



# Front-end PARA INICIANTES



# LuizTools

# SUMÁRIO

<b>SOBRE O AUTOR .....</b>	<b>5</b>
<b>ANTES DE COMEÇAR .....</b>	<b>9</b>
Para quem é este livro .....	10
<b>INTRODUÇÃO À PROGRAMAÇÃO WEB .....</b>	<b>11</b>
Um breve histórico .....	13
Mantenedores .....	15
Como funciona um sistema web? .....	15
Ambiente de Programação .....	16
Configurando o ambiente .....	17
<b>HTML .....</b>	<b>21</b>
Introdução ao HTML .....	22
As tags HTML .....	25
A tag HEAD .....	26
A tag BODY .....	28
As tags H1, H2, ... .....	28
As tags P, BR e HR .....	29
As tags B, STRONG, I e U .....	32
As tags UL, OL e LI .....	33
A tag A .....	35
A tag IMG .....	37
As tags DIV e SPAN .....	39
As tags TABLE, TR e TD .....	40
Formulários HTML .....	42
A tag FORM .....	42
A tag LABEL .....	43
A tag INPUT .....	44
A tag TEXTAREA .....	50

As tags SELECT e OPTION .....	50
Outras tags .....	51
<b>JAVASCRIPT BÁSICO .....</b>	<b>55</b>
A tag SCRIPT .....	57
Declaração de Variáveis .....	59
Tipos de dados .....	64
Comentários .....	65
Operadores .....	66
Operadores Aritméticos .....	66
Operadores Relacionais .....	67
Operadores Lógicos .....	69
Operador de Atribuição .....	70
Functions .....	71
O tipo String .....	73
Estruturas de Controle de Fluxo .....	76
Estruturas de repetição .....	77
Estruturas de desvio de fluxo .....	87
Arrays .....	97
O tipo Object .....	103
JavaScript Client-Side .....	106
Document Object Model .....	107
Como usar .....	108
O objeto window .....	109
Popups .....	110
O objeto document .....	110
Eventos JavaScript .....	110
onclick .....	111
onchange .....	112
onmouseenter e onmouseleave .....	113
onfocus e onblur .....	113

onkeypress, onkeydown e onkeyup .....	114
Calculadora em HTML+JS .....	115
Manipulando o DOM .....	122
Seletores .....	122
Propriedades de Elemento .....	124
Funções de Elemento .....	125
Exercitando .....	127
Ajax .....	136
Ajax com fetch .....	137
Opções do Fetch .....	139
<b>CSS .....</b>	<b>141</b>
CSS Inline .....	142
Estilos de texto .....	143
Classes de Estilos em CSS Internal .....	144
Dicas e Truques .....	147
Arquivos CSS Externos .....	148
Estilos de Caixas .....	149
CSS e JavaScript .....	153
Exercitando .....	154
<b>SEGUINDO EM FRENTE .....</b>	<b>164</b>

# **SOBRE O AUTOR**

---

Luiz Fernando Duarte Júnior é Bacharel em Ciência da Computação pela Universidade Luterana do Brasil (ULBRA, 2010) e Especialista em Desenvolvimento de Aplicações para Dispositivos Móveis pela Universidade do Vale do Rio dos Sinos (UNISINOS, 2013).

Carrega ainda um diploma de Reparador de Equipamentos Eletrônicos (SENAI, 2005), nove certificações em Métodos Ágeis de desenvolvimento de software por diferentes certificadoras (PSM-I, PSD-I, PACC-AIB, IPOF, ISMF, IKMF, CLF, DEPC, SFPC) e três certificações de coach profissional pelo IBC (Professional & Self Coach, Life Coach e Leader Coach).

Atuando na área de TI desde 2006, na maior parte do tempo como desenvolvedor, é apaixonado por desenvolvimento de software desde que teve o primeiro contato com a linguagem Assembly no curso de eletrônica. De lá para cá teve oportunidade de utilizar diferentes linguagens em diferentes sistemas, mas principalmente com tecnologias web, incluindo ASP.NET, JSP e, nos últimos tempos, Node.js.

### **Foi amor à primeira vista e a paixão continua a crescer!**

Trabalhando com Node.js desenvolveu diversos projetos para empresas de todos os tamanhos, desde grandes empresas como Softplan até startups como Busca Acelerada e Só Famosos, além de ministrar palestras e cursos de Node.js para alunos do curso superior de várias universidades e eventos de tecnologia.

Um grande entusiasta da plataforma, espera que com esse livro possa ajudar ainda mais pessoas a aprenderem a desenvolver softwares com Node.js e aumentar a competitividade das empresas brasileiras e a empregabilidade dos profissionais de TI.

Além de viciado em desenvolvimento, como consultor em sua própria empresa, a DLZ Tecnologia e é autor do blog [www.luiztools.com.br](http://www.luiztools.com.br), onde escreve semanalmente sobre métodos ágeis e desenvolvimento de software, bem como mantenedor do canal [LuizTools](https://www.youtube.com/user/LuizTools), com o mesmo propósito.

Entre em contato, o autor está sempre disposto a ouvir e ajudar seus leitores.

#### **SOBRE O AUTOR**



# MEUS CURSOS

## Curso online NODE.JS e MONGODB

[SAIBA MAIS...](#)

## Curso online Scrum e métodos Ágeis

[SAIBA MAIS...](#)

## Curso online Jira

[SAIBA MAIS...](#)

## Curso online Web Full Stack JavaScript

[SAIBA MAIS...](#)

## Curso online React Native com Firebase

[SAIBA MAIS...](#)

Conheça todos os meus cursos

# MEUS LIVROS



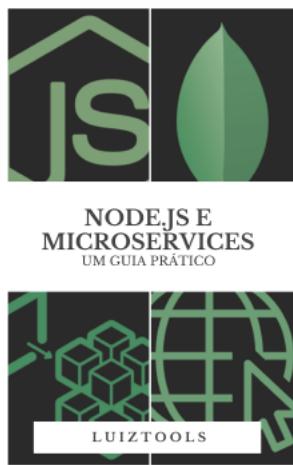
Programação  
Web com Node.js

[SAIBA MAIS...](#)



Programação  
Web com Node.js

[SAIBA MAIS...](#)



Node.js e  
Microservices

[SAIBA MAIS...](#)



MongoDB  
para Iniciantes

[SAIBA MAIS...](#)



Scrum e  
Métodos Ágeis

[SAIBA MAIS...](#)



Agile Coaching

[SAIBA MAIS...](#)



Criando apps  
para empresas  
com Android

[SAIBA MAIS...](#)



Java para  
iniciantes

[SAIBA MAIS...](#)

## Conheça todos os meus livros

Aproveita e segue nas redes sociais:



# ANTES DE COMEÇAR

---

“

Without requirements and design, programming is  
the art of adding bugs to an empty text file.

- Louis Srygley

”

Antes de começarmos, é bom você ler esta seção para evitar surpresas e até para saber se este livro é para você.

## Para quem é este livro

Primeiramente, este ebook vai lhe ensinar as bases fundamentais para construção de páginas web: o trio HTML+CSS+JS, mas não vai lhe ensinar lógica básica e algoritmos, nem vai entrar em aspectos mais avançados de frontend como frameworks e bibliotecas SPA.

Aliás, frontend (ou front-end) é a interface para o usuário, a cara da aplicação ou do site, a camada que executa e aparece no navegador do cliente, enquanto backend é o funcionamento no servidor, no banco de dados.

Segundo, este livro exige que você já tenha conhecimento técnico prévio sobre computadores, que ao menos saiba mexer em um e que preferencialmente possua um.

Parto do pressuposto que você é ou já foi um estudante de Técnico em informática, Ciência da Computação, Sistemas de Informação, Análise e Desenvolvimento de Sistemas ou algum curso semelhante. Usarei diversos termos técnicos ao longo do livro que são comumente aprendidos nestes cursos e que não tenho o intuito de explicar aqui.

Ao término deste ebook você estará apto a atuar na construção ou manutenção de páginas web simples, usando o mínimo e mais básico que há de front-end, independente do caminho que seguir depois: HTML, CSS e JavaScript.

*Quer fazer um curso online de Node.js e MySQL com o autor deste livro?*

*Acesse <https://www.luiztools.com.br/curso-fullstack>*

---

# INTRODUÇÃO À PROGRAMAÇÃO WEB

1

“

Good software, like wine, takes time.

- Joel Spolsky

”

A Internet é uma rede global de computadores interligados que utilizam um conjunto de protocolos padrões para servir vários bilhões de usuários no mundo inteiro. É uma rede de várias outras redes, que consiste de milhões de empresas privadas, públicas, acadêmicas e de governo, com alcance local e global e que está ligada por uma ampla variedade de tecnologias de rede eletrônica, sem fio e ópticas.

A Internet é muito mais do que o que vemos em nossos navegadores (os chamados browsers). Esta é a World Wide Web (ou simplesmente web hoje em dia), e é apenas uma das muitas faces da Internet. Dentro da Internet temos redes ponto-a-ponto, infraestrutura de apoio à e-mails e muito mais do que apenas os sites públicos que costumamos acessar todos os dias.

Programar para a Internet é um desafio muito maior do que criar softwares que rodam apenas em uma máquina local ou até mesmo em uma rede privada. Programar para Internet é ter de lidar com dispositivos heterogêneos, larguras de banda variadas, distâncias inimagináveis e diversas outras limitações. Mas ao mesmo tempo programar para a Internet lhe dá um alcance, um poder, muito maior do que visto nas disciplinas mais tradicionais de programação.

Em linhas gerais, esta disciplina irá trabalhar as competências e tecnologias necessárias para se programar sistemas para Internet, mais especificamente para a web, ou Internet comercial, essa que usamos tradicionalmente em nossos computadores e celulares, mas focando mais nos primeiros.

A tabela comparativa abaixo cita algumas vantagens e desvantagens em relação à programação tradicional para uma máquina desktop, que foi vista em disciplinas anteriores. Estes são fatos gerais, embora existam casos em que podemos criar sistemas para a Internet arrastando e soltando componentes, ou que podemos publicar um software desktop de maneira remota. É apenas para termos alguma ideia das diferenças gerais entre eles.

Programação Desktop	Programação Web
Construção de interfaces arrastando e soltando	Linguagem de marcação específica para construir interfaces
Curva de aprendizagem menor	Mais coisas para aprender
Publicação local, tem de estar em frente à máquina do cliente	Publicação remota, feita através da própria Internet
Pouca preocupação com segurança no sistema (geralmente AD/LDAP já dão conta de tudo)	Segurança é sempre preocupante pois o sistema está aberto ao mundo
Consumo de recursos do desktop não aumenta conforme o número de usuários	Consumo de recursos do servidor aumenta conforme o número de usuários
Somente acessa onde está instalado, máquina local ou rede empresarial	Acessa de qualquer navegador de qualquer dispositivo

## Um breve histórico

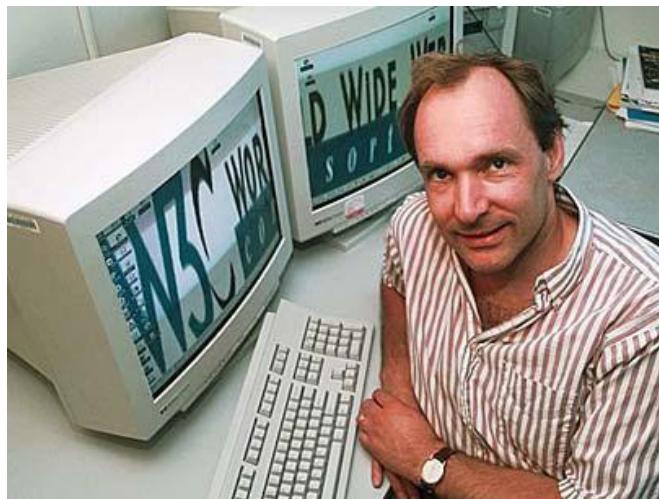
O embrião da Internet nasceu de motivos militares para comunicação entre as tropas, mas hoje seu uso é global e nada se assemelha ao que era no passado. A seguir, uma breve timeline dos fatos mais marcantes da história da Internet e da web para nós:

**Década de 1960**, Estados Unidos encomenda pesquisa para criar tecnologia que permitisse conectar seus computadores visando comunicação eficiente, rápida e barata entre longas distâncias.

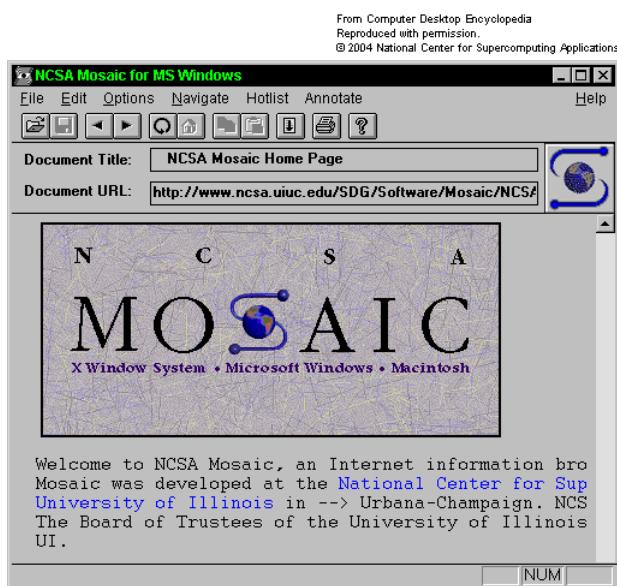
**Década de 1980**, universidades de diversos países que possuíam pesquisas com redes de computadores passam a se comunicar corriqueiramente usando terminais de Internet, apenas em modo texto (console). Aqui passou-se a investir muito dinheiro na aquisição de backbones para comunicação entre as redes de computadores internacionais.

**1989 a 1991**, Tim Berners-Lee criou um projeto hipertexto que permitia documentos com informações “linkasse” uns aos outros através da Internet, criando o embrião do que mais tarde se tornaria a World Wide

Web, ou WWW. Berners-Lee ainda montou a W3C em 1994, consórcio de empresas e entidades que regulamentam os padrões da web mundial, principalmente no quesito hipertexto.



**1993:** é lançado o Mosaic, primeiro navegador web que daria origem ao famoso Netscape Navigator, e que mais tarde licenciado pela Microsoft dando origem ao famoso Internet Explorer, que viria a “matar” seu “irmão”.



**Década de 1990**, a venda comercial em larga escala de conexões à Internet torna possível a não-militares e não-acadêmicos se conectarem à rede mundial à partir de suas casas. A partir daqui a Internet passou a afetar a cultura e comércio dos países de maneira drástica.

**Anos 2000**, o advento da banda-larga torna possível a transmissão de grandes volumes de dados com uma velocidade sem precedência,

permitindo transferências de grandes arquivos e streaming. Com isso as mídias tradicionais começam um processo drástico de reformulação.

**2004:** surge o Mozilla Firefox ameaçando o reinado do IE, criado nas férias de verão por um adolescente americano que se juntou a outros colegas do projeto Mozilla, que inclusive era financiado pela Netscape.

**2008:** surge o Google Chrome, browser que viria a tirar o reinado do IE e que hoje representa mais de 50% dos navegadores conectados à Internet mundialmente, seguido pelo IE com 22% e pelo Firefox com 19%.

**2014:** surge o Marco Civil da Internet no Brasil, onde pela primeira vez na história do país passam a existir leis específicas de atuação no ambiente digital da Internet, cuja legislação até então era vaga e praticamente inexistente (geralmente interpretavam-se disputas legais envolvendo a Internet com base em mídias como TV e rádio).

## MANTENEDORES

Cada país possui suas próprias regras com relação ao uso e distribuição da Internet, ou seja, a regulamentação da Internet cabe ao governo de cada país. Entretanto, toda a parte de endereçamento da Internet compete ao ICANN (nomes de domínio e DNS) e a parte técnica ao IETF (protocolos e tecnologias), entidades globais sem fins lucrativos que visam a padronização da Internet mundial.

Como falado anteriormente, também temos a W3C, que define os padrões técnicos da web.

## COMO FUNCIONA UM SISTEMA WEB?

Em linhas gerais temos o seguinte fluxo do lado do usuário:

- O usuário abre seu computador com um navegador de Internet e uma conexão com um provedor de Internet (geralmente uma linha telefônica).
- O usuário digita uma URL contendo o domínio de Internet (também chamado de domínio público ou “endereço do site”) do sistema.
- Com este domínio em mãos, seu computador envia uma requisição

para o servidor DNS do seu provedor de Internet (ISP) visando que ele lhe diga o endereço IP do servidor onde está aquele conteúdo. Os domínios públicos são para facilitar e organizar o acesso à Internet, mas na verdade, precisamos do endereço IP para acessar qualquer coisa na Internet.

- Caso o ISP já saiba qual o IP associado àquele endereço ele lhe retorna diretamente, caso contrário terá de perguntar a um servidor de DNS “mais alto” na hierarquia, sendo que as extensões dos domínios orientam os servidores DNS neste sentido.
- Com o IP em mãos, seu computador vai enviar uma requisição HTTP, o protocolo de transferência de hipertexto dizendo, entre outras informações, o endereço IP do servidor onde está o site e o arquivo que deseja acessar naquele servidor (geralmente uma tela do sistema).
- Quando a requisição chega ao servidor será avaliada o tipo de requisição realizado, pois podemos querer ler um arquivo, escrever informações, excluir etc. Além disso, dependendo do arquivo que estamos requisitando, pode ser necessário a presença de um manipulador (handler) específico para tratar aquela requisição, como é o caso de páginas Java e PHP, por exemplo. Este tratamento é feito pelos servidores web, como Apache e IIS.
- Após decidir o que fazer com a requisição, o servidor retorna uma resposta para o usuário. Aqui tem um ponto importante pois o navegador do usuário somente entende respostas HTTP com conteúdo em HTML (que veremos mais à frente). Ou seja, o servidor web terá que traduzir a resposta do sistema, escrito em Java por exemplo, para essa única linguagem que o browser entende.
- Aqui encerra-se o ciclo básico e genérico, com o usuário recebendo uma resposta em seu navegador após ter realizado uma ação no sistema web.

## Ambiente de Programação

Falando de tecnologias, a linguagem de marcação básica para criação de páginas web é o HTML, você precisará conhecê-la para construir as interfaces gráficas do seu sistema.

Visando tornar o HTML mais dinâmico você também precisará conhecer JavaScript, que já falamos bastante aqui anteriormente. Inicialmente a função do JavaScript era somente tornar o HTML dinâmico, mas tarde com o advento do Node.js, ele passou a desempenhar mais funções, mas não vamos falar de Node.js neste livro, pois ele é uma tecnologia de backend.

Visando tornar as interfaces HTML mais atraentes, você deverá conhecer CSS, um conjunto de estilos personalizáveis aplicados sobre o HTML.

Com esses três itens você tem o que chamamos de front-end ou apresentação do sistema web. Entretanto, com apenas estes três itens você não conseguirá fazer muito mais do que um site, embora caso você tenha acesso a web APIs, seja possível levar o seu front-end a outro nível. Mas aí começa a complicar a história, e ainda não é hora para isso...

## CONFIGURANDO O AMBIENTE

Para que seja possível criar páginas web usando HTML, CSS e JavaScript não é necessário nenhuma configuração “especial”. Mas como tenho certeza que alguns leitores podem estar começando exatamente agora suas carreiras, sou então migrando de outra área completamente alienígena ao desenvolvimento web, vale ressaltar alguns pontos.

Portanto, vamos começar do princípio, das ferramentas que eu recomendo que você possua para seguir com este ebook.

### Visual Studio Code

Para escrever TODOS os exemplos de código deste ebook você não precisa mais do que um bloco de notas, mas é especialmente mais produtivo usar um editor de código, como o Visual Studio Code, da Microsoft.

Esta não é a única opção disponível, mas é uma opção bem interessante e é a que uso, uma vez que reduz consideravelmente a curva de aprendizado, os erros cometidos durante o aprendizado e possui ferramentas de depuração muito boas, além de suporte a Git e linha de comando integrada. Apesar de ser desenvolvido pela Microsoft (uma

empresa que desenvolve tradicionalmente softwares pagos), é um projeto gratuito, de código-aberto, multi-plataforma e com extensões para diversas linguagens e plataformas, como HTML, CSS e JavaScript, as estrelas deste ebook.

E diferente da sua contraparte mais “parruda”, o Visual Studio original, ele é bem leve e pequeno.

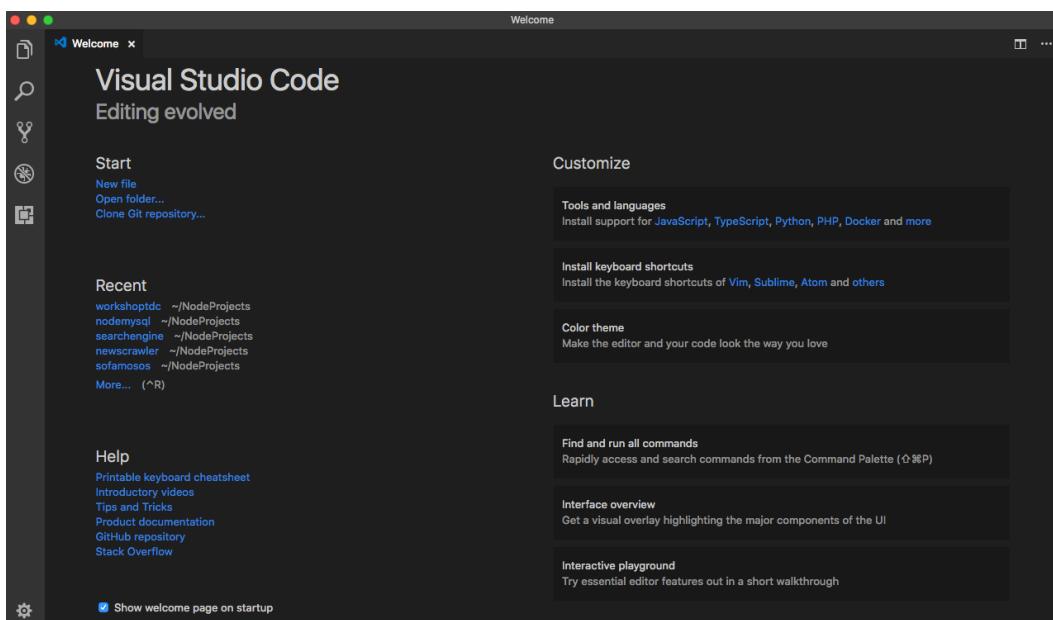
Outras excelentes ferramentas incluem Notepad++, Atom e o Sublime, mas o Visual Studio Code deixa eles para trás. Minha opinião.

Para baixar e instalar o Visual Studio Code, acesse o seguinte link, no site oficial da ferramenta:

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

Você notará um botão grande e verde para baixar a ferramenta para o seu sistema operacional. Apenas baixe e instale, não há qualquer preocupação adicional.

Após a instalação, mande executar a ferramenta Visual Studio Code e você verá a tela de boas vindas, que deve se parecer com essa abaixo, dependendo da versão mais atual da ferramenta. Chamamos esta tela de Boas Vindas (Welcome Screen).



Para fazer um rápido teste com esta ferramenta, vá no menu File > New File, escreva qualquer coisa dentro dele e depois vá no menu File > Save e dê o nome de index.html para o arquivo que será salvo.

Pronto, este é o modus operandi que você vai ter neste livro, o tempo todo.

### **Google Chrome**

Para testar as páginas web que vamos criar neste ebook, qualquer navegador serve. No entanto, existem navegadores que seguem mais os padrões da Internet e outros menos. O Google Chrome é um dos melhores neste quesito.

O Google Chrome é um navegador de internet, desenvolvido pela companhia Google com visual minimalista e compilado com base em componentes de código licenciado e sua estrutura de desenvolvimento de aplicações (framework).

Em 2 de setembro de 2008 foi lançado a primeira versão ao mercado, sendo uma versão beta e em 11 de dezembro de 2008 foi lançada a primeira versão estável ao público em geral. O navegador atualmente está disponível, em mais de 51 idiomas, para as plataformas Windows, Mac OS X, Android, iOS, Ubuntu, Debian, Fedora e openSUSE.

Atualmente, o Chrome é o navegador mais usado no mundo, com 49,18% dos usuários de Desktop, contra 22,62% do Internet Explorer e 19,25% do Mozilla Firefox, segundo a StatCounter. Além de desenvolver o Google Chrome, o Google ainda patrocina o Mozilla Firefox, um navegador desenvolvido pela Fundação Mozilla.

Durante muitos exemplos deste livro será necessária a utilização de um navegador de Internet. Todos os exemplos foram criados e testados usando o navegador Google Chrome, na versão mais recente disponível à época que era a versão 85. Caso não possuam o Google Chrome na sua máquina, baixe a versão mais recente no site oficial antes de avançar no livro:

<https://www.google.com.br/chrome/browser/desktop/index.html>

Além de um excelente navegador, o Google Chrome ainda possui uma série de ferramentas para desenvolvedor que são muito úteis como um

inspetor de código HTML da página, um depurador de JavaScript online, métricas de performance da página e muito mais.

Para ver se está tudo ok para nossos exercícios, experimente navegar no seu computador até chegar no arquivo index.html que você salvou no teste do Visual Studio Code. Dê um duplo clique no arquivo e ele vai abrir no Google Chrome, apresentando o texto que você escreveu.

*Quer fazer um curso online de Desenvolvimento Web FullStack JS com o autor deste ebook?  
Acesse <https://www.luiztools.com.br/curso-fullstack>*

---

# HTML

2

“

*A language that doesn't affect the way you think about  
programming is not worth knowing.*

- Alan J. Perlis

”

Em Ciência da Computação, front-end e back-end são termos generalizados que se referem às etapas inicial e final de um processo. O front-end é responsável por coletar a entrada do usuário em várias formas e processá-la para adequá-la a uma especificação em que o back-end possa utilizar.

Resumindo, o front-end é a camada de apresentação, que o usuário consegue ver e interage, muitas vezes sendo considerada a principal camada para ele que não costuma distinguir o que vê do que é o “sistema web de verdade”. Cabe ao profissional de front-end projetar a experiência do usuário, a identidade visual do sistema e muitas vezes acaba fazendo papel de web-designer e de programador ao mesmo tempo.

Este não é um ebook de web design, de user experience, mas existem uma série de conceitos básicos que todo programador web deve saber (mesmo que ele se especialize no back-end mais tarde), e é isso que exploraremos aqui.

Quando falamos de programação web, existem três principais tecnologias que são utilizadas e que devemos aprender, acima de todas as outras: HTML, CSS e JavaScript. Falamos de JavaScript de maneira genérica anteriormente e voltaremos a abordar ele mais pra frente. CSS também será visto mais adiante, enquanto neste capítulo falaremos de HTML.

## Introdução ao HTML

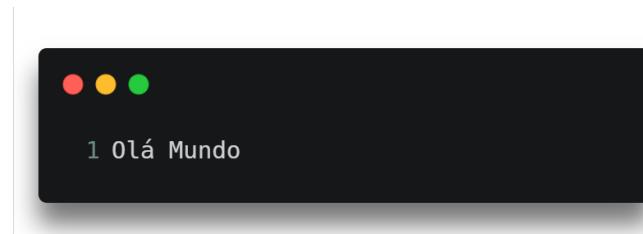
HTML (abreviação para a expressão inglesa HyperText Markup Language, que significa Linguagem de Marcação de Hipertexto) é uma linguagem de marcação utilizada para produzir páginas na Web. Documentos HTML podem ser interpretados por navegadores, como o Google Chrome, que pedi que você instalasse no capítulo 2.

Documentos HTML podem ser qualquer coisa que queiramos que seja exibida no browser de nossos usuários: a timeline do Facebook, a tela de pesquisa do Google ou a home-page da Microsoft. Todos são documentos HTML. Com exceção de telas criadas através de plug-ins, como o Flash (obsoleto), os documentos HTML são a cara da web e é de suma importância que entendamos a sua estrutura básica antes de começar a programar pra ela com Node.js.

Todo documento HTML é um arquivo de texto com a extensão .html (ou .htm, tanto faz). Pode ser criado e editado em qualquer editor de texto, desde que salvo com a extensão correta. Mas quando aberto no navegador, é que vemos o que o documento representa de verdade.

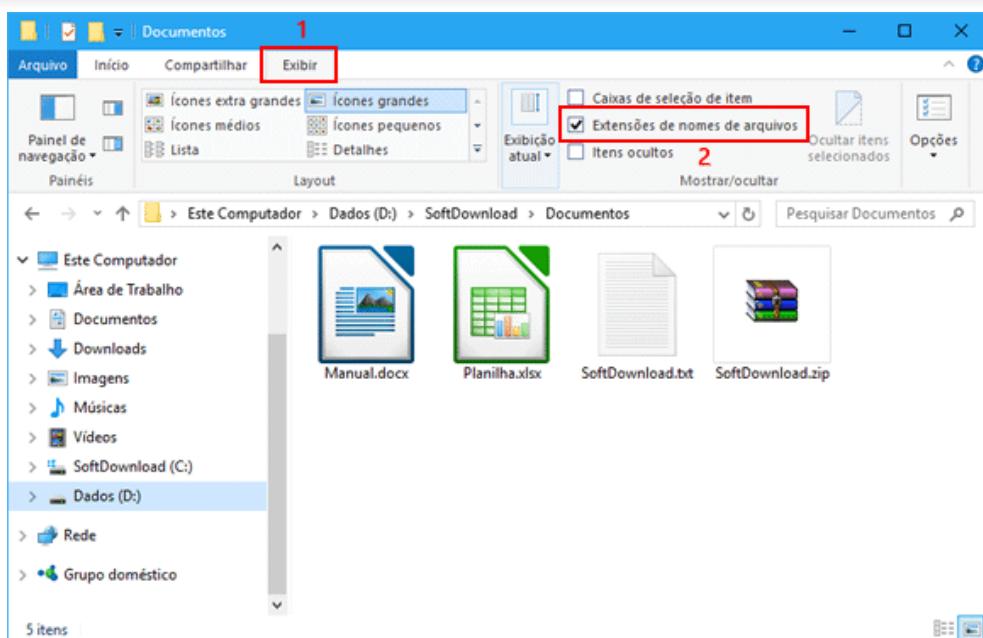
Isso porque todo documento HTML é composto de informações e de meta-informações. Estas últimas são organizadas em tags (rótulos) e cada tag possui uma propriedade especial no documento, indo desde alterar o tamanho ou cor de algo, até o seu posicionamento e comportamento.

Vamos começar simples, abra um editor de texto qualquer (podendo inclusive ser o VS Code) e digite o seguinte texto:



Depois salve com o nome index e a extensão '.html'. Agora dê um duplo-clique neste arquivo ou mande abrir com o Google Chrome. Você verá o seu arquivo de texto no browser.

**DICA:** No Windows, caso não consiga alterar a extensão (o notepad teima em salvar como .txt por padrão) pode ser necessário alterar a opção de 'Extensões de nomes de arquivos' no menu Exibir, do Windows Explorer como abaixo.



Note que não haverá qualquer diferença em relação ao texto originalmente escrito, talvez apenas a tipografia ligeiramente diferente. Isso porque colocamos apenas informações, mas nenhuma meta-informação no arquivo, que a partir de agora chamaremos simplesmente de tags.

