evolução da arquitetura MVW (Model-View-Whatever) para a arquitetura baseada em componentes, como o React. Hoje, o Angular é uma das estruturas de front-end JavaScript mais seguras para criar aplicativos de escala empresarial prontos para uso. Milhares de sites estão usando o Angular, incluindo Google, Forbes, IBM, Microsoft, entre outros.

## 5.4 Em desuso

Alguns frameworks em desuso são:

- JQuery: é ótimo para a manipulação do virtual DOM, sendo uma biblioteca JavaScript. Nos navegadores web mais modernos, seus recursos entraram em desuso.

- Ember.js: é uma estrutura JavaScript de código-fonte aberto usada para criar aplicativos da web de página única, escaláveis corporativamente. Ao contrário de outros frameworks que discutimos até agora, o Ember é baseado no padrão de arquitetura Model-View-Viewmodel (MVVM) que facilita a separação do desenvolvimento da interface.
- Polymer: é uma biblioteca JavaScript de código aberto desenvolvida pelo Google, que pode criar os elementos do site sem entrar em um nível complexo. Além disso, ele suporta vinculação de dados unilateral e bidirecional, tornando, assim, uma área de aplicação mais extensa.

## 5.5 Uso específico

Outros frameworks possuem usos específicos. São eles:

- React Native (também conhecido como RN): é um popular framework de desenvolvimento de aplicativo móvel baseado em JavaScript que permite construir aplicativos móveis renderizados nativamente para iOS e Android. O framework permite criar um aplicativo para várias plataformas usando a mesma base de código. O RN foi lançado pela primeira vez pelo Facebook como um projeto de código aberto. Em apenas alguns anos, ele se tornou uma das principais soluções para desenvolvimento móvel. O desenvolvimento do RN é usado para potencializar alguns dos principais aplicativos móveis do mundo, incluindo Instagram, Facebook e Skype.
- ThreeJs: é uma biblioteca composta de funções e APIs, usada para desenhos em 3D baseados em movimentação de cenários, que também pode ser utilizada para jogos. Essa biblioteca é um recurso WebGL (Web Graphics Library) em JavaScript para a

renderização com grande interatividade com gráficos em três e duas dimensões (3D e 2D).

- Electron: é um framework de fácil utilização e de rápida configuração, para o desenvolvimento de aplicações usando JavaScript. Utiliza algumas tecnologias como o Node.js.

## 6 Node.js

Com o nascimento e a rápida aceitação do Node.js, o JavaScript não se limita mais ao desenvolvimento de front-end. O desenvolvimento de back-end não é mais um “bicho de sete cabeças” para aqueles que buscam aprender a desenvolver o server-side.

O Node.js fornece recursos para criar seu próprio servidor web, que manipulará solicitações HTTP de forma assíncrona. É usado para criar diferentes tipos de aplicativos, como aplicativos de linha de comando, aplicativos da web e aplicativos de bate-papo em tempo real, entre outros.

Apesar da versatilidade e do vasto conjunto de aplicações que pode ser desenvolvido com o Node.js, ele é usado principalmente para criar programas de rede, como servidores da web, semelhantes ao PHP, Java ou ASP.NET.

O Node.js pode ser instalado no Windows, Mac, Linux e Solaris. Para o desenvolvimento, você precisa das seguintes ferramentas: Node.js, NPM (gerenciador de pacotes) e editor de texto (Notepad++, Visual Studio Code, Bloco de Notas, entre outros).

## 7 CDN e NPM

A combinação de JavaScript CDN (Content Delivery Network ou rede de entrega de conteúdo) para bibliotecas é usada quando o

desenvolvedor pretende puxar as bibliotecas diretamente de um site externo, usando o JavaScript ou o próprio HTML. Uma CDN funciona usando um grupo de servidores estrategicamente posicionados para ajudar a diminuir a distância entre os visitantes do seu site e os servidores que fornecem os dados solicitados.



### **PARA SABER MAIS**

Você pode começar a usar CDN aprendendo mais sobre alguns provedores, como Cloudflare, Amazon CloudFront e Akamai.

O registro público NPM (Node Package Manager, ou gerenciador de pacotes do node) é o gerenciador de dependência/pacote que você obtém ao instalar o Node.js, possibilitando que os desenvolvedores “puxem” as bibliotecas do NPM, gerando um *bundle* compactado, isto é, todos os módulos são compactados juntos.

Trata-se de uma ferramenta que ajuda a instalar esses pacotes e gerenciar suas versões e dependências. Existem milhares de bibliotecas e aplicativos Node.js no NPM e muitas mais são adicionadas diariamente. Ele utiliza pouco cache, garantindo uma entrega rápida e menos riscos com servidores de terceiros.

## **8 WebAssembly, Emscripten e ASM**

Por muito tempo, tivemos apenas o JavaScript como linguagem de programação disponível para uso nativo em um navegador da web. O desuso dos plug-ins binários de terceiros fez com que os desenvolvedores descartassem mais rapidamente as outras linguagens, como Java e ActionScript, do Flash. Outras linguagens da web, como CoffeeScript, são meramente compiladas em JavaScript.

Atualmente, temos uma nova possibilidade: WebAssembly (ou Wasm). WebAssembly é um formato binário pequeno e rápido que promete desempenho quase nativo para aplicativos da web. Além disso, é projetado para ser compilado em qualquer linguagem, sendo JavaScript apenas uma delas. Agora, com todos os principais navegadores suportando WebAssembly, é hora de começar a pensar seriamente em escrever aplicativos do client-side para a web que possam ser compilados como WebAssembly.

Note que WebAssembly não deve substituir os aplicativos JavaScript – pelo menos, ainda não. Em vez disso, pense no WebAssembly como um companheiro do JavaScript, uma vez que o JavaScript é flexível, com tipagem dinâmica e é entregue por meio de código-fonte legível, enquanto o WebAssembly é de alta velocidade, fortemente tipado e entregue por meio de um formato binário compacto.



### NA PRÁTICA

Os desenvolvedores devem considerar o WebAssembly para casos de uso de alto desempenho, como jogos, streaming de música e edição de vídeo.

---

O EMSScripten é um conjunto de ferramentas para compilar em asm.js e WebAssembly, construído usando LLVM (Low Level Virtual Machine), que permite executar C e C++ na web em velocidade quase nativa, sem plug-ins. Trata-se de um software impressionante que permitiu aos entusiastas portar muitos aplicativos C/C++ para o ambiente da web.

O asm.js não é uma linguagem, mas um conjunto restrito de definições que fornecem boas características de desempenho. Essas definições podem ser combinadas, como instruções em linguagem assembly, para criar aplicativos JavaScript muito rápidos. Ele não usa plug-ins

ou módulos para executar o código JavaScript, o que o faz ser compatível com versões anteriores.

Vale a pena mencionar que não é uma boa ideia escrever código asm.js manualmente. O resultado seria difícil de manter e demorado para depurar. Com base nessa observação, a questão que permanece é: como desenvolver aplicativos usando asm.js? É aí que o Emscripten entra. Podemos utilizar essa ferramenta para gerar código JavaScript de acordo com a especificação asm.js de outras linguagens, como C ou C++.

Desse modo, resumidamente, podemos dizer que o asm.js é uma especificação para criar um ambiente JavaScript de baixo nível, sendo um subconjunto mais rápido da linguagem JavaScript. Por outro lado, Emscripten é uma ferramenta para converter código escrito em C/C++ para bytecode LLVM e depois para código JavaScript asm.js.

## Considerações finais

Conhecemos, neste capítulo, a evolução da linguagem JavaScript, desde o surgimento da nomenclatura ECMAScript, em 1997. É importante observar o caminho percorrido para se tornar a linguagem mais utilizada no mundo pelos desenvolvedores web. A linguagem JavaScript passou por várias versões para atender a uma padronização a ser adotada pelos navegadores. Trata-se de uma linguagem usada tanto do lado do cliente quanto do servidor, apresentando funções mais fáceis e formas melhores de controle do client-side. Portanto, apresenta a característica de ser uma linguagem dinâmica, desenvolvida para suportar estilos de orientação a objetos.

# Referências

FLANAGAN, David. **JavaScript**: o guia definitivo. Porto Alegre: Bookman, 2012.

MOZILLA. MDN: **JavaScript**. [s. d.]. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 25 nov. 2020.

STATE OF JS. **JavaScript Flavors**: linguagens de programação que compilam para JavaScript. 2019. Disponível em: <https://2019.stateofjs.com/pt/javascript-flavors/>. Acesso em: 25 nov. 2020.

## Capítulo 2

# Sintaxe básica do JavaScript

Neste capítulo, iniciaremos nossos estudos da sintaxe básica da linguagem JavaScript e conhceremos o ambiente de execução do código e os tipos de constantes e variáveis que podemos implementar para a realização de operações aritméticas. Abordaremos também a manipulação de operadores lógicos através da utilização de laços e vetores.

## 1 Usando o developer tools e console do browser

Sabemos até aqui que a linguagem JavaScript possui vários recursos e é muito popular entre os desenvolvedores web. Trata-se de uma linguagem interpretada e, quando executada do lado cliente (client-side), não utiliza recursos do lado do servidor (server-side). Vamos

acompanhar duas formas de trabalhar com a linguagem JavaScript: utilizando um editor de texto e o console do browser.

## 1.1 Usando um editor de texto

É possível escrever um script usando qualquer editor de código-fonte, tais como bloco de notas, visual studio code ou o editor aberto Notepad++. Em nossos exemplos, utilizaremos o Notepad++ para criar arquivos com os scripts da linguagem, que necessitam de uma página HTML para serem executados dentro do código ou externamente a ele. Após escrito o código, para executá-lo, é necessário um interpretador de JavaScript (JS engine), como o NodeJS ou seu navegador web. Para isso, os arquivos devem ser salvos com a extensão <script>.js.

Neste primeiro passo, criaremos uma pasta em seu computador chamada “codigo\_curso”. Em seguida, abriremos o editor Notepad++ e digitaremos o código:

```
<!doctype html>
<html>
<head>
    <meta charset="UTF-8" />
    <title> Meu primeiro código do curso em javascript </title>
    <script type="text/javascript">

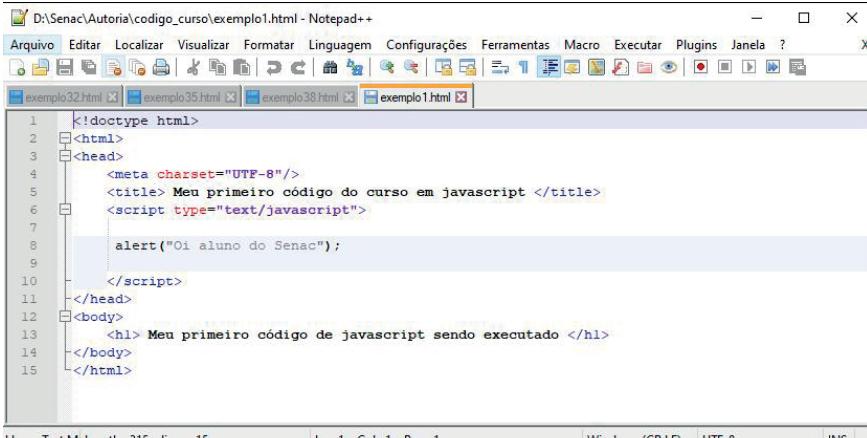
        alert("Olá aluno do Senac");

    </script>
</head>
<body>
    <h1> Meu primeiro código de javascript sendo executado
</h1>
</body>
</html>
```

O código em JavaScript dentro do HTML será acompanhado da tag `<script type = "text/javascript">` para representar o início. Para representar o fim, use a tag `</script>`. A instrução usada no exemplo é `alert("Oi aluno do Senac")`, para gerar uma caixa de alerta com a mensagem "Oi aluno do Senac".

Salve o código com o nome "exemplo1", conforme figura 1.

**Figura 1 – Salvar o primeiro programa**



The screenshot shows the Notepad++ interface with the file `exemplo1.html` open. The code contains a simple HTML structure with a script block that displays an alert message:

```
1 <!doctype html>
2 <html>
3 <head>
4   <meta charset="UTF-8"/>
5   <title> Meu primeiro código do curso em javascript </title>
6   <script type="text/javascript">
7     alert("Oi aluno do Senac");
8   </script>
9 </head>
10 <body>
11   <h1> Meu primeiro código de javascript sendo executado </h1>
12 </body>
13 </html>
```

The status bar at the bottom indicates the file has 315 characters and 15 lines, and the current position is Ln:1 Col:1 Pos:1. The encoding is set to Windows (CR LF) and the character set is UTF-8.

Para salvar no Notepad++, clique em Arquivo, Salvar Como ou `Ctrl+Alt+S`. Antes de salvar, mude o tipo do arquivo para *Hyper Text Markup Language file*, para salvar com a extensão `html`. Depois de salvo, para executar, encontre o arquivo "`exemplo1.html`" na pasta "`código_curso`" e clique duas vezes nele para seu navegador web padrão executar o código.

A segunda forma é separar o código JavaScript do arquivo em HTML. Salve o arquivo "`exemplo2.html`", com o código:

```
<!doctype html>
<html>
<head>
    <meta charset="UTF-8" />
        <title> Meu primeiro código do curso em javascript
    </title>
        <script type="text/javascript" src="codigo1.js">
            </script>
    </head>
    <body>
        <h1> Meu primeiro código de javascript sendo executado
    </h1>
    </body>
</html>
```

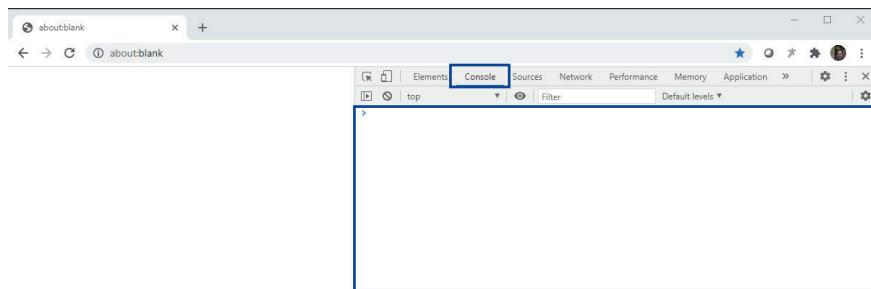
Note que a tag que marca o início do código JavaScript mudou para `<script type="text/javascript" src="codigo1.js">`, onde “src” representa o nome do arquivo separado que possui os procedimentos em JavaScript. Esse arquivo chamado “codigo1.js” está salvo com a extensão “js”, que é muito importante para o reconhecimento do código. O conteúdo do arquivo `codigo1.js` é apenas `“alert("Oi aluno do Senac")”`, isto é, uma janela será aberta com a mensagem.

## 1.2 Usando o console do browser

Para usar o console dos navegadores/browsers Microsoft Edge ou Chrome e executar instruções em JavaScript, primeiramente, abra o navegador e digite o endereço “about:blank” para exibir uma página em branco. Clicando no botão no lado superior direito representado pelo símbolo com três pontos, ao abrir a janela, selecione *Mais ferramentas/Ferramentas do Desenvolvedor*, pressione *Ctrl+Shift+i* ou *F12*.

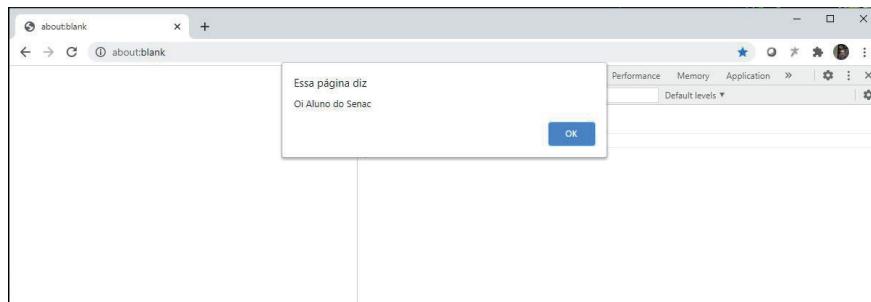
A figura 2 mostra o DevTools do navegador onde você pode depurar o código na aba *Source* ou executar um código JavaScript na aba *Console*, onde realizaremos nossos testes.

**Figura 2 – Console do navegador web**



Na aba *Console*, digite o código `alert("Oi aluno do Senac");`. Após digitar o ponto e vírgula, pressione *Enter* para executar. O resultado dessa ação é apresentado na figura 3.

**Figura 3 – Executando no console do navegador**



## 2 Tipagem dinâmica

A tipagem dinâmica representa uma sintaxe da linguagem em que o desenvolvedor não precisa inicializar uma variável com sua declaração de tipo antes de atribuir um valor. Isto é, o tipo de valor que será atribuído à variável determinará automaticamente o tipo de dados da variável.

Como exemplo utilizaremos as três linhas de código:

```
var idade = 34;  
idade = "Jovem";  
idade = false;
```

Na primeira linha, “var idade = 34”, a variável idade recebe o valor numérico 34; nesse momento, a variável é do tipo numérica Number. Na segunda linha é atribuído o texto “Jovem”. Logo, a partir da segunda linha, a variável idade passou a ser do tipo String. Na terceira linha foi atribuído o valor “false”, provocando novamente a mudança do tipo de variável para Boolean.

Portanto, você pode perceber que, em tempo de execução do seu código, o desenvolvedor pode alterar o tipo de variável conforme o valor atribuído.

## 3 Declaração de variáveis e constantes

A linguagem JavaScript, nesse ponto, não é diferente do que aprendemos em lógica de programação, pois a inicialização de variáveis é fundamental. Declarar variáveis antes de um procedimento assegura ao desenvolvedor o primeiro valor alocado e seu tipo. Na versão ES6, além do identificador var, foram criados o let e o const. Nessa versão, temos três formas de declarar uma variável: declaração var, declaração let e declaração const.

### 3.1 Declaração var

Esse tipo de declaração é a mais antiga. Vamos acompanhar, em nosso exemplo, a declaração de uma variável teste1 do tipo String.

No espaço reservado ao código script que inicia na linha 7, a variável de declaração var é iniciada, sendo teste1 do tipo String, com valor “Fora

do IF". Na instrução IF, caso a condição seja verdadeira, da linha 8 até a linha 10, onde está delimitado por chaves "{}", perceba que na linha 9 existe outra declaração var da mesma variável de nome teste1, alterando a string para "Dentro do IF".

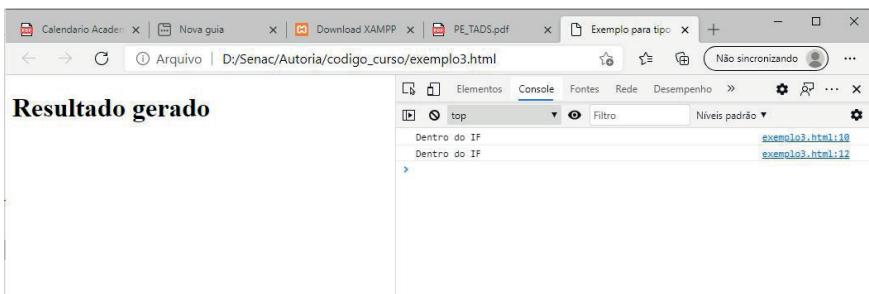
```
1  <!doctype html>
2  <html>
3  <head>
4  <meta charset="UTF-8"/>
5  <title> Exemplo para tipo var, Let e const </title>
6  <script type="text/javascript">
7  var teste1 = 'Fora do IF';
8  if (teste1 === 'Fora do IF') {
9      var teste1 = 'Dentro do IF';
10     console.log(teste1);
11 }
12 console.log(teste1);
13 </script>
14 </head>
15 <body>
16 <h1> Resultado gerado </h1>
17 </body>
18 </html>
```

Agora, observe as linhas 10 e 12: a linha 10 está dentro dos delimitadores da instrução IF, usada para mostrar o valor da variável teste1 através da instrução console.log(teste1); repare também que a linha 12 está fora da instrução IF.

A declaração da linha 7 é chamada de variável global, enquanto a declaração da linha 9 é chamada de variável local, porque foi declarada dentro da instrução IF.

Execute o programa em um navegador e, em seguida, abra a interface do console e teremos no navegador uma página HTML, com a janela console acionada, conforme figura 4.

**Figura 4 – Página HTML no navegador, com a janela console acionada**



Na figura 4, temos o resultado do código HTML que foi executado anteriormente; no lado direito, na aba *Console*, observamos o resultado da instrução `console.log`, usada para exibir o valor da variável `teste1`. Observe que, usando a declaração `var` na linha 9, variável local no IF, o valor exibido foi “Dentro do IF” pela linha 10. No entanto, quando a execução do código saiu do IF e chegou à linha 12, no mesmo valor da última declaração realizada dentro do IF, foi exibido novamente na linha 12, isto é, a declaração da variável local permaneceu fora da condição do IF, alterando o valor da variável global.

### 3.2 Declaração let

Utilizando a declaração `let` para inicializarmos uma variável, é possível perceber que se considera o bloco em que se instancia a variável, dentro da instrução `if()`, `switch()` ou `for()`. A declaração será compreendida como variável local no bloco, dentro do código do programa.

Para facilitar a compreensão, utilizaremos o mesmo código do exemplo da declaração `var`, porém, alteraremos, na linha 9, a declaração `var` para `let`.

```
1      <!doctype html>
2      <html>
```

```

3   <head>
4     <meta charset="UTF-8"/>
5     <title> Exemplo para tipo var, Let e const </title>
6     <script type="text/javascript">
7       var teste1 = 'Fora do IF';
8       if (teste1 === 'Fora do IF') {
9         let teste1 = 'Dentro do IF';
10        console.log(teste1);
11      }
12      console.log(teste1);
13    </script>
14  </head>
15  <body>
16    <h1> Resultado gerado </h1>
17  </body>
18 </html>

```

Execute novamente o programa e depois abra o console. A figura 5 representa o resultado no console do navegador: a primeira linha, com a mensagem “Dentro do IF” exibida pelo console.log(teste1) da linha 10, mostra que o valor de atribuição pelo let na linha 9 foi realizado. No entanto, perceba que a segunda linha é a mensagem “Fora do IF”, exibida pela instrução console.log(teste1) na linha 12, na qual permaneceu o valor atribuído pelo var da linha 7, localizado fora da instrução if().

**Figura 5 – Resultado no console**

## Resultado gerado

```

Dentro do IF
Fora do IF

```

Logo, podemos perceber que, se declararmos uma variável dentro de um bloco if() com a instrução let, a atribuição será considerada para a variável local. Enquanto isso, a declaração externa, considerada variável global, fica inalterada.

### 3.3 Declaração const

Essa declaração é utilizada quando o desenvolvedor precisa iniciar uma variável em seu código e não quer que o valor seja sobrescrito durante a execução do programa. Isto é, você declara uma variável como const para determinar que seu valor não pode ser alterado.

Vamos utilizar novamente o código do exemplo anterior. Desta vez, será alterado o código da linha 7, de var para const.

```
1  <!doctype html>
2  <html>
3  <head>
4  <meta charset="UTF-8" />
5  <title> Exemplo para tipo var, Let e const </title>
6  <script type="text/javascript">
7  const teste1 = 'Fora do IF';
8  if (teste1 === 'Fora do IF') {
9      Uncaught SyntaxError: Identifier 'teste1' has
10     already been declared
11     console.log(teste1);
12 }
13 console.log(teste1);
14 </script>
15 </head>
16 <body>
17 <h1> Resultado gerado </h1>
18 </body>
19 </html>
```

Salve, execute novamente o programa e depois abra o console. Observe que foi exibida uma mensagem de erro na linha 9, com o título “Uncaught SyntaxError: Identifier ‘teste1’ has already been declared”. Isto é, com a declaração const na linha 7, não foi permitida uma nova atribuição para a variável global na linha 9.

## 4 Números

A linguagem JavaScript não faz distinção entre valores inteiros e valores de ponto flutuante (FLANAGAN, 2012). Os números no código serão representados de acordo com as atribuições do desenvolvedor. Todos os números serão implementados como double, não existe especificação para inteiros. De acordo com Mozilla [s. d.], o tipo de número tem três valores simbólicos: +Infinity, -Infinity e NaN(not-number).

As atribuições numéricas podem ser:

```
var num = 20;  
num = 5.25;  
num = -12.34;  
num = 12e3; (Para tipo Exponencial, sendo 12000 ou 12 x  
103).
```

Você pode utilizar a linha de prompt no console para realizar seus testes. Observe na figura 4, do lado direito, o prompt do console representado pelo símbolo >. Realize a operação aritmética: > 2 + 2.12, tecle Enter ou observe que o cálculo será realizado.

### 4.1 Objeto Math

- **Objeto math:** apresenta propriedades e métodos para constantes matemáticas e funções (MOZILLA, [s. d.]), que podem complementar suas operações aritméticas, como:

$6 + \text{Math.sqrt}(25)$

Onde 6 será somado à raiz quadrada de 25.

- **NaN:** utilizado para determinar uma propriedade de um objeto global. O valor inicial de um NaN é um Not-a-Number. Quando é realizada uma operação matemática como Math.sqr(-1), o resultado

é um NaN; o mesmo ocorre quando transformamos um string em número. Para testar um valor desse tipo, usamos Number.isNaN() ou isNaN() para operações de comparação.

```
isNaN(NaN);
```

Onde o retorno será verdadeiro.

- **Operadores de comparação:** a linguagem JavaScript apresenta algumas particularidades para comparação de valores. Acompanhe:
- A === B representa uma comparação que somente é verdade se os operadores forem de tipos e conteúdos iguais, enquanto == converte os tipos antes de realizar a comparação. Já os operadores relacionais <=, >, <, >= e != são convertidos em tipos primitivos e depois para o mesmo tipo antes de ser comparados.



### PARA SABER MAIS

Para conhecer outras funções, consulte a obra de Flanagan (2012), em “Aritmética em Javascript”.

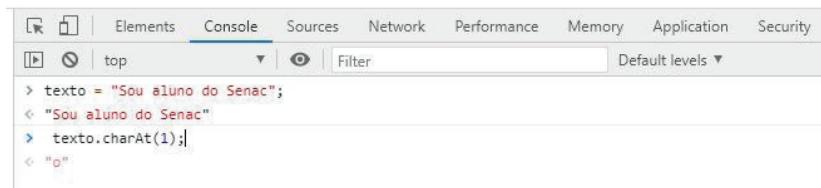
---

## 5 Strings

Strings são úteis para guardar dados que podem ser representados em forma de texto (MOZILLA, [s. d.]). Em JavaScript, o atributo string pode ser utilizado como um vetor, e os caracteres são identificados por um índice, iniciando na posição 0 do vetor.

No exemplo da figura 6, a variável texto é instanciada, do tipo String, e tendo atribuído o texto “Sou aluno do Senac”. No texto, a letra S representa a posição 0 (zero) no vetor e, na posição 17, está a letra c. Usando o console do navegador, temos:

Figura 6 – Visualizando a interação no console do navegador web



A screenshot of a browser's developer tools showing the 'Console' tab. The console window has a header with tabs: Elements, Console, Sources, Network, Performance, Memory, Application, and Security. Below the tabs is a toolbar with icons for back, forward, stop, and refresh, followed by a 'Filter' input field and a 'Default levels' dropdown. The main area of the console shows the following interaction:

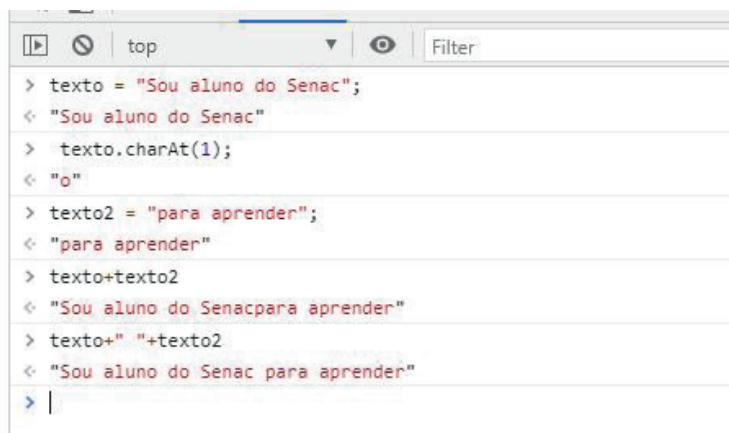
```
> texto = "Sou aluno do Senac";
< "Sou aluno do Senac"
> texto.charAt(1);
< "o"
```

Na primeira linha, estamos atribuindo uma string para a variável texto; em seguida, é exibido o resultado da atribuição. Na terceira linha, há um exemplo usando o método charAt para recuperar o caractere que é exibido na linha 4. Depois de realizada a atribuição na variável texto, sua criação como do tipo string possibilita a manipulação de vários métodos herdados, como o método charAt(), que fornece o caractere correspondente à posição numérica informada.

## 5.1 Concatenação

Esse processo representa a junção de um conjunto de strings ou caracteres. Para realizá-lo, utilizamos o operador (+), também utilizados para operações matemáticas. Como exemplo, daremos continuidade no código anterior:

Figura 7 – Soma de duas variáveis do tipo texto chamadas de texto e texto2, resultando em uma concatenação



A screenshot of a browser's developer tools showing the 'Console' tab. The console window has a header with tabs: Elements, Console, Sources, Network, Performance, Memory, Application, and Security. Below the tabs is a toolbar with icons for back, forward, stop, and refresh, followed by a 'Filter' input field and a 'Default levels' dropdown. The main area of the console shows the following interaction, continuing from Figure 6:

```
> texto = "Sou aluno do Senac";
< "Sou aluno do Senac"
> texto.charAt(1);
< "o"
> texto2 = "para aprender";
< "para aprender"
> texto+texto2
< "Sou aluno do Senacpara aprender"
> texto+" "+texto2
< "Sou aluno do Senac para aprender"
> |
```

Observe que estamos iniciando outra variável texto2 com o texto “para aprender”. Na linha 7, realizamos a operação de soma (+), resultando na concatenação de duas variáveis strings. Mas, na linha 9, foi realizada uma complementação do espaço entre aspas, para os textos não ficarem muitos próximos na concatenação.

## 5.2 Aspas simples e duplas

Na linguagem JavaScript, podemos utilizar tanto as aspas simples quanto as duplas, mas se você usar aspas duplas no início de um código para atribuir uma string, você deve terminar com o mesmo tipo de aspas.

Exemplo: var frase = “Estou estudando programação”;

Ou: var frase = ‘Estou estudando programação’;

Entretanto, se você utilizou as aspas duplas para atribuir string, você pode usar as aspas simples para o texto, como no exemplo abaixo:

Exemplo: var frase = “Meu nome é João”;

Ou: var frase = “Meu nome é ‘João’”;

## 5.3 Template strings

Na versão ES6 surgiu o template string ou template literals, que serve para construir cadeias ou conjuntos de caracteres sem utilizar o símbolo “+”. O conjunto de caracteres deve estar envolvido pelo acento grave do teclado(`), no lugar das aspas simples ou duplas. Nesse conjunto, podemos interpolar a string ou misturar os caracteres com o valor de variáveis, sem utilizar o símbolo “+”. Para isso, utilizamos dentro do template string o formato \${...}. Acompanhe o exemplo na figura 8.

**Figura 8 – Resultado mostrando uma string com a substituição de uma variável dentro do texto**

The screenshot shows a browser's developer tools console. At the top, there are buttons for play/pause, stop, and filter, followed by the word 'top'. Below the buttons, the console displays the following code and its execution:

```
> valor = 54;
< 54
> texto = `o resultado = ${valor}`;
< "o resultado = 54"
>
```

Nesse exemplo, realizado no console do navegador, na linha 1 é iniciada a variável de valor igual a 54, que gera o resultado na linha 2. Na linha 3, é atribuída à variável texto a frase `o resultado = \${valor}`. Observe que o acento grave está sendo usado no lugar das aspas. Na linha 4, é exibido o resultado da atribuição com a substituição da variável valor, formando "o resultado = 54".

## 6 Booleanos

Em JavaScript, há duas formas de se utilizar o tipo boolean. O primeiro e de mais fácil entendimento é o conhecido tipo primitivo, que representa dois estados: o verdadeiro (true) e o falso (false).

Exemplo:

```
var situa = false;
if (situa) {
    // este código dentro do if não é
executado
}
```

O segundo é o valor FALSY, que representa null, false, 0, -0, NaN, undefined ou um valor string vazio. O valor TRUTHY representa os valores

opostos de FALSY. Vista como um objeto Boolean, essa função é usada para converter um valor em um tipo boolean.

Exemplo:

```
1 let x = new Boolean('Tecnólogo');
2 console.log(x);
```

No exemplo, a função Boolean recebe um string para converter em booleano. Como no segundo caso, a string é considerada não vazio, e o valor de x vai ser *true*.

## 6.1 Operadores lógicos

Já aprendemos, neste capítulo, os operadores de comparação (==, >=, <=, >, < e !=). Os operadores lógicos nos ajudam a usar mais de um operador de comparação em um mesmo teste condicional. Utilizamos AND(&&), OR(||) ou NOT(!) nas expressões condicionais para comparar vários valores ao mesmo tempo nas operações algébricas matemáticas.

Exemplo:

```
var valor1 = true && true; (atribuído em valor1 o valor
true;)

Ou: if ((a>b) || (a>c)) {
    // este código será executado se uma das
    duas condições
    // forem atendidas
}
```

## 6.2 If-else e o operador ternário

Já acompanhamos vários exemplos de utilização básica do comando if, mas existem situações em que esse código pode gerar muitas linhas de código para se avaliar uma condição. O operador ternário representa,

a nível de código, um bit apenas em sua sintaxe, para quando o desenvolvedor precisar avaliar uma condição para gerar um valor ou expressão.

Sintaxe: <condição> ? <Caso verdadeiro> : <Caso falso>

Exemplo de código usando if-else:

```
3 var status;
4 if (situa) {
5     status = 1;
6 } else {
7     status = 2;
8 }
```

Podemos usar o operador ternário no exemplo anterior, resumido em uma linha:

```
var status = situa ? 1 : 2 ;
```

## 6.3 Operador de coalescência nula (null ou undefined)

Utilizamos “??” quando realizamos uma operação lógica em que, se o operador do lado esquerdo for null ou undefined, será atribuído como resultado o valor do lado direito.

Por exemplo, se a variável situa tiver valor null ou undefined, será atribuído valor 0 (zero) para a variável status:

```
var status = situa ?? 0;
```

# 7 Vetores

O vetor em JavaScript é um objeto usado para construir listas de valores ou arrays. Vamos acompanhar um exemplo que mostra como podemos inicializar uma lista:

```
var estado = ['SP', 'RJ', 'PA'];
var local = estado[0]; (a variável local recebe SP)
var tamanho = estado.length; (a variável tamanho recebe 3
- tamanho do array)
```

Também é possível utilizar o console do navegador, conforme figura 9.

**Figura 9 – Resultado de uma declaração de array como no exemplo**



The screenshot shows a browser's developer tools console window. At the top, there are buttons for back, forward, and refresh, followed by the text "top". To the right of these are dropdown menus for "Filter" and "D". Below the toolbar, the console output is displayed in a list format:

```
> estado = ['SP', 'RJ', 'PA'];
<  ▶ (3) ["SP", "RJ", "PA"]
> estado.push('MG');
<  4
> console.log(estado);
▶ (4) ["SP", "RJ", "PA", "MG"]
< undefined
> |
```

Na linha 1, a variável `estado` recebe o array 'SP', 'RJ', 'PA' entre colchetes `[]`. Na linha 3, usamos o método `push('MG')` para adicionar outro estado ao final da lista. Na linha 5, usamos o `console.log(estado)` para recuperar todo o vetor.

No vetor podemos utilizar, além de `length` e `push`, os seguintes métodos de acessos:

- `pop()` - remove o valor do último índice do vetor.
- `shift()` – remove o valor do primeiro índice do vetor.
- `unshift()` – adiciona novo valor no início do vetor.
- `indexof()` – recupera o índice no vetor da posição de determinado valor.

- splice(p,q) – remove o valor na posição p(indice do vetor), e q representa a quantidade de valores a partir da posição p.
- reverse() – inverte os valores do vetor.
- sort() – ordena os valores do vetor.

Para recuperar um vetor ou listar o array, o desenvolvedor pode utilizar os comandos for e while para manipular a lista de valores. Na figura 10, temos o resultado do uso do comando for.

**Figura 10 – Resultado usando for**

The screenshot shows a browser's developer tools console. At the top, there are icons for back, forward, and stop, followed by 'top' and a dropdown arrow. On the right, there is a 'Filter' input field. Below the toolbar, the console output is displayed in a list format:

- > `for(i=0;i<4;i++){console.log.estado[i]);}`
- SP
- RJ
- PA
- MG
- < undefined
- > |

Na linha 1, o comando for inicia a variável i em 0 (zero), que é o primeiro índice do vetor, e percorre a lista enquanto o valor de i for menor que 4 – ou seja, percorre os índices 0, 1, 2, 3.

**Figura 11 – Resultado usando while para o mesmo array**

The screenshot shows a browser's developer tools console window. At the top, there are buttons for back, forward, and refresh, followed by 'top' and a 'Filter' input field. Below the toolbar, the console output is displayed:

```
> i=0;
< 0
> while(i<4) { console.log.estado[i]); i++};
SP
RJ
PA
MG
< 3
> |
```

The code starts with `i=0;`, followed by the opening brace of a block. Inside the block, `console.log(estado[i]);` is called four times, resulting in the output: SP, RJ, PA, MG. After the fourth log statement, the value `3` is shown, indicating the current value of `i`. The final line shows the cursor at the end of the last log statement.

Utilizando o comando while, na linha 1, inicia-se a variável `i` em 0 (zero). Na linha 3, inicia-se o laço com `while(i<4)`, enquanto `i` for menor que 4. Dentro do laço, temos `console.log(estado[i])` para mostrar o valor e `i++` para incrementar o valor de `i`.

## 8 Objetos

Um objeto, conforme Flanagan (2012), é um valor composto, que agrupa diversos valores e permite armazená-los e recuperá-los pelo nome. Em Mozilla [s. d.], é uma coleção de dados e funcionalidades relacionadas, que são chamadas de propriedades e métodos.

A primeira forma de se criar um objeto em JavaScript é a notação literal, que é composta por chaves {} para constituir o que chamamos de propriedades, que possuem o formato “<nome>: <valor>”. Esse padrão é conhecido como JSON (JavaScript Object Notation). Por exemplo:

```
var veiculo={ ano : 2019,  
             tipo : "Sedan",  
             fabricante : "Honda",  
             status : function() {  
                 return this.fabricante + " "+this.  
ano;  
             }  
 }
```

Nesse exemplo, iniciamos o objeto veículo, que possui as propriedades ano, tipo e fabricante.

Quando iniciamos os objetos, precisamos saber como recuperar os valores das propriedades iniciadas. Existem duas formas: usando o ponto (.) ou os colchetes [] .

```
veiculo.tipo (recupera o valor Sedan)  
veiculo["tipo"]
```

Os métodos são funções que o desenvolvedor cria ao iniciar um objeto. O nosso objeto veículo possui um método de nome status; para recuperar o valor, escrevemos veiculo.status():

A sintaxe a seguir é utilizada para alterar o valor de uma propriedade criada para um objeto e é bem simples:

```
veiculo.fabricante = "Hyundai"
```



## PARA SABER MAIS

Consulte o capítulo 6 do livro *Javascript: o guia definitivo*, de Flanagan (2012), para conhecer outros exemplos de acessos a objetos.

# Considerações finais

Neste capítulo, apresentamos as formas de se executar um script utilizando um editor de texto para demonstrar a interação do código HTML com a chamada do código em JavaScript, de forma interna ou externa ao arquivo principal em HTML, criando o script em um arquivo separado. Também aprendemos a interagir com o código através do console do browser como uma forma de execução do código JavaScript. A declaração de variáveis torna o desenvolvimento do código prático e eficiente em modo de execução, tornando as operações aritméticas e lógicas rápidas para a manipulação de valores dentro de seu contexto de uso. Aqui, procuramos proporcionar uma visão básica para quem inicia seus estudos em JavaScript, finalizando com a manipulação de vetores e objetos.

## Referências

FLANAGAN, David. **JavaScript**: o guia definitivo. Porto Alegre: Bookman, 2012.