Cada tag é escrita usando colchetes angulares (<>), também chamados de símbolo “menor-maior”. Dentro dos símbolos de menor e maior temos o nome da tag. Ao escrever uma tag no documento HTML, temos de seguir o seguinte formato abre-e-fecha:



Ou nesse outro formato auto-fecha:



Note que a barra ‘/’ serve para demonstrar o fechamento da tag. No primeiro formato (abre-e-fecha) temos uma tag de container, que pode ter conteúdo dentro do seu interior. Este conteúdo pode ser um texto, ou uma outra tag por exemplo.

No segundo formato (auto-fecha), a tag não permite conteúdo, sendo apenas uma marcação isolada, podendo definir que naquele local do documento teremos a renderização de algo especial e não apenas informação pura.

Veremos tags de ambos tipos com detalhes, mas vale salientar desde já que toda tag que abre, tem de fechar. Ou ela fecha com uma tag de mesmo nome precedida de barra ‘/’ (caso abre-fecha) ou ela fecha em si mesma, com uma barra ‘/’ antes de ‘>’

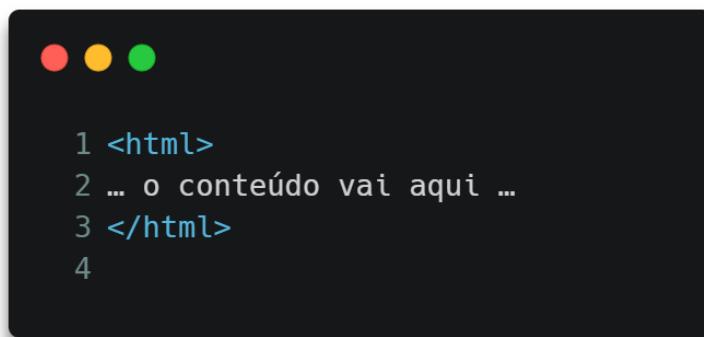
E por fim, algumas tags possuem atributos, ou seja, características personalizáveis em seu interior.

**Atenção:** por questões de implementação de cada um dos browsers de Internet, algumas tags podem ter seu efeito visual ligeiramente diferente, o que é normal. Além disso, alguns browsers são mais displicentes com relação à formação do documento HTML, ou seja, alguns aceitam formatações errôneas como tags que não fecham ou tags dentro de outras tags que não são containers. Isso não deve ser usado como uma desculpa para escrever HTML errado. Sempre que possível, siga os padrões de formatação de documentos HTML para garantir o máximo de compatibilidade entre os browsers. Todos estes padrões podem ser encontrados no site da W3C.

## As tags HTML

A primeira e mais importante tag que você deve conhecer é a tag HTML. Ela inicia e termina o documento HTML, ou seja, é a primeira e a última presente em um documento, uma tag contâiner que engloba todas as demais tags do documento, como segue:

Código 2.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
● ● ●
1 <html>
2 ... o conteúdo vai aqui ...
3 </html>
4
```

Sendo uma tag contâiner, ela permite dentro dela tanto conteúdo textual quanto outras tags, no entanto, como manda o padrão W3C, a tag HTML deve conter em seu interior essencialmente apenas outras duas tags: head e body, como veremos a seguir.

Além da tag HTML, sugere-se atualmente começar seu documento com a tag DOCTYPE, indicando que usaremos HTML5, a especificação mais recente da linguagem na data de escrita deste livro.

Código 2.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3 ... o conteúdo vai aqui ...
4 </html>
```

## A TAG HEAD

Todo documento HTML possui um cabeçalho, com informações e propriedades gerais do documento e essa é a função da tag contâiner HEAD. Ela deve vir imediatamente após a tag <html> e deve conter em seu interior apenas algumas tags especiais mas nenhum conteúdo solto. Como segue:

**Nota:** tabulações e quebras de linha não surtem qualquer efeito na apresentação do seu documento HTML no browser do usuário, então use-as para se organizar melhor, como faço abaixo quebrando a linha entre as tags e tabulando as que estão dentro de outras.

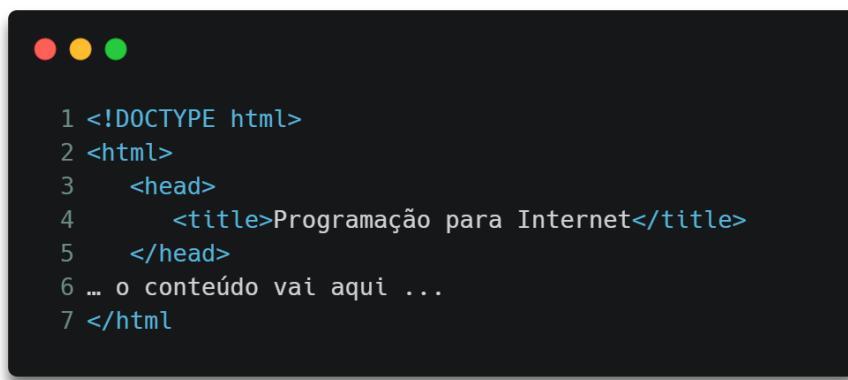
Código 2.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head> ... cabeçalho aqui ...
4   </head>
5   ... conteúdo aqui ...
6 </html>
```

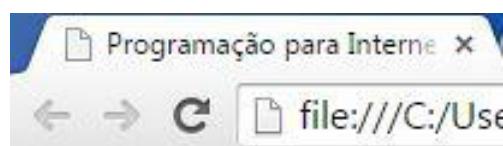
Dentro da tag HEAD podemos colocar, por exemplo, as tags TITLE, SCRIPT e STYLE além de tags para ajudar os mecanismos de busca, chamadas de meta-tags.

A tag TITLE denota o título deste documento e aparece na aba do seu navegador, indicando o que o usuário está vendo no momento. Já as tags SCRIPT e STYLE serão vistas mais pra frente para inserir Javascript e CSS no documento, respectivamente. A tag TITLE é um contâiner que permite colocar uma linha de texto entre seu abre-efecha com o título da página, veja abaixo e compare com o resultado:

Código 2.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5   </head>
6 ... o conteúdo vai aqui ...
7 </html>
```



... conteúdo vai aqui ...

Note na primeira imagem o código HTML e na segunda imagem o resultado no navegador. Agora entendeu a diferença entre informação e meta-information (tag)? A tag HEAD define que o conteúdo em seu interior é propriedade do documento, como seu título, que é colocado na aba do navegador. Já o que está fora da HEAD é interpretado como texto plano e é impresso diretamente na página.

**Atenção:** caso tenha problemas com acentos na sua página você pode utilizar uma tag no HEAD para dizer ao navegador qual a codificação de caracteres que está utilizando, da seguinte forma: <meta charset="UTF-8" /> neste caso, UTF-8. Outra codificação é ISO-8859-1.

## A TAG BODY

Assim como temos um cabeçalho no documento HTML, também precisamos ter um corpo, função da tag BODY, que deve vir logo após a tag HEAD, mas fechar antes da tag fecha-HTML como segue:

Código 2.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8 ... o conteúdo vai aqui ...
9   </body>
10 </html>
```

A tag BODY é uma tag container e aceita a maior parte das tags HTML existentes, pois ela representa o documento em si, o que o usuário vê e interage na tela. É dentro da tag body que construiremos as interfaces de nossos sistemas web.

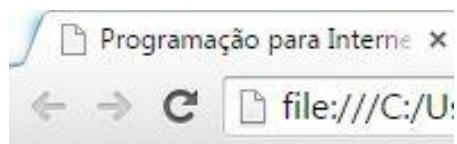
A tag BODY possui alguns atributos que permitem sua personalização. É incentivado que procure conhecer estes atributos, embora desnecessário no momento para o aprendizado de desenvolvimento de sistemas para Internet. Entenda apenas que as tags apresentadas a seguir devem ser inseridas dentro do BODY do documento HTML, mesmo que não seja citado isso explicitamente.

## AS TAGS H1, H2, ...

Um conjunto de tags iniciados com H são as tags de títulos e subtítulos de textos, containers que recebem texto plano em seu interior. Não confunda com a tag TITLE, que não aparece visivelmente na página web, as tags H aparecem sim e de forma bem chamativa, sendo a H1 a maior delas e quanto maior o número depois do H, menor seu tamanho, como mostra a imagem abaixo:

Código 2.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <h1>Teste</h1>
9     <h2>Teste</h2>
10    <h3>Teste</h3>
11    <h4>Teste</h4>
12  </body>
13 </html>
```



## Teste

Parágrafo 1

Parágrafo 2

Parágrafo 3

Note que houve quebra de linha entre os testes. Isso NÃO é devido à ter quebra de linha entre as tags H no HTML, mas sim porque as tags H possuem comportamento de bloco, ou seja, não permitem elementos ao seu lado, na mesma linha.

## AS TAGS P, BR E HR

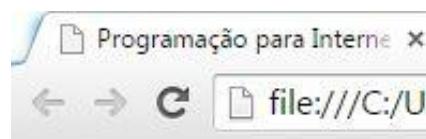
Eu falei anteriormente sobre o comportamento de bloco das tags H. Mas não é só elas que “quebram linha” após seu conteúdo. Existem algumas

tags especiais que também permitem fazer isso de uma maneira, digamos, mais controlada.

Sempre que queremos definir um parágrafo de texto em um documento HTML, com espaçamento antes e depois do parágrafo, usamos a tag P, que é um contâiner onde devemos colocar o texto a ser formatado em seu interior.

Código 2.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
● ● ●  
1 <!DOCTYPE html>  
2 <html>  
3   <head>  
4     <title>Programação para Internet</title>  
5     <meta charset="utf-8" />  
6   </head>  
7   <body>  
8     <h1>Teste</h1>  
9     <p>Parágrafo 1</p>  
10    <p>Parágrafo 2</p>  
11    <p>Parágrafo 3</p>  
12  </body>  
13 </html>
```



## Teste

Parágrafo 1

Parágrafo 2

Parágrafo 3

Note que a tag P não influencia no estilo do texto, apenas dá uma aparência de parágrafo, com espaçamentos antes e depois, para facilitar a leitura.

**Atenção:** a tag P deve ser usada para definir blocos de texto e somente texto. Para outros elementos veremos as tags DIV e SPAN mais à frente.

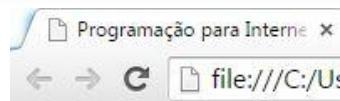
Já a tag BR apenas quebra a linha do documento exatamente naquele ponto, sem praticamente qualquer espaçamento. Note também no exemplo abaixo que a tag BR não é um contâiner, ou seja, ela não aceita conteúdo em seu interior. O browser apenas entende que quando encontrar uma tag BR ele deve mostrar um break-line no lugar (o nosso \n das linguagens de programação desktop).

Talvez você fique na dúvida de quando usar P e quando usar BR. A regra é simples: blocos de texto devem ser organizados usando tags P. Caso dentro de um bloco de texto haja a necessidade de quebrar linha, aí sim usamos a tag BR.

**Atenção:** a altura da quebra de linha feita com BR não é padronizado, diferenças podem ocorrer entre os browsers e sistemas operacionais.

Código 2.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <h1>Tópico 1</h1>
9     <p>Introdução ao HTML</p>
10    Linha 1<br />
11    Linha 2<br />
12  </body>
13 </html>
```



## Tópico 1

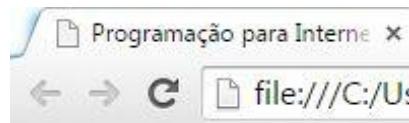
Introdução ao HTML

Linha 1  
Linha 2

E por fim, a tag HR é como se fosse uma BR, mas além de realizar a quebra ainda coloca uma linha contínua naquele ponto, o que pode ser útil em alguns casos.

Código 2.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <h1>Tópico 1</h1>
9     <p>Introdução ao HTML</p>
10    Linha 1
11    <hr />
12    Linha 2
13  </body>
14 </html>
```



## Tópico 1

Introdução ao HTML

Linha 1

---

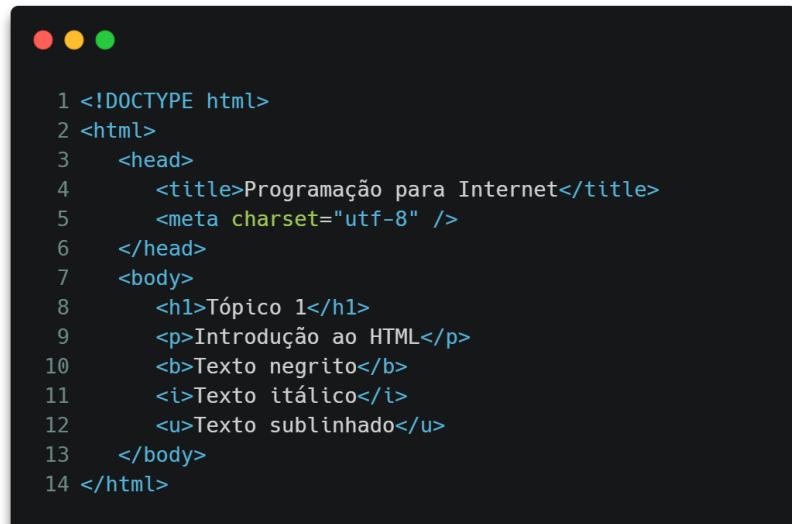
Linha 2

## AS TAGS B, STRONG, I E U

Assim como a tag P que formata um bloco de texto, existem outras tags que alteram a aparência do mesmo. As tags B e STRONG por exemplo, tornam o texto negrito (bold), sendo a segunda mais utilizada quando queremos dizer aos mecanismos de busca que essa palavra é importante para nosso site.

Já as tags I e U fazem o mesmo para itálico (italic) e sublinhado (underline), todas contâiners, que devem receber em seu interior o texto a ser formatado, como segue:

Código 2.5: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <h1>Tópico 1</h1>
9     <p>Introdução ao HTML</p>
10    <b>Texto negrito</b>
11    <i>Texto itálico</i>
12    <u>Texto sublinhado</u>
13  </body>
14 </html>
```



## Tópico 1

Introdução ao HTML

**Texto negrito** *Texto itálico* Texto sublinhado

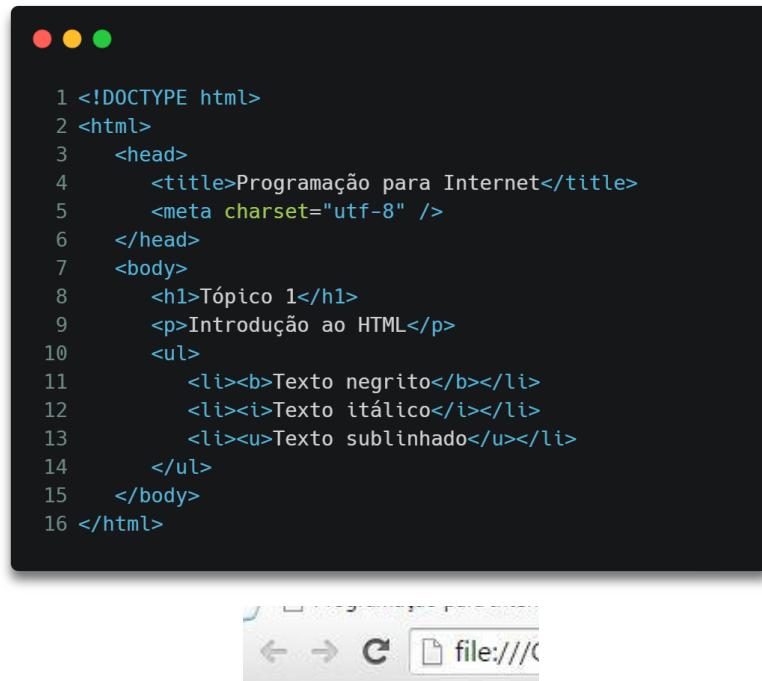
Note que os textos ficaram lado-a-lado. Isso porque as tags B, STRONG, I e U não são blocos de texto, mas apenas estilizações. Afinal, você não iria querer que no mesmo do seu texto a sua palavra sublinhada quebrasse todo o parágrafo, certo?

## AS TAGS UL, OL E LI

Quando queremos listar elementos, seja de texto ou de qualquer outra coisa (imagens, por exemplo), usamos as tags de lista UL (unordered list, lista desordenada) e OL (ordered list). Ambas listas são containers que aceitam somente tags LI dentro do seu interior, estas por sua vez também são containers que aceitam praticamente qualquer coisa no seu interior.

A diferença principal entre a lista desordenada (UL) e a ordenada (OL) não tem a ver com a ordem dos elementos, mas sim com o bullet, a marcação que vai antes de cada item, que no primeiro caso é um ponto preto e no segundo caso uma numeração crescente, como segue:

Código 2.6: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <h1>Tópico 1</h1>
9     <p>Introdução ao HTML</p>
10    <ul>
11      <li><b>Texto negrito</b></li>
12      <li><i>Texto itálico</i></li>
13      <li><u>Texto sublinhado</u></li>
14    </ul>
15  </body>
16 </html>
```



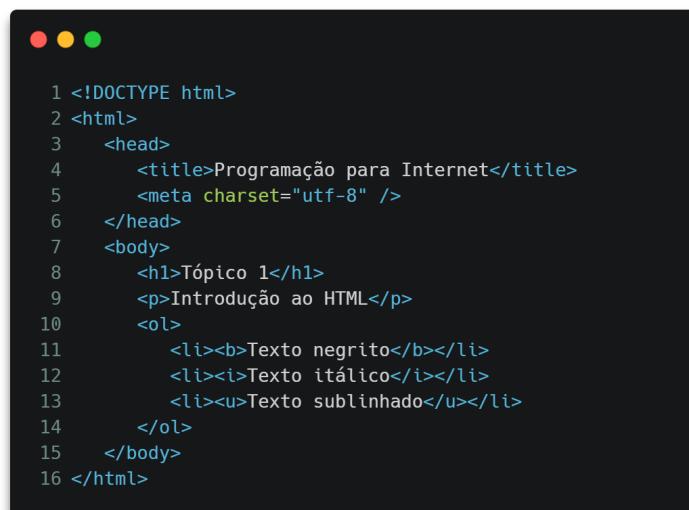
## Tópico 1

Introdução ao HTML

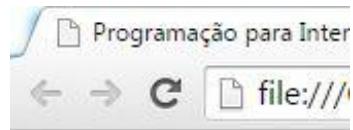
- **Texto negrito**
- *Texto itálico*
- Texto sublinhado

No exemplo acima usei uma UL, note que dentro dela somente existem tags LI, mas que dentro das tags LI coloquei uma combinação de texto e tags de formatação. O mesmo exemplo trocando a tag UL por OL pode ser visto abaixo, com seu efeito visível:

Código 2.7: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <h1>Tópico 1</h1>
9     <p>Introdução ao HTML</p>
10    <ol>
11      <li><b>Texto negrito</b></li>
12      <li><i>Texto itálico</i></li>
13      <li><u>Texto sublinhado</u></li>
14    </ol>
15  </body>
16 </html>
```



# Tópico 1

## Introdução ao HTML

1. **Texto negrito**
2. *Texto itálico*
3. Texto sublinhado

Note também que cada LI (list item) é colocado em uma linha separada, pois é um novo item da lista, o que nos leva a entender que as tags UL, OL e LI são tags de bloco, ou seja, quebram linha e garantem espaçamento automático antes e depois.

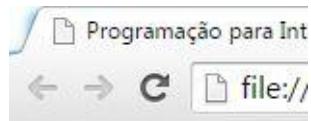
## A TAG A

A tag A é o que chamamos de âncora (anchor no original), mais conhecida popularmente como “link” (ligação), isso porque utilizamos tags A quando queremos levar o usuário de um ponto da web a outro, o que chamamos de hyperlink (em analogia ao hypertexto).

Falando de maneira prática, cada tag A é composta principalmente de duas partes: a URL de destino e o conteúdo do link, que pode ser textual ou gráfico. A URL deve ser colocada dentro do atributo href da âncora (hyperreference), enquanto que o conteúdo deve ficar entre as tags de abre-efecha da âncora (sim, ela é um contâiner). Como segue:

Código 2.8: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <h1>Tópico 1</h1>
9     <p>Introdução ao HTML</p>
10    <a href="http://www.google.com">Clique aqui</a>
11  </body>
12 </html>
```



# Tópico 1

Introdução ao HTML

[Clique aqui](#)

Note que o conteúdo da tag A ( a frase “Clique aqui”) é o texto que aparece na página, enquanto que o atributo href define para onde o usuário será levado no caso dele clicar no link, o que neste caso é o site do Google. Isto é o básico que você precisa saber sobre âncoras HTML.

Além do básico, existem ainda dois atributos que podem ser úteis às suas âncoras: o atributo title, que define a dica da âncora (quando você fica com o mouse parado sobre ela, a dica aparece) e o atributo target, que permite a você definir se ao ser clicado o link levará o usuário para outra aba ou permanecerá na mesma. Os valores possíveis para o atributo target são: \_blank e \_self

Código 2.8: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <h1>Tópico 1</h1>
9     <p>Introdução ao HTML</p>
10    <a href="http://www.google.com" title="Ir ao Google"
11        target="_blank">Clique aqui</a>
12  </body>
13 </html>
```

No exemplo acima, levemente modificado da versão anterior, colocamos um title “Ir ao Google” e um target “\_blank”, que fará com que uma nova aba se abra no navegador com o endereço do Google quando o usuário clicar neste link. Visualmente nenhum desses atributos mudará a aparência da âncora original, que mantém-se com o texto “Clique aqui” e a cor azul escuro sublinhado, denotando que é um hyperlink.

Embora o mais comum seja encontrarmos links textuais pela Internet, sendo a tag A um contâiner, ela permite agregar outras tags em seu interior, como tags de imagens, por exemplo, ou determinadas áreas do site como DIVs (a seguir).

**Atenção:** o atributo href aceita *links absolutos* e *links relativos*. *Links absolutos* são aqueles que começam com um protocolo (geralmente `http://`) e geralmente levam a outros sites da Internet. *Links relativos* começam com um nome de arquivo, pasta ou apenas uma '/' e fazem referência a um recurso existente no mesmo domínio/site. Ou seja, se tivermos um `href="http://www.google.com"` quer dizer que o link lhe levará ao site do Google. Agora se tivermos um `href="index.html"` o link nos levará ao arquivo `index.html` no site atual.

Experimente criar uma segunda página chamada `teste.html`, colocar ela na mesma pasta da sua `index.html` e mudar o href da âncora para `teste.html`. Quando clicar no link, a outra página vai se abrir no navegador.

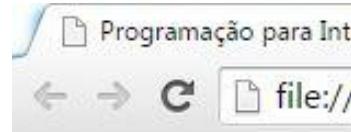
## A TAG IMG

O nome IMG vem de IMAGE e denota a tag que representa uma imagem na página web. Esta tag NÃO é um contâiner, ou seja, ela abre-e-fecha a tag nela mesma, e representando uma imagem seu atributo mais importante é o SRC (source, origem em inglês) que deve conter um endereço (absoluto ou relativo) de um arquivo de imagem (geralmente JPEG, GIF ou PNG).

Código 2.8: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <h1>Tópico 1</h1>
9     <p>Introdução ao HTML</p>
10    
11    <a href="http://www.google.com" title="Ir
ao Google" target="_blank">Clique aqui</a>
12  </body>
13 </html>
```

Por questões de organização costumamos guardar as imagens em uma subpasta do projeto, para separá-las das páginas HTML e demais arquivos do projeto. Nomes comuns para este subpasta incluem IMG (*tâãão criativo*), imagens, images e content. Crie uma pasta com o nome img ao lado do seu arquivo HTML e dentro coloque uma imagem qualquer com o nome de imagem.jpg, que no meu exemplo abaixo, é um carro.



# Tópico 1

Introdução ao HTML



Podemos também criar imagens com links. Para isso, basta colocar uma tag IMG contendo a sua imagem dentro de uma tag A, com o endereço do link. Assim, teremos uma imagem que quando clicada leva o usuário à outra página web.

Código 2.9: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <h1>Tópico 1</h1>
9     <p>Introdução ao HTML</p>
10    <a href="http://www.mitsubishimotors.com.br"
11        title="Ir à Mitsubishi" target="_blank">
12      
13    </a>
14  </body>
15 </html>
```

Outros atributos da tag IMG incluem ALT, que é uma legenda para a imagem (não visível), WIDTH (que representa sua largura, sendo por padrão a largura original da imagem) e HEIGHT (que representa sua altura. Caso apenas WIDTH seja especificado, a altura será redimensionada proporcionalmente e vice-versa. Note que isso não afeta o tamanho real da imagem, apenas o visual.

**Atenção:** procure usar caminhos absolutos apenas de imagens existentes na Internet e ainda assim, considere a hipótese de salvar as imagens localmente para poder utilizar caminhos relativos, mais confiáveis.

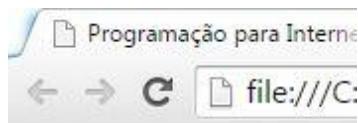
## AS TAGS DIV E SPAN

As tags DIV e SPAN são tags contâiner (ou seja, permitem que sejam colocadas outras tags em seu interior) utilizadas para organizar e formatar as áreas da sua página HTML, sendo que a tag DIV organiza uma área em bloco (in-block, ou seja, quebra a linha automaticamente) e a tag SPAN organiza uma área em linha (in-line, ou seja, não quebra a linha automaticamente). Por ora é o que precisamos saber.

Por exemplo, o seguinte layout abaixo, com 3 seções verticais: na primeira temos um título e parágrafo de texto, na segunda uma imagem e na terceira um rodapé.

Código 2.10: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
● ● ●
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <div>
9       <h1>Tópico 1</h1>
10      <p>Introdução ao HTML</p>
11    </div>
12    <div>
13      <a href="http://www.mitsubishimotors.com.br" title="Ir à Mitsubishi" target="_blank">
14        
15      </a>
16    </div>
17    <div><i>Apenas um rodapé</i></div>
18  </body>
19 </html>
```



# Tópico 1

Introdução ao HTML



*Apenas um rodapé.*

A especificação HTML diz que as tags SPAN e DIV somente devem ser usadas quando nenhuma outra tag HTML possui a função que desejamos.

## AS TAGS TABLE, TR E TD

Vimos na seção anterior que usamos DIVs e SPANS para organizar as seções da nossa página HTML. Entretanto, às vezes as informações de uma seção está organizada em linhas e colunas, como uma tabela Excel, de uma maneira que ficaria extremamente difícil (e trabalhoso) de fazer tudo usando DIVs (imagine uma DIV por célula da tabela). Nestes casos onde temos dados tabulares devemos usar a tag TABLE.

A tag TABLE é uma tag container que somente aceita em seu interior tags TR (table row ou linha da tabela). As tags TR, por sua vez, também são containers, mas que aceitam somente tags TD (table data ou dado da tabela, entenda como célula) em seu interior. Já as tags TD são containers que aceitam qualquer coisa em seu interior.

Assim, seguindo esta hierarquia de containers, podemos recriar a aparência de qualquer tabela que necessitarmos, como a abaixo:

	A	B
1	Nome	RA
2	Luiz	123
3	João	456
4	Maria	789
-		

# Tópico 1

Introdução ao HTML

**Nome RA**

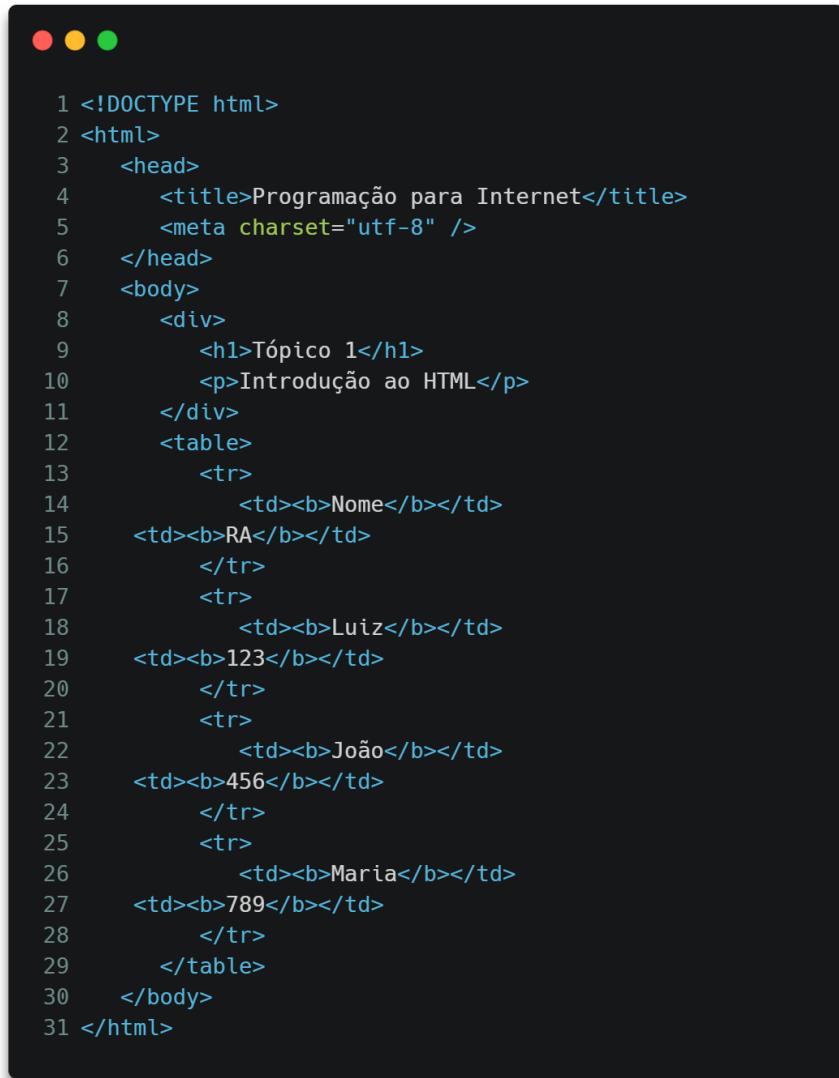
Luiz 123

João 456

Maria 789

Usando o código abaixo. Atente ao uso da tag B para tornar o texto do cabeçalho negrito.

Código 2.11: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <div>
9       <h1>Tópico 1</h1>
10      <p>Introdução ao HTML</p>
11    </div>
12    <table>
13      <tr>
14        <td><b>Nome</b></td>
15        <td><b>RA</b></td>
16      </tr>
17      <tr>
18        <td><b>Luiz</b></td>
19        <td><b>123</b></td>
20      </tr>
21      <tr>
22        <td><b>João</b></td>
23        <td><b>456</b></td>
24      </tr>
25      <tr>
26        <td><b>Maria</b></td>
27        <td><b>789</b></td>
28      </tr>
29    </table>
30  </body>
31 </html>
```

**Atenção:** apesar de parecer uma boa ideia, evite utilizar tabelas para organizar as seções da sua página HTML, pois elas não foram criadas com este propósito. O ideal é sempre utilizarmos DIVs e SPANs para essa finalidade, deixando as tabelas para a apresentação de dados tabulares.

## Formulários HTML

Quando falamos de programação para a web não temos como fugir da criação de formulários, tal qual na programação tradicional (desktop). A especificação HTML possui capítulos específicos para tratar disso e abaixo você confere um resumo das regras de criação e tags específicas de formulários HTML.