MOZILLA. MDN: **JavaScript**. [s. d.] Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 27 out. 2020.

## Capítulo 3

# JavaScript funcional

Neste capítulo, conhceremos os recursos para utilização de funções na linguagem JavaScript. Uma função representa uma tarefa que pode ser executada várias vezes dentro de um programa de computador, mas, para que isso aconteça, precisamos aprender como declarar e manipular essas funcionalidades internas da função. Aprenderemos também a manipulação de métodos existentes e como criar métodos internos na função. Aqui, o desenvolvedor verá como aprimorar seus conhecimentos na reutilização de códigos.

# 1 Funções, sintaxe básica e arrow-function

Uma função é um conjunto ou bloco de instruções em JavaScript que representa uma tarefa que será executada uma ou mais vezes durante a execução do projeto. Uma particularidade é que as funções em JavaScript podem ser atribuídas a outras funções; quando isso ocorre, a nova função tem acesso às variáveis definidas na outra função.

A declaração de uma função utiliza a sintaxe:

```
function <nome-da-função> (<lista de parâmetros>) {  
    ....  
    instruções....  
    return  
<valor-de-saída>  
}
```

Vamos a um exemplo. Acompanhe:

```
1   function quadrado(valor) {  
2       resultado = valor * valor;  
3       return     resultado;  
4   }
```

Você pode digitar também a função no console do navegador conforme figura 1.

Figura 1 – Função sintaxe básica

The screenshot shows a browser's developer tools console window. At the top, there are buttons for play/pause, stop, and refresh, followed by a 'Filter' input field and a 'Default levels' dropdown. The console output is as follows:

```
> function quadrado(valor) { let resultado = valor * valor; return resultado; }  
< undefined  
> quadrado(2);  
< 4  
> quadrado(4);  
< 16  
> |
```

Na versão ES6 do JavaScript, foi apresentada uma nova sintaxe para declaração de funções, mais simples e reduzida do que a anterior. Ela é conhecida como “arrow function”, devido à utilização dos símbolos de operação igual e maior que ( $=>$ ). Nesse caso, a sintaxe utilizada é:

```
<nome-da-função> = (<lista-de-parametros>) => <expressão>
```

Ou

```
<nome-da-função> = (<lista-de-parametros>) => {  
    ....instruções....  
    return <valor-de-saída>  
}
```

Vamos exemplificar:

```
quadrado_novo = (valor) => valor * valor;
```

Podemos aplicar a função também no console conforme figura 2, em que, na primeira linha, temos a sintaxe da arrow function e a função `quadrado_novo` declarada em apenas uma linha.

Figura 2 – Sintaxe básica de arrow function

The screenshot shows the 'Console' tab of a browser's developer tools. The console output is as follows:

```
> quadrado_novo = (valor) => valor * valor;
< (valor) => valor * valor
> quadrado_novo(4);
< 16
> quadrado_novo(5);
< 25
> |
```

## 2 Argumentos: valor e referência, rest, default values

O argumento representa uma variável interna da função ou variável local na função. Para recuperar os valores passados como parâmetros na função, você deve utilizar o objeto padrão chamado arguments. Assim como nos vetores, o primeiro argumento possui índice 0 (zero).

Em JavaScript, uma variável poderá armazenar um valor ou uma referência. Quando é considerado um valor, nós chamamos de argumento primitivo, que pode ser *undefined*, *null*, *boolean*, *number*, *string* ou *symbol*. Quando é atribuído a uma variável um valor do tipo *object*, ele é trabalhado na referência desse objeto, isto é, o valor é acessado por referência.

Exemplo para valores:

```
1 var a = 20;
2 var b = a;
3 b = 50;
4 console.log(a);    // (resultado será 20)
5 console.log(b);    // (resultado será 50)
```

Nesse exemplo, o valor de “a” não se alterou depois que “b” recebeu novo valor.

Exemplo para referência:

```
1 var registro = {nome: 'Romulo'};
2 var dados=registro;
3 dados.nome='Carolina';
4 console.log(registro.nome); // (o resultado será
Carolina)
```

Nesse exemplo, quando atribuído outro valor na linha 3 para dados.nome, por referência o valor do objeto registro também sofre a atualização.

Quando você precisar declarar uma função, mas não souber quantos argumentos serão informados na chamada da função, você deverá utilizar “...” antes do nome do parâmetro. Com isso, estará sendo utilizado o argumento rest na função e todos os argumentos passados serão agrupados no parâmetro, criando um vetor.

**Figura 3 – Rest na função**



The screenshot shows the 'Console' tab of a browser's developer tools. It displays the following JavaScript code and its execution results:

```
> function alunos(...nomes) { return console.log(nomes); }
< undefined
> alunos('João', 'Antônio', 'Carlos');
▶ (3) ["João", "Antônio", "Carlos"]
< undefined
> alunos('João', 'Antônio', 'Carlos', 'Maria');
▶ (4) ["João", "Antônio", "Carlos", "Maria"]
< undefined
> |
```

## 2.1 Default values

Por último, temos o argumento default values. Quando declaramos uma função e precisamos definir um valor padrão (um valor default) na inicialização da chamada da função, usamos a atribuição do valor inicial no momento da declaração.

Por exemplo, podemos executar a sintaxe a seguir, em que a função `calculo` possui os parâmetros `a=0`, `b=0` e `c=0`, retornando um resultado da soma de `a+b+c`, isto é, se um dos valores não existir, será assumido o valor zero na variável do parâmetro.

```
function calculo(a=0,b=0,c=0) { return a+b+c; }
```

Executando:

```
>calculo(2,3);
5
>calculo(2);
2
>calculo(2,4,5);
11
```

## 3 Hoisting

Criado especificamente na linguagem JavaScript, *hoisting* significa “içamento”, isto é, as declarações de variáveis e funções existentes no código do programa são elevadas à memória para o início do programa no momento da compilação, mas fisicamente permanecem no mesmo lugar dentro do código do programa.

```
1  <!doctype html>
2  <html>
3  <head>
4      <meta charset="UTF-8" />
5      <title> Exemplo de declaração de função </title>
6      <script type="text/javascript">
7          concatprof("Rômulo");
8          function concatprof(name) {
9              console.log(" O professor da disciplina é
" +name);
10         }
11     </script>
12 </head>
13 <body>
14     <h1> Exemplo de hoisting </h1>
15 </body>
16 </html>
```

Nesse código, a função *concatprof* está sendo solicitada dentro do código do programa antes de sua declaração na linha 7. Esse procedimento é chamado de hoisting, possível somente na linguagem JavaScript.



## IMPORTANTE

Somente as declarações de funções e variáveis serão elevadas. Não será elevada a inicialização do valor.

Exemplo certo:

1. `idade = 30;`
2. `console.log(idade);`
3. `var idade;`

Exemplo errado:

1. `console.log(idade);`
2. `var idade;`
3. `idade = 30;`

## 4 Contexto de variáveis locais

As declarações de variáveis e funções são processadas no início de todo código em linguagem JavaScript, ou seja, antes da execução do código. Caso a declaração de uma variável ocorra dentro do contexto de uma função, sua existência será local na função.

Não se engane com a atribuição de valor para uma variável. Caso você atribua um valor a uma variável fora do contexto de uma função, você estará iniciando essa variável como global.

```
1  <!doctype html>
2  <html>
3  <head>
4      <meta charset="UTF-8" />
5      <title> Exemplo de declaração de função com
6          variável local </title>
7      <script type="text/javascript">
8          function valor() {
9              a = 1;
```

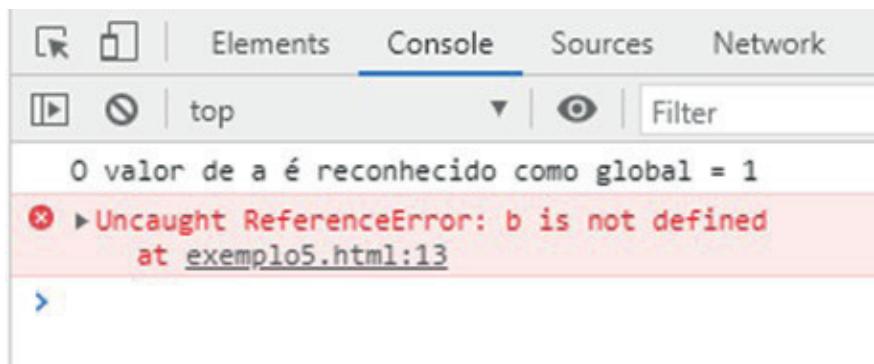
```

9         var b = 3;
10        }
11        valor();
12        console.log("O valor de a é reconhecido como
global = "+a);
13        console.log("Como b foi declarado dentro da
função como local = "+b);
14    </script>
15 </head>
16 <body>
17     <h1> Exemplo de declaração local </h1>
18 </body>
19 </html>

```

No código anterior, na linha 8, foi atribuído o valor 1 para a variável “a” dentro da função, o que significa que a variável “a” será declarada como global. Logo, na execução da linha 12, fora da função, o valor da variável existirá para o contexto geral do código. Enquanto isso, na linha 9, a declaração da variável “y” dentro do escopo da função, seguida de uma atribuição, será reconhecida somente dentro da função. Logo, como a chamada da variável “Y” na linha 13 está fora da função, será gerado um erro de variável não definida, conforme figura 4.

**Figura 4 – Erro de declaração local de variável na função, sendo usada fora do contexto da função**



## 5 Closures

Um closure representa uma função criada dentro de outra, como uma função filho dentro da função pai. No entanto, quando se trata de memória de contexto, é quando uma função interna tem acesso a todas as variáveis de uma função externa.

Na função closure, podemos utilizar o código:

```
1 var vezes = (n) => (x) => x * n;
2 var vezes2 = vezes(2);
3 console.log(vezes2(4)); // resultado 8
```

Por exemplo:

```
1 function pai() {
2     var nome_filho = "Pedro";
3     function exibir_filho(){
4         alert(nome_filho);
5     }
6     exibir_filho();
7 }
8 pai();
```

No código, estamos criando uma função *pai()*, na qual existem a declaração da variável *nome\_filho* e a definição da função interna *exibir\_filho()*. Chamamos essa função interna de closure. A função *exibir\_filho* está disponível para uso apenas dentro da função *pai*. A função *filho* não tem variáveis locais; ela acessa também a variável *nome\_filho* da função *pai*.

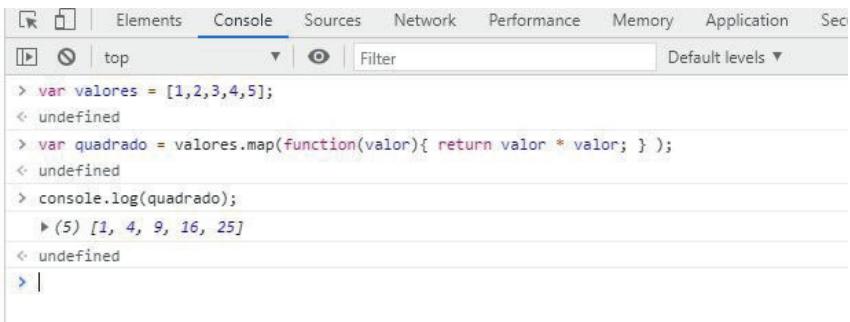
## 6 Métodos de vetores (map, reduce, filter, forEach)

Anteriormente, aprendemos como criar vetores, isto é, uma lista de valores conhecida pelos desenvolvedores como array. Agora,

conheceremos outros métodos que auxiliam na manipulação desses vetores: map, reduce, filter e forEach.

O método map é sempre utilizado a partir de um array, que usa uma função como parâmetro para receber os valores para um novo array, que é executado para cada elemento do array de origem. Na figura 5, temos um exemplo de map no console.

**Figura 5 – Exemplo método map**



The screenshot shows the 'Console' tab of a browser's developer tools. The code entered is:

```
> var valores = [1,2,3,4,5];
< undefined
> var quadrado = valores.map(function(valor){ return valor * valor; });
< undefined
> console.log(quadrado);
▶ (5) [1, 4, 9, 16, 25]
< undefined
> |
```

The output shows the array [1, 4, 9, 16, 25] printed to the console.

O método reduce é utilizado para reduzir os elementos de um array. Ele será executado com cada elemento do array original com o objetivo de gerar um único valor, não gerando outro array.

Neste caso, usamos a sintaxe:

```
array.reduce(callback(acumulador, valoratual[, index[, array]])  
[, valorinicial])  
Ou:  
array.reduce(function(acumulador, valoratual[, index[, array]])  
[, valorinicial]))
```

Já no console do navegador, temos:

**Figura 6 – Método reduce com arrow function**

The screenshot shows a browser's developer tools console tab labeled "Console". The code entered is:

```
> var numeros = [1,2,3,4,5,6];
< undefined
> var total=numeros.reduce((total,numero) => total + numero,0);
< undefined
> console.log(total);
21
< undefined
> |
```

```
Var total = números.reduce(function(total,numero) { return total + numero ; } , 0 );
```

O método filter é utilizado para gerar outro array conforme o elemento do array de origem atenda ao critério lógico especificado. Na figura 7, temos um exemplo de método filter aplicado no console do browser.

**Figura 7 – Método filter**

The screenshot shows a browser's developer tools console tab labeled "Console". The code entered is:

```
> var numeros = [1,2,3,4,5,6];
< undefined
> var numeroespecial = numeros.filter((valor)=> valor > 3);
< undefined
> console.log(numeroespecial);
▶ (3) [4, 5, 6]
< undefined
> |
```

Por fim, o método forEach executa uma função a partir de um array, atuando em cada elemento desse array. A diferença entre os métodos forEach e map é que o map gera o resultado em uma nova array, enquanto o forEach é executado na própria array. Na figura 8, temos um exemplo de utilização do método forEach.

**Figura 8 – Método forEach**



The screenshot shows the 'Console' tab of a browser's developer tools. The code entered is:

```
> var numeros = [1,2,3,4,5,6];
< undefined
> numeros.forEach((valor,indice,vetor) => vetor[indice] = valor * valor);
< undefined
> console.log(numeros);
▶ (6) [1, 4, 9, 16, 25, 36]
< undefined
> numeros
< ▶ (6) [1, 4, 9, 16, 25, 36]
> |
```

The output shows the original array and its modified version where each element is squared.

## 7 Generators e yield

O desenvolvedor declara uma função generator em um programa de computador quando pretende que sua execução possa ser interrompida em tempo de processamento, para que, em outro momento, possa continuar sua execução (MOZILLA, [s. d.]). Ao executar esse tipo de função, seu bloco de instruções não é executado imediatamente, a função retorna um objeto iterator. Depois, o desenvolvedor utiliza esse iterator para gerenciar todo o processo de execução. Por meio do método *next()* do iterator, o conteúdo inicia sua execução até a primeira ocorrência da expressão “yield”, que retornará o valor devolvido pelo iterator. Acompanhe um exemplo de função generator no código:

```
1  <!doctype html>
2  <html>
3  <head>
4      <meta charset="UTF-8" />
5      <title> Exemplo de Generator</title>
6      <script type="text/javascript">
7          function* disciplina() {
8              yield 'Matemática' ;
9              yield 'Programação' ;
10             yield 'Banco de Dados' ;
```

```

11      }
12      var matricula = disciplina();
13      console.log(" Sua primeira disciplina :
14          "+matricula.next().value);
15      console.log(" Sua Segunda disciplina :
16          "+matricula.next().value);
17      console.log(" Sua Terceira disciplina :
18          "+matricula.next().value);
19      </script>
20  </head>
21  <body>
22      <h1> Exemplo de declaração de função
23      Generator </h1>
24  </body>
25 </html>

```

Anteriormente, criamos um código completo em HTML com um bloco interno em JavaScript, iniciando na linha 6 e terminando na linha 16, e na linha 7 iniciamos a declaração da função generator. Observe que a sintaxe da função inicia `function*`, e cada parada de execução é representada pela expressão `yield`. Na linha 12, declararamos uma variável `matricula` que será atribuída à declaração da função generator. A linha 13 realiza a primeira chamada `matricula.next().value`, mostrando o primeiro valor na expressão `yield`, nas linhas 14 e 15. Observe que realizamos um `next()` para cada valor na função. A figura 9 apresenta o resultado da execução do código anterior.

**Figura 9 – Resultado da execução do código**

## Exemplo de declaração de função Generator



## 8 Funções assíncronas: `async` e `await`

Esse tipo de função possibilita que você escreva o seu código interno com base em uma promessa (`promise` é objeto que representa uma finalização ou falha na execução do código), conforme uma função síncrona, sem que ocorra uma parada ou bloqueio do código principal (FLANAGAN, 2012). Em outras palavras, se você executar duas funções declaradas no seu código, a segunda será executada somente depois que a primeira for finalizada, mas, se a primeira for uma função assíncrona (`async`), a segunda pode ser executada mesmo que a primeira ainda esteja em execução.

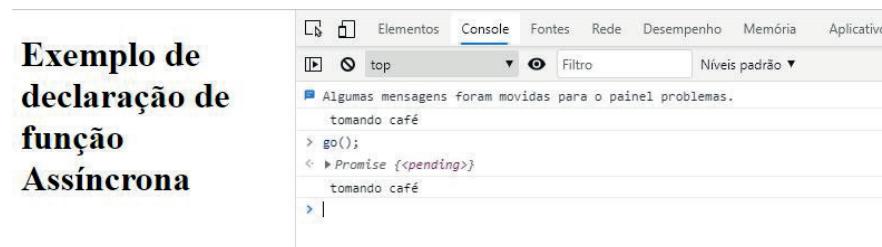
Podemos incluir um `await` dentro de uma função, que pode provocar uma pausa dentro da função até que uma `promise` seja finalizada.

```
1  <!doctype html>
2  <html>
3  <head>
4      <meta charset="UTF-8" />
5      <title> Exemplo de função assíncrona </title>
6      <script type="text/javascript">
7          function getcafe() { return new
Promise(resolva => {
8              setTimeout(() => resolva(' tomindo café
'), 2000));
9          });
10         async function go() {
11             const cafe = await getcafe();
12             console.log(cafe);
13         };
14         go();
15     
```

```
16     </script>
17 </head>
18 <body>
19         <h1> Exemplo de declaração de função
Assíncrona </h1>
20 </body>
21 </html>
```

No código, na linha 7, estamos criando uma função comum chamada `getcafe()` para gerar uma pausa por 2 segundos. Na linha 10, estamos declarando uma função assíncrona chamada `go()`, na qual, na linha 11, estamos utilizando o `await` dentro da função assíncrona, fazendo com que a execução da função seja pausada até o retorno da função `getcafe()`.

**Figura 10 – Exemplo de declaração de função assíncrona**



## **Considerações finais**

A utilização de funções contribui fortemente para a reutilização de blocos de códigos, facilitando a leitura e a compreensão do processo de execução. Aprender a sintaxe básica para a criação dessas funções oferece ao desenvolvedor uma escrita do código fácil e estruturada. Também é fundamental a manipulação de vetores com a utilização de métodos que facilitam a restruturação de seus elementos.

## Referências

FLANAGAN, David. **JavaScript**: o guia definitivo. Porto Alegre: Bookman, 2012.

MOZILLA. MDN: **JavaScript**. [s. d.] Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 22 fev. 2021.

## Capítulo 4

# Orientação a objetos em JavaScript

Neste capítulo, aplicaremos os conceitos de orientação a objetos utilizando a linguagem JavaScript e mostraremos a importância das funções construtoras para as classes no momento da instanciação dos objetos em tempo de execução. Isso contribui para que o profissional adquira uma boa prática de desenvolvimento orientado a objetos, no qual a classe representa o comportamento desses objetos com seus atributos e métodos, que podem ser herdados por outras classes dentro do código.

# 1 Função construtora (keywords new e this)

Uma das principais diferenças da linguagem JavaScript para as outras linguagens de programação orientada a objetos é que no JavaScript há a criação de funções especiais. Portanto, ela apresenta seu próprio estilo para a construção de objetos baseada em herança.

A função construtora é um tipo de função que também pode ser utilizada para instanciar objetos dentro de um programa. Para criar um objeto em JavaScript, primeiramente, é preciso definir a função construtora. Por exemplo:

```
1 function Discente(nome, idade) {  
2     this.nome = nome;  
3     this.idade = idade;  
4 }
```

No exemplo, a palavra “this” dentro da função está criando uma propriedade ao escopo global do Windows e representa que, no momento em que for instaciado um objeto em função da construtora, onde utilizamos o “this” no início, será criada uma propriedade ao objeto instanciado. Depois de criada a função construtora, podemos instanciar nosso objeto com o operador “new”, responsável pela criação do objeto, conforme código:

```
1 var aluno = new Discente("João", 23);  
2 console.log(aluno.nome);
```

# 2 Prototype

Sempre que desejarmos criar métodos para os objetos criados, que na verdade representam as funções internas da função principal, utilizadas para instanciar objetos, devemos usar o ponto (.) após a propriedade prototype.

Vamos acompanhar um exemplo de método utilizando a função construtora *discente* do exemplo anterior: suponhamos que queremos incluir a função *Matriculado()* como um método na função *Discente()*:

```
1 function Matriculado() {  
2     This.matricula = true;  
3 }  
4 Discente.prototype.matricula = Matriculado;
```

Caso o objeto já exista, como nosso exemplo “aluno”, é possível incluir o novo método como:

```
aluno.matricula = matriculado;
```

Um grande diferencial do JavaScript é que podemos adicionar novas propriedades ao objeto em tempo de execução. Por exemplo:

```
aluno.turma = "Desenvolvimento";  
console.log(aluno.turma);
```

## 3 Classes

Como visto anteriormente, as classes em JavaScript são funções especiais que, assim como as funções, podem ser declaradas de duas formas: *class declarations* e *class expression*. Vamos acompanhar como elas funcionam:

- **Class declarations:** é a primeira forma de se declarar uma classe dentro do código. No exemplo a seguir, apenas um método construtor deve existir por classe. Esse construtor de classe é um método especial que inicializa o objeto. Acompanhe:

```
class Area {  
constructor(base,altura) {  
    this.base = base;  
    this.altura = altura;  
}  
}
```

A principal diferença entre a declaração de classes e a declaração de funções é que as funções são hoisted, conforme já apresentado. As declarações de classes devem ser definidas antes de serem acessadas.

- **Class expression:** é a segunda forma de declarar uma classe, que pode ou não apresentar nome. Esse tipo de classe não é hoisted. Veja a seguir dois exemplos de sintaxes.

- Exemplo sem nome da classe:

```
let quadrado = class {  
constructor (base,altura) {  
    this.base = base;  
    this.altura = altura;  
}  
};
```

- Exemplo com nome da classe:

```
let quadrado = class Area{  
constructor (base,altura) {  
    this.base = base;  
    this.altura = altura;  
}  
};
```

## 3.1 Métodos

Dentro das funções e classes, podemos criar propriedades e funções internas chamadas de métodos dos objetos. Um exemplo é a criação do método `calculaarea` na classe `área`:

```
1  class Area {  
2      constructor(base,altura) {  
3          this.base = base;  
4          this.altura = altura  
5      }  
6      calculaarea(){  
7          return this.base * this.altura;  
8      }  
9  }  
10 const quadrado = new Area(5,4);
```

## 3.2 Getters e setters

As classes possuem propriedades e métodos para instanciar os objetos dentro de um código JavaScript, mas, para que os valores desses objetos sejam substituídos, utilizamos os métodos das classes chamados de getter e setter, conhecidos como métodos de acessos. Usamos o `get` para recuperar uma informação de uma classe e o `set` para a entrada de argumentos para o objeto instanciado. Vamos acompanhar um exemplo de utilização dos dois métodos:

```
1  <!doctype html>  
2  <html>  
3  <head>  
4      <meta charset="UTF-8" />  
5      <title> Exemplo de getter e setter </title>  
6      <script type="text/javascript">  
7          class area {  
8              constructor(base,altura) {  
9                  this.base = base;
```

```

10         this.altura = altura
11     }
12     get area() {
13         return this.calculaarea()
14     }
15     set entrabase(b) {
16         this.base = b;
17     }
18     set entraaltura(a) {
19         this.altura = a;
20     }
21     calculaarea() {
22         return this.base * this.altura;
23     }
24     static mensagem() {
25         return 'A mensagem do método estatico
foi chamada';
26     }
27 }
28 console.log(area.mensagem());
29 var quadrado = new area(5,6);
30 console.log(quadrado.area);
31 quadrado.entrabase = 7;
32 quadrado.entraaltura = 8;
33 console.log(quadrado.area);

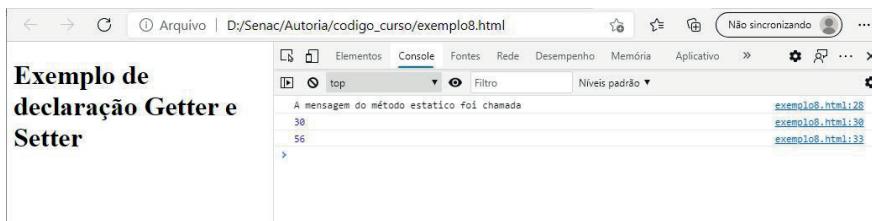
```

No código, na linha 12, declaramos o método `get area()` para recuperar ou extrair um retorno da multiplicação de dois parâmetros do objeto como base e altura. Nas linhas 15 e 18, declaramos dois métodos `set entrabase(b)` e `set entraaltura(a)` para enviar para dentro do objeto novos valores para os parâmetros dos objetos.

Na linha 26, instanciamos um objeto chamado `quadrado` em função da classe área, atribuindo os argumentos 5 e 6, que estão declarados como construtores. Logo, na linha 27, será mostrado no console o resultado 30.

Note que, na linha 28, em tempo de execução, são informados para o método `set entrabase` e `entraaltura` novos valores para gerar novo resultado no console na linha 30.

**Figura 1 – Resultado usando getter e setter**



### 3.3 Static

Os métodos e propriedades de uma classe podem ser definidos como estáticos quando precisamos executá-los sem a necessidade de instanciar um objeto para ser utilizado em tempo de execução do código. Vamos supor que criamos o método estático abaixo na linha 24 do código anterior.

```
24  static mensagem() {  
25      return 'A mensagem do método estático foi chamada';  
26  }
```

Podemos utilizar esse método antes da linha 26 da figura 1, onde instanciamos a classe. Por exemplo:

```
console.log(área.mensagem());
```

### 3.4 Herança

A linguagem JavaScript também utiliza o recurso de herança, muito comum nas linguagens orientadas a objetos. Com esse recurso, é possível criar uma nova classe que herde parâmetros e métodos de outra classe criada anteriormente. Por exemplo:

```
class Retangulo extends Area {  
    diagonal() {  
        return ' Retorna o valor da diagonal';  
    }  
}
```

## 4 JSON

Quando precisamos estruturar os dados manipulados pelas aplicações em um formato de texto, utilizamos o padrão JSON (JavaScript Object Notation, ou, em português, notação de objeto JavaScript), que é muito usado para a transferência de dados entre aplicativos web para a migração de informações entre bancos de dados diferentes. Sua sintaxe é uma serialização de objetos, matrizes, números, strings, booleanos e nulos. Por exemplo:

```
{  
    "nome" : "Antônio Silva",  
    "idade" : 18,  
    "disciplinas" : [ "Matemática", "Portugues", "Programação" ]  
}
```

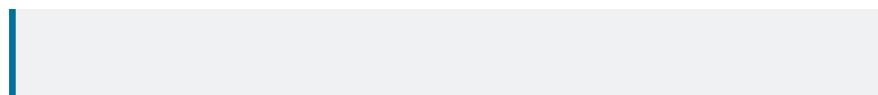
Nesse exemplo, temos um objeto limitado pelas chaves "{}". Entre aspas está o rótulo da informação seguido do valor, que pode ser número, string ou vetor.

Quando realizamos a conversão ou transformação de um objeto em um formato JSON, chamamos de serialização. Depois de enviadas as informações para outro aplicativo ou servidor, precisamos realizar o processo inverso, chamado de desserialização. Para isso, aprendemos dois métodos: *stringify()* e *parse()*.

## 4.1 Stringify()

É usado para converter a informação do formato JavaScript para o formato String JSON, isto é, converte os valores manipulados na linguagem acessível nos bancos de dados para o formato de texto no padrão JSON.

Sintaxe:



A função *replacer* pode ser utilizada para substituir alguns valores. É uma função opcional que altera a forma de conversão em uma string ou array. O espaço também é opcional, usado para inserir espaços em branco. Na figura 2, apresentamos um exemplo de conversão para JSON.

Figura 2 – Conversão para JSON

A screenshot of a browser's developer tools interface, specifically the 'Console' tab. The tab bar includes 'Elements', 'Console' (selected), 'Sources', 'Network', 'Performance', and 'Memory'. Below the tab bar, there is a toolbar with icons for play/pause, stop, and filter. The main area shows a command-line interface with the following code:

```
> aluno = { nome : "Antônio Silva", idade : 18 };
<  ▶ {nome: "Antônio Silva", idade: 18}
> texto = JSON.stringify(aluno, replacer);
<  ▶ {"nome": "Antônio Silva", "idade": 18}
> console.log(texto);
<  ▶ {"nome": "Antônio Silva", "idade": 18}
<  undefined
> |
```

The output of the JSON.stringify() function is shown in blue, indicating it is a string object.

## 4.2 Parse()

Usado para fazer a volta do formato JSON para o formato da linguagem JavaScript. A função *reviver* é opcional e serve para transformar uma informação.

Sintaxe:

Na figura 3, temos a transformação de um padrão JSON em JavaScript.

**Figura 3 – Transformando o padrão JSON em JavaScript**

The screenshot shows the browser's developer tools with the 'Console' tab selected. The console output demonstrates the conversion of a JSON object into a string and then back into an object. The code and its results are as follows:

```
> aluno = { nome : "Antônio Silva", idade : 18 };
< ◆ {nome: "Antônio Silva", idade: 18}
> texto = JSON.stringify(aluno);
< ◆ {"nome":"Antônio Silva","idade":18}
> console.log(texto);
  {"nome":"Antônio Silva","idade":18}
< ◆ undefined
> var novoobjeto = JSON.parse(texto);
< ◆ undefined
> console.log(novoobjeto);
  ◆ {name: "Antônio Silva", idade: 18}
< ◆ undefined
> console.log(novoobjeto.nome);
  Antônio Silva
< ◆ undefined
> console.log(novoobjeto.idade);
  18
< ◆ undefined
> |
```

## 5 Spread operator

O spread operator é conhecido como sintaxe de propagação e possibilita que um objeto, como um array ou string, seja expandido de zero ou mais argumentos para que sejam atribuídos.

Sintaxe:

```
função(...objetoIteravel);
[...objetoIteravel, 4, 5, 6]
```

Exemplo:

```
function funcaocalculo(a, b, c) { return a+b+c; }
var args = [3,6,7];
console.log(funcaocalculo(...args));
```

Quando tivermos um array e quisermos criar outro com o primeiro array dentro do segundo, a função spread facilitará esse processo. Por exemplo:

```
var equipe1 = ["João", "Antônio", "Carlos"];
var equipe2 = ["Roberto", "Heitor", "Luiza"];
var turma = ["Ana", ...equipe1, "Igor", ...equipe2, "Camila"];
console.log(turma);
```

O procedimento para incluir um array dentro de outro também pode ser realizado com objetos literais, para copiar as propriedades de um objeto para um novo objeto. Na figura 4, temos um exemplo de spread.

**Figura 4 – Exemplo de spread**

The screenshot shows a browser's developer tools console. The tabs at the top are Elements, Console, Sources, Network, Performance, Memory, Application, Security, and Lighthouse. The Console tab is selected. The console output is as follows:

```
> aluno = { nome : "Antônio Silva", idade : 18 };
< ◆ {nome: "Antônio Silva", idade: 18}
> endereco = { logradouro : "Av.Santo Amaro", numero : "777", Bairro : "Sto amaro" };
< ◆ {logradouro: "Av.Santo Amaro", numero: "777", Bairro: "Sto amaro"}
> var clonaendereco = {...endereco};
< ◆ undefined
> console.log(clonaendereco);
< ◆ {logradouro: "Av.Santo Amaro", numero: "777", Bairro: "Sto amaro"}
< ◆ undefined
> var discente = {...aluno,...endereco};
< ◆ undefined
> console.log(discente);
< ◆ {nome: "Antônio Silva", idade: 18, Logradouro: "Av.Santo Amaro", numero: "777", Bairro: "Sto amaro"}
< ◆ undefined
> console.log(discente.nome);
< ◆ Antônio Silva
< ◆ undefined
>
```

## 6 For in e for of

A criação de laços de repetição contribui muito para a simplificação e eficiência de um código. O comando *for* veio para controlar o número de repetições desse bloco de comandos dentro de um programa de computador. Na maioria das linguagens, são necessários a inicialização de uma variável, um controlador lógico e um incremento da variável.

Em nosso exemplo, vamos considerar o vetor *alunos* com três elementos:

```
1 var alunos = ["João", "Carlos", "ana"]
2 for(var i = 0; i < 3;i++) {
3     console.log(alunos[i]);
4 }
```

Na sintaxe *for of*, não será necessário utilizar o índice *i* para percorrer todos os elementos do vetor *alunos* nem controlar a finalização do índice. Na figura 5, temos um exemplo de laços de repetição.

**Figura 5 – Exemplos de laços de repetição**

The screenshot shows the Chrome DevTools Console tab. It displays two examples of iteration over an array named 'alunos'.

```
> alunos = [ "João", "Carlos", "ana" ];
<  ▷ (3) [ "João", "Carlos", "ana" ]
> for(var ind in alunos) { console.log(ind); }
0
1
2
< undefined
> for(var ind of alunos) { console.log(ind); }
João
Carlos
ana
< undefined
> for(var i = 0;i<3;i++) { console.log(alunos[i]); }
João
Carlos
ana
< undefined
> |
```

Na linha 3, utilizamos a sintaxe *for in* como o mesmo comando para imprimir, e o resultado são os índices dos elementos do vetor *alunos*. Se quisermos mostrar os elementos, precisamos informar o nome do vetor, como no exemplo:

```
for(var in alunos) {  
    console.log(alunos[ind]);  
}
```

Nesse exemplo, o resultado é igual ao exemplo usando *for of*.

## 7 Map e set

Já aprendemos muitas formas de representar um conjunto de valores, principalmente usando vetores. O *map* representa um objeto constituído de um conjunto de informações formadas por um par de dados, isto é, cada elemento será constituído de duas informações: uma chave e um valor. Temos outra vantagem: a possibilidade de manipular os elementos em tempo de execução de sua aplicação. Um objeto *map* pode ser instanciado conforme exemplo:

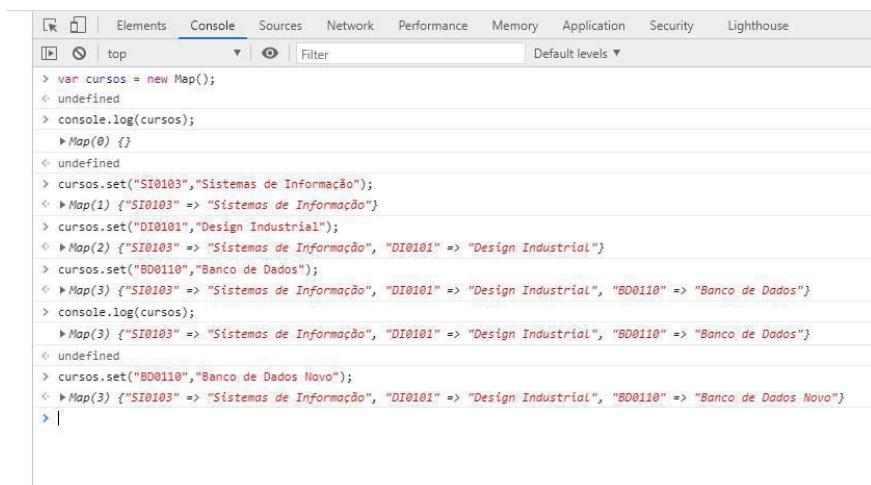
```
var cursos = new Map();
```

Esse exemplo retorna um *map* chamado “cursos” vazio. O método *set()* será utilizado para adicionar ou atualizar nosso *map* criado, conforme exemplo:

```
cursos.set("SI0103":"Sistemas de Informação");  
cursos.set("DI0101":"Design Industrial");  
cursos.set("BD0110":"Banco de Dados");
```

No exemplo anterior, a sintaxe é <Map>.set(<chave>,<valor>) e podemos acompanhar na figura 6.

**Figura 6 – Map e set**



The screenshot shows the browser's developer tools with the 'Console' tab selected. The code in the console is as follows:

```
> var cursos = new Map();
< undefined
> console.log(cursos);
Map(0) {}
< undefined
> cursos.set("SI0103","Sistemas de Informação");
< Map(1) {"SI0103" => "Sistemas de Informação"}
> cursos.set("DI0101","Design Industrial");
< Map(2) {"SI0103" => "Sistemas de Informação", "DI0101" => "Design Industrial"}
> cursos.set("BD0110","Banco de Dados");
< Map(3) {"SI0103" => "Sistemas de Informação", "DI0101" => "Design Industrial", "BD0110" => "Banco de Dados"}
> console.log(cursos);
Map(3) {"SI0103" => "Sistemas de Informação", "DI0101" => "Design Industrial", "BD0110" => "Banco de Dados"}
< undefined
> cursos.set("BD0110","Banco de Dados Novo");
< Map(3) {"SI0103" => "Sistemas de Informação", "DI0101" => "Design Industrial", "BD0110" => "Banco de Dados Novo"}
> |
```

Conforme Mozilla [s. d.], “o objeto Set permite que você armazene valores únicos de qualquer tipo”, ou seja, é uma nova forma de manipular vetores em um programa, sendo sua principal característica não aceitar a duplicação de valores atribuídos ao vetor.

**Figura 7 – Manipulando objeto set**

The screenshot shows the 'Console' tab of a browser's developer tools. The user has run several commands to demonstrate the behavior of the Set object:

- `> var conjunto = new Set([1,3,6,9,3,7,1]);`
- `< undefined`
- `> console.log(conjunto);`
- `▶ Set(5) {1, 3, 6, 9, 7}`
- `< undefined`
- `> conjunto.add(11);`
- `< ▶ Set(6) {1, 3, 6, 9, 7, ...} [1]`
- `▶ [[Entries]]`
- `▶ 0: 1`
- `▶ 1: 3`
- `▶ 2: 6`
- `▶ 3: 9`
- `▶ 4: 7`
- `▶ 5: 11`
- `size: (...)`
- `▶ __proto__: Set`
- `> conjunto.add(9);`
- `< ▶ Set(6) {1, 3, 6, 9, 7, ...}`
- `> conjunto.has(6);`
- `< true`
- `> conjunto.size;`
- `< 6`
- `> conjunto.add("Cursos");`
- `< ▶ Set(7) {1, 3, 6, 9, 7, ...} [1]`
- `▶ [[Entries]]`
- `▶ 0: 1`
- `▶ 1: 3`
- `▶ 2: 6`
- `▶ 3: 9`
- `▶ 4: 7`
- `▶ 5: 11`
- `▶ 6: "Cursos"`
- `size: (...)`
- `▶ __proto__: Set`

## 8 TypeScript

Representa um acréscimo nas funcionalidades do JavaScript, ou seja, é um conjunto de instruções que aperfeiçoa as instruções em JavaScript (FLANAGAN, 2012). Alguns chamam o TypeScript de *super-set*, um superconjunto de instruções. Imagine um código de uma função em JavaScript como o exemplo:

```
function total(a,b) { return a+b; }
console.log(total(3,6));
console.log(total("2", "4"));
```

Em JavaScript, as duas operações seriam realizadas, não importa se o valor é numérico ou string. No entanto, veja o exemplo a seguir, com o código escrito no padrão TypeScript:

```
function total(a:number, b:number) { return a+b; }
console.log(total(3,6));
console.log(total("2", "4"));
```

Perceba que, escrito em TypeScript, o parâmetro deverá receber somente *number*, para que a operação não seja realizada com strings.



### PARA SABER MAIS

Criado pela Microsoft para usar as instruções em TypeScript, o pacote necessita ser instalado no computador. O código TypeScript é compilado para JavaScript, que pode ser utilizado em outro código JavaScript.