### A TAG FORM

Quando queremos criar um formulário na nossa página HTML devemos utilizar a tag FORM. A tag FORM é um contâiner que indica que todas as demais tags e conteúdos no seu interior representam um único formulário HTML, por exemplo, um formulário de cadastro, de contato ou login.

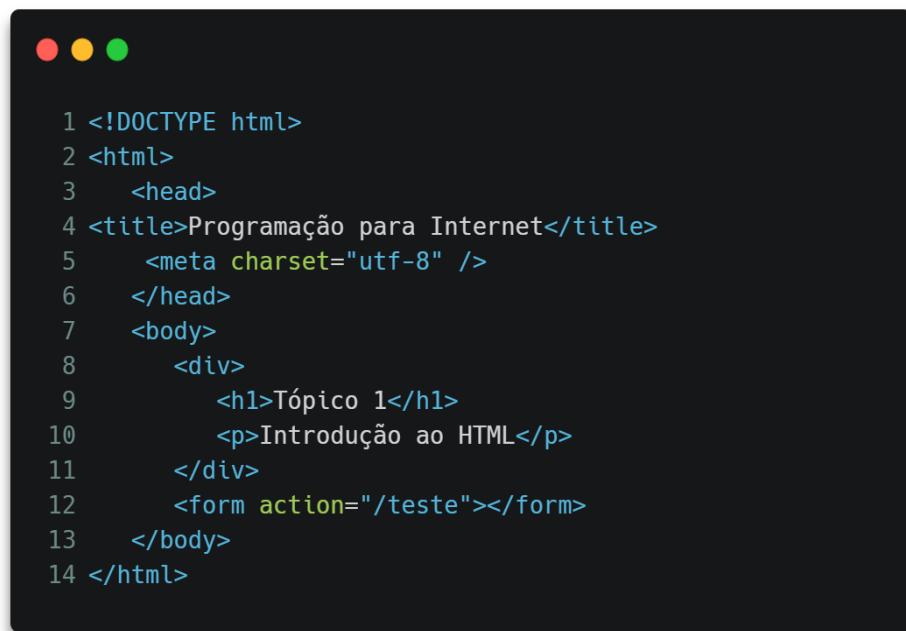
Dentro de um form, além das tags HTML comuns, que ajustam a aparência e organizam o conteúdo da página, usaremos as tags INPUT, que serão vistas a seguir.

O atributo mais importante de um form é sua ACTION (ação) que é a URL da página web para onde irão os dados preenchidos neste form, sendo que geralmente esta página é escrita em uma linguagem de programação como PHP e JAVA, ao invés de HTML, o que foge do escopo deste ebook. Por isso você vai ver que nos meus exemplos vai ter apenas “teste” escrito ali.

Além deste atributo temos o METHOD, que define o verbo HTTP que será utilizado para transmissão dos dados (geralmente GET ou POST). HTTP é o protocolo de transferência de hipertexto utilizado na Internet mundial. Ele define os dados e metadados que devem ser transmitidos a cada requisição e resposta web.

Usamos HTTP GET quando queremos obter dados de uma página. Usamos HTTP POST quando queremos enviar dados para uma página. Em ambos os casos, a informação a ser enviada é incluída junto à requisição, em pares de chave-valor como verá a seguir.

**Atenção:** existem outros verbos HTTP como DELETE, PUT, HEAD e DEBUG, sendo que GET e POST são os únicos suportados por HTML Forms e também os mais utilizados no geral em qualquer contexto web.



```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <div>
9       <h1>Tópico 1</h1>
10      <p>Introdução ao HTML</p>
11    </div>
12    <form action="/teste"></form>
13  </body>
14 </html>
```

Visualmente, o uso da tag FORM sozinha não renderiza nada no navegador do usuário, então passaremos às demais tags.

## A TAG LABEL

Uma tag que serve para criar um rótulo em seu FORM HTML geralmente acompanhada de outro componente. A LABEL é uma tag contâiner que permite texto ou um INPUT (veja a seguir) em seu interior. Geralmente a LABEL possui o mesmo comportamento visual de um texto escrito diretamente na página, e seu uso mais comum é como rótulo de campos de texto, sendo que segundo a especificação os pares LABEL + INPUT devem estar dentro de tags P (parágrafos) dentro do FORM.

Código 2.12: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <div>
9       <h1>Tópico 1</h1>
10      <p>Introdução ao HTML</p>
11    </div>
12    <form action="/teste">
13      texto comum <br />
14      <label>texto em label</label>
15    </form>
16  </body>
17 </html>
```

# Tópico 1

Introdução ao HTML

texto comum  
texto em label

Note no exemplo acima o uso da tag BR para quebrar a linha após o primeiro texto. Isso porque a tag LABEL tem comportamento inline (em-linha) ou seja, não quebra linha automaticamente. Essa seria outra alternativa em relação ao uso da tag P.

## A TAG INPUT

Na verdade INPUT não representa uma única coisa dentro de FORMs HTML, mas uma família de componentes utilizados para criação de formulários. A tag INPUT pode ou não ser um contâiner, dependendo de seu tipo, definido no atributo TYPE.

Além do TYPE, as tags INPUT costumam ter o atributo VALUE, que representa o seu valor, e o atributo NAME, que representa o seu nome

(invisível ao usuário final). Também temos o atributo DISABLED, que quando definido como TRUE torna o INPUT desabilitado, o que pode ser útil em algumas situações.

Os principais valores para o atributo TYPE e seus respectivos usos são:

### **TYPE="TEXT" e TYPE="PASSWORD"**

Utilizados para criar campos de texto e de senha, respectivamente, semelhantes ao JTextField do Java Swing e TextBox do ASP.NET. O atributo NAME define o nome do seu campo (não confundir com seu rótulo) e o atributo VALUE define o seu conteúdo. Este INPUT TYPE não aceita outros elementos em seu interior e seu comportamento visual é inline (ou seja, não quebra linha automaticamente).

Código 2.13: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <div>
9       <h1>Tópico 1</h1>
10      <p>Introdução ao HTML</p>
11    </div>
12    <form action="/teste">
13      <label>Usuário:</label>
14      <input type="text" name="txtUsuario" value="Luiz" /></label>
15      <br />
16      <label>Senha:</label>
17      <input type="password" name="txtSenha" value="123" /></label>
18    </form>
19  </body>
20 </html>
```

**Nota:** repare como coloquei o input dentro da tag label. Isso está perfeitamente correta e ajuda a dizer ao browser que "essa label é daquele input" e inclusive se você clicar no texto da label no navegador verá que o input fica selecionado pois o browser entende que deseja preenchê-lo.

Resultado do código acima (note a diferença entre o text e o password, além do fato de que o atributo NAME não aparece ao usuário):

# Tópico 1

Introdução ao HTML

Usuário:

Senha:

## TYPE="RADIO"

Em algumas ocasiões queremos que o usuário apenas uma dentre um número pequeno de opções. Assim, podemos definir estas opções como sendo botões de rádio HTML, os INPUT TYPE="RADIO", em analogia aos JRadioButtons e RadioButtons de outras linguagens.

Cada INPUT TYPE="RADIO" deve ser sucedido por um texto plano (pois ele não possui conteúdo próprio) e deve conter dois atributos: VALUE, que indica po valor da opção selecionada (não visível), e NAME, que indica o nome deste grupo de botões de rádio. Ou seja, todos os botões de rádio com o mesmo name representam o mesmo grupo de informação, e com isso o usuário poderá selecionar apenas um deles (bem mais fácil que o JButtonGroup do Java Swing!).

Código 2.13: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5   <meta charset="utf-8" />
6   </head>
7   <body>
8     <div>
9       <h1>Tópico 1</h1>
10      <p>Introdução ao HTML</p>
11    </div>
12    <form action="/teste">
13      <label>Usuário:</label>
14      <input type="text" name="txtusuario" value="Luiz" /></label>
15      <br />
16      <label>Senha:</label>
17      <input type="password" name="txtsenha" value="123" /></label>
18      <br />
19      <label>Sexo:</label>
20      <input type="radio" name="sexo" value="M" checked />Masculino
21      <input type="radio" name="sexo" value="F" />Feminino
22    </form>
23  </body>
24 </html>
```

Abaixo temos o resultado.

# Tópico 1

## Introdução ao HTML

Usuário:

Senha:

Sexo:  Masculino  Feminino

Note que o fato de ambos INPUT TYPE="RADIO" possuírem o mesmo valor de atributo NAME fará com que o usuário apenas possa selecionar um, bem como quando este formulário for submetido ao servidor, somente um deles será enviado sob a variável "sexo".

Note também que o atributo VALUE não é mostrado ao usuário, ele somente será conhecido quando o FORM for enviado ao servidor (e somente o VALUE do RADIO selecionado).

E por fim, note que foi utilizado um terceiro atributo, chamado CHECKED no RADIO masculino, mostrando que o mesmo deve vir selecionado por padrão. Este atributo não possui qualquer valor, a sua presença por si só no interior de uma tag INPUT RADIO já indica que o mesmo está selecionado.

### **INPUT TYPE="CHECKBOX"**

Vimos antes que o INPUT RADIO serve para o usuário selecionar uma dentre algumas opções. Mas e quando ele pode selecionar mais de uma opção? Neste caso podemos usar o INPUT CHECKBOX.

Neste INPUT nós temos o atributo NAME para definir o nome da variável que será enviado ao servidor e fora do INPUT (ele não é um contâiner) temos o texto descritivo para o usuário entender do que se trata o controle, como mostra a última linha de INPUT do código abaixo.

Código 2.13: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <div>
9       <h1>Tópico 1</h1>
10      <p>Introdução ao HTML</p>
11    </div>
12    <form action="/teste">
13      <label>Usuário:<input type="text" name="txtusuario" value="Luiz" /></label>
14      <br />
15      <label>Senha:<input type="password" name="txtsenha" value="123" /></label>
16      <br />
17      <label>Sexo:</label>
18      <input type="radio" name="sexo" value="M" checked />Masculino
19      <input type="radio" name="sexo" value="F" />Feminino
20      <input type="checkbox" name="chkSpam" />Quero receber spam por e-mail
21    </form>
22  </body>
23 </html>
```

Como resultado temos:

## Tópico 1

Introdução ao HTML

Usuário:

Senha:

Sexo:  Masculino  Feminino

Quero receber spam por e-mail

Note que poderíamos ter definido o atributo CHECKED neste INPUT CHECKBOX caso quiséssemos que ele viesse marcado por padrão, assim como fizemos com o INPUT RADIO anteriormente.

INPUT TYPE="SUBMIT" e TYPE="BUTTON"

Estes últimos INPUTs são tipos diferentes de botões. Ambos são representados visualmente da mesma forma, como botões com um texto dentro, definido pelo atributo VALUE desse input. Entretanto, suas funcionalidades variam: o INPUT BUTTON dispara um código Javascript quando clicado (que não veremos agora) e o INPUT SUBMIT submete todos os dados do formulário para o servidor, sendo este último o mais importante a ser estudado no momento.

Anteriormente, quando vimos a tag FORM foi falado que ela tinha dois atributos: ACTION e METHOD, sendo este último opcional. Quando um INPUT SUBMIT é clicado, o navegador coleta todas as informações contida nos campos do FORM e envia eles para o endereço fornecido no atributo ACTION, no formato NAME1=VALUE1&NAME2=VALUE2, ou seja, ele pega o name de cada componente HTML e seu valor e junta-os usando sinais de '=', e depois junta todos os pares de chave-valor usando sinais de '&'. Se o atributo METHOD não for definido, será enviado para o servidor via HTTP GET, caso contrário será enviado usando o verbo HTTP definido neste atributo (o que é irrelevante no momento).

Assim, quando esta informação chega em nosso backend, o nosso código programado em alguma linguagem de servidor irá dar cabo da mesma fazendo o que for necessário, como salvar dados no banco, por exemplo, inclusive retornando uma mensagem de sucesso ou fracasso na maioria das vezes.

Código 2.13: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
● ● ●
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <div>
9       <h1>Tópico 1</h1>
10      <p>Introdução ao HTML</p>
11    </div>
12    <form action="/teste">
13      <label>Usuário:<input type="text" name="txtusuario" value="Luiz" /></label>
14      <br />
15      <label>Senha:<input type="password" name="txtsenha" value="123" /></label>
16      <br />
17      <label>Sexo:</label>
18      <input type="radio" name="sexo" value="M" checked />Masculino
19      <input type="radio" name="sexo" value="F" />Feminino
20      <input type="checkbox" name="chkSpam" />Quero receber spam por email
21      <br />
22      <input type="submit" value="Salvar" />
23    </form>
24  </body>
25 </html>
```

Como resultado, temos a imagem abaixo. Note que se clicarmos no INPUT SUBMIT seremos enviados para a página /teste que não existe, exibindo um erro no navegador. Isso é normal, uma vez que ainda não vimos como fazer a programação desta página em Node.js.

## Tópico 1

Introdução ao HTML

Usuário:

Senha:

Sexo:  Masculino  Feminino

Quero receber spam por e-mail

Outros TYPES: não existem somente os TYPES mostrados acima. Existem muitos outros principalmente a partir da especificação 5 da linguagem HTML. Alguns TYPES especialmente interessantes vêm para substituir o INPUT TEXT quando queremos limitar o tipo de dados que pode ser informado pelo usuário, como o INPUT NUMBER (para somente números). Veja a lista completa em [http://www.w3schools.com/tags/tag\\_input.asp](http://www.w3schools.com/tags/tag_input.asp)

### A TAG TEXTAREA

Funciona de maneira idêntica ao INPUT TYPE="TEXT" porém permitindo múltiplas linhas, como em um JTextArea do Java Swing. O atributo VALUE do TEXTAREA define o seu conteúdo textual e seu NAME o nome da variável a ser enviada ao servidor.

### AS TAGS SELECT E OPTION

Quando queremos listar diversas opções ao usuário sem poluir demais o visual da tela podemos usar a tag SELECT, que é o equivalente ao JComboBox e DropDownList de outras linguagens. A tag SELECT é um contâiner que aceita tags OPTION em seu interior e possui um atributo name para representá-la quando o FORM é submetido para o servidor.

Já as tags OPTION por sua vez possuem um conteúdo textual (são containers que aceitam somente texto plano em seu interior) e um conteúdo oculto, seu atributo VALUE, que pode ser utilizado para guardar o código correspondente ao texto escolhido pelo usuário.

Quando o FORM é submetido ao servidor, é enviado uma variável com o NAME do SELECT e o VALUE do OPTION selecionado pelo usuário.

Código 2.14: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
● ○ ●
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Programação para Internet</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <div>
9       <h1>Tópico 1</h1>
10      <p>Introdução ao HTML</p>
11    </div>
12    <form action="/teste">
13      <label>Usuário:<input type="text" name="txtusuario" value="Luiz" /></label>
14      <br />
15      <label>Senha:<input type="password" name="txtsenha" value="123" /></label>
16      <br />
17      <label>UF:</label>
18      <select name="cmbUF">
19        <option value="RS">Rio Grande do Sul</option>
20        <option value="SC">Santa Catarina</option>
21        <option value="PR">Paraná</option>
22      </select>
23    </form>
24  </body>
25 </html>
```

Visualmente temos (onde o primeiro OPTION se torna o valor selecionado por padrão):

# Tópico 1

Introdução ao HTML

Usuário: luiz

Senha: ...

UF: Rio Grande do Sul ▾

## OUTRAS TAGS

Estas não são as únicas tags existentes. Além disso, a descrição de cada tag vista aqui é um resumo bem superficial e o estudo mais aprofundado é vital para todo profissional que deseje trabalhar com programação web.

Mais informações podem ser obtidas no site oficial da W3.org.

Recomendo que você exerceite bastante esta primeira etapa antes de avançar pois certamente tem coisas muito mais poderosas que você pode fazer com HTML do que apenas o que eu mostrei aqui. Por exemplo, como você poderia construir a imagem abaixo em HTML?



## Listagem de Clientes

Clientes já cadastrados no sistema.

Nome	Idade	UF	Ações
Nenhum cliente cadastrado.			
<a href="#">Cadastrar Novo</a>			

Abaixo, uma das possíveis soluções (note que usei um atributo STYLE que ainda não estudamos):

Código 2.15: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Listagem de Clientes</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <h1>Listagem de Clientes</h1>
9     <p>Clientes já cadastrados no sistema.</p>
10    <table style="width:50%">
11      <thead>
12        <tr style="background-color: #CCC">
13          <td style="width:50%">Nome</td>
14          <td style="width:15%">Idade</td>
15          <td style="width:15%">UF</td>
16          <td>Ações</td>
17        </tr>
18      </thead>
19      <tbody>
20        <tr>
21          <td colspan="4">Nenhum cliente cadastrado.</td>
22        </tr>
23      </tbody>
24      <tfoot>
25        <tr>
26          <td colspan="4">
27            <a href="novo.html">Cadastrar Novo</a>
28          </td>
29        </tr>
30      </tfoot>
31    </table>
32  </body>
33 </html>
```

Note que a última âncora da página está apontando seu href para novo.html. Que tal criarmos esta página para conseguir navegar de uma para outra?

A imagem abaixo lhe dá um exemplo de como essa novo.html poderia ser (salve na mesma pasta da anterior).

## Cadastro de Cliente

Preencha os dados abaixo para salvar o cliente.

Nome:

Idade:

UF:

[Cancelar](#) |

Se estiver com problemas no HTML, pode usar o código abaixo.

Código 2.16: disponível em <https://www.luisitools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Cadastro de Cliente</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <h1>Cadastro de Cliente</h1>
9     <p>Preencha os dados abaixo para salvar o cliente.</p>
10    <form action="index.html">
11      <p>
12        <label>Nome: <input type="text" name="nome" /></label>
13      </p>
14      <p>
15        <label>Idade: <input type="number" name="idade" /></label>
16      </p>
17      <p>
18        <label>UF: <select name="uf">
19          <option>RS</option>
20          <option>SC</option>
21          <option>PR</option>
22          <!-- coloque os estados que quiser -->
23        </select></label>
24      </p>
25      <p>
26        <a href="index.html">Cancelar</a> | <input type="submit" value="Salvar" />
27      </p>
28    </form>
29  </body>
30 </html>
```

Note que de uma página você pode navegar para outra e vice-versa, mas que nenhum cadastro acontece de verdade.

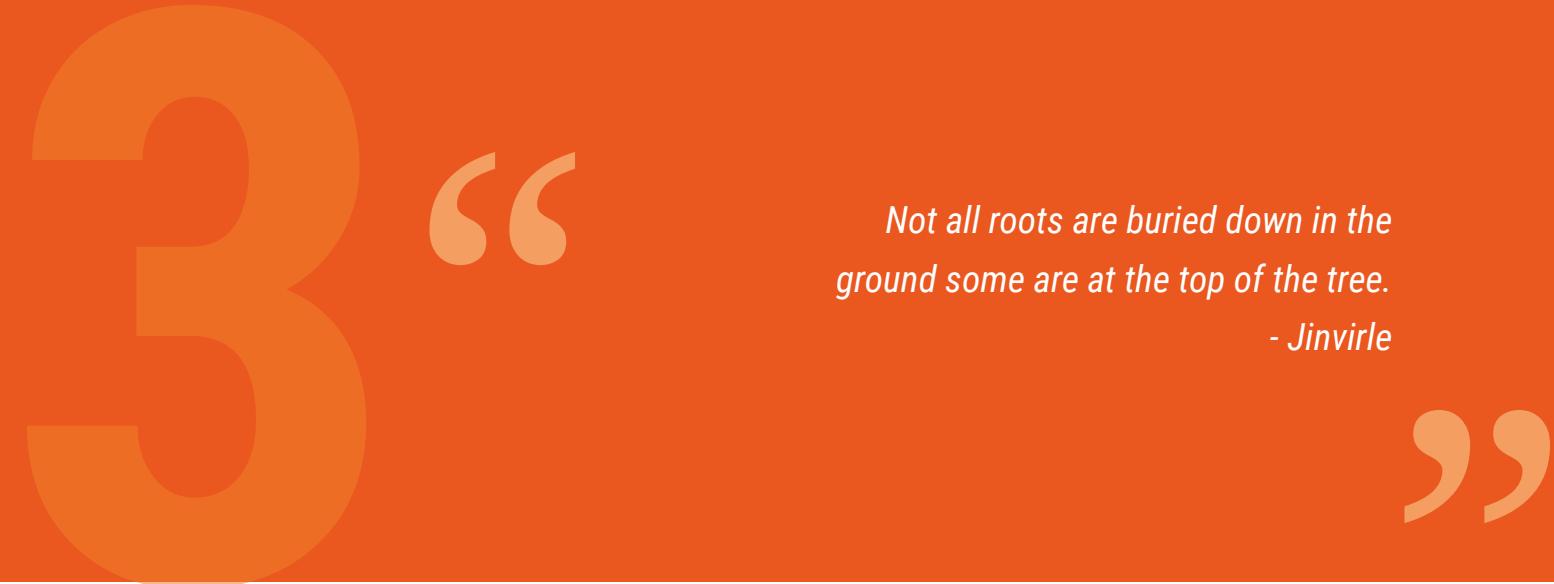
Deixe este projeto salvo, continuaremos ele mais adiante neste ebook.

*Quer fazer um curso online de Desenvolvimento Web FullStack JS com o autor deste ebook?*

Acesse <https://www.luiztools.com.br/curso-fullstack>

---

# JAVASCRIPT BÁSICO



*Not all roots are buried down in the  
ground some are at the top of the tree.*

*- Jinvirle*

O objetivo deste capítulo é lhe dar a base de programação em JavaScript. Ele é especialmente útil se você estiver começando agora com programação, mas também pode ser útil para quem possa estar migrando de uma linguagem para JS.

JavaScript foi originalmente desenvolvido por Brendan Eich quando trabalhou na Netscape sob o nome de Mocha, posteriormente teve seu nome mudado para LiveScript e por fim JavaScript. LiveScript foi o nome oficial da linguagem quando foi lançada pela primeira vez na versão beta do navegador Netscape 2.0 em setembro de 1995, mas teve seu nome mudado em um anúncio conjunto com a Sun Microsystems em dezembro de 1995 quando foi implementado no navegador Netscape versão 2.0B3.

A mudança de nome de LiveScript para JavaScript coincidiu com a época em que a Netscape adicionou suporte à tecnologia Java em seu navegador (Applets). A escolha final do nome causou confusão dando a impressão de que a linguagem foi baseada em java, sendo que tal escolha foi caracterizada por muitos como uma estratégia de marketing da Netscape para aproveitar a popularidade do recém-lançado Java.

JavaScript rapidamente adquiriu ampla aceitação como linguagem de script client-side de páginas web. JScript, um dialeto compatível e intercambiável criado pela Microsoft foi incluído no Internet Explorer 3.0, liberado em Agosto de 1996.

Em novembro de 1996 a Netscape anunciou que tinha submetido JavaScript para Ecma International como candidato a padrão industrial e o trabalho subsequente resultou na versão padronizada chamada ECMAScript.

JavaScript hoje é a linguagem de programação mais popular da web. Inicialmente, no entanto, muitos profissionais denegriram a linguagem pois ela tinha como alvo principal o público leigo. Com o advento do Ajax durante o boom da Web 2.0, JavaScript teve sua popularidade de volta e recebeu mais atenção profissional. O resultado foi a proliferação de frameworks e bibliotecas, práticas de programação melhoradas e o aumento no uso do JavaScript fora do ambiente de navegadores, bem como o uso de plataformas de JavaScript server-side.

## A tag SCRIPT

Utiliza-se a tag HTML SCRIPT para criar blocos de código Javascript em uma página HTML ou para carregar um arquivo com extensão .JS que exista em alguma pasta ou na Internet. Ambos exemplos são mostrados abaixo

```
1 <!--... código HTML qualquer ...-->
2 <script>
3     //código Javascript vai aqui
4 </script>
5 <!-- ... mais código HTML ... -->
```

No exemplo acima a tag script é usada em formato container. Tudo que for colocado dentro dela será interpretado como código Javascript. Para realizar os exercícios iniciais, recomendo esta abordagem, a de criar uma página HTML e colocar um bloco script dentro, como abaixo.

Código 3.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

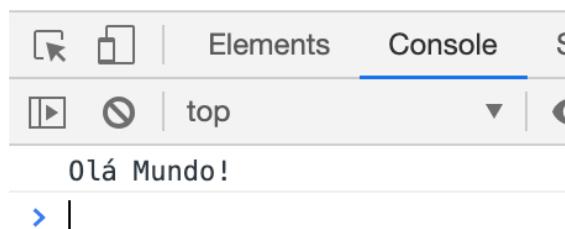
```
1 <!DOCTYPE html>
2 <html>
3     <head>
4         <title>Programação em JavaScript</title>
5         <meta charset="utf-8" />
6     </head>
7     <body>
8         <p>Apenas um texto</p>
9         <script>
10            console.log('Olá Mundo!');
11        </script>
12    </body>
13 </html>
```

Note que deixei um código console.log dentro do bloco script. Para testar esta página, faça do mesmo jeito que vem fazendo, abrindo o arquivo HTML no seu navegador, mas agora use o botão direito do mouse e escolha a opção de inspecionar a página.

Vai se abrir uma aba cheia de opções na parte inferior do seu Google Chrome. Escolha a opção Console e você verá algo como abaixo.



### Apenas um texto



`console.log` é um comando JavaScript para enviar mensagens para este “painel” do navegador, onde podemos ver as mensagens.

É através deste console que você vai verificar como os nossos exercícios com JavaScript estão funcionando, então deixe-o aberto e, entre um código e outro, salve o arquivo e apenas dê um refresh no navegador (F5).

Outra forma de fazer os testes, mais avançada, é criando um arquivo JS separado e referenciando-o no bloco script, ao invés de colocar os códigos diretamente. Não recomendo este formato se estiver começando agora com front-end.

A screenshot of a browser window. At the top, there are three colored window control buttons (red, yellow, green). Below the title bar, the page content is displayed. It shows three lines of code:  
1 <!-- ... código HTML qualquer ... -->  
2 <script src="funcoes.js" />  
3 <!-- ...código HTML qualquer ... -->

No exemplo acima a tag SCRIPT fecha em si mesma e usa o atributo SRC para apontar para um arquivo com extensão .JS presente no próprio projeto ou na internet (neste caso deve referenciar a URL absoluta do arquivo).

**Nota:** ro carregamento de scripts JS na página HTML possui comportamento bloqueante na requisição. Ou seja, até que o script termine de ser carregado o browser não carrega mais nada. Assim, caso seu script seja grande (algumas dezenas de KB por exemplo), é sempre indicado referenciá-lo apenas no final do documento HTML, o mais próximo possível da tag </BODY> (fecha-body) para evitar que o usuário tenha de esperar demais pelo carregamento para começar a usar a página.

A seguir apresentarei a sintaxe da linguagem JavaScript, abordando as estruturas de dados existentes, as regras para declaração de variáveis, as recomendações gerais para nomenclatura, os operadores e as estruturas de controle disponíveis.

Como será notado por quem já é programador de outras linguagens, a sintaxe da linguagem JavaScript é muito semelhante àquela usada pela linguagem C, porém muito mais dinâmica.

## Declaração de Variáveis

Uma variável é um nome definido pelo programador ao qual pode ser associado um valor pertencente a um certo tipo de dados.

Para que muitas coisas aconteçam em nossos sistemas, nós precisaremos de variáveis, tal qual operações aritméticas, armazenamento de dados e mensagens para o usuário, só para citar alguns exemplos comuns.

Em outras palavras, uma variável é como uma memória, capaz de armazenar um valor de um certo tipo, para a qual se dá um nome que usualmente descreve seu significado ou propósito. Desta forma toda variável possui um nome, um tipo e um conteúdo.

O nome de uma variável em JavaScript pode ser uma sequência de um ou mais caracteres alfabéticos e numéricos, iniciados por uma letra ou ainda pelo caractere ‘\_’ (underscore).

Os nomes não podem conter outros símbolos gráficos, operadores ou espaços em branco. É importante ressaltar que as letras minúsculas são consideradas diferentes das letras maiúsculas, ou seja, a linguagem JavaScript é sensível ao caixa empregado (case sensitive), assim temos

a	total	x2
idade	_especial	TOT
Maximo	ExpData	meuNumero

como exemplos válidos: Segundo as mesmas regras temos abaixo exemplos inválidos de nomes de variáveis:

- » 1x
- » Total geral
- » numero-minimo
- » function

A razão destes nomes serem inválidos é simples: o primeiro começa com um algarismo numérico, o segundo possui um espaço em branco, o terceiro contém o operador menos, mas por que o quarto nome é inválido?

Porque além das regras de formação do nome em si, uma variável não pode utilizar como nome uma palavra reservada da linguagem.

Mas o que é uma palavra reservada?

As palavras reservadas são os comandos, especificadores e modificadores pertencentes a sintaxe de uma linguagem.

As palavras reservadas da linguagem JavaScript, que portanto não podem ser utilizadas

abstract	arguments	await
boolean	break	byte
case	catch	char
class	const	continue
debugger	default	delete

do	double	else
enum	eval	export
extends	false	final
finally	float	for
function	goto	if
implements	import	in
instanceof	int	interface
let	long	native
new	null	package
private	protected	public
return	short	static
super	switch	synchronized
this	throw	throws
transient	true	try
typeof	var	void
volatile	while	with
yield		

como nome de variáveis ou outros elementos, são:

Algumas destas palavras não são mais reservadas em versões recentes do JavaScript, mas ainda assim não são utilizadas em nomes de variáveis por boa prática. Muitas dessas palavras são reservadas do JavaScript mas não são utilizadas de fato e existem ainda uma série de outras palavras que são nomes de funções ou objetos globais existentes em

diferentes ambientes como no navegador, por exemplo.

Desta forma para declararmos uma variável temos as seguintes opções:

- » nomeDaVariavel
- » var nomeDaVariavel
- » const nomeDaVariavel
- » let nomeDaVariavel

Na primeira opção, apenas escrevendo o nome da variável, declaramos a mesma como global na nossa aplicação. Isso não é exatamente verdade em Node.js, que possui algumas regras diferentes do JavaScript tradicional, e não é nem um pouco recomendado, mas existe.

Na segunda opção, a mais clássica, declaramos uma variável no seu sentido mais tradicional em JavaScript. Note que nós não declaramos o tipo da variável, ‘var’ simplesmente significa ‘variable’. Em JavaScript a tipagem é fraca e dinâmica, o que quer dizer que ela a princípio é inferida a partir do valor do dado que guardarmos na variável, mas que o tipo pode ser alterado durante a execução do programa.

Sendo assim, quando declaramos (este trecho deve estar dentro de um bloco script no seu HTML e o resultado deve ser observado no console do navegador):

Código 3.2: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 var x = 1;
```

a variável x passa a ser do tipo número inteiro, automaticamente.

Agora se declararmos na linha:

Código 3.2: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 x = 'teste';
```

a variável x passa a ser do tipo string (texto).

Já os modificadores let e const foram criados nas versões mais recentes do JavaScript para termos mais controle sobre o que acontece com nossas variáveis.