## Considerações finais

Aprendemos neste capítulo como instanciar objetos em função de uma determinada classe, que possui seus atributos e métodos, e que estes, em alguns momentos, precisam ser manipulados dentro de um código JavaScript. Esses atributos podem ser de acesso público ou não e contribuem com a utilização dos métodos getters e setters para a entrada e a saída de valores ao código da classe.

Vimos também a utilização de laços que facilitam a manipulação de vetores. Essas instruções são muito importantes na programação orientada a objetos.

## Referências

FLANAGAN, David. **JavaScript**: o guia definitivo. Porto Alegre: Bookman, 2012.

MOZILLA. MDN: **JavaScript**. [s. d.] Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 22 fev. 2021.

## Capítulo 5

# Document Object Model (DOM)

Neste capítulo, você compreenderá o processo de conversão de uma página HTML em uma plataforma conhecida como DOM (Document Object Model ou modelo de documento por objetos, em português) e conhecerá os recursos que essa nova interface oferece para manipular dinamicamente um site em HTML que interaja com a linguagem JavaScript. Uma das principais informações geradas é a nova estrutura em árvore do documento, que facilita a manipulação da informação.

# 1 Ligando o JavaScript ao HTML

Anteriormente, apresentamos alguns recursos para trabalhar com a linguagem JavaScript no HTML. Agora, explicaremos as formas de integrá-lo ao HTML.

A primeira forma de integrar o JS ao HTML é utilizando as tags `<script>` `</script>` para delimitar o início e o final do conjunto de instruções em JavaScript (FLANAGAN, 2012). As tags podem ser inseridas tanto no `<head>` do HTML quanto no `<body>`, sendo que, neste, a execução do código terá melhor tempo de resposta no carregamento. Exemplo:

```
1  <html>
2    <head>
3      <title> Primeiro local para o código JavaScript
4      </title>
5      <script>
6      </script>
7    </head>
8    <body>
9      <h1> Segundo Local para o código JavaScript
10     <script>
11     </script>
12   </body>
13 </html>
```

A segunda forma é carregar o código desenvolvido em JavaScript de um arquivo separado salvo com a extensão .js, que deverá ser chamado pelo atributo `src="arquivo.js"`, como no próximo exemplo. Essa prática facilita a reutilização do código por outras páginas em HTML. Acompanhe:

```
1  <html>
2    <head>
3    </head>
4    <body>
5      <h1> Abaixo Local para o código JavaScript      </h1>
```

```
6      <script type ="text/javascript"
7          src="programa.js">
8      </script>
9  </body>
</html>
```

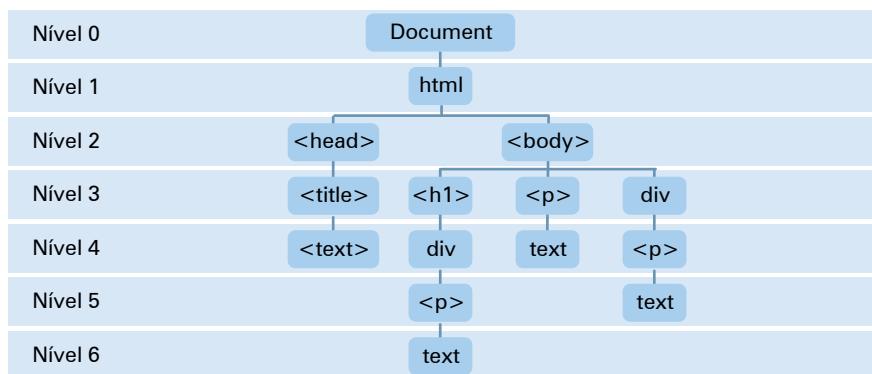
Existe também a tag `<noscript>`, utilizada para escrever um código alternativo caso as instruções JavaScript não possam ser executadas no seu navegador web. Por exemplo:

```
<noscript> Este programa utiliza script em JavaScript e
seu navegador não é capaz de interpretá-los </noscript>
```

## 2 Modelo de documento por objetos (DOM – Document Object Model)

O DOM é uma interface de desenvolvimento web que apresenta todos os objetos em uma página HTML. A manipulação desses objetos se torna possível pela sua representação em forma de árvore (figura 1), na qual cada objeto possui métodos e atributos que viabilizam os recursos que são capazes de alterar essa estrutura dinamicamente.

Figura 1 – Árvore DOM



Cada ramificação da árvore DOM visível no nível 2 é chamada de nó e, em cada nó, há outro nível hierárquico acima ou abaixo, constituído por outros elementos (*elements*), que são referenciados como elementos pai (*parent*), filhos (*children*) e irmãos (*siblings*), representados no nível 3 por `<h1>`, `<p>` e `<div>`, conforme expressado na árvore. Além dos elementos, podemos incluir os atributos e comentários que, agrupados em um nó, chamamos de objetos *nodes*. Cada agrupamento pode ser recuperado pelo seu nome, *NodeName*, ou pelo seu valor, *Nodevalue*. Portanto, o DOM identifica, para cada elemento na restruturação de uma página HTML, a existência da instanciação de um objeto que será referenciado no código JavaScript. Em um código HTML, as tags representam os elementos que são considerados pais, caso existam outras tags internas.

Por exemplo, vamos utilizar o código HTML:

```
<p> Na universidade temos os cursos <a href="#"> Graduação</a> em São Paulo</p>
```

No nó representado pelas tags `<p>` e `<a>`, o elemento `<p>` é pai do elemento `<a>`, que chamamos de filho.

As tags em HTML possuem atributos como *id*, cujo valor deve ser único dentro do código HTML. Quando um código HTML é transformado em DOM, cada nó tem suas próprias propriedades, como *parent* ou *childNodes*, que podem ser identificadas pela propriedade no nó *nodeType*, que fornece um número mapeado em uma classe *Node*. Portanto, dizemos que o DOM fica entre a página HTML e o código em JavaScript, no qual seus objetos possuem propriedades que representam os atributos do código HTML. Por exemplo:

```
var c = document.getElementById('id_titulo');
c.checked = false;
console.log(c.checked, c.getAttribute('checkad')) // 
(resultado: false true)
```

O DOM existe desde o período em que os navegadores passaram a suportar a linguagem JavaScript, época em que os desenvolvedores queriam acessar bits de HTML e alterar suas propriedades. A principal função do DOM é permitir esse acesso aos elementos HTML da página web. Desde seu surgimento, existem três níveis de modelo de documentos por objeto, que são:

- **Nível 0:** suportado pelo Netscape 2 desde seu surgimento até hoje. Surgiu no mesmo período do JavaScript. Por razões de compatibilidade com versões anteriores, os navegadores mais avançados ainda oferecem suporte a esse nível, forçando a Microsoft a copiar o Netscape DOM para o Internet Explorer 3.
- **Nível 2:** suportado pelo Netscape 4 e pelo Internet Explorer 4 e 5. Esse nível não é mais utilizado.
- **Nível 1:** o DOM nível 1, ou DOM W3C, é compatível com Mozilla e Internet Explorer 5. Pela primeira vez, não fornece apenas um modelo exato para todo o documento HTML, mas também permite alterar o documento, retirar um parágrafo e alterar layout de uma tabela.



### PARA SABER MAIS

Conforme a Mozilla ([s. d.]), o objeto *window* representa uma janela do navegador que contém um elemento DOM, cuja propriedade *document* se refere ao documento DOM visível na janela. Uma característica de alguns navegadores é o suporte a abas, sendo que cada aba terá seu objeto *window* e cada janela ou aba terá seu próprio objeto global.

## 3 Selecionando elementos: getElement e querySelector

São formas de selecionar os elementos em HTML: getElementById; getElementsByTagName; getElementsByClassName.

O método *getElementById* viabiliza a localização do elemento pelo atributo *id* criado no código HTML.

Exemplo:

```
1  <p id="id_parag" class="nome_classe"> Centro  
Universitário </p>  
2  var wp = document.getElementById('id_parag');  
3  wp.style.backgroundColor = 'blue'
```

O exemplo anterior apresenta, na linha 1, um exemplo de linha escrita em HTML, cuja tag de parágrafo *<p>* apresenta o parâmetro *id*, que o identifica como *id\_parag*. Na linha 2, temos um exemplo de linha em JavaScript usando o método *getElementById*, que faz referência ao *id* da linha 1. Na terceira linha, atribuímos à variável *wp* outra cor de fundo do texto que está no parágrafo identificado pelo *id*.

Pelo *getElementsByTagName*, também podemos localizar o elemento HTML pelo nome da tag, como *p* para parágrafos, *div* para divisões, *h1* para cabeçalhos e muitos outros.

Exemplo:

```
var parag = document.getElementsByTagName('p');
```

Note que está escrito *elements*, no plural, pois pode existir mais de um elemento pela tag *name*, o que possibilita que os elementos sejam gravados em um array, ao qual podemos nos referenciar, como no próximo exemplo, utilizando o índice do vetor.

```
parag[0].style.backgroundColor = 'red';
```

Por último, o `getElementsByClassName`, assim como o `getElementsByTagName`, retorna um array de elementos.

Outra forma de selecionar elementos é utilizando o `querySelector`, que retorna o valor do primeiro elemento dentro do documento em HTML que seja idêntico ao grupo ou conjunto de seletores. Vamos a um exemplo:

```
1  <html>
2      <head>
3          <title> Interface para entrada de dados </
title>
4      </head>
5      <body>
6          <input type="text" class="textocss">
7          <button onclick="recebeentrada()">Receber</
button>
8
9          <script>
10         function recebeentrada() {
11             var textorecebido = document.
querySelector('.textocss').value;
12             var recebefilho = textorecebido.
childNodes;
13             alert(textorecebido+rebefilho);
14         }
15         </script>
16     </body>
17 </html>
```

Na linha 11, dentro da função, o `document.querySelector` procura no código HTML a primeira ocorrência da classe `".textocss"`. Ao procurar uma classe, sempre se deve inserir o ponto antes do nome.

## 4 Navegando

Para percorrermos entre os nós de um documento DOM, podemos utilizar as palavras equivalentes aos membros de uma família. São eles:

- **Parent:** todo nó que possui o elemento deve ter um elemento pai. Para identificá-lo, usamos:

```
var textorecebido = documento.querySelector('.textocss')
var recebebai = textorecebido.parentNode;
```

- **Children:** um elemento pai pode ter filhos, e podemos identificá-los como no exemplo:

```
var textorecebido = documento.querySelector('.textocss')
var listadefilhos = textorecebido.childNodes; (array
de filhos)
var primeirofilho = textorecebido.firstChild;
var ultimofilho = textorecebido.lastChild;
```

- **Siblings:** assim como um elemento pode ter pai ou filhos, também existe a possibilidade de o elemento ter irmãos, conforme figura 1, nível 4, em que `<div>`, `<text>` e `<p>` são irmãos.

```
var textorecebido = documento.querySelector('.textocss')
var irmao = textorecebido.nextSibling;
```

## 5 Manipulando elementos

Até o momento, aprendemos como recuperar e localizar um elemento em um documento DOM. Agora, conhceremos uma parte importante de suas funcionalidades. Para facilitar a aprendizagem, utilizaremos como exemplo o código HTML:

```
1 <html>
2     <head>
3         <title> Interface para Manipulação do DOM </
title>
4         <script src="programa1.js"></script>
5     </head>
6     <body>
7         <div id="div_a"> </div>
8     </body>
9 </html>
```

Na linha 4, realizamos a chamada do código em JavaScript em arquivo separado chamado “programa1.js”, que será utilizado como exemplo para a chamada do próximo código, que representa o arquivo programa1.js.

Já entendemos que o DOM é uma interface gerada em função da página HTML, mas, se precisarmos criar um novo nó no documento durante a execução de um código em JavaScript, poderemos utilizar o método *createElement*. No código a seguir, utilizamos a instrução da linha 3 para localizar o nó *<div>* do código HTML. Na sequência, na linha 5, criamos o parágrafo *<p>* e instanciamos no objeto *novo\_parag*.

```
1 window.onload=function(){
2
3     var a_div = document.getElementById('div_a');
4
5     var novo_parag = document.createElement('p');
6
7     novo_parag.style.background = 'red';
8
9     var texto = document.createTextNode('Texto Exemplo');
10
11    novo_parag.appendChild(texto);
12
13    a_div.appendChild(novo_parag);
14
15 }
```

Na sequência do código anterior, utilizamos o método *appendChild* (adicionando filhos) para incluir um novo nó como filho de outro nó. Portanto, na linha 13, adicionamos um novo nó abaixo da localização do nó *<div>* do código HTML.

A utilização de texto no documento DOM para refletir na página em HTML oferece duas instruções importantes:

- **Criar um nó de elemento do tipo texto:** na linha 9 da figura 4, a instrução *createTextNode* é utilizada para criar um nó que será adicionado ao documento dentro do parágrafo novo na linha 11.
- **Ler ou alterar o texto:** se quisermos alterar o conteúdo HTML de um elemento, utilizaremos *innerHTML*, que possibilita ler e alterar o texto no documento. Exemplo:

```
document.getElementById('div_a').innerHTML = "Mudar o  
texto";
```

Ou:

```
var x = document.getElementById('div_a').innerHTML ;
```

A instrução `removeChild()` é utilizada para remover um nó filho de um elemento específico. Para isso, utilizaremos o exemplo do código na linha 15:

```
setTimeout(function() { a_div.removeChild(novo_parag); },  
5000 );
```

Esse comando causará uma pausa de 5 segundos antes de realizar a exclusão do novo nó.

```
1 window.onload=function(){  
2  
3     var a_div = document.  
4         getElementById('div_a');  
5  
6     var novo_parag = document.createElement('p');  
7  
8     novo_parag.style.background = 'red';  
9  
10    var texto = document.createTextNode('Texto  
Exemplo');  
11  
12    novo_parag.appendChild(texto);  
13  
14    a_div.appendChild(novo_parag);  
15  
16    setTimeout(function() { a_div.removeChild(novo_  
parag); }, 5000 );  
17  
18    document.getElementById('div_a').innerHTML =  
"Mudar o texto";  
19}
```

Sempre que precisarmos alterar a propriedade CSS de um elemento, devemos utilizar o objeto `style`, conforme linha 7 do código anterior, em que atribuímos a cor `red` para a propriedade `background` do objeto `style`. Também podemos alterar outras propriedades, como:

```
a_div.style.width = "200px";  
a_div.style.height= "100px";
```

## Considerações finais

Neste capítulo, aprendemos o quanto é importante e intuitiva a utilização da plataforma DOM para manipular dinamicamente as informações de uma página HTML em tempo de execução de uma linguagem JavaScript. Também aprendemos como adicionar ou remover novos elementos na página.

## Referências

FLANAGAN, David. **JavaScript**: o guia definitivo. Porto Alegre: Bookman, 2012.

MOZILLA. MDN: **JavaScript**. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 3 mar. 2021.

## Capítulo 6

# Eventos e interação

A interação do usuário com todos os sistemas que ofereçam uma interface gráfica, seja um navegador web ou sistema operacional, gera um evento. Um evento pode ser gerado, por exemplo, quando o cursor passa por cima de um objeto, quando se aciona um botão ou ao fechar uma janela, isto é, a partir de qualquer ação ou reação do usuário.

Se o desenvolvedor em JavaScript precisar realizar uma ação tomando como base algum evento realizado no sistema, essa ação poderá executar uma ou mais funções. Portanto, os eventos estão disponíveis em todas as interfaces, e cabe ao desenvolvedor utilizá-las quando precisar em um sistema.

# 1 Eventos no JavaScript

Os eventos representam as ocorrências realizadas pelo usuário durante sua interação na interface. Para o desenvolvedor, é muito importante entender quais são esses eventos ou a qual objetos eles estão associados. Em JavaScript, existe uma variedade de eventos, sendo estes os principais (FLANAGAN, 2012):

- **onchange:** quando muda o valor de um elemento.
- **onclick:** quando o usuário clica em cima do elemento.
- **onfocus:** quando o elemento recebe o foco, isto é, quando recebe o foco em um salto pela tecla *Tab*.
- **onkeypress:** quando o usuário está no elemento e tecla *Enter*.
- **onMouseOver:** quando o usuário passa o cursor do mouse por cima do elemento.

Evento de um clique no botão é representado no código:

```
1  <!doctype html>
2  <html>
3  <head>
4      <meta charset="UTF-8"/>
5      <title> Exemplo de Evento com Botão </title>
6  </head>
7  <body>
8      <h1> Exemplo de clique no botão </h1>
9      <button>Acionando o Botão</button>
10     <script>
11         var btn = document.querySelector('button');
12         btn.onclick = function() {
13             alert("Você clicou no botão !");
14         }
15     </script>
16 </body>
17 </html>
```

No código anterior, na linha 9, dentro do *body* existe uma tag `<button>`. Dentro da tag do script em JavaScript, que se inicia na linha 10, temos, na linha 12, o objeto *btn*, que faz com que, quando o botão recebe um clique, a função seja acionada. O código também pode ser:

```
8  <h1> Exemplo de clique no botão </h1>
9  <button onclick="acionando()">Acionando o Botão</
   button>
10 <script>
11 function acionando() {
12     alert("Você clicou no botão !");
13 }
14 </script>
```

Na linha 9, alteramos o código anterior. Agora, na tag *button* existe a chamada do evento *onclick*, apontando para a função em JavaScript.

## 2 Adicionando e removendo listeners

O listener (em português, ouvinte) é um método em que o desenvolvedor pode implementar uma ação que deverá ser executada na ocorrência de um determinado evento em um documento. Para fazer a associação do evento *listener*, utiliza-se a função *addEventListener()* relacionada ao evento de um objeto, conforme código:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="UTF-8" />
5      <title> Exemplo de Evento com Botão com
   Listener</title>
6  </head>
7  <body>
8      <h1> Exemplo de clique no botão com Listener </
   h1>
9      <button id = "botao"> Acionando o botão </
   button>
10     <script>
11         function aciona() {
```

```
12           alert("teste realizado");
13       }
14   document.getElementById("botao").
15     addEventListener("click", aciona);
16 </script>
17 </body>
18 </html>
```

No código anterior, na linha 9, utilizamos o *id* do botão para que ele seja localizado na linha 14. Já na linha 14, temos um *addEventListener* para receber a partir de um tipo como o “clique”, de forma a executar a função *aciona*.

Na função *addEventListener*, podemos ter tipos como *click*, *mouseover*, *mouseout*, *dbclick*, *mousemove*.

Caso o desenvolvedor necessite remover o listener em tempo de execução de um *addEventListener*, deverá utilizar a função *removeEventList*. Utilizando como base o código anterior, o exemplo será:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="UTF-8" />
5      <title> Exemplo de Evento com Botão com
6      Listener</title>
7  </head>
8  <body>
9      <h1> Exemplo de clique no botão com Listener </
10     h1>
11      <button id = "botao"> Acionando o botão </
12      button>
13      <script>
14          function aciona() {
15              alert("teste realizado");
16          }
17      document.getElementById("botao").
18      removeEventListener("click", aciona);
19      </script>
20  </body>
21 </html>
```

## 3 Bubble e capture

Os elementos em um documento HTML podem ser organizados ou aninhados dentro de outro elemento, como elementos pai e filho. Quando vinculamos um evento em JavaScript a um elemento filho de um documento, esse evento pode afetar o elemento pai.

Chamamos de bubble quando o efeito do evento atinge os elementos de dentro para fora ou de filho para pai. No próximo código, desenvolvemos um texto na tag parágrafo `<p>` na linha 9, em que está destacada com a tag `<strong>` uma palavra do texto. Tanto o parágrafo quanto a palavra destacada possuem um *id*, sendo o texto completo o pai, e a palavra, o filho, porque é interna ao parágrafo. Na linha 19, no código JavaScript, estamos localizando o elemento pelo *id* *pai* e atribuindo para um objeto *novopai*. Na linha 20, fazemos o mesmo com o elemento *filho*. Na linha 22, realizamos um listener para o pai e para o filho.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="UTF-8" />
5      <title> Exemplo de Bubble</title>
6  </head>
7  <body>
8      <h1> Exemplo de clique no botão Bubble </h1>
9      <p id="pai"> Este texto é para <Strong
  id="filho">demonstrar</strong> a
10     utilização do evento bubble e capture
11     </p>
12     <script>
13         function acionapai(){
14             console.log("Voce deu clique no Pai");
15         }
16         function acionafilho(){
17             console.log("Voce deu clique no filho");
18         }
19         var novopai = document.getElementById("pai");
20         var novofilho = document.
            getElementById("filho");
```

```

21           novopai.
22           addEventListener("click",acionapai,true);
23           novofilho.
24           addEventListener("click",acionafilho,false);
25       </script>
26   </body>
27 </html>

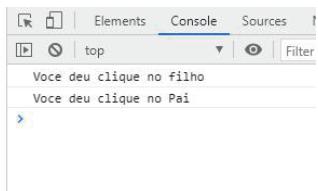
```

Na figura 1, quando executamos o código anterior em um navegador web e abrimos o console, podemos verificar o resultado do console clicando na palavra destacada. Percebemos, então, que os eventos são executados de filho para pai, ou de dentro para fora. Se clicar em qualquer outra palavra do parágrafo, só aparecerá o resultado pai.

**Figura 1 – Execução do bubble**

## Exemplo de clique no botão Bubble

Este texto é para demonstrar a utilização do evento bubble e capture



O evento capture representa o processo inverso do bubble, isto é, os eventos do parágrafo são executados de fora para dentro – primeiro o pai, depois o filho. Mas, para que esse processo funcione, a função `addEventListener` receberá mais um parâmetro, que será o valor `true`, como no exemplo:

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="UTF-8" />
5      <title> Exemplo de Bubble</title>
6  </head>
7  <body>
8      <h1> Exemplo de clique no botão Bubble </h1>

```

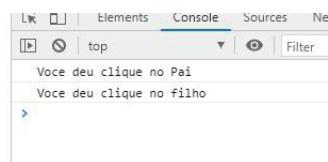
```
9      <p id="pai"> Este texto é para <Strong  
10     id="filho">demonstrar</strong> a  
11     utilização do evento bubble e capture  
12     </p>  
13     <script>  
14         function acionapai(){  
15             console.log("Vou clicou no Pai");  
16         }  
17         function acionafilho(){  
18             console.log("Vou clicou no filho");  
19         }  
20         var novopai = document.getElementById("pai");  
21         var novofilho = document.  
22             getElementById("filho");  
23             novopai.  
24             addEventListener("click",acionapai,true);  
25             novofilho.  
26             addEventListener("click",acionafilho,true);  
27         </script>  
28     </body>  
29 </html>
```

Na figura 2, depois de alterada a linha 22, incluindo o *true*, clicamos novamente na palavra “demonstrar”, mas o resultado gerado é inverso, isto é, primeiro o pai, depois o filho.

**Figura 2 – Execução do capture**

### Exemplo de clique no botão Bubble

Este texto é para **demonstrar** a utilização do evento bubble e capture



## 4 Prevent default e cancelamento

Quando desenvolvemos uma página web, pode surgir a necessidade de que uma tag exista na página, mas, por motivos especiais, não funcione. Por exemplo, um botão que, ao ser clicado, não cause nenhum

evento, ou uma tag de link que não funcione de forma padrão, abrindo uma outra página.

Para que esse evento seja cancelado, utilizamos o método *preventDefault*, que previne ou cancela o comportamento de uma tag na página web, conforme demonstrado no código:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="UTF-8"/>
5      <title> Exemplo de preventDefault</title>
6  </head>
7  <body>
8      <h1> Exemplo de Prevent Default </h1>
9      <h2> Como impedir que o exemplo funcione </h2>
10     <input type="checkbox" id="testecheck"/>
11     <label for="testecheck">Checkbox</label>
12     <input type="checkbox" id="permitir"/>
13     <label>volta a funcionar</label>
14     <script>
15         function pararAcao(evt) {
16             if(!funciona.checked)
17                 evt.preventDefault();
18         }
19         document.getElementById("testecheck") .
addEventListener("click", pararAcao);
20         var funciona = document.
getElementById("permitir");
21     </script>
22 </body>
23 </html>
```

No código, temos um *input checkbox* na linha 10 para o usuário selecionar com um clique do mouse. No entanto, quando usamos um listener para localizar o elemento *input*, chamamos uma função na qual o evento do clique é cancelado. Na linha 12, utilizamos outro *checkbox* para auxiliar a ativação e cancelamento do *preventDefault*. Assim, quando ele já estiver no estado desativado, ele se tornará ativo novamente.



## IMPORTANTE

Nem todos os eventos podem ser cancelados; você pode usar a propriedade `event.cancelable` para verificar se é possível aplicar um `preventDefault`.

## 5 Eventos de mouse

Quando o usuário interage em uma interface desenvolvida em JavaScript, o mouse é capaz de gerar uma série de ações que chamamos de “eventos de mouse”. Existem várias ações, algumas já citadas, sendo as mais importantes:

- **click:** o elemento recebe um clique do ponteiro do mouse.
- **mouseover:** quando se passa o ponteiro do mouse por cima do elemento.
- **mouseout:** quando o ponteiro sai de cima do elemento.
- **dblclick:** quando o usuário clica duas vezes em cima do elemento.
- **contextmenu:** quando o usuário clica com o botão direito do mouse em um elemento.

No código, nas linhas 23, 25, 27 e 29, estamos utilizando a função `addEventListener` para ler os eventos do mouse; conforme evento, uma função executa uma atividade no documento HTML. O objeto `MouseEvent` representa os eventos de interação do usuário com o mouse. Podemos alterar o exemplo do código anterior da seguinte maneira:

```
25 document.getElementById("botaao").  
  addEventListener("click", function(evt){  
26   document.getElementById("exemplotexto").innerHTML = "O  
     mouse foi utilizado"+evt.button; } );
```

Na linha 25, no evento *click* do mouse, a função recebe *evt* como parâmetro, a qual, com o método *button*, recupera o número do botão pressionado em *evt.button*.

## 6 Eventos de teclado

Os eventos de teclado, como o próprio nome diz, representam uma interação na interface do navegador web realizada via teclado. Alguns exemplos dessa ação são o *keydown*, o *keypress* ou *keyup*.

Os objetos do tipo *KeyboardEvent* são utilizados para identificar essa ação. As propriedades mais conhecidas são:

- **keycode**: retorna o código da tecla pressionada.
- **ctrlKey**: retorna *true* se a tecla *Ctrl* for acionada.
- **shiftKey**: retorna *true* se a tecla *Shift* estiver acionada.

No próximo código, na linha 13, temos outro exemplo de captura da tecla pressionada e exibida no parágrafo na linha 11.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="UTF-8"/>
5      <title> Exemplo de Eventos do teclado</title>
6  </head>
7  <body>
8      <h1> Exemplo de Eventos do Teclado </h1>
9      <p> Use o teclado no campo de digitação </p>
10     <input id="entrada" type="text" size="40">
11     <p id="tecla"></p>
12     <script>
13         document.getElementById("entrada").
14             addEventListener("keydown", function(evt) {
15                 document.getElementById("tecla").innerHTML
16                 = " Teclado usado : "+evt.key;
17             } );
18     </script>
19 </body>
20 </html>
```

## 7 Load, scrolling, change e outros eventos

O evento *load* será executado sempre depois que todos os recursos da página web forem carregados, conforme código:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="UTF-8" />
5      <title> Exemplo de Evento Load</title>
6  </head>
7  <body>
8      <h1> Exemplo de EventoLoad </h1>
9      <p id="mensagem"></p>
10     <script>
11         window.addEventListener("load", function(evt)
12 {
13     document.getElementById("mensagem") .
14     innerHTML =
15         " Todos os recursos terminaram de carregar";
16     } );
17     </script>
18 </body>
19 </html>
```

Na linha 11, o evento será imediatamente executado no momento que todos os recursos da página forem carregados.

O evento *scroll* é utilizado para realizar um scroll bar na página web, conforme código:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <style>
5  .test {background-color: yellow; }
6  </style>
7  </head>
8  <body style="height:1500px">
9  <h1>Página com Scroll</h1>
10 <p id="parag1" style="position:fixed">Esta pagina vai
    possuir um scroll bar</p>
11 <script>
12 window.onscroll = function() {meuScrool()};
13 function meuScrool() {
14     if (document.body.scrollTop > 50 || document.
    documentElement.scrollTop > 50) {
15         document.getElementById("parag1").className =
    "test";
16     } else {
17         document.getElementById("parag1").className = " ";
18     }
19 }
20 </script>
21 </body>
22 </html>
```

Na linha 12, a janela do Windows executará uma scroll bar na lateral.

O evento *change* será utilizado caso o desenvolvedor queira saber se um valor do elemento sofreu alteração. Vamos acompanhar o exemplo no código:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  </head>
5  <body>
6  <h1>Pagina para testar o Change mudando minúsculo para
maiúsculo</h1>
7  <input id="texto">
8  <script>
9  document.getElementById("texto") .
addEventListener("change", trocaTexto);
10 function trocaTexto() {
11     var t = document.getElementById("texto");
12     t.value = t.value.toUpperCase();
13 }
14 </script>
15 </body>
16 </html>
```

Na linha 9, se a informação na entrada de texto sofrer alguma alteração, a função *trocaTexto()* será executada, realizando uma substituição do texto.

## 8 Custom events

O *customEvent()* é uma classe que permite criar e disparar eventos DOM customizados nos navegadores web. Essa classe possui uma propriedade chamada *detail*, que é fundamental para fornecer dados customizados.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  </head>
5  <body>
6  <h1>Pagina para testar o customevents</h1>
7  <button id="botao"> Clique aqui </button>
8  <script>
9      let btn = document.getElementById("botao");
10     btn.addEventListener("pressiona", function() {
11         console.log("O botao foi acionado");
12     });
13     let evt = new CustomEvent("pressiona", {
14         detail:{autor:"Rômulo Maia" }
15     );
16     btn.dispatchEvent(evt);
17 </script>
18 </body>
19 </html>
```

Na linha 13, estamos customizando um novo evento nesse código, chamado *pressiona*. No *detail*, informamos que, na linha 10, o listener está aguardando pelo novo evento.

## 9 Interagindo com o usuário

Até aqui, conhecemos vários recursos interessantes para desenvolvimento em JavaScript. Agora, nós os reuniremos em um só programa para criar uma interação com o usuário. Vamos acompanhar o exemplo no código:

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4  </head>
5  <body>
6      <h1>Página para Interagir com o Usuário</h1>
7      <h2>Programa para somar dois valores</h2>
8      <p> Entre com o Primeiro número : </p>
9      <input type="text" id="valor1" value="Digite aqui">
10     <p> Entre com o segundo número : </p>
11     <input type="text" id="valor2" value="Digite aqui">
12     <input type="submit" id="soma" value="Somar">
13     <p id="resultado"></p>
14     <script>
15         document.getElementById("soma") .
addEventListener("click",ler);
16         function ler() {
17             var num1 = document.
getElementById("valor1").value;
18             var num2 = document.
getElementById("valor2").value;
19             total = parseInt(num1) + parseInt(num2) ;
20             document.getElementById("resultado") .
innerHTML = ' Resultado da Soma = '+total;
21         }
22     </script>
23     </body>
24     </html>

```

No código anterior, construímos uma página que interage com o usuário: nas linhas 9 e 11, utilizamos o *input* para o usuário digitar os valores que serão somados, conforme figura 3. Após clicar no botão *Somar*, que será lido na linha 15, a função *ler()* será executada e, nela, as variáveis *num1* e *num2* receberão os valores digitados. Na linha 19, os valores das variáveis serão convertidos para tipos numéricos inteiros, porque toda informação da página é considerada como tipo “texto”. Logo, para somar, precisamos converter para “inteiro”.

**Figura 3 – Tela de interação**

## Página para Interagir com o Usuário

### Programa para somar dois valores

Entre com o primeiro número :

2

Entre com o segundo número :

3

Somar

Resultado da Soma = 5

## 10 Timers: setTimeout, interval, requestAnimationFrame

A linguagem JavaScript possui os chamados “temporizadores”, que são funções utilizadas para executar outras funções após um determinado tempo, isto é, para executar um determinado bloco em intervalos de tempo ou aguardar sua execução por um tempo. Vamos conhecer alguns métodos:

- **setTimeout()**: usado para aguardar por um determinado tempo antes de executar determinada ação.

Sintaxe:

```
setTimeout(function() { código }, tempo em  
milissegundos);
```

Código:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="UTF-8" />
5      <title> Exemplo de preventDefault</
6      title>
7  </head>
8  <body>
9      <h1> Exemplo de setTimeout() </h1>
10     <input type="submit" id="aparecer">
11     <p id="mensagem"></p>
12     <script>
13         document.getElementById("aparecer").
14         addEventListener("click", execacao);
15         function execacao() {
16             setTimeout(function() {
17                 document.
18                 getElementById("mensagem").innerHTML = "Mensagem
19                 após 3 segundos";
20             }, 3000);
21         }
22     </script>
23     </body>
24 </html>
```

Na linha 14, estamos usando a função `setTimeout`, que define que a mensagem só vai aparecer depois de 3000 milissegundos. Para parar a execução, utiliza-se o método `clearTimeout()`.

- **setInterval()**: é utilizado para executar a função ou retornar valores da função em intervalos de tempos estabelecidos em milissegundos.

Sintaxe:

```
setInterval(function() {código}, tempo em
milissegundos);
```

- Utilizaremos o exemplo do código anterior, alterando a função para:

```
21 function execacao() {  
22     setInterval(function() {  
23         var datahoje = new Date();  
24         document.getElementById("mensagem") .  
innerHTML =  
25             "Horário: "+datahoje.  
toLocaleTimeString();},1000);  
26 }
```

No código, a função `setInterval()` será executada a cada 1000 milisegundos. Para parar a execução, é utilizado o método `clearInterval()`.

- **requestAnimationFrame():** esse método é utilizado para atualizar uma animação na página web. Sempre que acionada a função, ocorre uma chamada na página antes que o navegador realize uma atualização (repaint) nela.

Código:

```
1  <!DOCTYPE html>  
2  <html>  
3      <style>  
4          .quadrado{  
5              background-color:#00008b;  
6              color:#FFF;  
7              width:50px;  
8              height:50px;  
9              padding:30px; }  
10     </style>  
11     <head>  
12         <meta charset="UTF-8" />  
13         <title> Exemplo de preventDefault</title>  
14     </head>  
15     <body>  
16         <h1> Exemplo de requestAnimationFrame </h1>  
17         <div id="quad1" class="quadrado"></div>  
18         <script>  
19             var comeca = null;  
20             var element = document.  
getElementById("quad1");
```

```
21     element.style.position = 'absolute';
22     function step(timestamp) {
23         if (!comeca) comeca = timestamp;
24         var posicao = timestamp - comeca;
25         element.style.left = Math.
26             min(posicao/10, 200) + "px";
27         if (posicao < 4000) {
28             window.requestAnimationFrame(step);
29         }
30         window.requestAnimationFrame(step);
31     </script>
32 </body>
33 </html>
```

No código, utilizamos, a partir da linha 27, o método *requestAnimationFrame()* para chamar a função *step()* a cada 60 milissegundos. Na linha 30, realizamos nova chamada ao método *requestAnimationFrame* para atualizar a imagem. Na figura 4, temos o resultado da animação, em que o retângulo se movimenta na página.

**Figura 4 – Resultado da animação**



## Considerações finais

Neste capítulo, aprendemos a identificar várias ações que ocorrem em uma página web e que provocam a execução de um método. Essas ações podem ocorrer por meio do teclado ou do mouse, durante a interação do usuário na página apresentada no seu navegador.

Mais importante do que identificar a ação é saber o que executar quando ocorre a ação que estamos esperando durante essas interações. Aprender a controlar os eventos em uma página gera resultados que contribuem com a eficiência de seu projeto.

## Referências

FLANAGAN, David. **JavaScript**: o guia definitivo. Porto Alegre: Bookman, 2012.

## Capítulo 7

# Comunicação com o back-end

Neste capítulo, conhceremos todo o processo de comunicação de sua página web com um servidor web para a recuperação de informações armazenadas. A interação com o servidor depende de muitas variáveis, tais como o tempo de resposta e como essas informações estarão disponíveis para serem lidas. Do lado cliente, precisamos tratar essa comunicação cuja requisição nem sempre retorna quando é executada. Portanto, é fundamental saber como os dados estão disponíveis e como essas informações serão tratadas na página web.

# 1 Coletando e manipulando dados de formulários

Os formulários são criados para facilitar a interação do usuário com sua interface na página web, possibilitando o envio de informações da página para o servidor web, onde esta será processada e armazenada. Para criar um formulário em HTML, é necessário criar uma tag `<form>`. Dentro de um formulário, você pode utilizar as tags `<input>`, `<label>`, `<button>`, entre outras.

A sintaxe para criação de formulários é:

```
<form action="/url" method="post">  
  
</form>
```

Nesse formato, o atributo `action` representa o local para onde os dados preenchidos no formulário serão enviados; caso ele não seja fornecido, assume-se como local a URL da página do formulário. O atributo `post` representa o método HTTP, que será utilizado como GET ou POST.

Por exemplo:

```
1  <!DOCTYPE html>  
2  <html>  
3  <head>  
4      <meta charset="UTF-8" />  
5      <title> Exemplo Manipulando Formulários</title>  
6  </head>  
7  <body>  
8      <h1> Manipulando Formulários </h1>  
9      <form name="cadastro" id="cadastro" action="#"  
10         method="post">  
11             <label><br> Código : </label>  
12             <input type="text" name="campoCodigo"  
13             id="wCodigo" >  
14                 <label><br> Nome do Aluno : </label>
```

```

13          <input type="text" name="campoNome"
14          id="wNome" value="Digite o nome">
15          <label> <br>Email : </label>
16          <input type="email" name="campoEmail"
17          id="eEmail">
18          <input type="submit" name="botaoEnviar"
19          value="Enviar">
20      </form>
21      <script>
22          var formulario = document.forms.length;
23          console.log(formulario);
24      </script>
25  </body>
26 </html>

```

No código, da linha 9 até a 17, nós construímos o formulário de entrada de dados em HTML, e, na linha 19, utilizamos a propriedade *forms* da classe *document*, em que a propriedade *length* recupera a quantidade de elementos *forms*, que, no nosso caso, é 1.

A figura 1 representa a execução da aplicação descrita no código anterior para exibir um formulário em tela, contendo caixa de texto para digitação do código, nome do aluno e e-mail.

**Figura 1 – Tela do formulário**



## Manipulando Formulários

Código :

Nome do Aluno :

Email :

Dessa forma, começamos a manipular nosso formulário e, inserindo o próximo código entre as linhas 19 e 20, teremos o retorno no console do *id* do formulário.

```
19 var formulario=document.forms[0].id;  
20 console.log(formulario);
```

Em um novo exemplo, podemos mostrar que, na linha 19, é possível recuperar o layout do formulário e atribuir para uma variável. Também podemos recuperar os valores digitados nos *inputs* dentro do formulário como um vetor de elementos do formulário, como nas linhas 23, 24 e 25.

```
18 <script>  
19 var formulario = document.forms[0].innerHTML;  
20 document.write("<br>Copiando o formulário  
<br>"+formulario);  
21 var formulario = document.forms[0].innerHTML;  
22 document.write("<br>Valores dos campos <br>");  
23 document.write("<br>"+document.forms[0].elements[0].  
value);  
24 document.write("<br>"+document.forms[0].elements[1].  
value);  
25 document.write("<br>"+document.forms[0].elements[2].  
value);  
26 </script>
```

Como resultado, temos a figura 2.

Figura 2 – Resultado do script

# Manipulando Formulários

Código :

Nome do Aluno :

Email :

Copiando o formulário

Código :

Nome do Aluno :

Email :

Valores dos campos

1234  
Digite o nome  
xxx@xxxx.xx

## 2 Requisições assíncronas: XHR e fetch

O XHR ou XMLHttpRequest é um objeto que oferece recursos para a transferência de dados entre o usuário e o servidor. Ele permite atualizar os dados sem provocar uma atualização da página web. Para instanciar um objeto, utilizamos a sintaxe:

```
var minharesposta = new XMLHttpRequest();
```

No próximo código, na linha 11, realizamos uma requisição no servidor web. Na linha 21, realizamos uma requisição do arquivo texto da linha 20 para que ocorra uma requisição assíncrona sem travamento.