Quando declaramos uma variável com ‘const’ na frente, estamos dizendo que ela é uma constante, e que seu valor somente poderá ser atribuído uma vez, caso contrário incorrerá em erro. Curiosamente, por causa da tipagem dinâmica do JavaScript, recomenda-se declarar as variáveis sempre como const e, conforme a necessidade, muda-se para let (a seguir) ou var. Essa abordagem diminui a chance de mudanças de tipo que podem estragar a sua aplicação.

Já quando declaramos uma variável com ‘let’ na frente, queremos dizer que ela existe apenas no escopo atual. Ao contrário do var, que após sua declaração faz com que a variável exista para todo o código seguinte, ‘let’ garante um controle maior da memória e um ciclo de vida mais curto para as variáveis.

**Boa prática:** tente declarar suas variáveis como const. Se não puder, pois elas precisam ser alteradas, tente usar let. Se não puder usar let, porque a variável precisa ser usada fora do seu escopo (o que não é recomendado), use var.

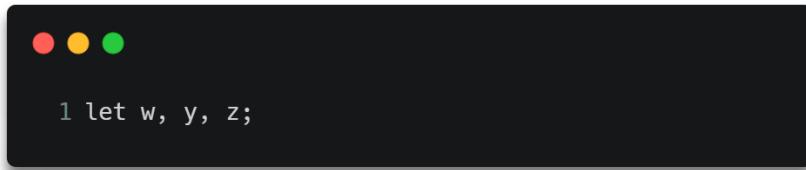
Por escopo entende-se o bloco (conjunto de comandos da linguagem) onde ocorreu a declaração da variável (geralmente delimitado por chaves).

Em JavaScript recomenda-se que a declaração de variáveis utilize nomes iniciados com letras minúsculas. Caso o nome seja composto de mais de uma palavras, as demais deveriam ser iniciadas com letras maiúsculas tal como nos exemplos:

- » contador
- » total
- » sinal
- » posicaoAbsoluta
- » valorMinimoDesejado
- » mediaGrupoTarefa2

E também é possível declarar mais de uma variável por vez:

Código 3.2: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 let w, y, z;
```

ou mesmo já atribuindo o seu valor, usando o operador de atribuição (sinal de igualdade):

Código 3.2: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 let a = 1;
```

## Tipos de dados

Ok, falei anteriormente que JavaScript possui tipagem dinâmica, certo? Ainda assim, existe tipagem, mesmo que fraca e inferida. Os tipos existentes em JavaScript são:

- » Numbers: podem ser com ou sem casas decimais, usando o ponto como separador. Ex: `let x = 10.1`
- » Strings: podem ser definidas com aspas simples ou duplas. Ex: `let nome = 'Luiz'`
- » Booleans: podem ter os valores literais `true` ou `false`, representando verdadeiro ou falso
- » Arrays: podem ter vários valores em seu interior
- » Objects: veremos mais pra frente, são tipos complexos que podem ter variáveis e funções em seu interior
- » Functions: veremos mais pra frente, são tipos complexos que recebem parâmetros, executam instruções e geram saídas

Não existem diferenças entre tipos primitivos e derivados em JavaScript, como existem em outras linguagens de programação. Todas variáveis são objetos, incluindo até mesmo funções e booleanos. O impacto disso será conhecido mais tarde, mas dois efeitos colaterais é que isso torna a

linguagem mais fácil de aprender e mais difícil de executar (mais lenta), em linhas gerais.

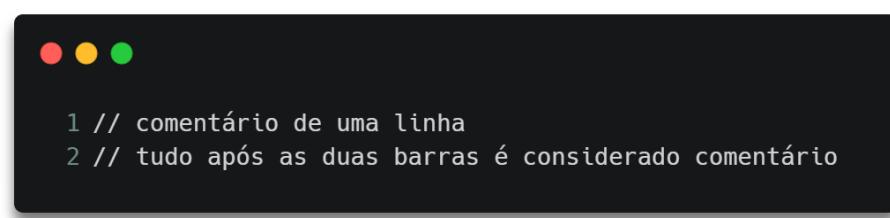
A tipagem dinâmica do JavaScript permite que você mude os tipos das variáveis em tempo de execução, embora isso não seja recomendado.

## Comentários

Comentários são trechos de texto, usualmente explicativos, inseridos dentro do programa de forma que não sejam considerados como parte do código, ou seja, são informações deixadas juntamente com o código para informação de quem programa.

O JavaScript aceita dois tipos de comentários: de uma linha e de múltiplas linhas. O primeiro de uma linha utiliza duas barras (//) para marcar seu início:

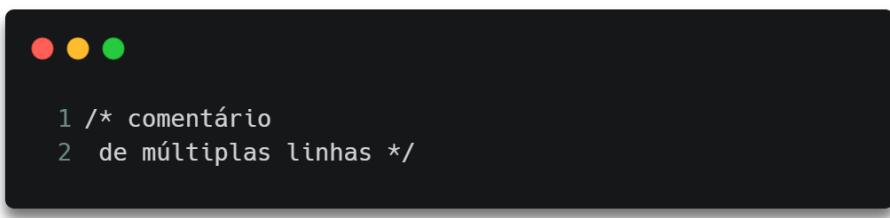
Código 3.2: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 // comentário de uma linha
2 // tudo após as duas barras é considerado comentário
```

O segundo usa a combinação /\* e \*/ para delimitar uma ou mais linhas de comentários:

Código 3.2: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 /* comentário
2 de múltiplas linhas */
```

Mas porque estou falando de comentários agora, logo após falar de declaração de variáveis?

Porque uma excelente maneira de melhorar os seus estudos é, a cada trecho de código escrito, comentar do que se trata a respectiva linha de código!

Conforme você for avançando e pegando experiência com JavaScript, não precisará usar mais tantos comentários, provavelmente apenas em blocos de código complexos, como forma de documentação para uso futuro. Por ora, comente tudo o que achar conveniente para seus estudos.

## Operadores

A linguagem JavaScript oferece um conjunto bastante amplo de operadores destinados a realização de operações aritméticas, lógicas, relacionais e de atribuição.

### OPERADORES ARITMÉTICOS

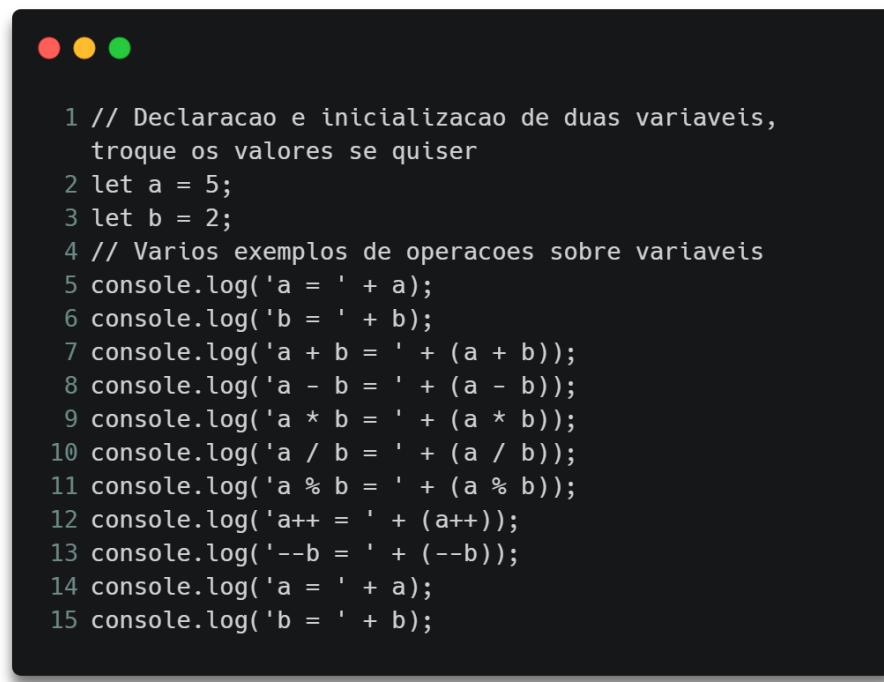
Como na maioria das linguagens de programação, o JavaScript possui vários operadores aritméticos. Considere nos exemplos que `a` e `b` são variáveis numéricas.

Operador	Significado	Exemplo
<code>+</code>	Adição, soma dois valores	<code>a + b</code>
<code>-</code>	Subtração, subtrai um valor de outro	<code>a - b</code>
<code>*</code>	Multiplicação, multiplica dois valores	<code>a * b</code>
<code>/</code>	Divisão, divide um valor pelo outro e retorna o resultado da divisão	<code>a / b</code>
<code>%</code>	Módulo, divide um valor pelo outro e retorna o resto da divisão	<code>a % b</code>
<code>++</code>	Incremento unário, aumenta o valor em 1	<code>a++ ou ++a</code>
<code>--</code>	Decremento unário, diminui o valor em 1	<code>a-- ou --a</code>
<code>+=, -=, *=, /=</code>	Auto-adição, auto-subtração, etc. Calcula o valor atual com o valor à direita e armazena o resultado em si mesmo.	<code>a += b</code> (soma <code>a</code> e <code>b</code> e guarda o resultado em <code>a</code> , equivalente a ' <code>a = a + b</code> '')

Estes operadores aritméticos podem ser combinados para formar expressões, fazendo uso de parênteses para determinar a ordem específica de avaliação de cada expressão.

A seguir um exemplo de aplicação que declara algumas variáveis, atribui valores iniciais e efetua algumas operações imprimindo os resultados obtidos.

Código 3.3: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
 1 // Declaracao e inicializacao de duas variaveis,
 2 // troque os valores se quiser
 3 let a = 5;
 4 let b = 2;
 5 // Varios exemplos de operacoes sobre variaveis
 6 console.log('a = ' + a);
 7 console.log('b = ' + b);
 8 console.log('a + b = ' + (a + b));
 9 console.log('a - b = ' + (a - b));
10 console.log('a * b = ' + (a * b));
11 console.log('a / b = ' + (a / b));
12 console.log('a % b = ' + (a % b));
13 console.log('a++ = ' + (a++));
14 console.log('--b = ' + (--b));
15 console.log('a = ' + a);
16 console.log('b = ' + b);
```

Executando o código fornecido (basta apertar F5 no VS Code ou Debug > Start Debugging) teremos os resultados 5, 2, 7, 3, 10, 2.5, 1, 5, 1, 6 e 1.

## OPERADORES RELACIONAIS

Além dos operadores aritméticos o JavaScript possui operadores relacionais, isto é, operadores que permitem comparar valores literais, variáveis ou o resultado de expressões retornando um resultado do tipo lógico, ou seja, um resultado falso ou verdadeiro. Os operadores relacionais disponíveis são:

Operador	Significado	Exemplo
<code>==</code>	Similaridade, retorna true se os valores forem iguais, independente do seu tipo	<code>a == b</code>
<code>!=</code>	Não similaridade, retorna true se os valores não forem iguais, independente do seu tipo	<code>a != b</code>
<code>===</code>	Igualdade, retorna true se os valores e tipos forem iguais	<code>a === b</code>
<code>!==</code>	Desigualdade, retorna true se os valores ou os tipos forem diferentes	<code>a !== b</code>
<code>&gt;</code>	Maior que, retorna true se o valor da esquerda for maior que o da direita	<code>a &gt; b</code>
<code>&gt;=</code>	Maior ou igual que, retorna true se o valor da esquerda for maior ou igual ao da direita	<code>a &gt;= b</code>
<code>&lt;</code>	Menor que, retorna true se o valor da esquerda for menor que o da direita	<code>a &lt; b</code>
<code>&lt;=</code>	Menor ou igual que, retorna true se o valor da esquerda for menor ou igual que o da direita	<code>a &lt;= b</code>

Note que temos diferentes usos para o sinal de igualdade (`=`). Se usarmos ele em duplas (`==`), comparamos valores de variáveis, independente do seu tipo. Assim ‘1’ `==` 1 é verdadeiro, pois ambos são o número 1. Por causa dessa interpretação do JavaScript, recomenda-se sempre utilizar a igualdade em trios (`===`), que além de comparar o valor entre as variáveis, também compara o seu tipo, para ver se coincidem. Em ambos os casos, não confunda com o uso isolado do sinal de igualdade (`=`), que é o operador de atribuição, que falarei mais tarde. Esse erro é um dos mais comuns por programadores JS desatentos.

**Boa prática:** use sempre `===` e `!==` para comparações. Diminui a chance de ‘falsos positivos’ e se tem um pequeno ganho de performance entre algumas comparações.

O operador de desigualdade é semelhante ao existente na linguagem C, ou seja, é representado por “`!=`” ou “`!==`” na versão mais ‘completa’. Os demais são idênticos a grande maioria das linguagens de programação em uso.

É importante ressaltar também que os operadores relacionais duplos, isto é, aqueles definidos através de dois caracteres ( $\geq$  por exemplo), não podem conter espaços em branco.

A seguir um outro exemplo simples de aplicação envolvendo os operadores relacionais.

Código 3.4: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 let a = 15;
2 let b = 12;
3 let c = '15';
4 console.log('a = ' + a);
5 console.log('b = ' + b);
6 // aqui a igualdade é apenas sobre o valor, mas são valores diferentes
7 console.log('a == b : ' + (a == b));
8 // aqui, o interpretador acha que são iguais, pois o valor é o mesmo, mas com tipos diferentes
9 console.log('a == c : ' + (a == c));
10 // agora sim, valida-se o tipo primeiro e depois o valor
11 console.log('a === c : ' + (a === c));
12 // ou seja, use sempre === e !==
13 console.log('a !== b : ' + (a !== b));
14 console.log('a < b : ' + (a < b));
15 console.log('a > b : ' + (a > b));
16 console.log('a <= b : ' + (a <= b));
17 console.log('a >= b : ' + (a >= b));
```

## OPERADORES LÓGICOS

Como seria esperado o JavaScript também possui operadores lógicos, isto é, operadores que permitem conectar logicamente o resultado de diferentes expressões aritméticas ou relacionais construindo assim uma expressão resultante composta de várias partes e portanto mais complexa.

Nos exemplos abaixo, a e b não são necessariamente variáveis, mas sim condições booleanas (true ou false), ou seja, podem ser qualquer tipo de expressões que retornem true ou false, incluindo variáveis booleanas simples.

Operador	Significado	Exemplo
&&	E lógico (AND), retorna true se ambas expressões retornarem true	a && b
	Ou lógico (OR), retorna true se ao menos uma expressão retornar true	a    b
!	Negação lógica (NOT), retorna o valor oposto ao da expressão ou uma expressão retornar true	!a

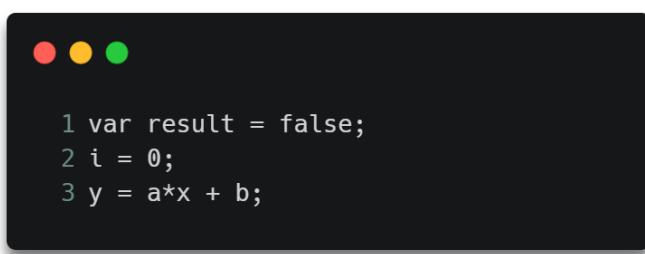
Os operadores lógicos duplos, isto é, definidos por dois caracteres, também não podem conter espaços em branco.

## OPERADOR DE ATRIBUIÇÃO

Atribuição é a operação que permite definir o valor de uma variável através de um literal ou através do resultado de uma expressão envolvendo operações diversas. Geralmente lemos a expressão ‘a = b’ como ‘a recebe b’ e ‘a = 1’ como ‘a recebe 1’. Note que isto é diferente de ‘a == b’ e ‘a == 1’, que lemos como ‘a é igual a b’ e ‘a é igual a 1’ (o mesmo para a === b e a === 1 como vimos anteriormente).

Exemplos de atribuições válidas são mostradas no trecho de código abaixo.

Código 3.5: disponível em <http://www.luiztools.com.br/ebook-frontend>



```

1 var result = false;
2 i = 0;
3 y = a*x + b;

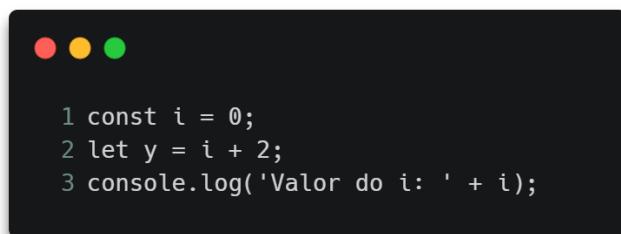
```

Note como podemos fazer uma atribuição no momento que declaramos a variável, para inicializá-la com algum valor. Também podemos atribuir valores mais tarde, com valores literais ou como resultado de expressões. No entanto, se apenas declaramos uma variável e não assumimos um valor à ela, ela será considerada undefined (não definida) pelo interpretador JS. Que tal exercitar o aprendizado de

**Boa prática:** sempre initialize suas variáveis no momento da declaração, o que costuma melhorar a performance dos otimizadores de código JS presentes no V8, que é o motor de JS usado pelo Node.js e também evita erros futuros por usar variáveis não inicializadas.

operadores? Modifique os valores abaixo à vontade:

Código 3.6: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
● ● ●

1 const i = 0;
2 let y = i + 2;
3 console.log('Valor do i: ' + i);
```

Realize testes ao menos uma vez com cada operador, para entender na prática como eles funcionam. Se não conseguir realizar testes com os operadores lógicos, não tem problema, revisitaremos eles em breve.

## Functions

Outro tipo de variável existente em JavaScript são as functions ou funções. São objetos especiais que, quando invocados, executam uma série de comandos a partir de parâmetros, produzindo saídas. Functions são extremamente úteis para reaproveitamento de código e organização de projetos. Se uma mesma lógica de programação precisa ser usada em mais de um ponto do seu código, ao invés de repetir os mesmos comandos JavaScript, você pode criar uma função com eles e chamar a função as duas vezes.

A própria linguagem JavaScript possui diversas funções globais que estão à disposição do programador, bem como outras funções que estão atreladas a tipos específicos de dados como Strings e Arrays, que veremos mais adiante.

Em JS as funções são tratadas como objetos de primeiro nível, diferentemente de outras linguagens. Uma vez definidas em um arquivo, esta função torna-se pública para aquele arquivo e pode-se com as configurações adequadas torná-la disponível em outros arquivos. Sendo o JS uma linguagem procedural, uma function somente existe depois

que o interpretador JS leu ela em algum arquivo ou bloco de código. Chamar uma função JS que ainda não foi carregada para a memória do navegador retornará undefined.

A sintaxe não é muito complicada: basicamente temos a palavra reservada function que define que o trecho de código a seguir refere-se à uma função. Damos um nome à função (seguindo as mesmas regras de nomenclatura de variáveis) e depois definimos os seus parâmetros entre parênteses, sendo que precisamos apenas dos nomes dos mesmos. Como abaixo:

Código 3.7: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
● ● ●  
1 function somar(num1, num2){ return num1 + num2; }
```

Note que neste exemplo usamos a instrução return para retornar o valor da soma dos dois números, mas isso é opcional. Funções JS não necessariamente precisam retornar algum valor, assim como também não precisam ter parâmetros.

Functions JS são tratadas como objetos, assim como as variáveis. Dessa forma, podemos guardar functions em variáveis para serem usadas depois ou até mesmo passá-las por parâmetro. Como abaixo:

Código 3.7: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
● ● ●  
1 const somar = function(num1, num2) { return num1 + num2; }
```

Mais tarde, no mesmo arquivo, podemos usar esta função chamando:

Código 3.7: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
● ● ●  
1 somar(1,2);
```

Ou passá-la por parâmetro como abaixo:

Código 3.7: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 console.log(somar(1,2));
```

Em versões mais recentes de JavaScript, adicionou-se a notação “arrow functions”, que permitem escrever as mesmas funções mas de maneira mais sucinta e direta, como no método somar abaixo:

Código 3.7: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 const somar = (num1,num2) => num1 + num2;
```

É bem útil para agilizar a leitura e a escrita de funções pequenas, principalmente quando precisamos definir elas de maneira anônima na passagem de parâmetro para outras funções.

Diferente, não?!

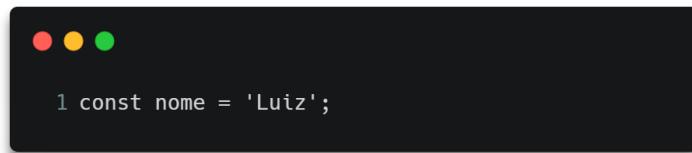
Note que nos exemplos que declarei a function como uma variável, eu usei const na declaração. Isso porque essa function não será alterada, é uma constante.

## O tipo String

Existem alguns tipos de dados em JavaScript que são mais complexos que outros. Na curva de aprendizado, logo após os numéricos e booleanos temos as Strings. O tipo String é um dos mais utilizados, isso porque ele é utilizado para criar variáveis que guardam texto dentro de si, algo muito recorrente em desenvolvimento de software.

O tipo String permite armazenar um número virtualmente infinito de caracteres, formando palavras, frases, textos, etc. Apesar de todo esse “poder”, sua declaração é tão simples quanto a de qualquer outra variável:

Código 3.8: disponível em <http://www.luiztools.com.br/ebook-frontend>



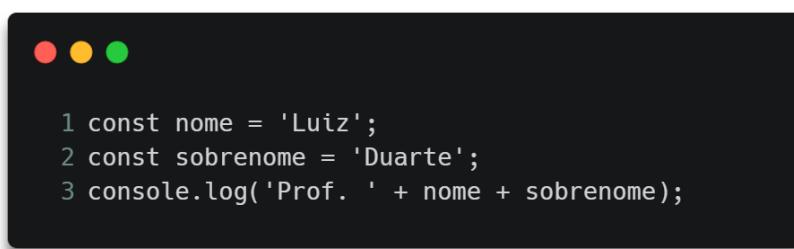
```
1 const nome = 'Luiz';
```

Note que a palavra ‘Luiz’ está entre aspas (simples) e que poderia estar entre aspas-duplas, sem qualquer diferença. No entanto, se você começou a string com aspas simples, deve terminar da mesma forma e vice-versa.

O ideal é sempre inicializarmos nossas Strings, nem que seja com aspas vazias (”) para não corrermos o risco de termos erros de execução mais pra frente, pois por padrão o conteúdo das variáveis é ‘undefined’.

Assim como vimos anteriormente na prática, quando usamos o operador ‘+’ com textos (que agora chamaremos de Strings), ele não soma os seus valores, mas sim concatena (junta) eles. Isso vale tanto para Strings literais (textos entre aspas) quanto para variáveis Strings.

Código 3.8: disponível em <http://www.luiztools.com.br/ebook-frontend>

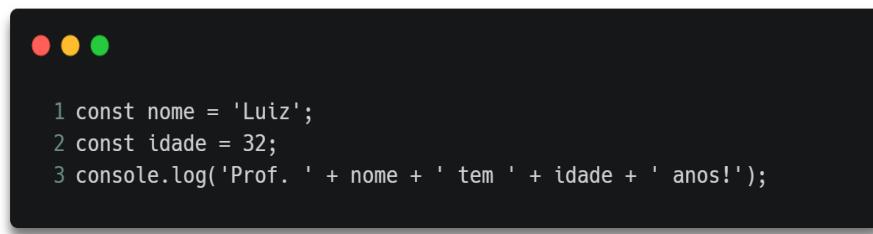


```
1 const nome = 'Luiz';
2 const sobrenome = 'Duarte';
3 console.log('Prof. ' + nome + sobrenome);
```

O código acima, se executado pelo terminal interativo ou dentro de um index.js deve imprimir o texto Prof. LuizDuarte na janela do console. Isso porque ele concatenou (juntou) a String literal “Prof. ” com as variáveis nome e sobrenome, resultando em uma nova String, maior e mais completa “Prof. LuizDuarte”, que foi impressa pelo console.log.

Outra característica muito interessante das Strings é que qualquer variável que a gente concatene com uma String, acaba gerando outra String, como abaixo.

Código 3.8: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
 1 const nome = 'Luiz';
 2 const idade = 32;
 3 console.log('Prof. ' + nome + ' tem ' + idade + ' anos!');
```

Isso vai imprimir o texto “Prof. Luiz tem 32 anos！”, copiando o valor da variável `idade` pra dentro da nova String completa. Note que isso não altera o tipo da variável `idade` ou seu valor, apenas faz uma cópia dela em forma de texto, automaticamente.

Legal, não é mesmo?!

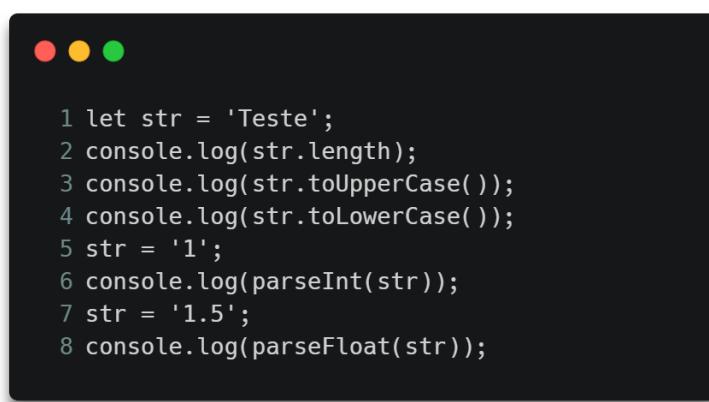
Mas as Strings são bem mais complexas e interessantes que isso, existindo uma série de características e funções que podemos chamar a partir delas para obter variados efeitos, como nos exemplos abaixo (considere `str` uma variável inicializada com uma String):

- » `str.length`: retorna a quantidade de caracteres da String
- » `str.toUpperCase()`: retorna a String toda em maiúsculas
- » `str.toLowerCase()`: retorna a String toda em minúsculas
- » `str.endsWith('x')`: retorna true se a String termina com o caracter (ou palavra) passado por parâmetro
- » `str.startsWith('x')`: retorna true se a String começa com o caracter (ou palavra) passado por parâmetro
- » `str.replace('x', 'y')`: retorna uma String cópia da original, mas com os caracteres `x` substituídos por `y`
- » `str.trim()`: retorna uma String cópia da original, mas sem espaços no início e no fim
- » `str.concat(str2)`: faz a fusão dos caracteres de `str2` no final de `str`
- » `parseInt(str)`: retorna um número a partir da String (se for possível)
- » `parseFloat(str)`: retorna um número decimal a partir da String (se for possível)
- » `x.toString()`: retorna uma String criada a partir de outra variável qualquer (`x`)

Note que nenhuma dessas funções muda a String em si, todas retornam um valor a partir dela. Strings são imutáveis na maioria das linguagens de programação, e JavaScript não é exceção. Quando colocamos um texto dentro de uma variável que já possuía um texto, não mudamos o texto original, mas criamos um novo.

Para exemplificar estes conceitos, escreva e execute o código abaixo:

Código 3.8: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 let str = 'Teste';
2 console.log(str.length);
3 console.log(str.toUpperCase());
4 console.log(str.toLowerCase());
5 str = '1';
6 console.log(parseInt(str));
7 str = '1.5';
8 console.log(parseFloat(str));
```

E as Strings possuem muitas outras características incríveis devido à sua natureza textual, que é interpretada internamente como um array (vetor) de caracteres. Exploraremos essas características mais pra frente, quando entrarmos em estruturas de dados mais complexas.

Está começando a ficar mais interessante!

## Estruturas de Controle de Fluxo

Um programa de computador é uma sequência de instruções organizadas de forma tal a produzir a solução de um determinado problema. Naturalmente tais instruções são executadas em sequência, o que se denomina fluxo sequencial de execução. Em inúmeras circunstâncias é necessário executar as instruções de um programa em uma ordem diferente da estritamente sequencial. Tais situações são caracterizadas pela necessidade da repetição de instruções individuais ou de grupos de instruções e também pelo desvio do fluxo de execução.

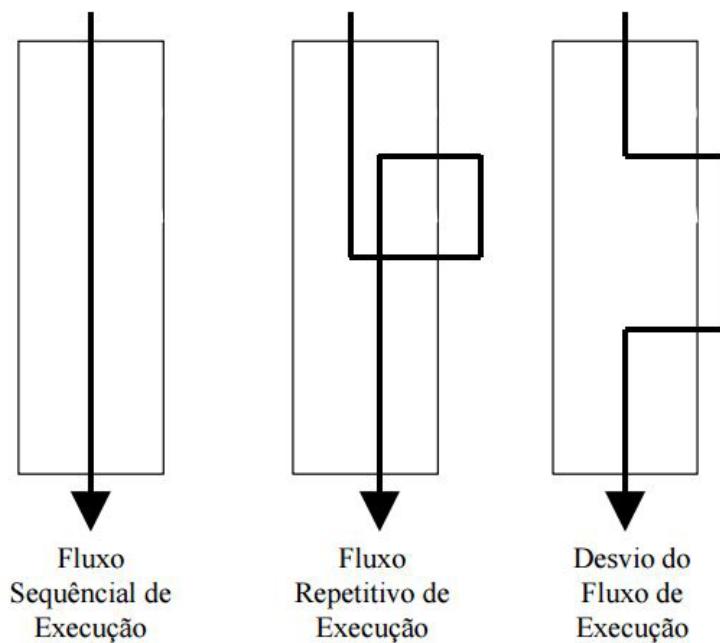
As linguagens de programação tipicamente possuem diversas estruturas de programação destinadas ao controle do fluxo de execução, isto é, estruturas que permitem a repetição e o desvio do fluxo de execução.

Geralmente as estruturas de controle de execução são divididas em:

**Estruturas de repetição:** destinadas a repetição de um ou mais comandos, criando o que se denomina laços. O número de repetições é definido por uma condição, que pode ser simples como uma comparação numérica, ou complicado como uma expressão lógica. No JavaScript dispõe-se das diretivas **for**, **while** e **do/while**.

**Estruturas de desvio de fluxo:** destinadas a desviar a execução do programa para uma outra parte, quebrando o fluxo sequencial de execução. O desvio do fluxo pode ocorrer condicionalmente, quando associado a avaliação de uma expressão, ou incondicionalmente. No JavaScript dispomos das diretivas **if/else** e **switch/case**.

A imagem abaixo ilustra os três tipos possíveis de fluxos de execução em um software: o sequencial, que seria o fluxo padrão; um fluxo repetitivo, onde temos um laço de repetição em determinado momento; e um fluxo com desvio, cuja “direção” é decidida com base em uma condição lógica.

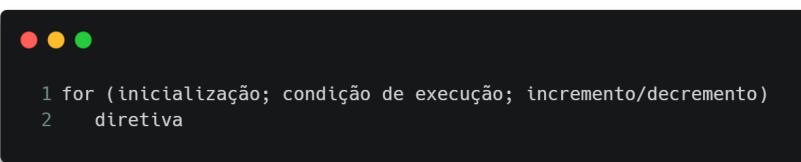


Agora vamos estudar cada um dos tipos de estruturas de controle básicas que possuímos no JavaScript.

## ESTRUTURAS DE REPETIÇÃO

Estas estruturas podem ser divididas entre simples e condicionais. Independente da classificação, elas repetem um ou mais comandos JavaScript durante um número de vezes definido na sua declaração ou baseado em uma expressão lógica, como veremos a seguir.