Já na linha 20, ocorre a conexão com o servidor e o parâmetro deve ser *true*. Na linha 12, a propriedade *onreadystatechange* recebe uma função que será executada sempre que um evento ocorrido pela linha 21 *for* disparado. Na linha 13, o *estado 4* informa que a informação já está disponível pelo servidor, e, na linha 14, fazemos o teste para saber se a informação foi concluída e se toda a comunicação terminou com sucesso.

```
1  <!DOCTYPE html>
2  <html>
3  <body>
4      <h2>Usando XMLHttpRequest Object</h2>
5      <div id="texto">
6          <button type="button" onclick="lerDoc()">Change
Content</button>
7      </div>
8
9      <script>
10     function lerDoc() {
11         var texto1 = new XMLHttpRequest();
12         texto1.onreadystatechange = function() {
13             if (this.readyState == 4 ) {
14                 if (this.status == 200) {
15                     document.getElementById("texto").innerHTML =
16                     this.responseText;
17                 }
18             }
19         };
20         texto1.open("GET", "http://httpbin.org/get", true);
21         texto1.send();
22     }
23     </script>
24     </body>
25     </html>
```



## IMPORTANTE

Uma comunicação síncrona representa a comunicação em tempo real, em que se envia a mensagem e a resposta retorna no mesmo momento. Já na comunicação assíncrona, envia-se a mensagem, mas a resposta pode não retornar no mesmo momento da comunicação.

A API Fetch surgiu para oferecer uma funcionalidade melhor e de utilização mais fácil por outras tecnologias, como as usadas por Service Workers. Por exemplo, vamos utilizar o código onde substituiremos o código da função *lerDoc()*:

O método fetch, no momento de sua execução, realiza uma requisição HTTP, trazendo a informação da URL informada. Esse resultado retorna uma promise para que o desenvolvedor o trate. Para tratar o resultado, utilizamos o método then, que recebe uma função para mostrá-lo no console.

Figura 3 – Resultado do método fetch

```
Welcome Elementos Console Fontes Rede Desempenho Memória Aplicativo Segurança
top Filtro Níveis padrão ▾
Algunas mensagens foram movidas para o painel problemas.
Response {type: "cors", url: "http://httpbin.org/get", redirected: false, status: 200, ok: true, ...} ⓘ
  body: (...)

  headers: Headers {}
    ok: true
    redirected: false
    status: 200
    statusText: "OK"
    type: "cors"
    url: "http://httpbin.org/get"
  __proto__: Response
```

Na figura 3, temos o resultado do método fetch no console do navegador, onde constam várias informações do servidor conforme a URL. Observem o status 200, informando que a requisição foi um sucesso.

No entanto, ainda não é possível trabalhar com essa informação, pois precisamos transformar os dados de acordo com o tipo de informação que queremos trabalhar (HTML ou JSON), aplicando o código:

```
20  function lerDoc() {  
21      fetch("http://httpbin.org/get")  
22          .then((resultado)=> resultado.json())  
23          .then((json) => console.log(json) )  
24  }
```

## 2.1 Requisições (URL, HTTP, query strings, body e headers)

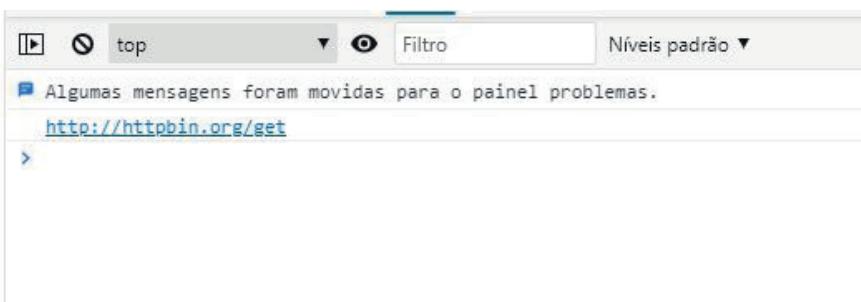
Agora, precisamos entender alguns métodos das requisições assíncronas. O método `fetch` retorna um objeto do tipo `response`, e podemos perceber o retorno de várias informações, como URL, o método http, body e headers.

O resultado de uma requisição ou `response` conta com uma variedade de métodos para realizar a conversão da informação recebida. Portanto, se você quiser no formato URL, deve alterar a função para:

```
20  function lerDoc() {  
21      fetch("http://httpbin.org/get")  
22          .then((resultado)=> resultado.url)  
23          .then((json) => console.log(json) )  
24  }
```

Note que, na linha 22, o `response` será uma URL, conforme figura 4.

**Figura 4 – Resultado da requisição sendo URL**



Até agora, compreendemos que o fetch realiza uma requisição HTTP e, por default, utiliza o método GET para receber as respostas. A sintaxe completa do fetch é:

```
fetch(input: string, {
    method?: "GET" | "POST" | "PUT" | "DELETE",
    mode?: "navigate" | "same-origin" | "no-cors" |
    "cors",
    headers?: { [ key: string ]: any }
}): Promise<Response>
```

Nessa sintaxe, temos um parâmetro fundamental chamado *mode*, que representa o tipo de requisição executada no servidor, que informa se as solicitações de origem podem receber respostas válidas do servidor e quais respostas podem ser acessadas. Essa informação fornecida pelo servidor pode ser *same\_origin*, em que o domínio da solicitação de origem é o mesmo do servidor, e *Cors* (Cross-Origin Resource Sharing), em que as solicitações são de origem cruzada – as APIs são acessadas por fornecedores diferentes do servidor.

Portanto, também podemos utilizar o método POST para atualizar as informações na URL, conforme código:

```
20  function lerDoc() {
21      fetch('http://httpbin.org/post', { method:'POST',
22          body:'Romulo'})
23      .then((resultado)=>resultado.json())
24      .then((json) => console.log(json)
25  }
```

Cujo resultado é demonstrado na figura 5.

**Figura 5 – Resultado do método POST**



O query string é o processo utilizado para executar um GET na URL e capturar ou enviar um parâmetro nessa leitura, isto é, quando se pretende obter o valor do parâmetro *id* passado na URL. Exemplo:

```
25  function lerDoc() {
26      fetch("http://httpbin.org/get?t=1")
27          .then((resultado)=> resultado.json())
28          .then((json) => console.log(json) )
29  }
```

Nesse exemplo, o GET será executado para obter dados específicos do parâmetro *t=1*. No resultado do exemplo, é apresentado o parâmetro no campo *args*.

**Figura 6 – Resultado do query string**



Uma requisição *body* representa o corpo de requisição, para onde enviamos as informações que desejamos armazenar no servidor web. Apesar de poder ser utilizada no método GET, essa requisição é mais

usada com os métodos POST e PUT. Nesse local, estão as informações de um formulário usado para cadastro. No exemplo, enviaremos um *body* na requisição:

```
20 function lerDoc() {  
21     fetch('http://httpbin.org/post', { method:'POST',  
22         body:'Romulo'})  
23     .then((resultado)=>resultado.json())  
24     .then((json) => console.log(json))  
25 }
```

No código, quando realizamos um método POST e declaramos o objeto *body* para armazenar a informação do autor “Romulo”, no resultado exibido no console a informação aparece como do tipo *data*, conforme figura 7.

Segundo Mozilla ([s. d.]), “um objeto headers é um mapa multidimensional simples dos nomes para valor”. No próximo código, temos um exemplo de um header no servidor web. Nesse objeto, é possível armazenar a informação do cabeçalho da requisição para seu arquivo no servidor web. Por exemplo, na linha 21 podemos complementar a instrução:

```
fetch('http://httpbin.org/post', {  
    method:'POST',  
    headers: {'Content-Type': 'text/plain;charset=UTF-8'  
},  
    body:'Romulo' })
```

Na figura 7, expandimos as informações no servidor web de nosso arquivo e mostramos o cabeçalho com outras informações.

**Figura 7 – Cabeçalho do arquivo**



The screenshot shows a browser's developer tools Network tab. A POST request to "httpbin.org/post" is selected. The Headers section displays the following:

```
Alumas mensagens foram movidas para o painel problemas.
Visualizar problemas
exemplo36.html:13

{
  args: {},
  data: "Romulo",
  files: {},
  form: {},
  headers: {
    Accept: "*/*",
    Accept-Encoding: "gzip, deflate",
    Accept-Language: "pt-BR,pt;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6",
    Content-Length: "6",
    Content-Type: "text/plain;charset=UTF-8",
    Host: "httpbin.org",
    Origin: "null",
    User-Agent: "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4389.72 Safari/537.36 Edg/89.0.774.45",
    X-Amzn-Trace-Id: "Root=1-604a191a-2b0bf440053eeced04b9e376"
  },
  json: null,
  origin: "201.95.113.163",
  url: "http://httpbin.org/post"
}
```

## 3 Promises

Uma promise representa um objeto de uma requisição concluída em um servidor web ou uma falha na requisição executada. Quando executamos o código de uma função assíncrona, essas requisições não acontecem no mesmo momento em que o código é executado. Uma promise possui duas propriedades, chamadas de *state* e *result*. A seguir, veremos o que representa cada tipo de state:

- Quando o *state* for *pending*, o *result* é *undefined* (iniciou e não foi concluído).
- Quando o *state* for *fulfilled*, o *result* é a *result value* (iniciou e foi concluído).
- Quando o *state* for *rejected*, o *result* é *an error object* (ocorreu falha na requisição).

Na figura 8, note que, se executarmos o método `fetch` diretamente no console, retornará uma promise que ainda não foi retornada pelo servidor web, seu *state* é *pending*. Quando essa promise for resolvida, o método `fetch` executará a função *then*, que retornará um callback.

**Figura 8 – Retorno de uma promise**



A screenshot of a browser's developer tools console. At the top, there are buttons for 'top', 'Filtro' (Filter), and 'Níveis padrão' (Default levels). Below the buttons, the console shows the following code and its execution result:

```
> fetch("http://httpbin.org/get")
< Promise {<pending>}
> |
```

Somente o comando `fetch ("http://httpbin.org/get")` executado no console do navegador, como exemplificado na figura 8, retornou a promise como resultado.

```
20   fetch("http://httpbin.org/get").then((resultado)=>{
21       console.log(resultado);
22   })
```

## 4 Tratando o resultado – do JSON para o DOM

O JSON é uma formatação de dados baseada em texto para serialização de objetos, números e outros. Facilita a troca de dados de forma legível, sem depender de uma plataforma de banco de dados. Exemplo:

```
" nome " : " Senac "
```

Ou:

```
var turma = { "nome":"Tecnologia em
Analise","periodo":"Noite"};
```

Na figura 9, temos o resultado do formato JSON no console, em que, atribuindo à variável `var` uma array no formato JSON, se exemplifica a recuperação da informação atribuída.

**Figura 9 – Formato JSON**



A screenshot of a browser's developer tools console. The tabs at the top are Elements, Console, Sources, Network, Performance, Memory, Application, and Security. The Console tab is selected. The console window shows the following JavaScript code and its output:

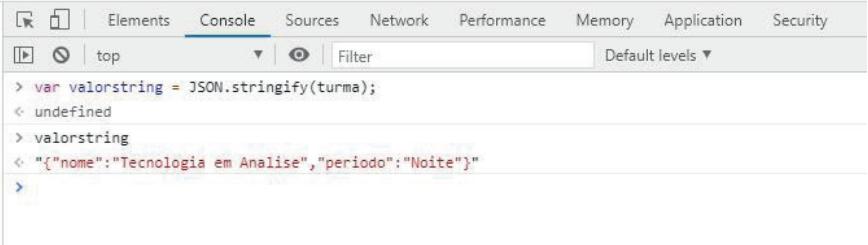
```
> var turma = { "nome": "Tecnologia em Analise", "periodo": "Noite"};
< undefined
> turma.nome
< "Tecnologia em Analise"
> |
```

Para converter o formato JSON em uma string, utilizamos a função *stringify()*. Por exemplo:

```
var valorstring = JSON.stringify(turma);
```

Na figura 10, temos a conversão do formato JSON para string, em que se atribui um *JSON.stringify* para uma variável. Na linha seguinte, quando solicitado o valor da variável, é possível identificar seu valor de memória.

**Figura 10 – Conversão de JSON para string**



A screenshot of a browser's developer tools console. The tabs at the top are Elements, Console, Sources, Network, Performance, Memory, Application, and Security. The Console tab is selected. The console window shows the following JavaScript code and its output:

```
> var valorstring = JSON.stringify(turma);
< undefined
> valorstring
< "{"nome": "Tecnologia em Analise", "periodo": "Noite"}"
>
```

Para converter uma string JSON em um valor ou objeto, utilizamos a função *parse()* em um programa JavaScript, conforme figura 11.

Figura 11 – Valores convertidos com parse()



A screenshot of a browser's developer tools console. The title bar says "Default levels". The console has the following content:

```
> const valoresjson = '{"situacao":true,"idade":34,"curso":"TADS"}';
< undefined
> const dadosobjt = JSON.parse(valoresjson);
< undefined
> console.log(dadosobjt.curso+" "+dadosobjt.idade+" "+dadosobjt.situacao);
TADS 34 true
< undefined
> |
```

Na figura 11, o objeto *dadosobjt*, na linha 3, é criado em função da conversão usando *parse()*. Na linha 5, conseguimos manipular os elementos do objeto.

No próximo código, nas linhas 16 e 17, tratamos o objeto JSON para atribuirmos os valores como uma string completa ou usando o *parse()* para separar cada campo do objeto JSON. Na linha 18, temos um exemplo de como exibir no documento toda a string. Nas linhas 19, 20 e 21, identificamos cada elemento do objeto *osdados* para exibi-los separadamente.

```
1  <!DOCTYPE html>
2  <html>
3  <body>
4      <h2>Tratando o JSON para DOM</h2>
5      <input type="submit" id="exibir" value="Mostrar">
6      <div id="texto">
7          </div>
8          <p id="tx1"> </p>
9          <p id="tx2"> </p>
10         <p id="tx3"> </p>
11         <p id="tx4"> </p>
12         <script>
13             document.getElementById("exibir") .
addEventListerner("click",exibir);
14             function exibir() {
```

```

15     var turma = '{ "nome": "Tecnologia em
16         Analise", "periodo": "Noite", "idade": 25,
17         "situacao": true}';
18     var completo = JSON.stringify(turma);
19     var osdados = JSON.parse(turma);
20     document.getElementById("texto").innerHTML =
21         "Completo : " + completo;
22     document.getElementById("tx1").innerHTML = "Nome
23         do curso : " + osdados.nome;
24     document.getElementById("tx2").innerHTML =
25         "Periodo : " + osdados.periodo;
26     document.getElementById("tx3").innerHTML =
27         "Idade : " + osdados.idade;
28 }
29 </script>
30 </body>
31 </html>

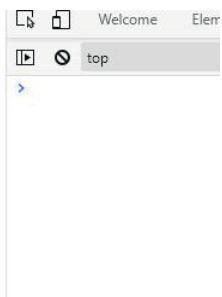
```

Na figura 12, temos o resultado exibindo o JSON com o DOM.

**Figura 12 – JSON para DOM**

## Tratando o JSON para DOM

Completo : "{ \"nome\": \"Tecnologia em  
 Analise\", \"periodo\": \"Noite\", \"idade\": 25, \"situacao\": true}"  
 Nome do curso : Tecnologia em Analise  
 Periodo : Noite  
 Idade : 25



## Considerações finais

A comunicação entre a página web que está no lado cliente com as informações que estão armazenadas no servidor web exige uma troca de pacotes em que, além dos dados, existem as informações de identificação do processo que está sendo executado. Se o servidor tem certificado, pode-se usar HTTP ou HTTPS, principalmente com o envio de

mensagens ao servidor e aguardando uma resposta. Do lado cliente, há a preocupação com a interface, com o usuário, com a construção do formulário e com a manipulação dessas informações através do DOM. Portanto, essa comunicação exige uma atenção redobrada para que todo o processo possa funcionar de forma rápida e leve para a aplicação e, principalmente, transparente para o usuário.

## Referências

MOZILLA. MDN: **JavaScript**. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 3 mar. 2021.

## Capítulo 8

# Tópicos avançados

Neste capítulo, conhceremos algumas bibliotecas e aprenderemos recursos avançados que podem contribuir para deixar os aplicativos com um layout diferenciado, criando animações e efeitos na interface que tornem a interação do usuário mais fácil. Em algumas aplicações, o uso de bibliotecas oferece agilidade para o desenvolvimento e eficiência da página web.

# 1 Bibliotecas: uso, criação e distribuição (CDN e NPM)

A rede de fornecimento de conteúdo ou CDN (Content Delivery Network) representa uma rede de servidores de conteúdos localizados geograficamente em locais estratégicos no mundo, com o objetivo de entregar mais rapidamente os arquivos para seus usuários. Essas bibliotecas permitem acelerar o conteúdo de uma página, de arquivos CSS ou de JavaScript (FLANAGAN, 2012).

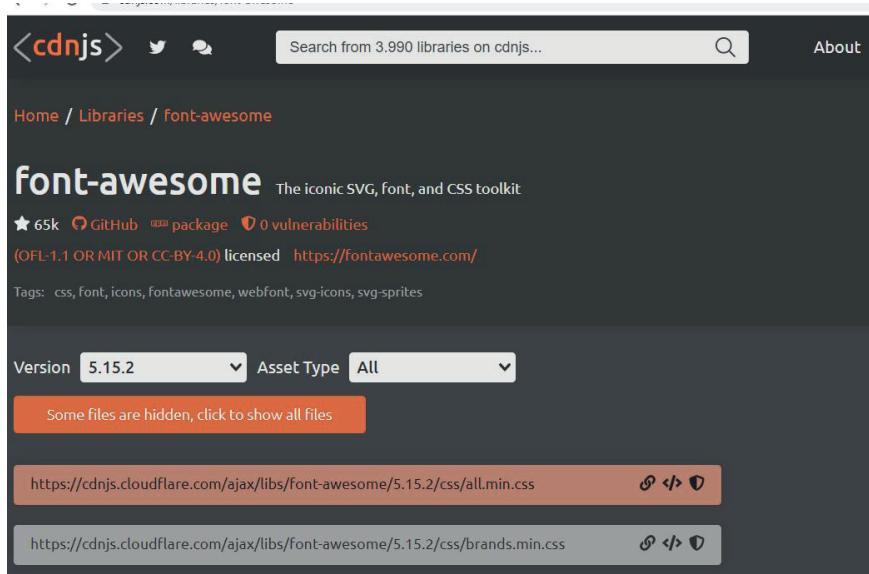
Para exemplificar a CDN, utilizaremos o endereço do servidor <https://cdnjs.com/>. No link *bibliotecas (libraries)*, teremos uma lista de endereços com conteúdo para inúmeras funcionalidades e, neste exemplo, incluiremos uma imagem de ícone no botão, como apresentado na figura 1.

Figura 1 – Tela do exemplo CDN



O primeiro passo é incluir o link da biblioteca em seu código, o copiando a linha 6 do código a seguir no site da figura 2, onde está destacado. Note que o código da linha 6 está na tag `<head>`.

**Figura 2 – Site da biblioteca CDN**



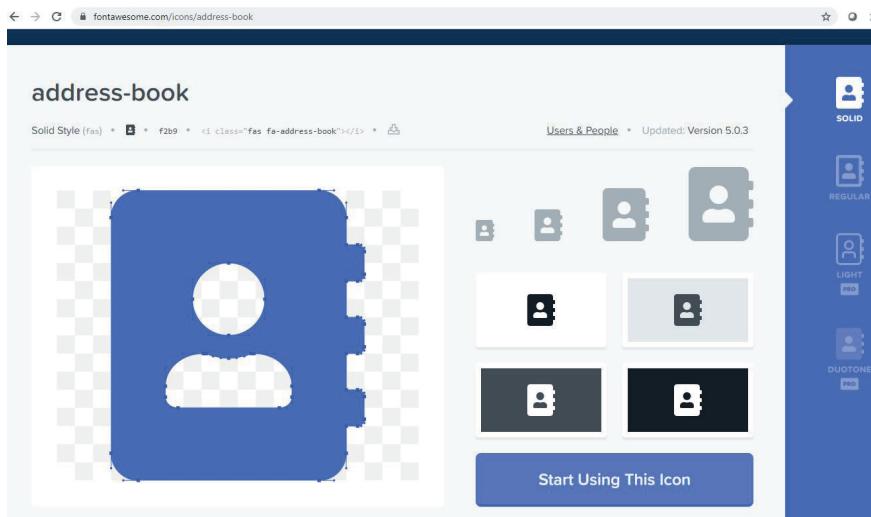
No exemplo a seguir, na linha 11, utilizamos dentro da tag `<h1>` o código fornecido no site da biblioteca CDN, que possibilita a utilização do ícone da figura 1. Portanto, utilizamos o código `<i class="fas fa-book"></i>` para capturar a imagem para o botão, conforme figura 3.

```

1  <!doctype html>
2  <html>
3  <head>
4      <meta charset="UTF-8" />
5      <title> Clic no botão para Exibir o texto</title>
6      <link rel="stylesheet" href="https://cdnjs.
cloudflare.com/ajax/libs/font- awesome/5.15.1/css/all.
min.css"
7          integrity="sha512-+4zCK9k+qNFUR5X+cKL9EIR+Z0htIloN
19GIKS57V1MyNsYpYcUrUeQc9vNfzsWfV28IaLL3i96P9sdNyeRssA =="
8          crossorigin="anonymous" />
9  </head>
10 <body>
11     <h1> Exemplo de Bibliotecas CDN <i class="fas fa-
address-book "></i></h1>
12     <h2 id="mensagem"> Nova Mensagem</h2>
13     <button onclick="exibir()" <i class="fas fa-
book"></i>> Veja aqui </button>
14     <script>
15         function exibir() {
16             document.getElementById("mensagem").innerHTML
= "Nova Mensagem ";
17         }
18     </script>
19 </body>
20 </html>
```

Utilize o link <https://fontawesome.com/> que foi apresentado na figura 2 como fonte CDN. Nesse link você encontra imagens de ícones gratuitos para serem utilizados como exemplos no código, como na linha 11 da figura 3, onde temos: *<i class="fas fa-address-book "></i>*, que foi copiado do site depois de localizar a imagem.

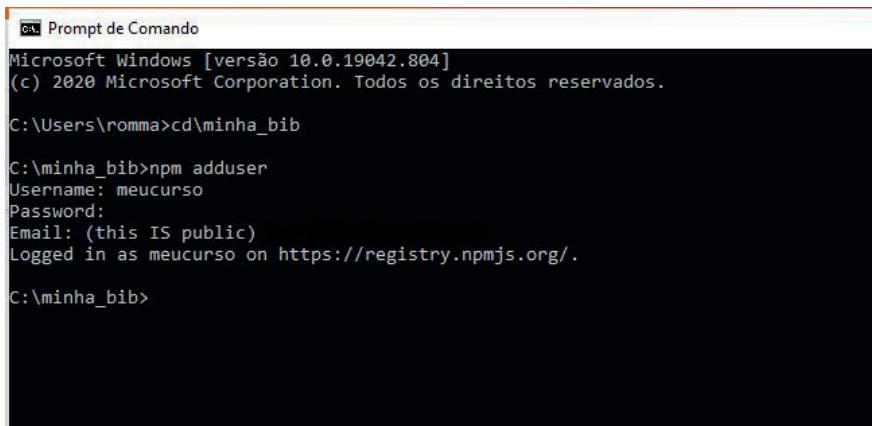
**Figura 3 – Site do ícone**



O NPM (Node Package Manager) é conhecido como gerenciador de pacotes no Node.js. Ele funciona de duas formas: como um repositório público de projetos Node.js e como uma plataforma de interação de linha de comando. Para utilizar esse gerenciador, precisamos criar uma conta no endereço <https://www.npmjs.com/> e, em seguida, instalar no computador o Node.js.

Para iniciar o processo de publicação de uma nova biblioteca, desenvolva o conteúdo em um diretório em seu computador e instale o pacote *npm*. Após instalado, crie uma conta de usuário *npm adduser* com um *<nome\_usuario>*, uma senha e o e-mail da conta criada no site [npmjs.com](https://www.npmjs.com). Assim, teremos uma página onde publicaremos nosso pacote, conforme figura 4.

**Figura 4 – Criando o usuário npm**



```
Prompt de Comando
Microsoft Windows [versão 10.0.19042.804]
(c) 2020 Microsoft Corporation. Todos os direitos reservados.

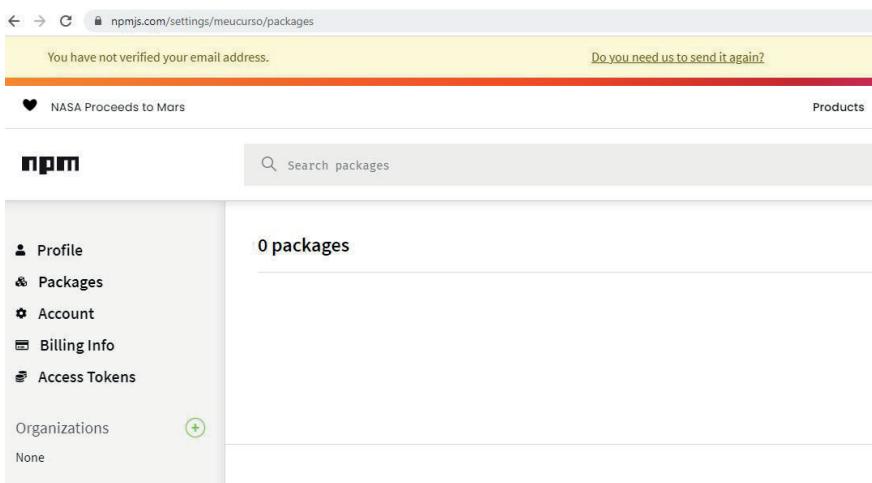
C:\Users\romma>cd\minha_bib

C:\minha_bib>npm adduser
Username: meucurso
Password:
Email: (this IS public)
Logged in as meucurso on https://registry.npmjs.org/.

C:\minha_bib>
```

Na sequência, temos a figura 5, com a página web da biblioteca, acessada pelo endereço NPM.

**Figura 5 – Página web da biblioteca**



Para iniciar o pacote, executamos *npm init*, conforme figura 6, em que devem ser identificados o nome do pacote, a versão, a descrição e um comando de teste que chamaremos de *mensagem()*, mas o principal é o *entry point*, que representa o programa principal que usa sua biblioteca, que pode ser o *index.js*. Outra informação pedida é o GitHub, onde estará o endereço de seu repositório no git.

**Figura 6 – Criando repositório no GitHub**

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

---

Owner \*      Repository name \*

/  ⚠

Great repository names are [The repository minha\\_bib already exists on this account.](#) out [redesigned-waddle?](#)

Description (optional)

---

 **Public**  
Anyone on the internet can see this repository. You choose who can commit.

 **Private**  
You choose who can see and commit to this repository.

---

Initialize this repository with:

Skip this step if you're importing an existing repository.

**Add a README file**  
This is where you can write a long description for your project. [Learn more](#).

**Add .gitignore**  
Choose which files not to track from a list of templates. [Learn more](#).

**Choose a license**  
A license tells others what they can and can't do with your code. [Learn more](#).

Na tela do prompt de comandos apresentada na figura 7, temos as informações para sincronizar a biblioteca *minha\_bib*, que estão no formato JSON, como nome, versão, descrição e nome do script.

**Figura 7 – Criando a biblioteca**

```
C:\minha_bib>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.
See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (minha_bib)
version: (1.0.0)
keywords:
license: (ISC)
About to write to C:\minha_bib\package.json:

{
  "name": "minha_bib",
  "version": "1.0.0",
  "description": "Minha função de teste de biblioteca",
  "main": "index.js",
  "scripts": {
    "test": "mensagem()"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/rommaia/minha_bib.git"
  },
  "author": "Romulo Maia",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/rommaia/minha_bib/issues"
  },
  "homepage": "https://github.com/rommaia/minha_bib#readme"
}

Is this OK? (yes)
C:\minha_bib>
```

No diretório, deve constar seu arquivo index.js com o seguinte exemplo de conteúdo. Dessa forma, nossa biblioteca estará criada.

```
1 exports.mensagem = function() {
2   console.log("Biblioteca funcionando");
3 }
```

Para criar o repositório, usamos o código fornecido no github.com.

```
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/rommaia/minha_bib.git
git push -u origin main
```

## 2 Componentes web

Os componentes web representam um conjunto de elementos de diversas tecnologias criadas com funcionalidades independentes do código principal de sua aplicação web, que podem ser customizados para serem usados em novas aplicações. A reutilização de códigos de componentes possibilita resolver novos problemas durante o processo de desenvolvimento. As tecnologias envolvidas nessa customização são:

- Elementos customizados: são os elementos criados pelo desenvolvedor com a reutilização de códigos.
- Shadow DOM: representa uma parte do código do elemento customizado que é encapsulado, considerado como um DOM paralelo que não é visível pelo usuário.
- Templates HTML: são pequenos códigos ou módulos reutilizados na estrutura de um elemento.

## 3 Shadow DOM

O shadow DOM representa um subconjunto de nós existentes no DOM que apresenta seu próprio conteúdo. Além disso, seu HTML, CSS e JS estão isolados de todos os outros nós do DOM, a fim de acabar com complicações como a duplicação de um id ou classe no HTML, que pode causar conflito na sua página. Facilita a criação de novos componentes, tornando-os independentes. Vamos acompanhar um exemplo no código:

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <title>Exemplo Shadow DOM</title>
```

```

5      <link rel="stylesheet" href="exemplo40_css.css">
6  </head>
7  <body>
8    <section>
9      <input type="text" />
10     <p>  </p>
11     <div id="test" class="cor_vermelho">
12       #shadow-primeiro (open)
13       <input type="text">
14     </div>
15   </section>
16   <script>
17     test = document.querySelector('#test');
18     let text = document.createElement("input");
19     text.type = 'text';
20     test.attachShadow({mode: 'open'}) .
appendChild(text);
21   </script>
22 </body>
23 </html>
```

No código, criamos um elemento na linha 12 chamado *shadow-primeiro* com um *input* na linha 13. Esse elemento é interpretado separadamente, onde temos um código de JavaScript isolado.

```

1  input[type="text"] {
2    border: 2px solid black;
3  }
4  #test
5  {
6    height: 30px;
7    width: 200px;
8  }
9  .cor_vermelho {
10    background: blue;
11 }
```

Esse código representa o arquivo “*shadow-primeiro*”, com seu código em HTML como exemplo.

## 4 Storage

O storage é uma API que representa uma nova forma de armazenamento de dados locais com o navegador web. O procedimento anterior, muito usado pelos desenvolvedores, era a utilização de cookies. No entanto, a utilização dessa API oferece mais segurança e permite armazenar maior volume de informações. Existem duas propriedades: local storage e session storage.

O local storage permite que o desenvolvedor salve, recupere ou exclua os dados localmente em um navegador web. As informações são armazenadas em pares contendo chave e valor, em que os valores são do tipo string. Os métodos principais são:

- `setItem(chave,valor)`: para gravar a chave e o valor.
- `getItem(chave)`: para ler o valor que corresponda à chave fornecida.
- `removeItem(chave)`: para excluir o valor correspondente a chave.
- `clear()`: remove todas as chaves armazenadas.

No código a seguir, na linha 17, identificamos a referência para o armazenamento da chave e o valor da informação que está sendo solicitada pelas instruções em HTML nas linhas 10 e 11, que representam os valores das variáveis *chave* e *valor*, conforme figura 8.

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4  <title>Exemplo Storage</title>
5  </head>
6  <body>
7      <P> Entre com a chave : </p>
8      <input type="text" id="chave" /><br>
9      <P> Entre com o valor : </p>
```

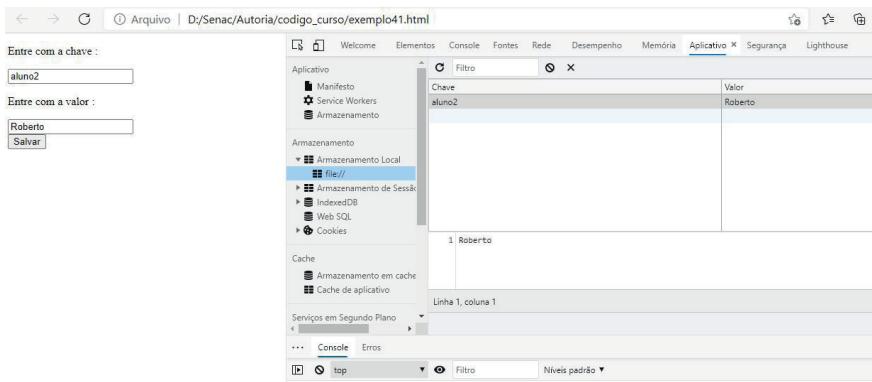
```

10      <input type="text" id="valor" /><br>
11      <input type="submit" id="Salvar" value="Salvar">
12      <script>
13          document.getElementById("Salvar") .
addEventLister("click",salve);
14          function save() {
15              var chave = document.getElementById("chave") .
value;
16              var valor = document.getElementById("valor") .
value;
17              localStorage.setItem(chave,valor);
18          }
19      </script>
20  </body>
21 </html>

```

Na figura 8, temos o resultado da execução do código, em que, ao clicar em *Salvar*, a informação é salva no local storage, na aba *application* do console do seu navegador web. Note que a informação está salva, conforme o método da linha 17 do código.

**Figura 8 – Resultado do código para salvar em local storage**



Na session storage, o funcionamento é idêntico ao do local storage, exceto pelo fato de que as informações são armazenadas enquanto sua sessão estiver aberta. Ao final da sessão, os dados são excluídos.

Outra API de armazenamento de dados do lado cliente é o indexedDB, que permite armazenar uma maior quantidade de dados e fornece mais facilidade de recuperação através de uma chave na qual os dados estão indexados.

O primeiro passo é saber se o seu navegador apresenta suporte para esse recurso. Para isso, temos o exemplo do código:

```
window.indexedDB = window.indexedDB || window.mozIndexedDB  
|| window.webkitIndexedDB ||  
window.msIndexedDB;  
if(!window.indexedDB)  
{  
    console.log(`O navegador não oferece suporte ao  
IndexedDB`);  
}
```

O segundo passo é abrir a conexão com o banco de dados para realizar os acessos. Utilizamos o método `open` para abrir essa conexão, com o banco chamado `turma`, e o segundo parâmetro representa a versão do banco. O código a seguir apresenta a linha de conexão com o banco de armazenamento do navegador com a execução da instrução `open`:

```
var request = window.indexedDB.open("turma", 1);
```

O código da função a seguir apresenta a variável `request`, que estabelece uma transação de dados para ler ou gravar os dados do “aluno”, que estão no formato JSON. As instruções `onsuccess` e `onerror` representam o retorno da transação, que pode ter sido bem-sucedida ou não.

```
function incluir() {  
    var request = db.transaction(["aluno"], "readwrite")  
        .objectStore("aluno")  
        .add({ id: "01", name: "Roberto", age: 19});  
  
    request.onsuccess = function(event) {  
        alert("Informações gravadas");  
    };  
  
    request.onerror = function(event) {  
        alert("As informações não foram salvas ");  
    }  
}
```

## 5 Websockets e webworkers

O API Websockets é uma tecnologia que abre uma sessão de comunicação do lado cliente com o lado servidor, viabilizando o envio e o recebimento de mensagens. Ele estabelece uma conexão persistente para o envio de mensagens.

O primeiro passo é estabelecer a comunicação com o servidor com a instrução:

```
var conexao = new WebSocket('ws://www.senac.exemplo.  
com.br/socketServidor', 'protocolOne');
```

Essa instrução será inserida em seu código conforme o objetivo de sua aplicação.

Do lado cliente, a resposta será compreendida por um `conexao.readyState`, que apresentará um estado de `open`.

Estabelecida a conexão, podemos utilizar os métodos:

- `Send`: para o envio de mensagens.

- Onmessage: para receber a resposta do servidor.

A comunicação pode ser realizada com qualquer domínio, sendo mais utilizada com servidores confiáveis e apresentando compatibilidade com servidores proxy com conexões HTTP.

Já os webworkers são mecanismos da linguagem JavaScript que possibilitam que um script seja executado em um plano diferente da sua aplicação web. São utilizados para executar tarefas demoradas sem provocar o bloqueio da aplicação do lado cliente. Os web workers podem ser do tipo dedicado, representado pelo objeto *DedicatedWorkerGlobalScope*, ou do tipo compartilhado, usando *SharedWorkerGlobalScope*.

Em nosso exemplo, criamos um arquivo chamado *contador.js*:

```
1 var i = 0;
2 function timedCount() {
3     i = i + 1;
4     postMessage(i);
5     setTimeout("timedCount()", 500);
6 }
7 timedCount();
```

A comunicação entre sua aplicação e o arquivo worker é realizada pelo método *postMessage*, na linha 4 do código.

```
1 <!DOCTYPE html>
2 <html>
3 <body>
4 <p>Contador Numero : <output id="resultado"></
output></p>
5 <button onclick="iniciar()">Iniciar</button>
6 <button onclick="parar()">Parar</button>
7 <script>
8 var w;
9 function iniciar() {
10     if (typeof(Worker) !== "undefined") {
11         if (typeof(w) == "undefined") {
12             w = new Worker('contador.js');
```

```

13         }
14         w.onmessage = function(event) {
15             document.getElementById("resultado").
16             innerHTML = event.data;
17         };
18     } else {
19         document.getElementById("resultado").innerHTML
20         = "Seu navegador não oferece suporte";
21     }
22     function parar() {
23         w.terminate();
24         w = undefined;
25     }
26 </script>
27 </body>
28 </html>

```

O *postMessage* é manipulado por *onmessage*, linha 14. Na aplicação principal, um objeto *worker* é criado na linha 12, o arquivo *contador.js* será encontrado e o navegador gerará, em segundo plano, uma nova sequência do worker.

## 6 Recursos multimídia

Já aprendemos muitos recursos trabalhando com textos e figuras para o desenvolvimento de uma página web. No entanto, também é possível incluir animações em sua página e é importante explorar alguns recursos que podem ser utilizados com o JavaScript, como:

- SVG (Gráficos Vetoriais Escaláveis): utilizado para gerar gráficos e baseado em XML, desenvolvido com o HTML, CSS e JavaScript.
- Canvas 2D: CanvasRenderingContext2D, para desenhos em duas dimensões, textos, imagens e outros objetos.
- WebGL: permite utilizar um API baseado em OpenGL para renderização de figuras em três dimensões em um canvas no código HTML do seu navegador web.

- WebVR: oferece recursos para desenvolver realidade virtual, facilitando a interação do usuário com sua aplicação web.
- Webaudio: oferece recursos de áudio em sua aplicação web, permitindo o roteamento modular.
- WebRTC (Web Real-Time Communications): oferece recursos para que sua aplicação web capture e transmita áudios e vídeos. Permite o compartilhamento e a realização de teleconferência.

## Considerações finais

A utilização de pacotes de bibliotecas disponíveis para a linguagem JavaScript oferece recursos que facilitam a vida do desenvolvedor. Outro ponto fundamental é a oportunidade de criar componentes com a reutilização de códigos já existentes. Com isso, a interface de sua aplicação web pode ser inovadora e criativa. O armazenamento de informações locais contribui com a interface da aplicação, facilitando a interação e a navegação e oferecendo ao usuário velocidade e segurança. Portanto, neste capítulo, percebemos que a linguagem JavaScript oferece ao desenvolvedor recursos que podem ser aplicados ao projeto, tornando-o rápido, seguro e eficiente.

## Referências

FLANAGAN, David. **JavaScript**: o guia definitivo. Porto Alegre: Bookman, 2012.

## Sobre o autor

**Rômulo Francisco de Souza Maia** é mestre em Ciência da Computação pela Universidade Federal de Minas Gerais (UFMG). Possui especialização em Ensino Superior (1991) pela Universidade da Amazônia e aperfeiçoamento em Análise de Sistemas pelo Centro de Serviços Educacionais do Pará (Cesep). Link para o Currículo Lattes: <http://lattes.cnpq.br/8792772462731111>.