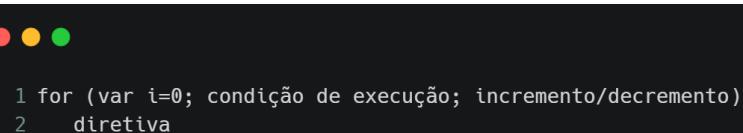
Como repetição simples consideramos um trecho de código, isto é, um conjunto de diretivas que deve ser repetido um número conhecido e fixo de vezes. A repetição é uma das tarefas mais comuns da programação utilizada para efetuarmos contagens, para obtenção de dados, para impressão etc. Em JavaScript dispomos da diretiva for cuja sintaxe é dada a seguir:



```
● ● ●
1 for (inicialização; condição de execução; incremento/decremento)
2   diretiva
```

O for (PARA) possui três campos ou seções, delimitados por um par de parênteses que efetuam o controle de repetição de uma diretiva individual ou de um bloco de diretivas (neste caso, circundado por chaves). Cada campo é separado do outro por um ponto e vírgula.

O primeiro campo é usado para dar valor inicial a uma variável de controle (um contador). Nele declaramos nossa variável de controle, geralmente um inteiro, e inicializamos ele, geralmente com 0, como no exemplo abaixo.



```
● ● ●
1 for (var i=0; condição de execução; incremento/decremento)
2   diretiva
```

O segundo campo é uma expressão lógica que determina a execução da diretiva associada ao for, geralmente utilizando a variável de controle e outros valores. Resumindo: o segundo campo vai determinar quantas vezes a diretiva do for será executada. Atente ao exemplo abaixo:

```
1 for (var i=0; i < 10; incremento/decremento)
2   diretiva
```

O que você consegue abstrair a partir do segundo campo do for?

Considerando que `i` é a nossa variável de controle, usamos ela em uma expressão lógica para determinar quantas vezes a diretiva irá ser executada. Neste caso, enquanto `i` for menor que 10, executaremos a diretiva (um comando JavaScript qualquer).

Mas se `i = 0`, quando que ele será maior ou igual à 10 para o for encerrar a repetição?

Aí que entra o terceiro campo do for: o incremento/decremento. Neste último campo nós mudamos o valor da variável de comando, para mais ou para menos. Esse incremento/decremento acontece uma vez a cada repetição do laço, logo após a execução da diretiva e antes da condição (segundo campo do for) ser analisada novamente. Isto faz com que, em determinado momento, o laço pare de se repetir, como no exemplo abaixo.

```
1 for (var i=0; i < 10; i++)
2   diretiva
```

Neste caso, a diretiva será executada 10 vezes e sequência, uma vez que a cada execução o valor de `i` aumentará em uma unidade até alcançar o valor 10, o que fará com que a condição '`i < 10`' se torne falsa e o for acabe sua execução.

Não acredita? Copie o código abaixo, onde substituí a diretiva por um comando de impressão do JavaScript e execute no VS Code (ou no terminal interativo):

Código 3.9: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
● ● ●

1 for (let i=0; i < 10; i++)
2   console.log('i=' + i);
```

A saída esperada é uma sequência de impressões iniciadas em ‘i=0’ e terminando em ‘i=9’, pois quando o i for igual a 10, o for será encerrado.

A tradução literal do for é PARA, e lemos o código anterior como “para uma dada variável i, inicializada com 0, até ela se tornar 10, imprima o valor de i e incremente-o logo após”.

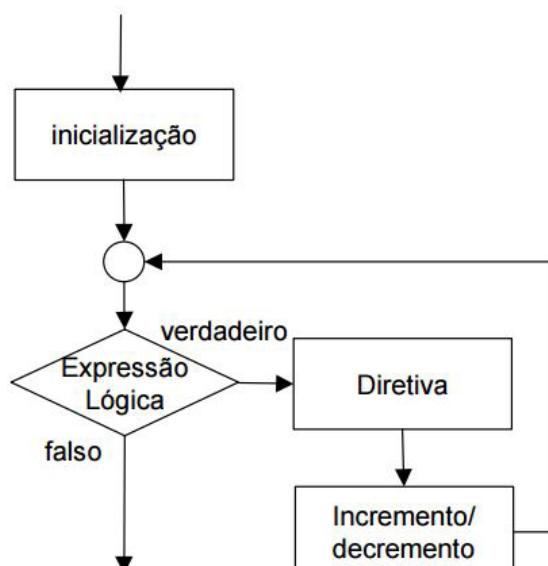
Caso quiséssemos executar mais de uma instrução dentro do for, repetidas vezes, basta apenas circundarmos elas com chaves, como no exemplo abaixo:

Código 3.9: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
● ● ●

1 for (let i=0; i < 10; i++)
2   {
3     console.log('i=' + i);
4   }
```

Podemos resumir o funcionamento da estrutura for através do fluxograma abaixo:

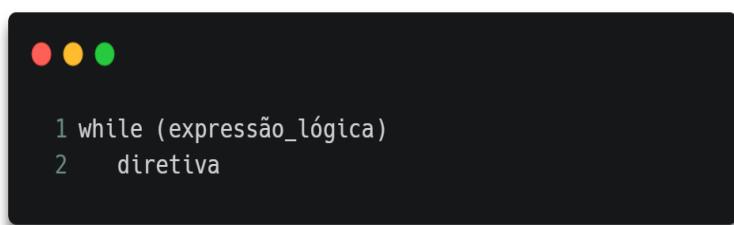


Mas e as estruturas de repetição mais complexas, baseadas em condições?

As estruturas de repetição condicionais são estruturas de repetição cujo controle de execução é feito pela avaliação de expressões condicionais. Estas estruturas são adequadas para permitir a execução repetida de um conjunto de diretivas por um número indeterminado de vezes, isto é, um número que não é conhecido durante a fase de programação mas que pode ser determinado durante a execução do programa tal como um valor a ser fornecido pelo usuário, obtido de um arquivo ou ainda de cálculos realizados com dados alimentados pelo usuário ou lido de arquivos.

Existem duas estruturas de repetição condicionais: **while** e **do/while**.

O **while** (ENQUANTO) é o que chamamos de laço condicional, isto é, um conjunto de instruções que é repetido enquanto o resultado de uma expressão lógica (uma condição) é avaliado como verdadeiro. Abaixo segue a sintaxe desta diretiva:

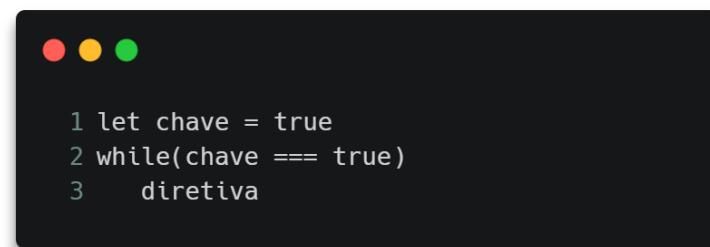


```
1 while (expressão_lógica)
2   diretiva
```

Note que a diretiva `while` avalia o resultado da expressão antes de executar a diretiva associada, assim é possível que a diretiva nunca seja executada caso a condição seja inicialmente falsa.

Mas que tipo de expressão lógica usamos geralmente como condição do `while`?

Abaixo temos um exemplo bem simples:

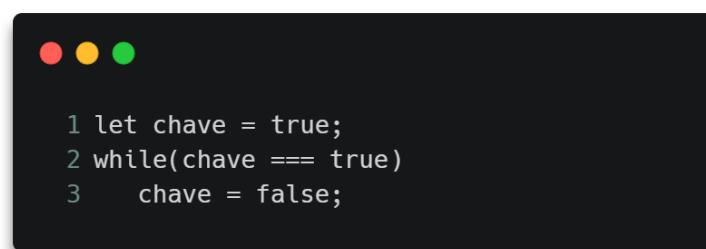


```
1 let chave = true
2 while(chave === true)
3   diretiva
```

Neste exemplo inicializamos uma variável chave como true (verdadeiro) e depois usamos a própria variável como condição de parada do while. Enquanto chave for true, o while continuará sendo executado (i.e. a diretiva será executada). No entanto, note que um problema típico relacionado a avaliação da condição da diretiva while é o seguinte: se a condição nunca se tornar falsa o laço será repetido indefinidamente.

Para que isso não aconteça, devemos nos certificar de que exista algo na diretiva que, em dado momento, modifique o valor da condição, como no exemplo abaixo:

Código 3.9: disponível em <http://www.luiztools.com.br/ebook-frontend>



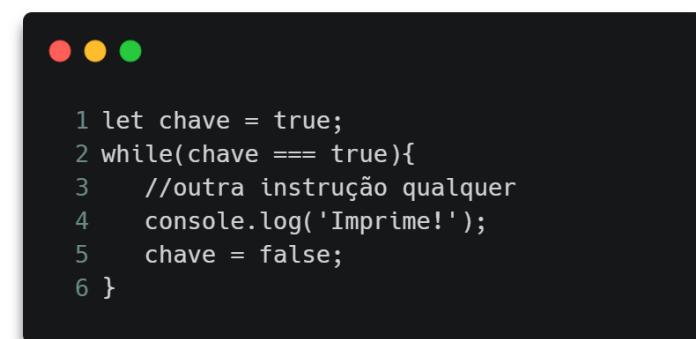
```
1 let chave = true;
2 while(chave === true)
3     chave = false;
```

Nesse caso, o while executará apenas uma vez e depois será encerrado, pois na segunda vez que a condição for analisada, a variável chave terá valor false e ele não executará sua diretiva novamente.

Lemos o trecho de código anterior como “enquanto a variável chave for verdadeira, troque o valor dela para false”.

Claro, esse é um exemplo pouco útil, uma vez que a única instrução existente é para encerrar o while. O que acontece geralmente é que o while possua mais de uma diretiva e, assim como for, elas devem ser circundadas por chaves:

Código 3.9: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 let chave = true;
2 while(chave === true){
3     //outra instrução qualquer
4     console.log('Imprime!');
5     chave = false;
6 }
```

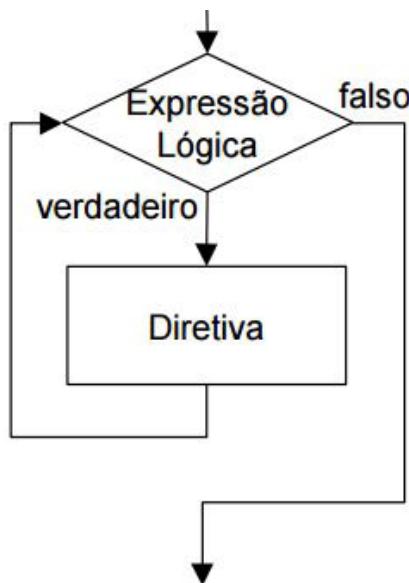
É possível ainda, usar o while de maneira semelhante ao for, definindo como condição uma comparação lógico-aritmética:

Código 3.9: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
 1 let i = 0;
 2 while(i < 10){
 3     //outra instrução qualquer
 4     console.log('Imprime!');
 5     i++;
 6 }
```

Nesse caso, o texto “Imprime!” será impresso 10 vezes, assim como seria possível fazer com um for.

O funcionamento do while pode ser resumido através do fluxograma abaixo:



O **do/while** também é um laço condicional, isto é, tal como o while é um conjunto de instruções que são repetidas enquanto o resultado da condição é avaliada como verdadeira mas, diferentemente do while, a diretiva associada é executada antes da avaliação da expressão lógica e assim temos que esta diretiva é executada pelo menos uma vez.

Por exemplo:

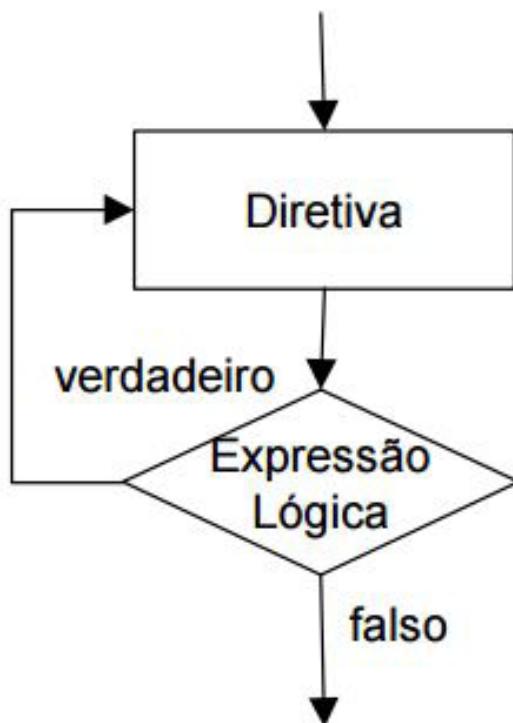
Código 3.10: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 let chave = false;
2 do{
3     //outra instrução qualquer
4     console.log('Imprime!');
5 }while(chave === true)
```

Note que a variável chave foi declarada como false logo na sua inicialização, ou seja, quando ela for comparada na condição do **while** ela ainda será false e consequentemente sairá do **while**. Porém, como o bloco **do** é executado primeiro, teremos a impressão da palavra “Imprime!” na saída do console.

Lemos o código anterior como: “faça (DO) a impressão da palavra imprime, enquanto (WHILE) a variável chave for verdadeira”.

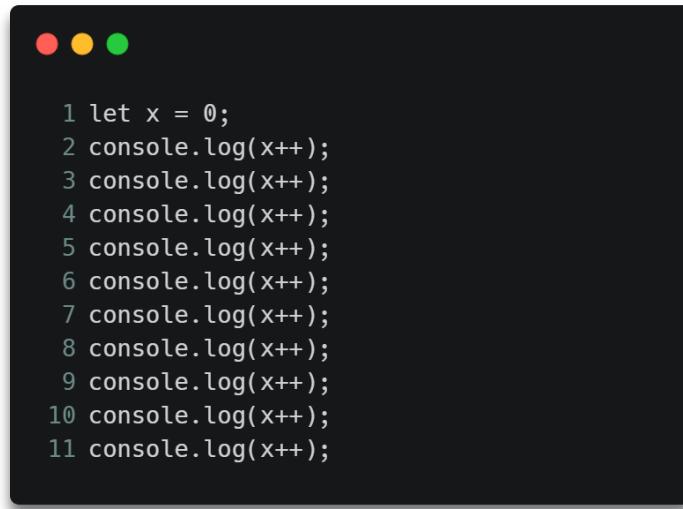
Basicamente esta é a única diferença entre os laços **while** e **do/while**. O **do/while** sempre executa o laço ao menos uma vez, antes de fazer a comparação, como ilustrado pelo fluxograma abaixo:



Um exercício que você pode fazer para testar o for, o while e o do/while é o seguinte: antes de conhecer os laços de repetição, como você faria para declarar uma variável e imprimi-la de 0 a 9?

O código abaixo exemplifica como você poderia fazer:

Código 3.10: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 let x = 0;
2 console.log(x++);
3 console.log(x++);
4 console.log(x++);
5 console.log(x++);
6 console.log(x++);
7 console.log(x++);
8 console.log(x++);
9 console.log(x++);
10 console.log(x++);
11 console.log(x++);
```

Copie e cole o código acima dentro de um index.js, execute e você verá, que graças ao operador de incremento unário a variável x vai do valor 0 ao 9, sendo impressa durante o processo.

Repetitivo, não?!

Seria tão bom se tivéssemos uma maneira de repetir comandos JavaScript sem repetir código... É nessas horas, quando você precisa repetir comandos, que os laços de repetição são úteis!

Pare um minuto e pense como esse mesmo exercício (imprimir um número de 0 a 9) poderia ser feito usando laços de repetição...

Aqui vai uma solução, agora usando um laço for:

Código 3.10: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 for(var i=0; i < 10; i++)
2   console.log(i);
```

Copie e cole esse código e verá que o resultado no console é o mesmo.

Muito menor, não?!

Como é uma tarefa repetitiva cuja condição de parada é numérica e os valores vão incrementando unitariamente, o for é a melhor opção. Mesmo assim, a título de exercício, vou mostrar implementações usando while:

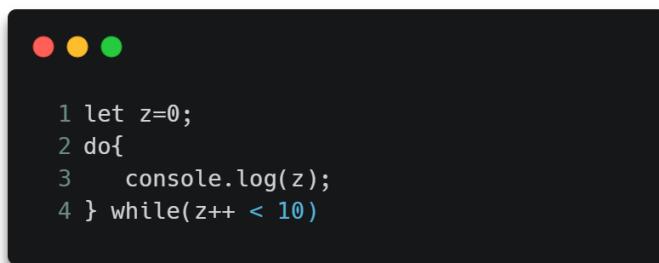
Código 3.10: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 let y=0;
2 while(y < 10)
3   console.log(y++);
```

E outra usando do/while:

Código 3.10: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 let z=0;
2 do{
3   console.log(z);
4 } while(z++ < 10)
```

Ambos exemplos fazem exatamente a mesma coisa. Estudar algoritmos de programação é exatamente isso: entender que diversos caminhos levam à mesma solução. Cabe ao programador identificar qual o melhor, neste caso, o for.

Um último ponto digno de nota é a forma como usei o incremento unário nestes exemplos. Note que incrementei o valor de x dentro de um console.log e dentro de uma estrutura while. Isso é perfeitamente possível e lhe permite estratégias interessantes em seus algoritmos. Temos dois operadores de incremento unário, o `++x` e o `x++`. O primeiro aumenta o valor de x em 1 e depois retorna o valor do mesmo, enquanto que o segundo retorna o valor de x e depois o aumenta em 1. Pode parecer uma pequena diferença, mas analise esse código:

Código 3.11: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 let x=0;
2 console.log(x++);
```

O que será impresso?

A resposta é 0.

Já neste exemplo...

Código 3.11: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 let y=0;
2 console.log(++y);
```

A resposta é 1. Capicce?

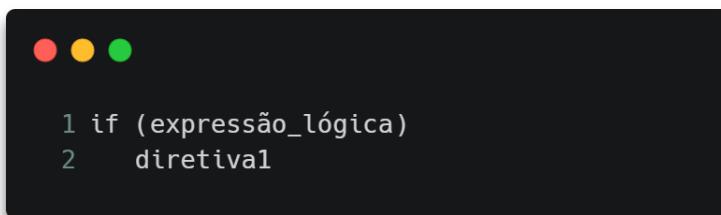
Então vamos passar à próxima estrutura, temos muito pra ver ainda dentro do básico de JavaScript!

## ESTRUTURAS DE DESVIO DE FLUXO

Existem várias estruturas de desvio de fluxo que podem provocar a modificação da maneira com que as diretivas de um programa são executadas conforme a avaliação de uma condição. O JavaScript dispõe de duas destas estruturas: **if/else** e **switch/case**.

O **if/else** é uma estrutura simples de desvio de fluxo de execução, isto é, é uma diretiva que permite a seleção entre dois caminhos distintos para execução dependendo do resultado falso ou verdadeiro resultante de uma expressão lógica.

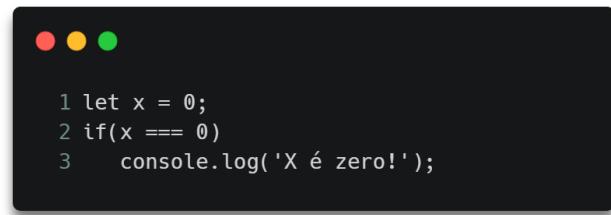
Vamos começar do **if** básico:



```
1 if (expressão_lógica)
2   diretiva1
```

A tradução literal de um if é SE. SE a expressão entre parênteses for verdade (true), então a diretiva1 será executada. Caso contrário, ela é ignorada. Exemplo:

Código 3.12: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 let x = 0;
2 if(x === 0)
3   console.log('X é zero!');
```

O que acontece quando executamos esse código?

E se alterarmos o valor inicial de x (na primeira linha), fazendo ele receber 1 ao invés de 0?

A frase “X é zero” somente será impressa se a variável x for zero. Em qualquer outra circunstância, não.

Note que, assim como acontece nos laços de repetição, se vamos executar mais de uma diretiva, devemos agrupá-las dentro de chaves, como no exemplo abaixo, levemente modificado do anterior.

Código 3.12: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 let x = 0;
2 if(x === 0){
3     console.log('X é zero!');
4     x++;
5 }
```

Neste exemplo, como queremos executar dois comandos no caso do x ser zero, devemos circundar os comandos com chaves. Podemos ler esta estrutura como “se a variável x for igual a zero, imprimimos que X é zero e depois incrementamos o mesmo”.

Note também que a expressão lógica que vai dentro dos parênteses do if aceitam qualquer expressão que retorne true ou false, usando os operadores lógicos ( $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $==$ ,  $\neq$  e  $!$ ). Sendo assim, podemos fazer coisas muito mais complexas com um if do que apenas uma comparação de igualdade, como no exemplo abaixo, onde vemos se um número é PAR (isto é, divisível por 2):

Código 3.12: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 let y = 0;
2 if(y % 2 === 0){
3     console.log('Y é PAR!');
4 }
```

Você conhece o operador de módulo (%)?

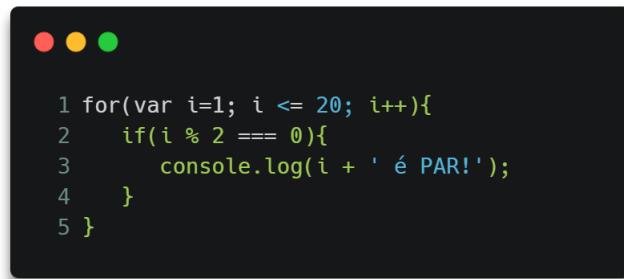
Falamos dele brevemente no tópico sobre operadores, mas só agora que estou mostrando uma utilidade dele: descobrir se um número é par ou não. O funcionamento do mod (apelido popular do % entre programadores) é fazer a divisão de um número pelo outro, mas ao invés de retornar o resultado, ele retorna o resto.

Ora, se dividirmos um número por 2 ( $x / 2$ ) e o resto for zero, é porquê foi uma divisão exata e ele é PAR, certo?

E agora, unindo o conceito do if e do for, como você faria para imprimir todos os números pares de 1 a 20?

Tente fazer, e depois compare a sua solução com a apresentada abaixo:

Código 3.12: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 for(var i=1; i <= 20; i++){
2     if(i % 2 === 0){
3         console.log(i + ' é PAR!');
4     }
5 }
```

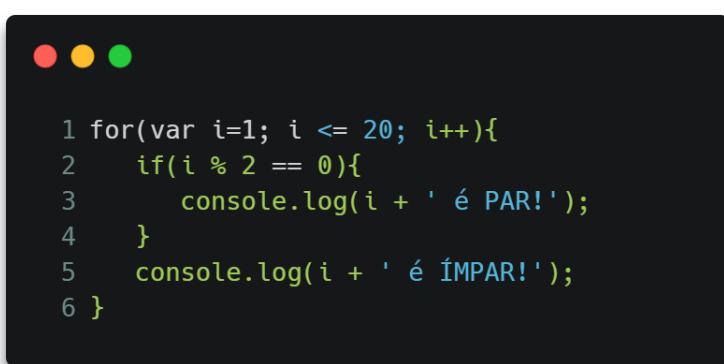
Este **for** que criei no exemplo anterior está ligeiramente diferente. Primeiro, eu initializei a variável de controle com 1 ao invés do 0 tradicional. Além disso, coloquei na expressão de controle o operador ‘menor ou igual que’, comparando com o número 20, para que o 20 em si seja impresso também. Já o **if** eu aproveitei do exemplo anterior, então nenhuma novidade aqui.

Copie e cole este código e veja como ele imprime corretamente somente os números pares de 1 a 20.

Mas e se quiséssemos imprimir a mensagem “é ÍMPAR” para os demais números?

Como você faria?

Código 3.12: disponível em <http://www.luiztools.com.br/ebook-frontend>

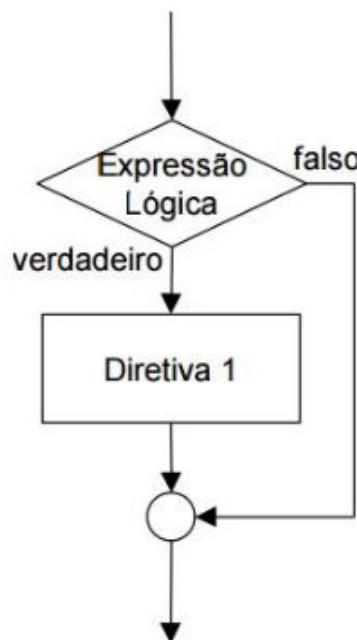


```
1 for(var i=1; i <= 20; i++){
2     if(i % 2 == 0){
3         console.log(i + ' é PAR!');
4     }
5     console.log(i + ' é ÍMPAR!');
6 }
```

Essa talvez tenha sido uma solução pensada por você, mas está errada. Execute ela dentro de um arquivo index.js e notará que ela sempre imprime que todos os números são ÍMPARES, além de imprimir quando algum é PAR também.

Porquê?

O **if** é um desvio de execução, mas ao término da execução da sua diretiva, ele volta ao fluxo tradicional, como o fluxograma abaixo nos mostra.



Uma vez que a impressão de que o número é ÍMPAR está no fluxo normal, ela sempre será executada após o **if**.

Mas então, como resolvemos isso?

Com a segunda parte da construção **if/else**, o **else**.

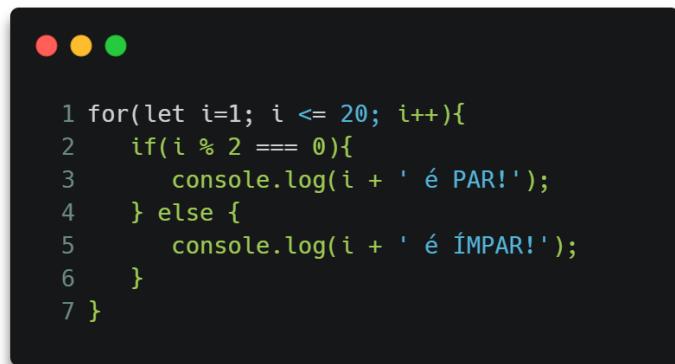
A tradução literal de **else** é SENÃO, e ele define basicamente o caminho alternativo que seu programa tomará caso a condição do **if** não seja verdadeira, como abaixo.

```
1 if (expressão_lógica)
2   diretiva1
3 else
4   diretiva2
```

Lemos essa estrutura como: “se a expressão lógica for verdadeira, executamos a diretiva 1, senão, executamos a diretiva 2”.

Dessa forma, conseguimos modificar nosso programa anterior para imprimir que o número é ÍMPAR, quando ele não entrar no if que diz que ele é PAR, como abaixo, usando else.

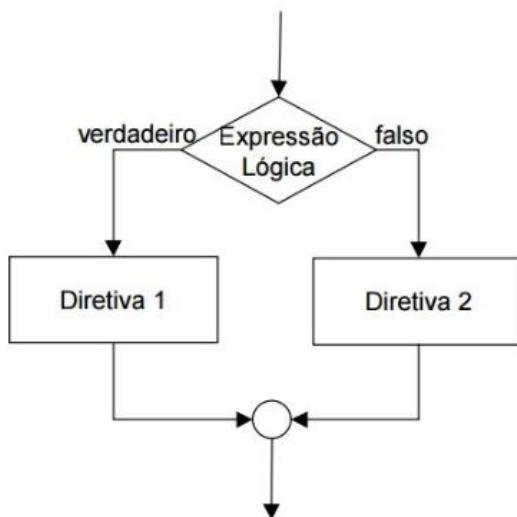
Código 3.13: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 for(let i=1; i <= 20; i++){
2   if(i % 2 === 0){
3     console.log(i + ' é PAR!');
4   } else {
5     console.log(i + ' é ÍMPAR!');
6   }
7 }
```

Lemos o trecho mais interno do algoritmo acima como: “se  $i$  for divisível por 2, então imprima que ele é PAR, senão, imprime que ele é ímpar”.

O fluxograma que melhor define o comportamento de uma estrutura if/else como essa é visto abaixo.



Neste modelo simples de if/else e no anterior, com apenas if, consideramos que há sempre um ou no máximo dois caminhos alternativos para um dado fluxo de execução. No entanto, existem fluxos ainda mais complexos, em que podemos precisar de um encadeamento de execuções. Como no caso abaixo, em que queremos que a palavra “FUNCIONOU” seja impressa somente quando o número estiver entre 5 e 7.

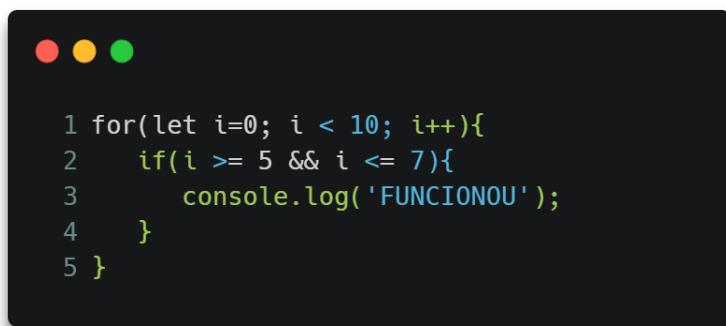
Código 3.13: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 for(let i=0; i < 10; i++){
2     if(i >= 5){
3         if(i <= 7){
4             console.log('FUNCIONOU');
5         }
6     }
7 }
```

Alguns problemas lógicos como esse são facilmente resolvidos apenas estudando e combinando os operadores lógico e relacionais, como no caso abaixo, que faz exatamente a mesma coisa que o algoritmo anterior, mas de maneira mais eficiente.

Código 3.13: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 for(let i=0; i < 10; i++){
2     if(i >= 5 && i <= 7){
3         console.log('FUNCIONOU');
4     }
5 }
```

Ou seja, somente entrará no if se ambas condições forem verdadeiras, uma vez que usei o operador AND (E lógico). O estudo dos operadores lhe abre um leque enorme de opções para otimizar o seu código e conseguir ‘mais’ com ‘menos’.

Para finalizar o estudo do **if/else**, cabe ressaltar que muitas vezes também precisamos que nossos **else**’s também possuam condições, que o algoritmo não entre dentro do else simplesmente porque não atendeu à condição do **if**. Neste caso, podemos adicionar um **if** ao **else**, o que chamamos de **else if**.



```
1 if(x === 1){  
2     diretiva1  
3 } else if(x === 2){  
4     diretiva2  
5 }
```

Lemos esta estrutura como: “se x é igual a 1, executamos a diretiva 1, senão, se x for igual a 2, executamos a diretiva 2”.

Neste caso, se o x for igual a 1, executaremos a primeira diretiva. Senão, caso o x seja igual a 2, executaremos a segunda diretiva. Agora, se x não for 1 ou 2, ele passará ‘reto’ pelo if, apenas usando o fluxo principal da execução.

Podemos ainda criar um fluxo default (padrão), colocando um (e apenas um) **else** no final da cadeia de **ifs** e **else ifs**.



```
1 if(x === 1){  
2     diretiva1  
3 } else if(x === 2){  
4     diretiva2  
5 } else {  
6     diretiva3  
7 }
```

Neste caso, se o x não for 1, nem 2, executaremos a diretiva3.

A construção **if/else** é muito poderosa e permite múltiplos fluxos de execução alternativos, uma vez que podemos combinar inúmeras condições diferentes.

**Boa prática:** tentar manipular variáveis ‘*undefined*’ provoca erros em tempo de execução. Sempre que quiser saber se uma variável está *undefined* ou não, basta fazer um if sobre ela, pois variáveis *undefined* retorna *false* em ifs. Como no exemplo a seguir:  
*if (!x) { console.log('x is undefined') }*

No entanto, se suas comparações são sempre de uma variável em relação a um valor, existe uma estrutura mais eficiente para isso: a switch/case.

O switch/case é uma diretiva de desvio múltiplo de fluxo, isto é, baseado na avaliação de uma variável é escolhido um caminho de execução dentre vários possíveis. O switch/case equivale logicamente a um conjunto de diretivas if encadeadas, embora seja usualmente mais eficiente durante a execução.

A sintaxe desta diretiva é a seguinte:

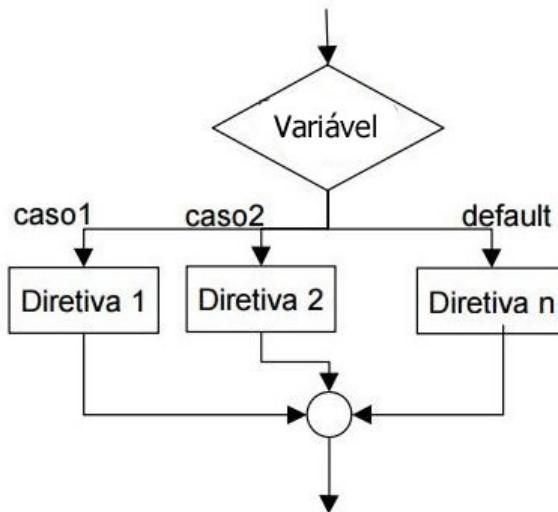
```
1 switch (variavel) {  
2     case valor1: diretiva1  
3         break  
4     case valor2: diretiva2  
5         break  
6     default: diretiva_default  
7 }
```

As diretivas encontradas a partir do caso (**case**) escolhido são executadas até o final da diretiva **switch** ou até uma diretiva **break** que encerra o **switch/case**. Se o valor resultante não possuir um caso específico, é executado a diretiva default colocada, opcionalmente, ao final da diretiva switch/case.

O mesmo efeito obtido pela construção switch/case mostrada antes também pode ser realizado (para fins didáticos) apenas com ifs e elses:

```
1 if(variavel === valor1)  
2     diretiva1  
3 else if(variavel === valor2)  
4     diretiva2  
5 else  
6     diretiva_default
```

Em um fluxograma, temos a estrutura switch/case representada da seguinte maneira:



Note que o ponto de início de execução é um caso (case) cujo valor é aquele da variável avaliada. Após iniciada a execução do conjunto de diretivas identificadas por um certo caso, tais ações só são interrompidas com a execução de uma diretiva break ou com o final da diretiva switch.

A seguir temos uma aplicação simples que exemplifica a utilização das diretivas switch/case e break. Como de costume, sugiro criar um novo arquivo index.js dentro de uma pasta ExemploSwitch e abri-la no VS Code para trabalhar melhor:

Código 3.13: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 let letra = 'A';
2 switch (letra) {
3     case 'A': console.log('Vogal A'); break;
4     case 'E': console.log('Vogal E'); break;
5     case 'I': console.log('Vogal I'); break;
6     case 'O': console.log('Vogal O'); break;
7     case 'U': console.log('Vogal U'); break;
8     default: console.log('Não é uma vogal');
9 }
```

Note que neste exemplo eu usei ‘;’ (ponto-e-vírgula) logo após a impressão no console. Isso porque eu quis usar outra diretiva na mesma linha, a break, o que não seria permitido caso eu não usasse o ‘;’. A quebra de linha automaticamente é entendida como uma nova instrução, porém nos casos em que queremos colocar mais de uma instrução na mesma linha, devemos usar o ‘;’, sem limite de instruções que eu poderia colocar na mesma linha.

Troque o valor da variável letra e veja os resultados no console. A saída esperada depende do valor que você colocar na variável ‘letra’, declarada logo no início do algoritmo.

Podemos ler a estrutura **switch/case** anterior da seguinte forma: “considerando a variável letra, caso seja A, imprima ‘Vogal A’, caso seja B, imprima ‘Vogal B’...etc”.

## Arrays

Aprendemos nos tópicos anteriores como declarar variáveis e manipular os valores delas, certo?

Mas e quando o que precisamos armazenar em memória não é apenas um valor, mas diversos deles?

E se eu lhe pedir para guardar 10 números, como você faria?

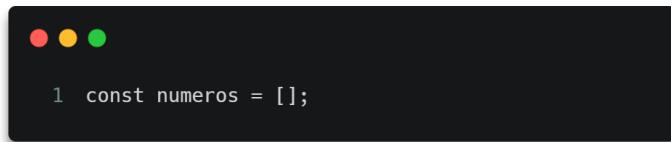
Seguindo uma lógica recente que utilizamos, você declararia 10 variáveis, certo?

Mas e se começássemos com 10 números e esse valor tivesse de aumentar em tempo de execução? Como faria neste caso?

É aí que entram os arrays!

Um array (também chamado de vetor em algumas linguagens de programação) é um tipo especial de objeto que permite que a gente armazene diversos valores dentro dele, cada um em uma posição indexada dentro do array. Declarar um array é muito simples, basta inicializarmos uma variável qualquer como um par de colchetes:

Código 3.14: disponível em <http://www.luiztools.com.br/ebook-frontend>



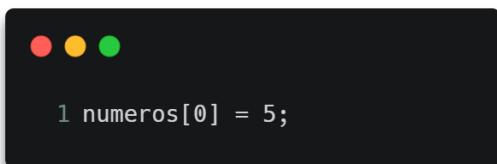
```
1 const numeros = [];
```

Aqui declaramos uma variável que é na verdade um array, demonstrado pelo uso de colchetes ([]). Essa declaração não indica qual o tipo de dado que armazenaremos nesse array, embora recomende-se que você armazene sempre dados de mesmo tipo em um mesmo array. Ao contrário de outras linguagens de programação, em JS os arrays não possuem tamanho fixo e nem mesmo precisam ser pré-dimensionados.

Calma, eu vou explicar.

Cada elemento que quisermos guardar em nosso array deve ficar em uma posição, começando na posição 0, depois 1, 2 e assim por diante. Sendo assim, se quisermos guardar o número 5 na primeira posição (0) faremos:

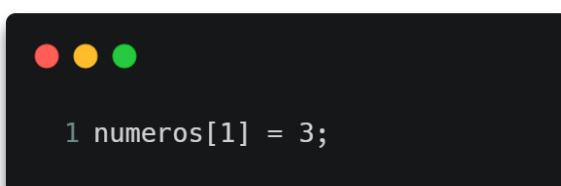
Código 3.14: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 numeros[0] = 5;
```

Agora se quisermos guardar o número 3 na segunda posição (1), fazemos:

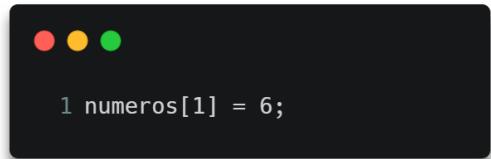
Código 3.14: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 numeros[1] = 3;
```

Se por algum motivo quisermos substituir o valor contido em alguma das posições, basta definirmos o novo valor sobre ela, como neste caso, em que estamos substituindo o valor guardado na posição 1 do array por um novo valor:

Código 3.14: disponível em <http://www.luiztools.com.br/ebook-frontend>

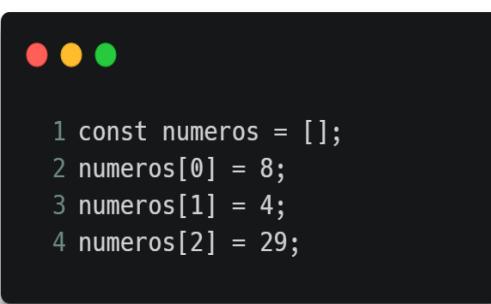


```
1 numeros[1] = 6;
```

Os elementos do array são numerados iniciando com zero (zero-based), e índices válidos variam de zero ao número de elementos menos um. O elemento do array com índice 1, por exemplo, é o segundo elemento no array. O número de elementos em um array é seu comprimento (length).

Resumindo: para guardar valores dentro do array, usamos o nome que demos a ele, seguido de colchetes e o número da posição onde queremos guardar o valor, como abaixo:

Código 3.14: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 const numeros = [];
2 numeros[0] = 8;
3 numeros[1] = 4;
4 numeros[2] = 29;
```

Aqui temos um array de comprimento 3, onde guardamos valores nas posições 0 (a primeira), 1 (a segunda) e 2 (a terceira e última).

Agora outro exemplo, onde crio um array e populo com nomes de pessoas:

Código 3.14: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 const inventores= [];
2 inventores[0] = 'Einstein';
3 inventores[1] = 'Edson';
4 inventores[2] = 'Galileu';
5 inventores[3] = 'Da Vinci';
```

**Atenção:** note que apesar de declarar a variável `inventores` com 'const' (constante) eu consigo adicionar elementos nela. Isso porque os elementos não são a variável `inventores` em si, mas estão sendo referenciados por ela. Sem entrar em muitos detalhes de baixo nível aqui (baixo nível = nível de máquina), basta entender que adicionar um item no array não muda o que o array é e por isso podemos declará-lo como `const` sem problemas. O que geraria um erro? Tentar reinicializar `inventores` com um novo valor, pois daí estariamos mudando quem/o quê ele é.

Mais tarde, quando quisermos acessar os elementos do array, basta utilizarmos a sua posição (índice) para retornarmos o valor contido dentro dela:

Código 3.14: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 const inventor = inventores[3];
2 console.log(inventor);
```

Considerando os exemplos anteriores de código, o que o trecho acima irá imprimir no console quando executado?

Da Vinci! Pois esse foi o nome do inventor armazenado na quarta posição do array (índice 3).

Os Arrays possuem algumas características e funções especiais que podemos utilizar para manipulá-los mais facilmente, como as abaixo (considere que `arr` é um variável Array):

- » **arr.length**: retorna a quantidade atual de elementos do array
- » **arr.push(x)**: adiciona o valor x ao final do Array
- » **arr.splice(i)**: remove o elemento na posição i do array
- » **arr.splice(i, y)**: remove y elementos a partir da posição i do array
- » **arr.concat(arr2)**: faz a fusão dos elementos de arr2 (outro array) no final de arr
- » **arr.indexOf(x)**: retorna a posição do elemento x dentro de arr, ou -1 caso ele não exista dentro de arr
- » **arr.forEach(function(x){})**: percorre todas as posições do array, da primeira à última, e à cada iteração executa a função passada por parâmetro onde x representa o item da iteração atual.

Mas voltando ao desafio inicial deste tópico, como faríamos para armazenar um número arbitrário de valores?

Você lembra dos laços de repetição?

Uma maneira muito inteligente de popular arrays é com laços de repetição, como um for, por exemplo.

Código 3.14: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 const numeros = [];
2 for(let i=0; i < 10; i++)
3     numeros[i] = 1
```

Note que usamos a própria variável de controle ‘i’ para acessar a posição do array de números onde queremos guardar os valores.

Um código equivalente, mas usando a função ‘push’ seria:

Código 3.14: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
1 cconst numeros = [];
2 for(let i=0; i < 10; i++)
3     numeros.push(1);
```

**Atenção:** reparou que usei `let` para a variável de controle do `for`? Isso porque não posso usar `const`, uma vez que o valor dela altera a cada iteração. Podem parecer coisas bobas e você até pode se sentir tentado a só usar `var`, mas recomendo fazer o esforço de usar corretamente `const/let/var` nessa ordem pois vale a pena a longo prazo.

Notou algo estranho? Sim, eu guardei o número 1 em todas posições do array.

Não acredita?

Use o seguinte código para imprimir todas posições do array:

Código 3.14: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
● ● ●  
1 for(let i=0; i < numeros.length; i++)  
2   console.log(numeros[i]);
```

Note que aqui eu usei a propriedade `length` acessada a partir do array de números. Eu citei ela nas características e funções úteis do array e essa é a mesma propriedade existente no tipo String que vimos anteriormente. Hmm, por que você acha que Strings e Arrays possuem uma propriedade `length` em comum?

Porque ambos são Arrays!

Sim, a diferença é que Strings apenas guardam caracteres dentro de si, mas ambos possuem muitas características em comum, muito mais do que o `length` em si. Ou seja, considerando que `str` é uma variável String:

- » `str.length`: retorna a quantidade atual de caracteres da String
- » `str.split('x')`: retorna um array de Strings quebrando a original pelo caracter (ou palavra) informado na função;
- » `str.charAt(i)`: retorna o caracter na posição informada na String (zero-based)

- » **str.slice(start,end)**: retorna uma substring da original começando na posição start e terminando na posição end
- » **str.indexOf('x')**: retorna a posição do caracter (ou palavra) x dentro de str, ou -1 caso ele não exista dentro de str

Muito legal, não?!

## O tipo Object

Tudo é tratado como objeto em JavaScript, mesmo os números e textos literais. Além disso, os objetos em JavaScript são extremamente dinâmicos como você já deve ter percebido, podendo alterar o seu conteúdo livremente.

No entanto, existe uma forma de criar seus próprios objetos em Javascript, embora ela não seja uma linguagem orientada à objetos da mesma forma que Java e C#, por exemplo. Objetos JS são declarados como qualquer outra variável (embora exista a possibilidade de criá-los a partir de classes), os atributos e métodos podem ser definidos no mesmo momento em que são lidos e/ou escritos.

Sendo assim, para criar um objeto considere a seguinte declaração de variável:

Código 3.15: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 const cliente = {};
```

Nada de especial, certo? Mas o que acontece se na linha seguinte fizermos o seguinte:

Código 3.15: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 cliente.nome = 'Luiz';
```

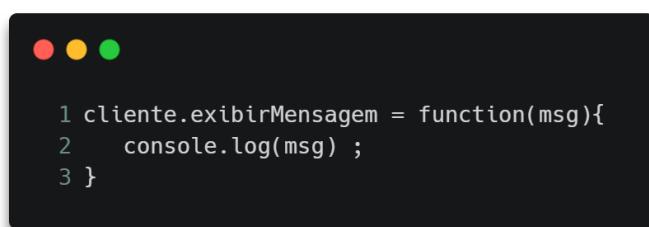
No Java e outras linguagens compiladas e tipadas isso daria errado, certo?

Mas no JS não.

Simplesmente será criado um atributo nome dentro do objeto cliente contendo o valor da string ‘Luiz’. Prático, não?! Quando digo que cliente recebe as chaves, estou dizendo que cliente é um object (objeto complexo), e poderei colocar dentro dele atributos e funções. Note que se eu mandasse imprimir no console o nome deste cliente, antes do mesmo ser definido, o atributo nome não existiria e com isso teríamos undefined como resultado.

Assim como definimos atributos conforme vamos precisando deles, também o fazemos com funções. Assim, podemos ter instruções “estranhas” como esta:

Código 3.15: disponível em <http://www.luiztools.com.br/ebook-frontend>



```
● ● ●
1 cliente.exibirMensagem = function(msg){
2   console.log(msg) ;
3 }
```

Ou seja, a partir da linha seguinte podemos chamar cliente.exibirMensagem(‘teste’) que irá disparar um console.log com a mensagem teste dentro.

Quando queremos criar um objeto já com seus atributos e funções pré-definidos podemos utilizar a notação JSON. JSON é uma sigla para JavaScript Object Notation e registra um padrão de escrita e leitura de objetos. Um objeto cliente pode ser definido em JSON da seguinte maneira:

Código 3.15: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 const cliente2 = {  
2     nome: 'Luiz',  
3     saldo: 100.0,  
4     idade: 32,  
5     gaúcho: true  
6 }
```

Assim, a partir da linha seguinte à este trecho JS, se mandarmos um console.log com cliente.saldo, irá aparecer o valor ‘100.0’ no navegador.

E para criar arrays de objects (algo perfeitamente possível), podemos fazer como segue:

Código 3.15: disponível em <http://www.luiztools.com.br/ebook-frontend>

```
1 const clientes = [cliente1, cliente2, { nome:"cliente3",  
saldo:5.0, idade:21, gaúcho:false }];
```

**Nota:** o padrão JSON (<http://json.org>) define que o nome do atributos deve estar entre aspas duplas. Entretanto, o JS entende os atributos que não possuam aspas, desde que os mesmos não possuam espaços ou caracteres especiais.

Quer fazer um curso online de Desenvolvimento Web FullStack JS com o autor deste ebook? Acesse <https://www.luiztools.com.br/curso-fullstack>

---

# JAVASCRIPT CLIENT-SIDE

“

*Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?*

*- Brian W. Kernighan*

”

JavaScript é uma linguagem de programação interpretada originalmente implementada como parte dos navegadores web para que scripts pudessem ser executados do lado do cliente e interagissem com o usuário sem a necessidade deste script passar pelo servidor, controlando o navegador, realizando comunicação assíncrona e alterando o conteúdo do documento exibido.

É atualmente a principal linguagem para programação client-side em navegadores web e foi concebida para ser uma linguagem script com orientação a objetos baseada em protótipos, tipagem fraca e dinâmica e funções de primeira classe. Possui suporte à programação funcional e apresenta recursos como clojures e funções de alta ordem comumente indisponíveis em linguagens populares como Java e C++.

Atualmente considera-se o JavaScript como sendo a linguagem de programação mais utilizada no mundo, sendo padronizada pela Ecma International sob o nome de ECMAScript, nas especificações ECMA-262 e ISO/IEC 16262.

Em capítulo anterior aprendemos o básico da sintaxe do JavaScript, bem como aprendemos a fazer uma série de provas de conceito usando o console do Google Chrome. Neste capítulo aprenderemos como utilizar o JavaScript como linguagem client-side para tornar as nossas páginas HTML mais dinâmicas.

Como client-side entenda do lado do cliente, ou seja, no front-end, na interface do navegador que está rodando na máquina do seu usuário. O oposto disso seria server-side, ou no lado do servidor, código que roda no datacenter, longe da máquina do cliente.

## Document Object Model

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

Uma página da Web é um documento. Este documento pode ser exibido na janela do navegador ou como a fonte HTML. Mas é o mesmo

documento nos dois casos. O DOM (Document Object Model) representa o mesmo documento para que possa ser manipulado. O DOM é uma representação orientada a objetos da página da web, que pode ser modificada com uma linguagem de script como JavaScript.

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

Os padrões W3C DOM e WHATWG DOM são implementados na maioria dos navegadores modernos. Muitos navegadores estendem o padrão; portanto, é necessário ter cuidado ao usá-los na Web, onde os documentos podem ser acessados por vários navegadores com diferentes DOMs.

## COMO USAR

Você não precisa fazer nada de especial para começar a usar o DOM. Navegadores diferentes têm implementações diferentes do DOM, e essas implementações exibem graus variados de conformidade com o padrão DOM real (um assunto que tentamos evitar nesta documentação), mas todo navegador usa um modelo de objeto de documento para tornar as páginas da web acessíveis via JavaScript.

Quando você cria um script - seja embutido em um elemento(tag) <script> ou incluído na página da web por meio de uma instrução de carregamento de script - você pode começar imediatamente a usar a API para o document, visando manipular o próprio documento ou obter os filhos desse documento, que são os vários elementos na página da web. Sua programação DOM pode ser algo tão simples quanto o exemplo seguinte, que exibe uma mensagem de alerta usando a função alert() da função window ou pode usar métodos DOM mais sofisticados para criar realmente novo conteúdo, como nos comando que você verá a seguir.

**Nota:** sugiro criar uma página HTML, colocar uma tag SCRIPT nela e testar cada um dos recursos abaixo para entender realmente o que eles fazem. Mais para frente teremos um exercício, assim como já vínhamos fazendo no capítulo anterior.

## O OBJETO WINDOW

O objeto window é o contâiner por fora do DOM, contém informações e funções que afetam a janela atual do navegador:

### **window.location.href**

Pega a URL atual do navegador ou faz com que o navegador acesse a URL especificada. Experimente usar um console.log(window.location.href).

### **window.location.search**

Pega a querystring atual do navegador. Experimente usar um console.log(window.location.search).

### **window.location.reload()**

Atualiza a página atual (F5).

### **window.history**

Acessa o objeto de histórico do navegador, permite inclusive voltar para a página anterior.

### **setTimeout(function, delay)**

Função nativa do objeto window que atrasa o disparo de uma função JS em um tempo definido em milissegundos.

### **setInterval(function, interval)**

Semelhante ao setTimeout, mas dispara a mesma function a cada x milissegundos. Como retorno à chamada deste método é passado o ID do timer, que pode ser cancelado usando a função clearInterval(id).

### **setImmediate(function)**

Executa a function exatamente agora, mas por uma thread em background. Muito útil para processamento não-bloqueante.

## POPUPS

Popups são aquelas janelinhas, muitas vezes irritantes, que saltam no navegador e são fornecidas pela API DOM da janela do navegador (window). Se bem utilizadas podem ser muito úteis para exibir informações ao usuário, por exemplo.

### **alert(message)**

Exibe um popup com uma mensagem dentro e um botão de ok. É um atalho para `window.alert(message)`.

### **confirm(message)**

Exibe um popup com uma mensagem dentro (geralmente um questionamento) e dois botões: ok e cancelar. Caso o usuário clique em ok, esta função irá retornar true, caso contrário false. É um atalho para `window.confirm(message)`.

## O OBJETO DOCUMENT

O objeto document contém informações e funções que afetam a página HTML atual que está sendo exibida na janela (window) navegador.

### **document.getElementById(id)**

Busca no documento HTML o objeto com o atributo Id passado por parâmetro (string). Este objeto resultante tem os mesmos atributos e funções do objeto original. Note que o atributo Id não é a mesma coisa que o atributo name. O id serve justamente para diferenciar os objetos HTML entre si (não temos dois objetos com mesmo id), enquanto que o name serve para definir o nome dos valores a serem passados dentro de um FORM (ou seja, não possui sentido em tags que não sejam INPUTs e derivados).

Mais à frente ver mais sobre o document, assim que exercitarmos o que acabamos de ver.

## EVENTOS JAVASCRIPT

É possível associar funções JS à eventos que possam ocorrer com seus elementos HTML. Assim, quando um botão clicar ou quando digitarmos sobre um campo de texto, algo interessante pode ser executado.

Para configurar um evento em um elemento HTML basta fazer o mesmo que fazíamos com os atributos dos elementos: dentro da tag em que queremos configurar o evento, adicionamos o nome do evento seguido do sinal de igualdade e entre aspas colocamos a chamada à função JS que deve existir em nossa página (incluindo parâmetros, se houverem). Isso ficará mais claro no detalhamento à seguir.

**Nota:** uma dica de como testar os seus códigos JS rapidamente é usando o site <http://jsfiddle.net> que permite criar código HTML, JS e CSS e testá-lo diretamente no navegador.

## ONCLICK

O evento onclick pode ser configurado em qualquer elemento HTML que possa ser clicado. Quando o click do mouse ocorrer sobre o elemento, a função JS configurada será disparada. Veja o exemplo à seguir de uma função JS que apenas exibe uma mensagem (essa função deve estar em um arquivo):

Código 4.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 function exibirMensagem() {
2     alert('funcionou!');
3 }
```

Agora veja um botão HTML com o evento onclick configurado para a função acima:

Código 4.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 <input type="button" value="Teste" onclick="exibirMensagem()" />
```

Assim, quando este botão for clicado, a mensagem ‘Funcionou!’ irá aparecer no navegador do cliente. Note que usamos um TYPE=”BUTTON” ao invés de um SUBMIT. Isso porque botões do tipo SUBMIT iriam submeter o FORM HTML ao invés de simplesmente exibir a mensagem, estragando nosso teste. Entretanto, não pense que o onclick funciona apenas com botões, ele funciona com qualquer elemento visível que possa ser clicado com o mouse.

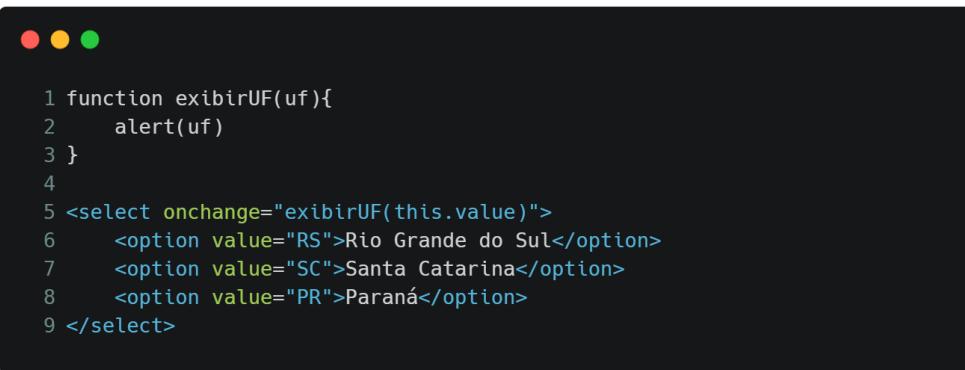
Analogamente, temos o evento ondblclick que é quando o elemento HTML recebe um clique-duplo do mouse.

## ONCHANGE

O evento onchange pode ser configurado em qualquer elemento HTML que possa ter uma opção selecionada. Quando o usuário escolhe uma nova opção em um SELECT, esse evento é disparado, por exemplo.

Abaixo, a função JS que você deve colocar no bloco script e o código HTML que deve ir na página.

Código 4.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
● ● ●

1 function exibirUF(uf){
2     alert(uf)
3 }
4
5 <select onchange="exibirUF(this.value)">
6     <option value="RS">Rio Grande do Sul</option>
7     <option value="SC">Santa Catarina</option>
8     <option value="PR">Paraná</option>
9 </select>
```

Note que esta função JS é ligeiramente diferente da anterior, ela recebe um parâmetro UF, que é o texto que será exibido no alert. Quando configurarmos qualquer evento podemos passar parâmetros para a função JS que será chamada e neste caso optei por passar o atributo value do próprio elemento HTML. Com isso, ao selecionar a opção ‘Santa Catarina’, a uf ‘SC’ irá aparecer na mensagem.

## ONMOUSEENTER E ONMOUSELEAVE

Quando queremos mapear os eventos do ponteiro do mouse sobre nossos elementos HTML podemos usar os eventos onmouseenter e onmouseleave. O primeiro é disparado quando o ponteiro do mouse entra na área do elemento, e o segundo quando o ponteiro do mouse sai da área do elemento.

Código 4.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

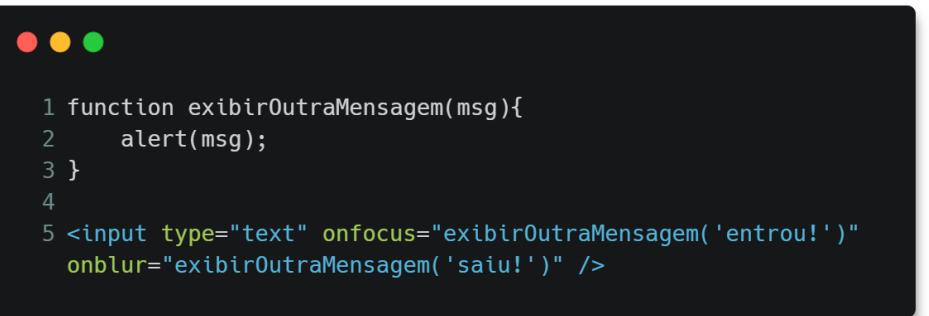
```
1 function exibirOutraMensagem(msg){  
2     alert(msg);  
3 }  
4  
5 <div onmouseenter="exibirOutraMensagem('entrou!')"  
     onmouseleave="exibirOutraMensagem('saiu!')">Teste</div>
```

No exemplo acima, quando o usuário passar o mouse sobre a DIV a mensagem exibida será ‘entrou’ e quando o ponteiro do mouse sair da área da DIV aparecerá ‘saiu’. Note que aqui usamos apenas uma função JS mapeada em dois eventos diferentes, apenas mudando a string literal que passamos por parâmetro. Note também que usamos aspas simples para passar o parâmetro string, isso porque já havíamos usado aspas duplas na definição dos eventos.

## ONFOCUS E ONBLUR

Quando queremos mapear os eventos de ganho de foco e perda de foco em um elemento HTML, usamos os eventos onfocus, que ocorre quando um elemento ganha o foco do usuário, e onblur, quando o elemento HTML perde o foco do usuário. Por ‘ganhar’ e ‘perder’ o foco entenda que o elemento recebe o foco quando o cursor para sobre ele, seja com o uso do mouse ou do teclado (usando TAB).

Código 4.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
function exibirOutraMensagem(msg){  
    alert(msg);  
}  
  
<input type="text" onfocus="exibirOutraMensagem('entrou!')"  
onblur="exibirOutraMensagem('saiu!')"/>
```

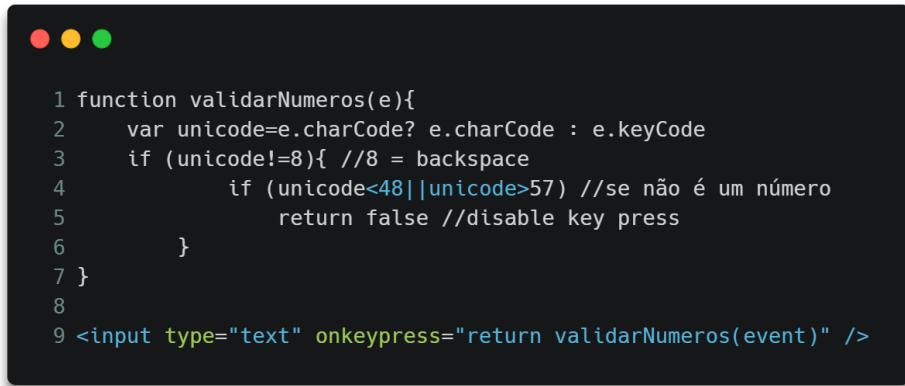
No exemplo acima nosso campo de texto exibirá o alert ‘entrou’ quando ganhar o foco e a mensagem ‘saiu’ quando ele perder o foco. Experimente no seu navegador usando a tecla TAB e usando o mouse para ver como o comportamento é o mesmo.

## ONKEYPRESS, ONKEYDOWN E ONKEYUP

Também podemos mapear os eventos do teclado quando o foco está sobre nosso elemento HTML. O evento onkeydown é disparado quando uma tecla está atualmente sendo pressionada no teclado, o onkeyup quando uma tecla está atualmente sendo solta no teclado e o onkeypress corresponde à ação completa de pressionar e soltar uma tecla.

Estes eventos são especialmente úteis em campos de texto, para verificar em tempo real o que fazer com as informações que o usuário está digitando. Com isso é possível criarmos validações e máscaras em nosso campo. Estes eventos disparam funções JS comuns, porém através do objeto event podemos saber qual a tecla pressionada e opcionalmente alterar o conteúdo do campo de texto ou até invalidar a última tecla digitada, como abaixo.

Código 4.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 function validarNumeros(e){  
2     var unicode=e.charCodeAt : e.keyCode  
3     if (unicode!=8){ //8 = backspace  
4         if (unicode<48||unicode>57) //se não é um número  
5             return false //disable key press  
6     }  
7 }  
8  
9 <input type="text" onkeypress="return validarNumeros(event)" />
```

A função validarNumeros retorna false quando a tecla digitada não é um número. Como o evento onkeypress está mapeado para retornar o resultado de validarNumeros, se a função retornar false a tecla digitada não irá aparecer no campo de texto, isso porque o onkeypress acontece antes da tecla realmente aparecer no campo.

Note também que passamos o objeto event por parâmetro na função, que possui informações sobre o evento em si, incluindo o caracter digitado ou o código da tecla, de acordo com o que foi digitado.

Existem muitos outros eventos JS que você pode mapear, como o arrastar e soltar do mouse (onmousedown e onmouseup), o redimensionar (onresize) de elementos, o carregar/descarregar (onload e onunload) do body HTML e muitos outros.

Vamos exercitar o que vimos até aqui?

## CALCULADORA EM HTML+JS

O que acha de fazermos uma calculadora que roda no navegador?

Crie uma página HTML contendo um INPUT de texto para o display, botões para os número de 0 a 9 e para as operações elementares. Também inclua um botão para a igualdade e um de limpar. Ela deve se comportar exatamente como uma calculadora digital que estamos acostumados a usar.

Em termos de aparência, tente construir da seguinte forma:



Não se preocupe tanto com a beleza da interface. Veremos estilização de HTML mais à frente para dar uma aparência mais profissional às nossas aplicações web.

Se não conseguir construir por si próprio este layout HTML, use os fontes abaixo:

Código 4.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Calculadora</title>
5     <meta charset="utf-8" />
6   </head>
7   <body>
8     <table>
9       <tr>
10      <td colspan="6"><input type="text" id="display" value="0" disabled></td>
11    </tr>
12    <tr>
13      <td><input type="button" value="7" /></td>
14      <td><input type="button" value="8" /></td>
15      <td><input type="button" value="9" /></td>
16      <td><input type="button" value="*" /></td>
17    </tr>
18    <tr>
19      <td><input type="button" value="4" /></td>
20      <td><input type="button" value="5" /></td>
21      <td><input type="button" value="6" /></td>
22      <td><input type="button" value="-" /></td>
23    </tr>
24    <tr>
25      <td><input type="button" value="1" /></td>
26      <td><input type="button" value="2" /></td>
27      <td><input type="button" value="3" /></td>
28      <td><input type="button" value="+" /></td>
29    </tr>
30    <tr>
31      <td><input type="button" value="C" /></td>
32      <td><input type="button" value="0" /></td>
33      <td><input type="button" value="/" /></td>
34      <td><input type="button" value="/" /></td>
35    </tr>
36  </table>
37 </body>
38 </html>
```

Agora vamos aos comportamentos. Cada vez que um botão numérico for pressionado, o número do botão deve ser adicionado à direita do input/display. Note que coloquei o id display no input, para podermos manipulá-lo via JavaScript e também coloquei o atributo disabled nele, para que não seja permitido digitar diretamente nele (o usuário terá de usar os botões).

Para fazer isso, abra uma tag SCRIPT antes de </body> e vamos criar uma função de atualização do display nele:

Código 4.2: disponível em <https://www.luisztools.com.br/ebook-frontend-fontes>



```
1 <script>
2     function atualizarDisplay(btn){
3         const display = document.getElementById('display')
4         if(display.value.length === 9) return
5         if(display.value === '0') display.value = btn.value
6         else display.value += btn.value
7     }
8 </script>
```

O que fazemos aqui: a função recebe um objeto btn que representa um botão por parâmetro. Daí usamos document.getElementById para carregar o input display em uma variável local. Com essa variável local display podemos ler e escrever no value do display, alterando o seu texto.

Ao invés de editar o texto diretamente, coloquei uma verificação para impedir que sejam digitados mais de 9 números e também fiz uma lógica para que caso o display esteja zerado, que o primeiro número vai substituir esse zero. Por fim, faço com que o value do botão seja adicionado ao value do display.

Mas como fazer para que essa função seja adicionada a todos os botões numéricos?

Basta usar o atributo onclick dos botões que permite dizer qual a função JS será disparada toda vez que o botão for clicado. O uso de ‘this’ no parâmetro faz com que o próprio botão seja passado por parâmetro, como no exemplo abaixo do botão 7 (você deve replicar para todos os numéricos):

Código 4.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 <td><input type="button" value="7" onclick="atualizarDisplay(this)" /></td>
```

Para testar é muito simples, abra a sua calculadora.html no navegador e pressione os botões numéricos, verá que já estão funcionando.

Para fazer o botão de limpar (C de Clear) é bem simples, basta criar uma função que zera o display novamente (coloque essa função no mesmo bloco SCRIPT da outra função):

Código 4.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1     function limparDisplay(){
2         document.getElementById('display').value = '0'
3     }
```

E depois referenciar esta função no atributo onclick do botão C:

Código 4.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 <td><input type="button" value="C" onclick="limparDisplay()"/></td>
```

Teste e verá que funciona perfeitamente.

Agora é a hora de fazer as operações funcionarem. Para isso, vamos criar uma função para o click dos botões de operação, bem como algumas variáveis:

Código 4.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 let operador = '';
2 let valor1 = 0;
3 function atualizarOperacao(btn){
4     const display = document.getElementById('display');
5     operador = btn.value;
6     valor1 = parseInt(display.value);
7     display.value = '0';
8 }
```

Aqui guardamos a operação que foi digitada em uma variável, bem como o valor numérico que estava no display no momento que o operador foi pressionado. Vincule esta função aos quatro botões de operação, como neste exemplo com o botão de multiplicação:

Código 4.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <td><input type="button" value="*" onclick="atualizarOperacao(this)" /></td>
```

Visualmente isso não tem efeito algum, exceto zerar o display e permitir que você digite outro número enquanto que o primeiro ficou armazenado em memória. Para fazer com que as operações funcionem, vamos criar mais uma função, que usaremos para o botão de igualdade:

Código 4.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 function calcularOperacao(){
2     const display = document.getElementById('display');
3     const valor2 = parseInt(display.value);
4     valor1 = eval(valor1+operador+valor2);
5     display.value = valor1;
6     operador = '';
7 }
```

Essa função merece uma atenção especial!

Primeiro, carregamos o input de display e lemos o número atualmente nele (já convertendo para numérico). Depois, usamos a função eval do JS (que interpreta uma string como sendo código JS) para calcular a equação matemática composta pelo valor1 (que guardamos anteriormente), o operador matemático e o valor2 (que acabamos de ler). Guardamos o resultado eval no valor1 caso o usuário queira encadear operações e por fim limpamos a variável operador.

Para fazer o botão de igualdade funcionar, vamos vincular esta última função no onclick dele:

Código 4.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 <td><input type="button" value="=" onclick="calcularOperacao()" /></td>
```

E com isso você terá uma calculadora web perfeitamente funcional!

**Nota:** o HTML não armazena estado, logo, a cada refresh do seu navegador ou troca de página você perderá os dados que tinha em memória. Isso é normal da web.

Vamos colocar uma perfumaria? É muito chato não poder usar o teclado do computador para usar a calculadora, certo?

Então vamos criar uma última função no bloco SCRIPT da nossa calculadora.html que vai pegar as teclas digitadas do teclado e decidir se elas vão ou não aparecer no display:

Código 4.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 function manipularTeclado(){
2     if(/[0-9]/.test(event.key))
3         atualizarDisplay({value: event.key});
4 }
```

event é um objeto que representa o evento JS que acabou de ser disparado. Do jeito que está não dá pra entender direito o que event vai representar mas confie em mim. Já event.key é um atributo que existe apenas quando o evento é de teclas pressionadas.

O que fiz ali no if é uma expressão regular. Existem livros apenas sobre a construção e uso de expressões regulares como o fantástico Expressões Regulares: Uma Abordagem Divertida, de Aurélio Marinho Jargas, mas basta entender por ora que uma expressão regular é um padrão de texto sobre o qual testamos para ver se outro texto está de acordo.

Quando uso barras no JavaScript ele entende que vou definir uma expressão regular entre as barras. Aqui no caso usei [0-9] que é uma expressão regular bem comum para validar se um texto é um número de 0-9. O teste é feito usando a função test, passando por parâmetro o texto que queremos validar. No nosso caso, somente entrará no if se a tecla que foi pressionada é um número.

Para que essa função passe a valer em nossa calculadora, devemos colocar ela no atributo onkeypress do body da sua página. Sim, isso mesmo, no body. Estamos definido no body, essa função irá manipular TODOS os pressionamentos de teclado do usuário para ver se algum dele é um número que possa ser colocado no display:

Código 4.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <body onkeypress="manipularTeclado()">
```

Teste agora e verá que, com exceção da aparência, nossa calculadora ficou bem legal.

O JavaScript client-side é muito poderoso e mesmo antes da invenção do Node.js ele já era uma linguagem muito utilizada mundialmente, uma vez que a tecnologia padrão para o client-side dos browsers.

Agora vamos aprender como fazer alguns truques bem interessantes usando recursos de manipulação do DOM. Sim, eu vou explicar o que é DOM também...

## Manipulando o DOM

Vimos o básico do DOM anteriormente e entendemos do que se trata, agora chegou a hora de extrair mais poderes dele usando JavaScript.

### SELETORES

O primeiro e mais fundamental conceito de manipulação do DOM é o de seletores. Isso porque, quando vamos manipular algo, a primeira coisa que devemos aprender é como encontrar o elemento que vamos manipular.

Para fazer isso, geralmente os programadores têm de definir um id no elemento e depois carregá-lo usando document.getElementById, como já citei antes.

Considere o seguinte trecho HTML:

Código 4.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <!DOCTYPE html>
2 <html>
3   <head><meta charset="utf-8" /></head>
4   <body>
5     <input type="text" name="nome" id="txtNome" /><br />
6     <input type="text" name="nome" id="txtNome2" />
7   </body>
8 </html>
```

Para então exercitar os comandos abaixo.

#### **document.getElementById(id)**

Busca no documento HTML o objeto com o atributo Id passado por parâmetro (string). Este objeto resultante tem os mesmos atributos e funções do objeto original. Experimente usar document.getElementById("txtNome") no console do navegador com esta página aberta e verá ser retornada a respectiva tag input.

Mas além desta função, temos:

### **document.getElementsByName(name)**

Busca no documento HTML todos os objetos com o name referenciado. Experimente usar `document.getElementByName("nome")` no console do navegador com esta página aberta e verá ser retornada ambas tags `input`, pois elas possuem o mesmo nome.

### **document.getElementsByTagName(name)**

Quando queremos trazer todos os elementos do HTML que sejam de uma mesma tag, usamos este seletor. Experimente trazer todos inputs ou algum outro elemento à sua escolha, como em `document.getElementsByTagName("input")`.

### **document.getElementsByClassName(name)**

Ainda não vimos CSS, mas existe outro atributo chamado class, que também pode ser usado como seletor do DOM.

E por último, temos a função mais poderosa...

### **document.querySelector(filtro) e document.querySelectorAll(filtro)**

Tudo o que fizemos com as demais funções e coisas ainda mais poderosas podem ser feitas com as funções acima, sendo que a primeira sempre retorna apenas um elemento e a segunda pode retornar vários. Em ambas, devemos passar por parâmetro o filtro a ser utilizado. Este filtro pode ser (usando o HTML acima como exemplo):

- » um id, prefixado com sustenido: (“#txtNome”);
- » tipo de tag, sem prefixo algum: (“input”);
- » um atributo definido após a tag, entre colchetes: (“input[name='nome']”);
- » começo de um valor de atributo, usando circunflexo: (“input[name^='nom']”);
- » final de um valor de atributo, usando cifrão: (“input[name\$='ome']”);
- » contendo um valor de atributo, usando asterisco: (“input[name\*= 'om']”);
- » um elemento dentro de outro elemento: (“body > input”);
- » mais de um elemento ao mesmo tempo: (“head,body”);

Todos os seletores exemplificados acima carregam o elemento `input` presente no HTML de exemplo.

O uso da função de seleção irá retornar um objeto JS contendo o elemento HTML e todas suas propriedades e funções. Além disso, a busca é feita em todo o documento, logo, se mais de um elemento atender ao filtro, todos serão retornados, motivo pelo qual devemos sempre buscar utilizar filtros específicos, como por id, por exemplo.

Uma vez que selecionamos o elemento que queremos, podemos manipulá-lo usando os atributos e funções nativos do JS ou nativas do DOM Element, como as funções a seguir.

## PROPRIEDADES DE ELEMENTO

Todo elemento HTML/DOM possui uma lista de propriedades que é muito grande para eu referenciar aqui. No próprio console do Google Chrome, quando você manipula o DOM e obtém um elemento, ele tem um recurso de autocomplete que lhe mostra todas as possibilidades daquele elemento.

Ainda assim, tentando trazer as propriedades mais comuns e úteis, temos:

### **element.value**

Esta propriedade, existente em inputs, define ou retorna o conteúdo do atributo value dele. Em alguns inputs, o value é o texto no seu interior. Em outros, é o item selecionado.

### **element.getAttribute(attributeName) e setAttribute(attributeName, newValue)**

Esta propriedade retorna o conteúdo de um atributo da tag HTML, como por exemplo o src de uma imagem ou o href de uma âncora.

O oposto de getAttribute é o setAttribute, que espera o nome do atributo e o seu novo valor como parâmetros.

### **element.style**

Esta propriedade define tudo relacionado à aparência do elemento. Qualquer subpropriedade de estilo que você quiser aplicar pode ser feito através de element.style.propriedade, como abaixo (exercite escrevendo código JavaScript):

- » `element.style.display`: retorna a visibilidade atual do elemento;
- » `element.style.display = "none"`: esconde o elemento em questão;
- » `element.style.display = "block"`: exibe o elemento em questão;

### **element.innerHTML**

Esta propriedade define o conteúdo HTML de uma tag, útil para ler ou alterar sua aparência também:

- » `element.innerHTML`: retorna o conteúdo HTML do componente;
- » `element.innerHTML = "<p>teste</p>"`: insere um parágrafo dentro do componente;

### **element.innerText**

Esta propriedade define o conteúdo texto de uma tag, útil para ler ou alterar sua aparência também.

## **FUNÇÕES DE ELEMENTO**

Para usar as funções abaixo, você deve primeiro carregar o elemento (ou documento) ao qual deseja disparar a função. Algumas funções esperam outras funções por parâmetro, o que indica que elas irão disparar a sua função em algum momento futuro, quando algo acontecer, como sendo um evento.

A primeira coisa que você tem de entender é que o exemplo acima é bem básico e que algumas páginas podem ser tão complexas que o navegador pode demorar um pouco até carregá-las. Se você quiser saber o exato momento em que o DOM de uma página está 100% carregado, o código abaixo pode lhe dizer isso.

Código 4.3: disponível em <https://www.luisitools.com.br/ebook-frontend-fontes>



```
1 document.addEventListener("DOMContentLoaded", function(event) {  
2   //faz alguma coisa  
3 });
```

Neste caso, a função passada como segundo parâmetro será disparado uma única vez, assim que o DOM terminar de ser carregado.

Talvez vá demorar algum tempo até você precisar usar um código como o acima, mas tenha em mente que todas as funções de manipulação de DOM que eu mostrei nesta seção, dependem que o DOM da página já esteja carregando, ok?

### **element.onclick = funcao**

Define uma função JavaScript que será disparado quando este element for clicado. Mesmo efeito do atributo onclick no HTML. Opcionalmente, você pode disparar este evento como qualquer outra função, para forçar o clique do elemento.

### **element.onchange = funcao**

Define uma função que será disparada quando este element for alterado (selects principalmente). Mesmo efeito do atributo onchange.

### **element.onfocus e element.onblur**

Define uma função para quando o usuário coloca ou tira o foco de um elemento, respectivamente.

### **element.onkeypress, onkeydown e onkeyup**

Define funções para manipulação de teclas pressionados quando o foco está sobre o elemento em questão. Mesmo efeito dos atributos onkeypress, onkeydown e onkeyup.

### **element.onmouseenter, onmouseleave**

Adiciona um gatilho no elemento que irá disparar uma função toda vez que o mouse entrar ou sair do elemento respectivamente (passando por cima, sem clicar).

### **element.onsubmit = funcao**

Com esta propriedade você pode definir uma função JS que vai ser disparada toda vez que um formulário HTML for submetido, ou seja, que seu input[type=submit] for pressionado.

### **element.addEventListener(eventName, callback)**

Esta função é a genérica para definir qualquer gatilho existente nos elementos. O primeiro parâmetro é o evento a ser mapeado e o segundo é a função que vai ser disparada quando esse evento acontecer.

## EXERCITANDO

Vamos exercitar o que vimos até aqui de manipulação de DOM criando um exercício que mais tarde evoluiremos com outra tecnologia importantíssima chamada Ajax.

Crie um arquivo HTML padrão chamado index.html, com HEAD e BODY, como abaixo:

Código 4.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
 1 <!DOCTYPE html>
 2 <html>
 3   <head>
 4     <title>Exemplo DOM</title>
 5     <meta charset="utf-8" />
 6   </head>
 7   <body>
 8     <h1>Exemplo DOM</h1>
 9     <p>Sistema para exemplificar manipulação de DOM</p>
10   </body>
11 </html>
```

Agora coloque duas DIVs no BODY deste HTML, uma com id ‘divCadastro’ e outra com id ‘divListagem’. A ideia aqui é construir um único HTML que contenha tanto um formulário de cadastro quanto a listagem dos clientes cadastrados. Tornaremos essa tela dinâmica apenas usando JavaScript e nossos conhecimentos de DOM, sem qualquer tipo de backend ou banco de dados.

Código 4.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
 1 <div id="divCadastro">
 2   <h2>Cadastro</h2>
 3 </div>
 4 <div id="divListagem">
 5   <h2>Listagem</h2>
 6 </div>
```

Note que se você abrir este arquivo HTML no seu navegador verá as duas DIVs visíveis, mas não é o que queremos, certo? De alguma forma temos que garantir que apenas uma delas esteja visível de cada vez.

Para fazer isso, vamos adicionar botões de navegação em ambas DIVs, como abaixo, visando ocultar/exibir as DIVs de acordo com cada botão pressionado:

Código 4.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
● ● ●  
1 <div id="divCadastro">  
2     <h2>Cadastro</h2>  
3     <input type="button" id="btnListar" value="Listar">  
4 </div>  
5 <div id="divListagem">  
6     <h2>Listagem</h2>  
7     <input type="button" id="btnCadastrar" value="Cadastrar">  
8 </div>
```

Mas e como podemos programar esses botões? Com JavaScript, é claro!

Para não ficarmos escrevendo JavaScript no meio do HTML, o que é considerado uma má prática, crie um arquivo scripts.js em uma pasta js ao lado do seu index.html e referencie-o na index.html:

Código 4.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
● ● ●  
1 <script src="js/scripts.js"></script>
```

**Nota:** a tag *SCRIPT* exige que ela seja fechada com uma */SCRIPT*, jamais fechada nella mesma. Se você não respeitar essa regra, sua página não irá funcionar.

Agora abra seu scripts.js e vamos definir o evento DOMContentLoaded, que é disparado assim que os componentes do documento estão todos prontos para serem manipulados:

Código 4.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
● ● ●  
1 document.addEventListener("DOMContentLoaded", function(event) {  
2  
3});
```

O que vamos fazer aqui?

Primeiro, esconder a divListagem, pois queremos que o HTML inicie exibindo somente a divCadastro. Fazemos isso facilmente em JS selecionando o elemento que queremos esconder e depois alterando o estilo de display dele como none, que serve para esconder elementos HTML:

Código 4.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
● ● ●  
1 document.addEventListener("DOMContentLoaded", function(event) {  
2     const divListagem = document.getElementById("divListagem");  
3     divListagem.style.display = "none";  
4});
```

Implementando este código você notará que quando abrir seu arquivo index.html no navegador, ele exibirá somente uma das DIVs, mas que o botão dela não funciona ainda.

Para fazê-lo funcionar é simples, precisamos manipular o evento click deles! Ainda dentro do evento DOMContentLoaded do seu scripts.js, inclua os seguintes eventos nos botões (note que sempre começamos selecionando o componente que vamos manipular):

Código 4.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
 1 document.addEventListener("DOMContentLoaded", function(event) {  
 2     const divListagem = document.getElementById("divListagem");  
 3     divListagem.style.display = "none";  
 4  
 5     const divCadastro = document.getElementById("divCadastro");  
 6  
 7     document.getElementById("btnListar").onclick = (evt) => {  
 8         divListagem.style.display = "block";  
 9         divCadastro.style.display = "none";  
10    }  
11  
12    document.getElementById("btnCadastrar").onclick = (evt) => {  
13        divListagem.style.display = "none";  
14        divCadastro.style.display = "block";  
15    }  
16});
```

Teste agora e verá que é perfeitamente possível manipular a exibição das divs usando os botões.

Com esses conceitos e conhecimentos dominados, podemos avançar no nosso exercícios. Agora vamos criar um formulário de cadastro de cliente (nome, idade e UF) na primeira div e uma tabela de clientes na segunda. Sim, já fizemos algo parecido antes, você pode aproveitar a ‘base’, mas será ligeiramente diferente desta vez.

Dentro da divCadastro, segue o formulário:

Código 4.5: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <form id="frmCadastro" action="" method="">
2     <p>
3         <label>Nome: <input type="text" name="nome" /></label>
4     </p>
5     <p>
6         <label>Idade: <input type="number" name="idade" /></label>
7     </p>
8     <p>
9         <label>UF: <select name="uf">
10            <option>RS</option>
11            <option>SC</option>
12            <option>PR</option>
13            <!-- coloque os estados que quiser -->
14        </select></label>
15     </p>
16     <p>
17         <input type="button" id="btnListar" value="Listar" | <input
18             type="submit" value="Salvar" />
19     </p>
20 </form>
```

Tomei a liberdade de trocar o btnListar de lugar, colocando-o dentro do form também. Note que este form não possui action nem method, pois não vamos enviá-lo ao servidor, vamos fazer coisas bem mais legais. :)

Já a divListagem, segue o HTML também:

```
1 <table style="width:50%">
2   <thead>
3     <tr style="background-color: #CCC">
4       <td style="width:50%">Nome</td>
5       <td style="width:15%">Idade</td>
6       <td style="width:15%">UF</td>
7       <td>Ações</td>
8     </tr>
9   </thead>
10  <tbody>
11    <tr>
12      <td colspan="4">Nenhum cliente cadastrado.</td>
13    </tr>
14  </tbody>
15  <tfoot>
16    <tr>
17      <td colspan="4">
18        <input type="button" id="btnCadastrar" value="Cadastrar">
19      </td>
20    </tr>
21  </tfoot>
22 </table>
```

Como ainda não vimos como estilizar nossas páginas HTML, ignore os atributos style que usei e não se importe se a aparência de tudo ficar muito feia, como nas imagens abaixo:



## Cadastro de Cliente

Preencha os dados abaixo para salvar o cliente.

Nome:

Idade:

UF:  RS

[Cancelar](#) |

E nessa outra aqui:

## Listagem

Nome	Idade	UF	Ações
Nenhum cliente cadastrado.			
<a href="#">Cadastrar</a>			

Agora vamos programar o nosso formulário usando JavaScript!

Como não temos back-end nesse exercício, quando o FORM for submetido, vamos capturar os dados enviados usando JavaScript e usá-los para preencher a tabela com a lista de clientes. Claro, esse armazenamento dos cadastros será efêmero, e toda vez que atualizarmos essa página no navegador ela virá zerada.

No entanto, servirá para nosso intuito de exercitar o que vimos de JavaScript. Como já estamos fazendo, aliás!

Para programar a submissão do form, vamos novamente abrir nosso scripts.js e inserir novos códigos dentro do evento DOMContentLoaded do document (logo abaixo daqueles scripts de clique dos botões das divs).

```
1 const frmCadastro = document.getElementById("frmCadastro");
2 frmCadastro.onsubmit = (evt) => {
3
4     let linha = '<tr>';
5     var data = new FormData(frmCadastro)
6     for(let item of data)
7         linha += `<td>${item[1]}</td>`;
8     linha += `<td><input type="button" value="X" /></td></tr>`;
9
10    //se tem apenas uma TD, é a default
11    const tbody = document.querySelector('table > tbody');
12    if(tbody.querySelectorAll('tr > td').length === 1)
13        tbody.innerHTML = "";
14
15    tbody.innerHTML += linha;
16
17    divListagem.style.display = "block";
18    divCadastro.style.display = "none";
19
20    frmCadastro.reset();
21
22    evt.preventDefault();
23 }
```

Esse código ficou extenso, mas vou explicar.

Primeiro, o evento submit do form é disparado toda vez que o form é submetido, o que no nosso caso acontece apenas quando o input de submit é pressionado. Para evitar o refresh na página, comportamento natural de submissões de formulário, coloquei um evt.preventDefault() ao final da função, para cancelar a “submissão de verdade”.

Para pegar os dados que foram submetidos temos duas opções: usando formData (que foi o que eu fiz, que retorna uma coleção de chave-valor ) ou pegando campo a campo.

Usei um for na coleção formData para construir uma string html de uma linha que precisa ser inserida na tabela. Mas antes de inserir, fiz um teste para ver quantas TDs (células) existiam dentro da tr de tbody. Se existir só uma, é porque é aquela default (não existem cadastros...) e devemos removê-la apagando o innerHTML do tbody.

Para finalizar, coloquei nossa linha HTML em conjunto com o restante de HTML que já existia no tbody, para que a tabela seja atualizada e troquei a visibilidade das divs para que seja possível ver o cliente recém cadastrado. Note que em um momento usei um seletor com o formato ‘element > child’, onde consigo descer níveis dentro da hierarquia de elementos HTML.

Se você testar agora, verá que funciona perfeitamente.

Mas...e aquele botão de exclusão que eu coloquei na última coluna da TD e que não funciona ainda?

Adicione um último código dentro da function de submit, uma linha antes do evt.preventDefault() que fará com que todos os botões de exclusão da tabela removam a própria linha onde estão (adicionei uma confirmação JavaScript também, só pra evitar cliques acidentais):

Código 4.5: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
 1 const buttons = document.querySelectorAll("input[value='X']");
 2 for(let btn of buttons){
 3     if(btn.onclick === null) continue;
 4     btn.onclick = (evt) => {
 5         if(confirm('Tem certeza que deseja excluir este cliente?')){
 6             btn.closest('tr').remove();
 7         }
 8     }
 9 }
```

Basicamente seleciono todos os botões com “X” e aqueles que ainda não possuem uma função de click, eu adiciono a nossa de remover a tr mais próxima com as funções closest (busca um elemento determinado mais próximo) e remove (remove o elemento em questão).

Bem simples.

## Listagem

Nome	Idade	UF	Ações
Luiz	29	RS	
<a href="#">Cadastrar</a>			

O funcionamento é muito legal na minha opinião. Uma aplicação simples e dinâmica para o usuário, muito rápida e eficiente, sem aqueles refreshs de tela, redirecionamentos, etc.

Mas como podemos manter esse mesmo nível de experiência agradável tendo um back-end com banco de dados e tudo mais? Ou carregando de uma web API da Internet?

Com Ajax.

## Ajax

AJAX (acrônimo em inglês de Asynchronous JavaScript and XML , em português “JavaScript e XML Assíncrono”) é o uso metodológico de tecnologias como Javascript e XML, providas por navegadores, para tornar páginas Web mais interativas com o usuário, utilizando-se de solicitações assíncronas de informações. Foi inicialmente desenvolvida pelo estudioso Jessé James Garret e mais tarde encabeçada por diversas associações. Apesar do nome, a utilização de XML não é obrigatória (JSON é frequentemente utilizado) e as solicitações também não necessitam ser assíncronas obrigatoriamente.

Quando submetemos um formulário ao servidor, seja via GET ou via POST, acabamos enviando junto muito mais informações do que gostaríamos (geralmente a página inteira é enviada), bem como direcionando a resposta do usuário para longe da página atual. Muitas vezes gostaríamos de consumir conteúdo do servidor sem ter de trocar de página e mesmo que nossa tecnologia de back-end permita redirecionar o fluxo novamente para a mesma tela, temos aquelas ‘piscadas’ inconvenientes no navegador que fazem com que todos os dados que estavam na página se percam.

Essa é a motivação por trás do Ajax: economia de dados trafegados e melhor experiência para o usuário.

A ideia por trás da tecnologia é que quando for necessário enviar alguma informação para o servidor, que somente a própria informação seja enviada. Que quando queremos obter alguma informação do servidor, que somente esta informação seja obtida. Em ambos os casos, que isso ocorra sem que haja um carregamento de toda a página

atual, mas sim somente das partes que devem sofrer mudanças após a requisição assíncrona tiver sido concluída.

## AJAX COM FETCH

Ajax é uma tecnologia embutida na linguagem de scripting Javascript, através do objeto XMLHttpRequest (XHR), que permite ao navegador fazer chamadas assíncronas a recursos em outro endereço do servidor.

Atualmente, com JavaScript moderno (suportado pelos navegadores mais modernos também), é extremamente simples de fazer requisições AJAX usando a função fetch.



Esta função pode ser usada apenas passando a URL que se deseja requisitar de maneira assíncrona. Nesse caso, será feito um HTTP GET na URL e o fluxo de execução seguirá normalmente. Quando a requisição retornar com uma resposta (o que pode ser praticamente instantâneo ou demorar minutos), devemos executar a função de retorno usando um conceito chamado Promises do JavaScript.

Usaremos neste primeiro exemplo uma web API aberta e gratuita existente na Internet chamada IPStack. Ela permite que você informe um endereço IP e ela vai lhe responder com a geolocalização daquele IP.

Acesse <https://ipstack.com> e crie uma conta free/gratuita para obter uma API Key, que é uma forma de segurança bem popular na Internet. É bem rápido.

Vamos criar uma página HTML simples, para que um usuário possa inserir um endereço IP em um input e, pressionando um botão, descobrir de onde ele é.

Código 4.6: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <label>IP:  
2     <input type="text" id="txtIP" value="8.8.8.8" />  
3 </label>  
4 <p>  
5     <input type="button" value="Buscar" onclick="buscarIP()" />  
6 </p>  
7 <p id="paragrafo"></p>
```

Nenhuma novidade aqui, mas note que o botão está esperando uma função buscarIP que ainda não criamos. Assim, crie um bloco script no final desta página HTML com o código do nosso fetch e incluindo a sua API Key como abaixo.

Código 4.6: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <script>  
2     function buscarIP() {  
3         const apiKey = "coloque a sua API Key aqui";  
4         const ip = document.getElementById('txtIP').value;//ex: 8.8.8.8  
5         const paragrafo = document.getElementById('paragrafo');  
6         fetch(`http://api.ipstack.com/${ip}?access_key=${apiKey}&format=1`)  
7             .then(req => req.json())  
8             .then(json => paragrafo.innerText = json.country_name)  
9             .catch(err => paragrafo.innerText = err);  
10    }  
11 </script>
```

Neste bloco, além das primeiras linhas que não devem ser novidade para você, está o fetch. O primeiro parâmetro dele é a string com a URL de acesso à web API. Aqui, eu monto ela usando um conceito chamado Template Literals que permite juntar variáveis com texto de forma mais organizada do que usando +.

A novidade aqui fica por conta do then e do catch.

Como eu mencionei antes, o fetch é assíncrono, ou seja, ele vai buscar a sua informação e libera sua página para fazer outra coisa. Mas como saber quando ele voltou com a resposta que eu queria?

Através do `then`. O `then` só é executado quando a resposta retorna com sucesso. Se der erro, vai ser executado o `catch`, logo abaixo.

Mas por que eu coloquei dois `thens`?

O `fetch` é uma função bem “crua”, serve para fazer requisições de qualquer coisa. Como essa API retorna dados no formato JSON (experimente acessar aquela URL do `fetch` no navegador, para ver ela por completo), precisamos do primeiro `then` para converter a resposta para JSON (por padrão ela vem em bytes). Essa conversão também é assíncrona e cairá a resposta para o segundo `then`, que aí sim exibe a informação que queremos na tela (o nome do país).

O resultado você confere na imagem abaixo.

## OPÇÕES DO FETCH

IP: `8.8.8.8`

`Buscar`

**United States**

Como eu mencionei antes, o `fetch` tem um segundo parâmetro opcional que são as opções ou configurações da requisição. Isso porque nem sempre você vai querer um HTTP GET e muitas vezes você vai precisar enviar dados no corpo da requisição. Isso tudo são opções do `fetch`, sendo que abaixo listo algumas delas que devem ser enviadas em um único objeto JSON como segundo parâmetro do `fetch`.

`method`: string com GET, POST, etc

`headers`: um objeto do tipo Headers;

`mode`: usado para cors, por exemplo (conceito avançado);

`body`: o conteúdo da sua requisição (deve coincidir com o header);

Um exemplo de requisição hipotética, com opções, pode ser visto abaixo.

```
1 fetch('URL', {  
2   headers: myHeaders,  
3   method: 'POST',  
4   mode: 'cors',  
5   body: myData  
6 })
```

Note duas coisas aqui: primeiro, em headers eu passei um objeto. Isso porque a propriedade headers do fetch espera um objeto do tipo Headers, como este que criei abaixo.

```
1 const myHeaders = new Headers();  
2 myHeaders.append("Content-Type", "application/json");
```

E segundo, que no body eu passei um objeto também, que na verdade deve ser uma string. Se o que eu quiser enviar for um objeto JavaScript, eu posso serializá-lo usando JSON.stringify, como abaixo.

```
1 const myData = JSON.stringify({numero: 1})
```

Infelizmente não conheço nenhuma API pública na Internet que permita POST para testarmos essas possibilidades de uso do fetch. Mas se você souber desenvolver para back-end, pode facilmente criar uma API e usar com este exemplo.

*Quer fazer um curso online de Desenvolvimento Web FullStack JS com o autor deste ebook? Acesse <https://www.luiztools.com.br/curso-fullstack>*

---

# CSS

5

“

*Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?*

- Brian W. Kernighan

”

Outra tecnologia muito comum quando o assunto é front-end é o CSS. CSS significa Cascading Style Sheets, ou Folhas de Estilo em Cascata. Enquanto que HTML é o esqueleto da página, o CSS são os órgãos e principalmente, a pele, o que dá a aparência, o estilo.

Estilos podem ser adicionados de três maneiras nos elementos HTML:

- » Inline: usando o atributo style
- » Internal: usando a tag STYLE
- » External: usando arquivos CSS externos

A forma mais comum de ser utilizado CSS é a terceira, principalmente por ser a mais organizada, em analogia ao que fazemos com JavaScript também. Ainda assim, veremos as três formas neste capítulo.

## CSS Inline

Todo elemento HTML possui um estilo padrão. A cor dos textos é preta, e a cor do plano de fundo é branca, só para citar dois exemplos. Alterar o estilo de um elemento HTML pode ser feito usando o atributo style, presente em todos elementos.

O exemplo a seguir mostra como alterar a cor do plano de fundo de branco (default) para cinza claro.

Código 5.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <body style="background-color: lightgrey">
2   <h1>This is a heading</h1>
3   <p>This is a paragraph.</p>
4 </body>
```

**Nota:** sugiro abrir uma página HTML e testar todos os exemplos de estilos CSS vendo como eles se comportam no navegador. Pode inclusive usar a index.html que criamos no capítulo anterior.

```
1 style="property:value"
```

O atributo style possui a seguinte sintaxe:

Sendo property uma propriedade CSS e value um valor CSS. Entenderemos melhor do que se trata CSS mais tarde neste mesmo material. Por enquanto, vamos ver como podemos mudar o estilo de nossos textos.

## ESTILOS DE TEXTO

Para alterar a cor do texto usamos a propriedade color.

```
1 <h1 style="color:blue">This is a heading</h1>
2 <p style="color:red">This is a paragraph.</p>
```

**Nota:** todas os valores de cores CSS aceitam literais em Inglês (blue, red, black, etc) e valores hexadecimais iniciados com # (como #CCCCCC para cinza claro e #FFFFFF para branco, por exemplo).

Para alterar a tipografia, usamos a propriedade font-family.

```
1 <h1 style="font-family:verdana">This is a heading</h1>
2 <p style="font-family:courier">This is a paragraph.</p>
```

**Nota:** todas as fontes são aceitas pelo CSS, no entanto, uma vez que os estilos são interpretados client-side, você não deve utilizar fonts que não existam por padrão nos navegadores e/ou sistemas operacionais de uso comum. Caso a font que você definiu não exista na máquina do usuário, o browser irá renderizar como um fonte comum, provavelmente estragando seu layout.

Código 5.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>  
O tamanho do texto pode ser definido pela propriedade font-size.



```
1 <h1 style="font-size:300%">This is a heading</h1>
2 <p style="font-size:160%">This is a paragraph.</p>
```

**Nota:** o tamanho pode ser definido em porcentagem, pontos e pixels, conforme a sua preferência.

Código 5.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

Enquanto que o alinhamento é definido pela propriedade text-align.



```
1 <h1 style="text-align:center">Centered heading</h1>
2 <p>This is a paragraph.</p>
```

Código 5.1: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

## Classes de Estilos em CSS Internal

Muitas vezes temos o mesmo estilo aplicado a diversos elementos em nossas páginas HTML. Da mesma forma, às vezes queremos que um determinado estilo se aplique a todas ocorrências de um elemento HTML no documento. Em todas essas ocasiões em que teríamos atributos style

com propriedades e valores repetidos devemos utilizar classes de estilo, mais comumente chamadas classes CSS.

Existe uma tag que ainda não utilizamos em nossas lições que é a tag STYLE. A tag STYLE define um bloco no seu documento HTML onde você poderá definir os estilos-base da sua página bem como as suas classes de estilo. A sua utilização é bem semelhante à da tag SCRIPT, ou seja, a tag STYLE é um container que conterá dentro dela estilos aplicados aos elementos desta página HTML em específico.

Código 5.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <style>
2     /* seu estilo vai aqui. A propósito, isto é um comentário CSS */
3 </style>
```

A tag STYLE pode ir em qualquer ponto do HTML, mas preferivelmente dentro da tag HEAD da página, para que seja carregado antes dos elementos HTML, para que quando eles sejam renderizados, já o sejam na maneira correta (com o novo estilo ao invés do padrão).

Dentro da tag STYLE podemos definir um estilo que se aplica a todos elementos de uma tag HTML específica, como abaixo, onde definimos que a cor dos textos dentro de todos parágrafos será verde (usei o nome da cor, mas você também pode usar valores hexadecimais).

Código 5.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <style>
2 p {
3     color: red;
4 }
5 </style>
```

A sintaxe é bem simples: definimos a tag à qual vamos aplicar o estilo e abrimos chaves para colocar o estilo dentro. A sintaxe para definição do estilo é a mesma que vimos no atributo style anteriormente, com

```
1 element { property:value; property:value; }
```

propriedades e valores CSS.

Note que esse estilo será aplicado automaticamente sobre todas tags P (os parágrafos), sem exceção. Muitas vezes não queremos isso, queremos que um determinado estilo seja aplicado a todos os parágrafos que iniciam um novo capítulo, por exemplo. Neste caso, devemos criar uma classe de estilo, como abaixo.

Código 5.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <style>
2 .paragrafoInicial {
3     color: green;
4 }
5 </style>
```

Note que as classes começam com um ponto, ao invés de simplesmente conterem o nome de uma tag. Além disso, sua nomenclatura é a mesma de variáveis Java e JavaScript (Camel Case) sem espaços, acentos ou caracteres especiais, raramente usando \_ (underline) ou – (hífen). O estilo da classe .paragrafoInicial será aplicado em todos os parágrafos que nós definirmos explicitamente através do uso do atributo CLASS, presente em todos elementos HTML, da seguinte maneira.

Código 5.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <p class="paragrafoInicial">
2     Este texto ficará verde.
3 </p>
4 <p>
5     Este texto terá a cor padrão de parágrafo.
6 </p>
```

Note que o atributo class dispensa o ponto do início da classe, colocamos apenas o nome dela em todos os elementos HTML que quisermos aplicar seu estilo.

Vale ressaltar também que os estilos funcionam em cascata, ou seja, podemos definir um estilo global para todos parágrafos, mais um estilo específico apenas para os parágrafos iniciais. Por exemplo:

Código 5.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <style>
2   p { color: red; font-family: Arial; }
3
4   .paragrafoInicial{ color: green; }
5 </style>
```

Assim, todos os parágrafos da aplicação terão a fonte Arial, mas somente os parágrafos com o atributo class definido como paragrafoInicial é que terão a cor da fonte verde.

## DICAS E TRUQUES

Existem diversos truques que podemos utilizar para aumentar ainda mais o poder dos estilos. Podemos definir estilos específicos para tags específicas dentro da hierarquia HTML de uma página, como abaixo, onde definimos que somente a cor dos textos dos parágrafos dentro de tabelas é que será verde, os demais manterão a cor normal (preta).

Código 5.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <style>
2   table p {
3     color: green;
4   }
5 </style>
```

Também podemos definir que um elemento possua mais de uma classe de estilo, apenas separando o nome das nossas classes por espaços:

Código 5.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <p class="paragrafoInicial paragrafoImportante">Parágrafo cheio de estilos.</p>
```

Assim o parágrafo receberá ambos estilos.

E por fim, quando queremos que somente um elemento receba determinado estilo, mas ao mesmo tempo não queremos usar o atributo style nele (pois é um estilo complexo, por exemplo), podemos criar um estilo baseado no atributo id do elemento, como abaixo.

Código 5.2: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <style>
2 #paragrafoLegal{
3     color: blue;
4 }
5 </style>
```

Assim, o elemento HTML cujo atributo id for ‘paragrafoLegal’ receberá automaticamente o estilo determinado na tag STYLE acima.

**Nota:** JavaScript usa essa mesma notação em seus seletores, usando o nome literal das tags, classes CSS iniciadas em ‘.’ e ids de componentes iniciados com '#'.

## ARQUIVOS CSS EXTERNOS

O uso de arquivos CSS externos (que nada mais são do que arquivos de texto com a extensão .CSS), geralmente em uma pasta chamada ‘styles’ ou ‘css’, é preferível uma vez que provavelmente você irá querer aproveitar os estilos entre mais de uma página HTML da mesma aplicação web. Com o uso correto de arquivos CSS você pode alterar a aparência inteira de um website ou sistema web apenas mexendo

em um arquivo, da mesma forma que com arquivos JS alteramos o comportamento sem mexer no HTML manualmente.

Para definir um arquivo CSS à um documento HTML devemos referenciá-lo usando a tag LINK dentro do HEAD da página, como abaixo.

Código 5.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 <!DOCTYPE html>
2 <html>
3 <head><link rel="stylesheet" href="estilos.css" /></head>
4 <body>
```

Neste exemplo estamos referenciando um arquivo estilos.css que está na mesma pasta do arquivo HTML, caso contrário nosso href seria maior, indicando o caminho até o mesmo.

Imagine cada arquivo CSS como um bloco STYLE. Escreva quantos estilos quiser e sinta-se livre para usar identação, quebras de linha e comentários para tornar seu CSS mais legível.

Para conseguirmos construir estilos que façam sentido para nossos componentes, não basta apenas conhecermos o nome das propriedades. Temos de entender o conceito de ‘Styling Boxes’ do CSS.

## ESTILOS DE CAIXAS

Imagine que todo elemento HTML está dentro de uma caixa. Existem diversos estilos que podemos aplicar à essa caixa imaginária e vamos ver alguns logo abaixo.

### Exibição

Quando queremos alterar a exibição da nossa caixa usamos a propriedade display, com os valores none ou block, seja para esconder ou exibir, respectivamente.

Código 5.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 p {  
2   display: none; /* todos p's ficarão invisíveis*/  
3 }
```

**Nota:** esse é exatamente o comportamento da função `hide()` do JQuery (e inversamente o efeito da `show()`)

## Plano de Fundo

Também podemos alterar o plano de fundo das nossas caixas, usando as propriedades `background-color` e `background-image`:

Código 5.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 p {  
2   background-color: #d0e4fe;  
3   background-image: url('url da imagem');  
4   background-repeat: no-repeat;  
5 }
```

No exemplo acima definimos a cor do plano de fundo de todos nossos parágrafos para um valor hexadecimal com a propriedade `background-color`. Também definimos que todos nossos parágrafos terão como imagem de fundo a imagem cuja url deverá estar entre as aspas dentro dos parênteses, com as mesmas regras do atributo `src` da tag `IMG`, isso tudo com a propriedade `background-image`. E por fim, apenas dissemos que a imagem de fundo não se repetirá, caso a caixa seja maior que a imagem em si.

**Nota:** e por padrão, caso a caixa seja maior que a imagem, a imagem se repetirá tanto na horizontal quanto na vertical, o que muitas vezes não é um comportamento interessante pois fica feio.

## Largura e Altura

Caso queiramos definir uma altura e largura fixa para nossa caixa, usamos as propriedades width e height, que permitem definir porcentagens ou tamanho fixo em pixels.

Código 5.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 p {  
2     width: 100%;  
3     height: 50px;  
4 }
```

Caso queiramos definir largura ou altura máxima ou mínima, neste caso utilizamos as variantes das propriedades width e height: max-width/max-height e min-width/min-height.

## Borda, espaçamento e margem

Para definir uma borda ao redor do nosso elemento HTML usamos a propriedade CSS border, onde definimos a espessura, o tipo de borda e sua cor, sendo os três valores separados por espaço.

Código 5.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 p { border: 1px solid black; }
```

Aqui, todos os parágrafos terão uma borda de 1px preta e sólida ao seu redor.

Ainda pensando em nossos parágrafos-caixas (embora o conceito de box se aplique a qualquer elemento), que tal colocarmos um espaçamento do texto do parágrafo (seu conteúdo) em relação às suas bordas? Fazemos isso com padding (margem interna).

Código 5.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 p { border: 1px solid black; padding: 10px; }
```

Agora todos os parágrafos manterão seus textos 10px distantes das quatro bordas da caixa imaginária (top, bottom, left e right).

No exemplo abaixo, além do espaçamento interno, definiremos uma margem (externa) de 30px de cada um dos nossos parágrafos em relação à qualquer elemento à sua volta.

Código 5.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 p {  
2   border: 1px solid black;  
3   padding: 10px;  
4   margin: 30px;  
5 }
```

**Nota:** todos esses estilos de caixa aplicam-se às 4 bordas da caixa: top (topo), bottom (rodapé), left (esquerda) e right (direita). Assim, caso queira aplicar um estilo de caixa à somente uma borda específica, use a propriedade correspondente seguida de um traço e o nome da borda, como abaixo, onde definimo que somente a margin do rodapé será de 15px.

Código 5.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 p {  
2   margin-bottom: 15px;  
3 }
```

Para uma referência completa de todas propriedades CSS e como utilizá-las, acesse: <http://www.w3schools.com/cssref/default.asp>

## CSS e JavaScript

Podemos definir estilos em nossos elementos HTML utilizando JavaScript também, de maneira dinâmica. Já falei disso no capítulo anterior, onde mostrei como fazer usando a propriedade `element.style`. Para reforçar, aqui pegamos um parágrafo com id 'myP' e trocamos a cor do seu texto dinamicamente.

Código 5.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
● ● ●  
1 document.getElementById( '#myP' ).style.color = 'red';
```

A propriedade `style` está presente em todos elementos HTML e possui como sub-propriedades as mesmas características disponíveis no CSS, com a única atenção é que estilos CSS com hífen mudam de nomenclatura, pois o padrão do JS é camel case.

- » `background-color` no CSS vira `backgroundColor` no JS;
- » `border-left` no CSS vira `borderLeft` no JS;

E assim por diante.

Além disso, podemos adicionar e remover classes CSS a um elemento usando a propriedade `classList` do `element`, como abaixo.

Código 5.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
● ● ●  
1 document.getElementById( '#myP' ).classList.add( 'nomeClasse' );
```

Outra questão importante é que quando vimos o capítulo de JavaScript client-side aprendemos a utilizar seus seletores. Pois bem, um poderoso seletor que está disponível no JS é o seletor por classe, como abaixo.

Código 5.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 document.getElementsByClassName( '.paragrafoInicial' );
```

Aqui selecionamos todos os elementos HTML que possuam a classe paragrafoInicial definida em seu atributo class. Simples assim. Essa é uma maneira extremamente útil de, com uso de laços de repetição, executar operações sobre grupo de elementos, como esconder todos...

Código 5.3: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 const col = document.getElementsByClassName( "teste" );
2 for(let item of col)
3     item.style.display = "none";
```

Note que com o exemplo acima já é possível abrir a sua cabeça para as possibilidades de manipulação de elementos em lote.

Repare também como usei o for com uma construção ligeiramente diferente. Isso porque o retorno do getElements (ByClassName, ByTagName, etc) é um Iterator e não um array comum. Assim sendo, aquela construção de for ali em cima, que chamaos de for-each percorrerá todos os elementos do Iterator independente da quantidade, ordem, etc.

## Exercitando

Vamos fazer um exercício bem rápido e simples com CSS, apenas para ver se você conseguiu entender como CSS funciona. Vamos usar aqui o mesmo HTML final do capítulo anterior, aquele com o formulário de cadastro e tabela de listagem.

Primeiro, vamos entender onde queremos chegar:

## Cadastro

Nome:

Idade:

UF: RS

Listar

| Salvar

Nenhuma obra-prima, mas bem mais agradável que o layout anterior na minha opinião.

E a tabela de clientes:

## Listagem

Nome	Idade	UF	Ações
Luiz	29	RS	<input type="button" value="x"/>
Teste	28	RS	<input type="button" value="x"/>
jquery	15	SC	<input type="button" value="x"/>
ajax	12	SC	<input type="button" value="x"/>
mais um	80	RS	<input type="button" value="x"/>
outro cliente	30	RS	<input type="button" value="x"/>

**Nota:** não investirei pesado em CSS aqui pois este não é exatamente um livro de design, algo que requer habilidades além da minha capacidade enquanto programador. :P

Para avançar mais rapidamente com estilos, sugiro dar uma olhada em Bootstrap

Você acha que consegue sozinho? Tente!

Comece criando uma pasta css na raiz do seu projeto e dentro dela um arquivo estilos.css vazio, referenciando-o no HEAD do seu index.html:

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 <link href="css/estilos.css" rel="stylesheet" />
```

Agora dê uma olhada no HTML antigo desse projeto. Notará que algumas vezes usei o atributo STYLE em algumas tags. Embora isso funcione perfeitamente, o ideal é não utilizar estilo inline, ou seja, estilos ao longo do HTML. O mais elegante e correto é colocar todos os estilos em arquivos CSS, referenciados no HTML, assim como estamos fazendo com nossos JavaScripts.

**Nota:** a regra geral diz que HTML é a forma, CSS o estilo e JS o comportamento e que cada um desses três elementos que formam a base do front-end moderno deve estar em seus respectivos arquivos.

Primeiro, note o uso do atributo STYLE da nossa table de listagem. Ele está indicando que a table deve ocupar no máximo metade da largura do contêiner mais externo à ela (a DIV).

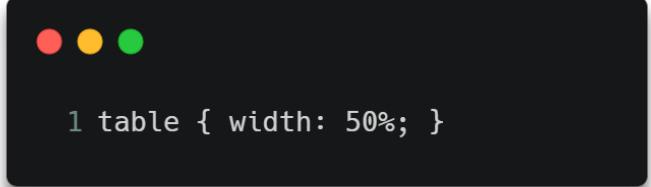
Código 4.5: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 <table style="width:50%">
```

Remova esse atributo style e vamos colocar esse estilo de tabela em nosso arquivo estilos.css:

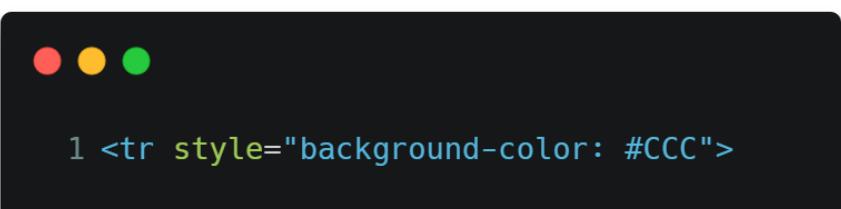
Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 table { width: 50%; }
```

Agora, repare no style da TR que fica no THEAD da table. Ele diz que a cor de fundo desta TR deve ser cinza claro (#CCC é um tom de cinza claro).

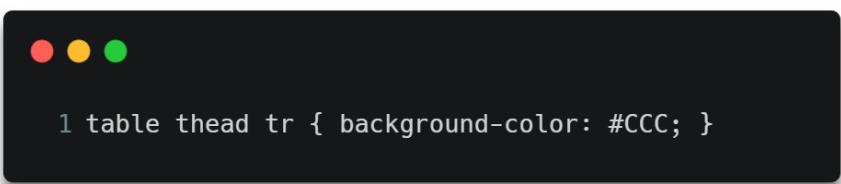
Código 4.5: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 <tr style="background-color: #CCC">
```

Remova este estilo e vamos colocá-lo via arquivo estilos.css:

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>



```
1 table thead tr { background-color: #CCC; }
```

Note que aqui eu tive de dizer toda a hierarquia até chegar no componente que desejo alterar. Experimente colocar apenas `tr { ... }` e verá que TODAS as TRs da página ficarão com fundo cinza e não é o que queremos.

Agora, repare no estilo das colunas dessa TR. A primeira é maior que as outras, ocupando 50% do tamanho total da tabela. As demais estão com 15% cada, sendo que a última terá o restante que é 20%.

Código 4.5: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <td style="width:50%">Nome</td>
2 <td style="width:15%">Idade</td>
3 <td style="width:15%">UF</td>
4 <td>Açōes</td>
```

Como removemos esses estilos individuais mantendo a diferença de tamanho entre eles?

Uma ideia é criando classes CSS para as duas variações que temos (sendo que a última célula recebe o que sobrar). No entanto, isso fica um tanto verboso demais no HTML e podemos fazer algo mais elegante usando alguns recursos do CSS. Remova os estilos daquelas TDs e coloque o seguinte no seu estilos.css:

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 table thead tr td:first-child { width:50%; }
2
3 table thead tr td:last-child { width:20%; }
```

Aqui usei os modificadores first-child e last-child para dizer que a primeira e última células da linha (tr) da thead possuem tamanho fixado em 50% e 20%, respectivamente, sobrando outros 15% para cada uma das células intermediárias.

Essa é uma notação mais profissional e caso você tenha resolvido com classes, não há problema.

Com esse último ajuste conseguimos eliminar todos os estilos que haviam sido escritos diretamente no HTML. Agora é a hora de fazer novos estilos para adequar nosso HTML atual à imagem do início desse exercício.

Primeiro, não é exatamente um estilo mas incomoda: use o atributo cellspacing da table para remover os espaços entre as células:

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 <table cellspacing="0">
```

Agora, vamos começar adicionando uma margem interna lateral pois tudo está muito “grudado” na esquerda da página. Fazemos isso definindo um estilo para a tag body no estilos.css:

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 body{ padding-left: 10px; }
```

Já que a primeira coisa que vemos é o formulário de cadastro, vamos começar por ele, estilizando o form:

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 form{
2     width: 50%;
3     text-align: right;
4 }
```

Dei uma largura máxima de 50% e alinhei-o à direita, pois isso deixará o alinhamento de labels e campos mais interessante.

Agora para deixar todos os nossos inputs mais agradáveis visualmente, crie um estilo global para todos eles, deixando-os mais arredondados (border-radius) e com mais espaçamento interno (padding):

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 input {  
2     padding: 5px;  
3     border-radius: 4px;  
4 }
```

Você vai notar que isso não afeta o select, uma vez que ele não é um input. Mesmo que você coloque essas mesmas propriedades CSS em um estilo para o select não vai adiantar pois os navegadores possuem aparências próprias pra ele. Para conseguir estilizar o select você precisa primeiro remover o estilo atual dele, como abaixo:

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 select{  
2     -webkit-appearance: none;  
3     -moz-appearance: none;  
4     appearance: none;  
5     padding: 5px;  
6     width: 53%;  
7 }
```

As propriedades iniciadas com ‘-’ são exclusivas de alguns browsers como Firefox (moz) e Chrome (webkit). Assim conseguiremos que inputs e selects fiquem mais parecidos visualmente.

Opa, mas a largura deles está diferente, não? Isso porque o select está com um width de 53% da largura do container externo. Vamos criar estilos para a largura dos inputs do formulário também:

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 input[type="text"], input[type="number"] { width: 50%; }
```

Aqui usei um truque de CSS para aplicar um estilo a dois tipos de componentes diferentes, separados por vírgulas. Assim, todos os textos (como o campo de nome) e os números (como o campo de idade) terão a mesma largura, que apesar de ser 3% menor que o select, é visualmente idêntica.

Mas e os botões? Se você olhar a imagem verá que eles são do mesmo tamanho. Sendo assim, vamos criar um estilo idêntico para os dois:

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 #btnListar, input[type="submit"] { width: 25%; }
```

Como o btnListar tem id, usamos ele. Já o outro botão tratei como um submit genérico, visto que é o único na página.

Pra completar nossa página, vamos apenas alinhar o título de maneira mais adequada ao formulário, definindo o seguinte estilo:

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 h2 {  
2     text-align: right;  
3     padding-right: 50%;  
4 }
```

Se eu colocasse apenas um text-align para direita, ele ia grudar no lado direito da tela. Então resolvi colocar um padding de 50% na direita para empurrá-lo de volta mais para o centro. Note que isso é completamente diferente de usar um “text-align: center”, teste no seu navegador verá a diferença.

E agora a nossa tabela!

Vamos começar botando um padding para todas as informações das células ficarem menos coladas em relação às bordas, bem como vamos centralizar essas informações:

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 table tr td{  
2     padding: 5px;  
3     text-align: center;  
4 }
```

Eu particularmente não gosto do nome centralizado pois ele é muito comprido, então vou adicionar outro estilo apenas para a primeira célula da tabela, usando o recurso ‘:first-child’ novamente:

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 table tr td:first-child{ text-align: left; }
```

E por fim, a cereja do bolo, vamos colocar um efeito CSS que vai trocar a cor do fundo de uma linha da tabela quando o mouse estiver passando por cima dela:

Código 5.4: disponível em <https://www.luiztools.com.br/ebook-frontend-fontes>

```
1 table tbody tr:hover{  
2     background-color: #ccc;  
3 }
```

Isso é particularmente útil em tabelas muito grandes, para facilitar a leitura. Obtemos este efeito de sobreposição graças ao efeito ‘:hover’ que aplicamos em todas as TRs, definindo uma nova cor de fundo.

E com isso terminamos mais este capítulo de front-end!

*Quer fazer um curso online de Desenvolvimento Web FullStack JS com o autor deste ebook? Acesse <https://www.luiztools.com.br/curso-fullstack>*

# SEGUINDO EM FRENTE

---

“

A code is like love, it has created with clear intentions at the beginning, but it can get complicated.

- *Gerry Geek*

”

Este livro termina aqui.

Pois é, certamente você está agora com uma vontade louca de aprender mais, criar aplicações web incríveis com front-ends maneiras e de quebra que o deixem cheio de dinheiro na conta bancária, não é mesmo?

Pois é, eu também! :)

Este livro é pequeno se comparado com o universo de possibilidades que a web nos traz. Como professor, costumo dividir o aprendizado de alguma tecnologia em duas grandes etapas: aprender o básico e executar o que foi aprendido no mercado, para alcançar os níveis intermediários e avançados. Acho que este guia atende bem ao primeiro requisito, mas o segundo só depende de você.

De nada adianta saber muita teoria se você não aplicar ela. Então agora que terminou de ler este livro e já conhece uma série de formas de criar páginas web com estas fantásticas tecnologias, inicie hoje mesmo (não importa se for tarde) um projeto que as use. Caso não tenha nenhuma ideia, cadastre-se agora mesmo em alguma plataforma de freelancing. Mesmo que não ganhe muito dinheiro em seus primeiros projetos, somente chegarão os projetos grandes, bem pagos e realmente interessantes depois que você tiver experiência.

Me despeço de você leitor com uma sensação de dever cumprido. Caso acredite que está pronto para ainda mais tutoriais bacanas, sugiro dar uma olhada em meu blog <https://www.luiztools.com.br>.

Outras fontes excelentes de conhecimentos sobre HTML, CSS e JavaScript é o site da <https://developer.mozilla.org/pt-BR/>, lá possui tudo que você pode precisar!

Caso tenha gostado do material, envie esse ebook a um amigo que também deseja aprender a programar para web. Não tenha medo da concorrência e abrace a ideia de ter um sócio que possa lhe ajudar nos projetos.

Caso não tenha gostado tanto assim, envie suas dúvidas, críticas e sugestões para contato@luiztools.com.br que estou sempre disposto a melhorar.

Um abraço e até a próxima!

# MEUS CURSOS

## Curso online NODE.JS e MONGODB

[SAIBA MAIS...](#)

## Curso online Scrum e métodos Ágeis

[SAIBA MAIS...](#)

## Curso online Jira

[SAIBA MAIS...](#)

## Curso online Web Full Stack JavaScript

[SAIBA MAIS...](#)

## Curso online React Native com Firebase

[SAIBA MAIS...](#)

Conheça todos os meus cursos

# MEUS LIVROS



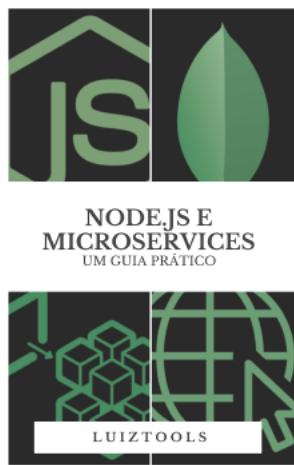
Programação  
Web com Node.js

[SAIBA MAIS...](#)



Programação  
Web com Node.js

[SAIBA MAIS...](#)



NODEJS E  
MICROSERVICES  
UM GUIA PRÁTICO



MongoDB  
para  
Iniciantes

POR LUIZTOOLS

MongoDB  
para Iniciantes

[SAIBA MAIS...](#)



Scrum e  
Métodos Ágeis

[SAIBA MAIS...](#)



Agile Coaching

[SAIBA MAIS...](#)



Criando apps  
para empresas  
com Android

[SAIBA MAIS...](#)



Java para  
iniciantes

[SAIBA MAIS...](#)

## Conheça todos os meus livros

Aproveita e segue nas redes sociais:

