

Instituto Nacional de Telecomunicações – INATEL

**Pós-graduação em Desenvolvimento de Aplicações para Dispositivos
Móveis e Cloud Computing**

DM121 – Introdução ao desenvolvimento Web: HTML5, CSS3 e Javascript

Sobre a apostila

Este é um material de apoio para o curso de Pós-graduação em Desenvolvimento de Aplicações para Dispositivos Móveis e Cloud Computing do Instituto Nacional de Telecomunicações – INATEL, referente a disciplina DM121 – Introdução ao desenvolvimento Web: HTML5, CSS3 e Javascript.

Sumário

1. Introdução	4
2. HTTP e HTML	5
3. Desenvolvimento Web.....	6
4. Ambiente de desenvolvimento	8
5. Introdução ao HTML.....	9
5.1. Sintaxe HTML.....	9
5.1.1. Composição das tags	9
5.1.2. Visão geral dos elementos do HTML e seus atributos.....	10
5.1.3. Estruturando conteúdo	12
5.1.4. Tags HTML5.....	14
5.1.5. Novos atributos, atributos descontinuados e atributos não permitidos	17
6. Introdução ao CSS	20
6.1. Sintaxe CSS	20
6.1.1. Vinculando CSS ao documento HTML	20
6.1.2. Visão geral dos seletores e propriedades do CSS	21
6.2. CSS3 Grid e Flexbox.....	47
7. Introdução ao Javascript	72
7.1. Sintaxe Javascript	72
7.2. DOM API	90
8. Tópicos Avançados.....	97
8.1. ECMA Script versões	97
8.2. ECMA Script 2015 – ES6	98
8.3. ECMA Script 2016 – ES7	108
8.4. ECMA Script 2017 – ES8	108
8.5. ECMA Script 2018 – ES9	109
8.6. ECMA Script 2019 – ES10	109
9. REFERÊNCIAS	112

1. Introdução

Aplicações web são baseadas em interações entre duas partes que são conhecidas como cliente e servidor, onde o cliente pode ser qualquer navegador de internet (Google Chrome, Firefox, Internet Explorer, Safari, Opera entre outros). O servidor é responsável em prover recursos para o cliente, ou seja, basicamente irá processar requisições dos clientes e baseado nessas requisições irá prover os recursos solicitados.

A fim de viabilizar essa interação entre as partes é necessário que ambos cliente e servidor “falem a mesma língua”, ou seja, utilizem o mesmo protocolo de comunicação que no contexto de aplicações Web o protocolo utilizado é o HTTP.



Podemos caracterizar aplicações web da seguinte maneira:

- Aplicações web são fundamentadas no conceito cliente e servidor
- Utilizam o protocolo HTTP para se comunicar
- A camada cliente é o navegador
- A camada servidor é responsável por processar as requisições e servir os conteúdos

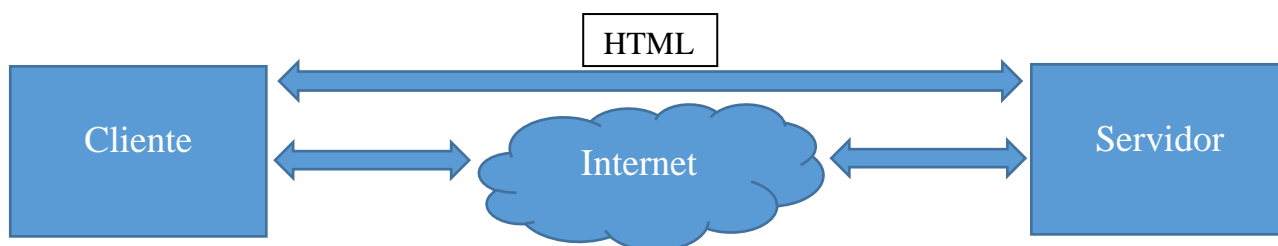
2. HTTP e HTML

O HTTP (*Hypertext Transfer Protocol*) é o protocolo padrão da Web cujo a finalidade é possibilitar a transferência de documentos *Hypermedia* (gráficos, áudio, vídeo, textos).

Uma das extensões de documentos *Hypermedia* é o *Hypertext* que pode ser categorizado em *Hypertext Markup Language* (HTML). Não está no escopo desse documento abordar detalhes profundos do protocolo HTTP, para mais detalhes consulte [aqui](#).

Dentro do contexto de aplicações web que é o objeto do nosso estudo podemos chegar as seguintes conclusões:

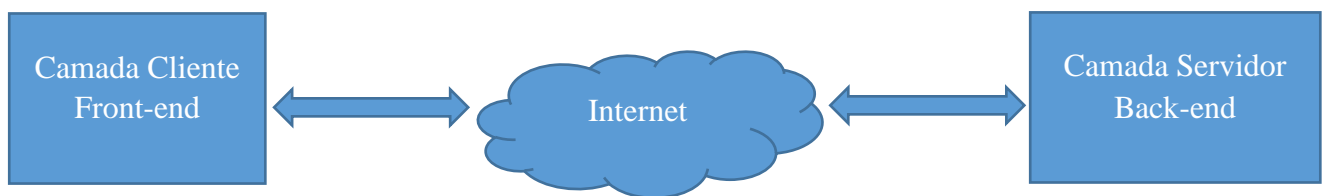
- O protocolo HTTP transfere *medias* (gráficos, áudio, vídeo, textos)
- O protocolo HTTP transfere *Hypertext*, em outras palavras HTML (podemos considerar paginas Web aqui)
- O protocolo HTTP é o protocolo padrão da Web



3. Desenvolvimento Web

Conforme visto nos capítulos anteriores, aplicações web são baseadas no conceito cliente/servidor e são fundamentadas no protocolo padrão da web, o HTTP. Com base nessas características o processo de desenvolvimento pode ser dividido em duas frentes: Desenvolvimento na camada cliente e Desenvolvimento na camada servidor.

É comum referenciar o desenvolvimento na camada cliente como “Desenvolvimento Front-end” e referenciar o desenvolvimento na camada servidor como “Desenvolvimento Back-end”. A separação do desenvolvimento Web em camadas, além de facilitar na divisão de responsabilidades também nos apresenta de maneira mais direta o funcionamento da Web.



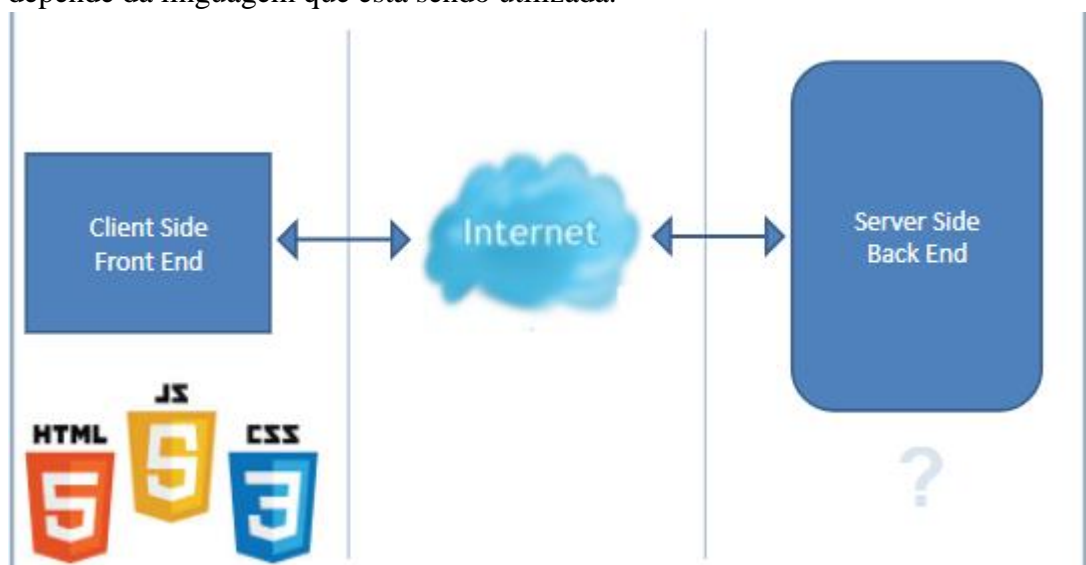
Do ponto de vista de linguagens de programação e tecnologias podemos resumir as duas camadas da seguinte maneira:

Front-end

- Linguagens: HTML, CSS e Javascript
- O navegador é a fronteira com o usuário

Back-end

- Linguagens: Java, PHP, Javascript (node.js), Python, Ruby, entre outras.
- O servidor é que irá lidar com as requisições, ou seja, irá responder as requisições originadas do cliente e a forma com que o servidor irá lidar com as requisições depende da linguagem que está sendo utilizada.



Em resumo conclui-se que:

- A camada Fron-end é fortemente acoplada com o navegador do usuário
- O navegador do usuário interpreta e apresenta páginas ao usuário no formato HTML
- A camada Back-end é dependente do servidor e pode suportar várias linguagens de programação.
- Uma vez que a camada front-end é executada em cima do navegador, basicamente a saída sempre será em HTML, CSS e Javascript.

4. Ambiente de desenvolvimento

O foco será no desenvolvimento web especificamente no Front-end, ou seja, HTML, CSS e Javascript. A maioria das IDE's (Ambiente Integrado de Desenvolvimento) de mercado suportam facilmente essas linguagens, editores de textos também podem ser utilizados como ferramentas de desenvolvimento Web. A ferramenta de edição de código que será utilizada é o editor [Visual Studio Code](#) que é um editor de texto leve, suporta várias linguagens de programação e é free.

O Visual Studio Code possui versões para MacOS, Linux e Windows. O procedimento de instalação para cada plataforma pode ser encontrado [aqui](#).

Para executar e depurar páginas web apenas o navegador de internet é necessário. Ao decorrer deste curso o navegador utilizado será o [Google Chrome](#), porém pode ser utilizado qualquer navegador de sua escolha.

5. Introdução ao HTML

O HTML5 é a nova versão do HTML4. Enquanto o WHATWG define as regras de marcação que usaremos no HTML5 e no XHTML, eles também definem APIs que formarão a base da arquitetura web.

Um dos principais objetivos do HTML5 é facilitar a manipulação do elemento possibilitando o desenvolvedor a modificar as características dos objetos de forma não intrusiva e de maneira que seja transparente para o usuário final.

Ao contrário das versões anteriores, o HTML5 fornece ferramentas para a CSS e o Javascript fazerem seu trabalho da melhor maneira possível. O HTML5 permite por meio de suas APIs a manipulação das características destes elementos, de forma que o website ou a aplicação continue leve e funcional, sem a necessidade de criações de grandes blocos de scripts.

O HTML5 também cria novas tags e modifica a função de outras. As versões antigas do HTML não continham um padrão universal para a criação de seções comuns e específicas como rodapé, cabeçalho, sidebar, menus e etc.

Não havia um padrão de nomenclatura de IDs, Classes ou tags. Não havia um método de capturar de maneira automática as informações localizadas nos rodapés dos websites.

Há outros elementos e atributos que sua função e significado foram modificados e que agora podem ser reutilizados de forma mais eficaz. Por exemplo, tags como B e I que foram descontinuados em versões anteriores do HTML agora assumem funções diferentes e entregam mais significado para os usuários.

O HTML5 modifica a forma de como escrevemos código e organizamos a informação na página. Seria mais semântica com menos código. Seria mais interatividade sem a necessidade de instalação de plugins e perda de performance. É a criação de código inter-operável, pronto para futuros dispositivos e que facilita a reutilização da informação de diversas formas.

2.2. HTML 5 E SUAS MUDANÇAS

Quando o HTML4 foi lançado, o W3C alertou os desenvolvedores sobre algumas boas práticas que deveriam ser seguidas ao produzir códigos client-side. Desde este tempo, assuntos como a separação da estrutura do código com a formatação e princípios de acessibilidade foram trazidos para discussões e à atenção dos fabricantes e desenvolvedores.

O HTML4 ainda não trazia diferencial real para a semântica do código. O HTML4 também não facilitava a manipulação dos elementos via Javascript ou CSS. Se você quisesse criar um sistema com a possibilidade de Drag'n Drop de elementos, por exemplo, era necessário criar um grande script, com bugs e que muitas vezes não funcionavam de acordo em todos os browsers.

5.1. Sintaxe HTML

5.1.1. Composição das tags

A linguagem HTML é um conjunto de tags (etiquetas). Essas tags podem ser divididas em:

Tags que recebem conteúdo:

```
<h2 class="titulo">Tag H2</h2>
```

Tags que não recebem conteúdo:

```
<br>
```

5.1.2. Visão geral dos elementos do HTML e seus atributos

A estrutura de um arquivo HTML (página HTML) tem a seguinte estrutura de tags:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>

</body>
</html>
```

Doctype

O DOCTYPE indica ao navegador e para outros meios qual especificação de código utilizar. Isso determina como será exibido o conteúdo lido. Em outras versões, era necessário informar o DTD, mas agora é responsabilidade dos navegadores referenciar o tipo de documento.

O Doctype não é uma tag do HTML, mas uma instrução para que o browser tenha informações sobre qual versão de HTML que o código foi escrito.

	HTML 4	HTML 5
DOCTYPE	<pre><!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd"></pre>	<pre><!DOCTYPE html></pre>

Head

Dentro da tag Head é possível encontrar as tags de metadados (<meta>), que são as informações sobre a página e sobre o conteúdo ali publicado.

Meta

	HTML 4	HTML 5
Meta	<code><meta http-equiv="Content-Type" content="text/html; charset=utf-8"></code>	<code><meta charset="utf-8"/></code>

Com HTML 5 ficou muito mais fácil declarar o CHARSET, visto que não é preciso decorar uma linha inteira de código. O CHARSET é usado para definir o conjunto de caracteres que está sendo usado em uma página web. Como os sites são acessados por várias partes do mundo, foi necessário criar um conjunto único de caracteres que poderia ser usado para interpretar as páginas de maneira correta. O UNICODE fornece um número para cada caractere, fazendo com que, independentemente da plataforma ou navegador, esse caractere seja interpretado de maneira correta. Um dos CHARSET mais utilizados e que, inclusive, é usado no exemplo acima, é o UTF-8 (Unicode Transformation Format 8 bits), uma variação do UNICODE.

Link

A tag Link, usada para importar elementos externos, como arquivos CSS e RSS, sofreu uma pequena alteração. Não é mais necessário utilizar o type na hora de declarar uma folha de estilo, e também não é mais necessário o uso da barra no final da declaração.

	HTML 4	HTML 5
Link	<code><link rel="stylesheet" type="text/css" href="estilo.css"></code>	<code><link rel="stylesheet" href="estilo.css"></code>

Script

A regra de exclusão do type também se aplica para a tag script

	HTML 4	HTML 5
Script	<code><script type="text/javascript" src="script.js"></script></code>	<code><script src="script.js"></script></code>

5.1.3. Estruturando conteúdo

Há regras básicas que nós já conhecemos e que estão no HTML desde o início. Estas regras definem onde os elementos podem ou não estar. Se eles podem ser filhos ou pais de outros elementos e quais os seus comportamentos. Essas regras são chamadas de Modelos de Conteúdo.

Dentre todas as categorias de modelos de conteúdo, existem dois tipos de elementos: elementos de linha e os elementos de bloco.

Os elementos de linha marcam, na maioria das vezes, textos, elementos de formulários, imagens. Alguns exemplos de tags: **a**, **strong**, **em**, **img**, **input**, **br**, **span**.

Já os elementos de blocos são como caixas, que dividem o conteúdo em seções do layout.

Abaixo segue algumas premissas que você precisa relembrar e conhecer:

- Os elementos de linha podem conter outros elementos de linha, dependendo da categoria que ele se encontra. Por exemplo: o elemento **a** não pode conter o elemento **LABEL**. Mas o inverso é possível.
- Os elementos de linha nunca podem conter elementos de bloco.
- Elementos de bloco sempre podem conter elementos de linha.
- Elementos de bloco podem conter elementos de bloco, dependendo da categoria que ele se encontra. Por exemplo, um parágrafo não pode conter um **div**. Mas o inverso é possível.

Categoria

Estes dois grandes grupos podem ser divididos em categorias. Estas categorias dizem qual modelo de conteúdo o elemento se encaixa e como pode ser seu comportamento.

Os navegadores utilizam muito estas regras para definir o fluxo de informação e quais as suas utilizações. Alguns navegadores podem ajudar o desenvolvedor tentando consertar algum erro de sintaxe. Outros simplesmente quebram o layout ou a aplicação no local que o erro de sintaxe acontece.

Cada elemento no HTML pode ou não fazer parte de um grupo de elementos com características similares. As categorias estão a seguir.

- Metadata content
- Flow content
- Sectioning content
- Heading content
- Phrasing content
- Embedded content
- Interactive content

Metadata content

Este conteúdo vem antes da apresentação, formando uma relação com o documento e seu conteúdo com outros documentos que distribuem informação por outros meios.

Base Command Link Meta noscript Script, style, title.

Flow content

A maioria dos elementos utilizados no **body** e aplicações são categorizados como Flow Content. Cada um desses elementos deve conter, ao menos, um elemento textual ou um elemento descendente que faça parte da categoria.

a abbr address area article aside audio b bdo blockquote br button canvas cite code command datalist del details dfn div dl Em embed fieldset figure footer form h1 até h6 header hgroup hr i iframe img input ins kbd keygen label link map mark math menu meta meter nav noscript object ol output p pre progress q Ruby samp script section select small span strong style sub sup svg table textarea time ul var video wbr text

Sectioning content

São elementos que definem seções do código, como rodapés, cabeçalhos, área de textos principais, links de navegação, entre outros. Nesta categoria, o HTML5 inseriu algumas tags para melhorar a semântica estrutural. Com esses elementos, é mais fácil identificar que seção da página está sendo feita com determinado código.

header footer article aside nav

Heading content

Os elementos da categoria Heading definem uma seção de cabeçalhos, que podem estar contidos em um elemento na categoria Sectioning.

h1 até h6 hgroup header

Phrasing content

Fazem parte desta categoria elementos que marcam o texto do documento, bem como os elementos que marcam este texto dentro do elemento de parágrafo.

a abbr area audio b bdo br button canvas vite code command datalist del dfn em embed i iframe img input ins kbd keygen label link map mark math meta meter noscript object output progress q ruby samp script sup select small span strong sub svn textarea time var video text time var wbr

Embedded content

Na categoria Embedded, há elementos que importam outra fonte de informação para o documento.

audio canvas embed iframe img math object svg vídeo

Interactive content

São elementos que interagem com o usuário através de algum evento. Estes elementos podem ser ativados por algum comportamento, ou seja, o usuário pode ativá-los de alguma maneira. O user-agent permite que o usuário ative estes elementos manualmente através de um teclado, mouse, comando de voz, toque, etc.

a audio(se o atributo control for utilizado) button details embed iframe img(se o atributo usemap for utilizado) input(se o atributo type não tiver o valor hidden) keygen label menu(se o atributo type tiver o valor toolbar) Object(se o atributo usemap for utilizado) select textarea video(se o atributo control for utilizado)

É possível que um elemento faça parte de mais de uma categoria, dependendo do contexto em que está inserido. O elemento áudio, por exemplo, está na categoria Embedded Content por incluir um arquivo de áudio na página, mas também está na categoria Interactive Content, pois possui controles que interagem com o usuário como o play, pause, etc.

5.1.4. Tags HTML5

O HTML5 criou uma série de elementos que nos ajudam a definir os setores principais no documento. Com a ajuda destes elementos, podemos por exemplo diferenciar diretamente pelo código HTML5 áreas importantes do site como sidebar, rodapé e cabeçalho. Conseguimos seccionar a área de conteúdo indicando onde exatamente está o texto do artigo. Estas mudanças simplificam o trabalho de sistemas como os dos buscadores.

Com o HTML5 os buscadores conseguem vasculhar o código de maneira mais eficaz, procurando e guardando informações mais exatas e levando menos tempo para armazenar essa informação.

Elementos de seção

A tag **div** tem a função de criar divisões.

Quando criamos um website, dividimos as áreas do layout em seções. O problema desta tag, é que ele não tem nenhum significado semântico. Os sistemas de busca, por exemplo, não têm como saber o que é um rodapé, um cabeçalho, sidebar e etc, porque tudo é feito com div e assim damos o mesmo nível hierárquico de informação para todas as seções.

Para resolver esse problema, foi criado um conjunto novo de elementos que além de dividir as áreas do layout, ele entrega significado. Esses elementos são chamados de Conteúdos de Seções ou Sectioning content.

Esse conjunto de tags tem a função de dividir as áreas do layout como fazíamos com a tag div, mas cada uma delas carrega um significado específico.

Tag Section

O elemento **section** define uma nova seção genérica no documento. Por exemplo, a home de um website pode ser dividida em diversas seções: introdução, destaque, novidades, informação de contato e chamadas para conteúdo interno.

Basicamente o elemento section substitui o div em muitos momentos.

```
<section id="rock">
  <h2>Rock</h2>
</section>

<section id="blues">
  <h2>Blues</h2>
</section>

<section id="jazz">
  <h2>Jazz</h2>
</section>
```

Tag Nav

O elemento **nav** representa uma seção da página que contém *links* para outras partes do website. Nem todos os grupos de *links* devem ser elementos nav, apenas aqueles grupos que contém *links* importantes. Aqueles *links* que são considerados principais, por exemplo o menu principal do *site*. No caso dos portais, aquele menu lateral com uma série de *links* separados por assuntos poderiam ser uma nav. Isso também pode ser aplicado para aqueles blocos de *links* que geralmente são colocados no rodapé.

```
<nav>
  <h3>Esportes</h3>
  <ul>
    <li><a href="#">Futebol</a></li>
    <li><a href="#">Natação</a></li>
    <li><a href="#">Vôlei</a></li>
    <li><a href="#">Basket</a></li>
  </ul>
  <h3>Educação</h3>
  <ul>
    <li><a href="#">Português</a></li>
    <li><a href="#">Matemática</a></li>
  </ul>
  <h3>Notícias</h3>
  <ul>
    <li><a href="#">Esporte</a></li>
    <li><a href="#">Política</a></li>
    <li><a href="#">Tecnologia</a></li>
  </ul>
</nav>
```

Com a tag **nav**, há uma indicação de que aquele grupo é uma seção (**nav** é um tipo de **section**. Enquanto a tag **section** serve para indicar seções no site, a tag **nav** indica que um determinado grupo é uma seção de navegação) é um bloco de navegação.

Entenda que o **nav** não carrega qualquer tipo de links. Tenha em mente que sempre que usar o nav, ele irá carregar grupos de links ligados ao site. Normalmente estes links são links internos.

Tag article

O elemento **article** é onde colocamos o texto ou a informação principal.

Imagine que você está visitando um blog, há uma sidebar e há também o bloco de texto, que seria o bloco que carrega o conteúdo principal do site.

Este bloco seria marcado como article. Dentro deste **article** haverá toda a informação necessária sobre o artigo: data de publicação, título, autor, o texto, informações de outros artigos relacionados etc.

Para entender melhor, a informação que vai dentro do **article** é exatamente a mesma informação que os leitores de feeds capturam do seu RSS/Atom. O leitor de FEED não disponibiliza o menu do site, a sidebar e etc, ele apenas disponibiliza o texto principal do post, e será esse texto que haverá no elemento **article**.

```
<article>
  <h1>HTML 5</h1>
  <p>
    Lorem ipsum dolor sit amet consectetur adipiscing elit.
    Ea architecto totam ex recusandae laboriosam facere quod ducimus
  </p>
</article>
```

Tag Aside

O elemento **aside** representa um bloco de informação relativa ao conteúdo principal.

Algumas utilidades do **aside**: citações ou sidebars, agrupamento de publicidade, grupos e blocos de navegação ou para qualquer outro conteúdo que tenha importância secundária e relativa ao conteúdo principal da página.

Dentro do **aside** você pode agregar por exemplo grupos de elementos nav, headers, sections e etc, permitindo produzir um menu lateral com os grupos de informações.

```
<aside id="menu-lateral">
  <nav>
    <h3>Esportes</h3>
    <ul>
      <li><a href="#">Futebol</a></li>
      <li><a href="#">Natação</a></li>
      <li><a href="#">Vôlei</a></li>
      <li><a href="#">Basket</a></li>
    </ul>
    <h3>Educação</h3>
    <ul>
```



```

        <li><a href="#">Português</a></li>
        <li><a href="#">Matemática</a></li>
    </ul>
    <h3>Notícias</h3>
    <ul>
        <li><a href="#">Esporte</a></li>
        <li><a href="#">Política</a></li>
        <li><a href="#">Tecnologia</a></li>
    </ul>
</nav>
</aside>

```

5.1.5. Novos atributos, atributos descontinuados e atributos não permitidos

Alguns elementos ganharam novos atributos:

- O atributo autofocus pode ser especificado nos elementos input (exceto quando há atributo hidden atribuído), textarea, select e button.
- A tag a passa a suportar o atributo media como a tag link.
- A tag form ganha um atributo chamado novalidate. Quando aplicado o formulário pode ser enviado sem validação de dados.
- O elemento ol ganhou um atributo chamado reversed. Quando ele é aplicado os indicadores da lista são colocados na ordem inversa, isto é, da forma descendente.
- O elemento fieldset agora permite o atributo disabled. Quando aplicado, todos os filhos de fieldset são desativados.
- O novo atributo placeholder pode ser colocado em inputs e textareas.
- O elemento area agora suporta os atributos href, lang e rel como os elementos a e link
- O elemento base agora suporta o atributo target assim como o elemento a. O atributo target também não está mais descontinuado nos elementos a e area porque são úteis para aplicações web.
- Os atributos abaixo foram descontinuados:
 - O atributo border utilizado na tag img.
 - O atributo language na tag script.
 - O atributo name na tag a. Porque os desenvolvedores utilizam ID em vez de name.
 - O atributo summary na tag table.
- Estes atributos foram descontinuados porque modificam a formatação do elemento e suas funções são melhores controladas pelo CSS:
 - align como atributo da tag caption, iframe, img, input, object, legend, table, hr, div, h1, h2, h3, h4, h5, h6, p, col, colgroup, tbody, td, tfoot, th, thead e tr.
- alink, link, text e vlink como atributos da tag body.
- background como atributo da tag body.
- bgcolor como atributo da tag table, tr, td, th e body.
- border como atributo da tag table e object.
- cellpadding e cellspacing como atributos da tag table.
- char e charoff como atributos da tag col, colgroup, tbody, td, tfoot, th, thead e tr.

- clear como atributo da tag br.
- compact como atributo da tag dl, menu, ol e ul
- frame como atributo da tag table.
- frameborder como atributo da tag iframe.
- height como atributo da tag td e th.
- hspace e vspace como atributos da tag img e object.
- marginheight e marginwidth como atributos da tag iframe.
- noshade como atributo da tag hr.
- nowrap como atributo da tag td e th.
- rules como atributo da tag table.
- scrolling como atributo da tag iframe.
- size como atributo da tag hr.
- type como atributo da tag li, ol e ul.
- valign como atributo da tag col, colgroup, tbody, td, tfoot, th, thead e tr.
- width como atributo da tag hr, table, td, th, col, colgroup e pre.
- Alguns atributos do HTML4 não são mais permitidos no HTML5.
- rev e charset como atributos da tag link e a.
- shape e coords como atributos da tag a.
- longdesc como atributo da tag img and iframe.
- target como atributo da tag link.
- nohref como atributo da tag area.
- profile como atributo da tag head.
- version como atributo da tag html.
- name como atributo da tag img (use id instead).
- scheme como atributo da tag meta.
- archive, classid, codebase, codetype, declare e standby como atributos da tag object.
- valuetype e type como atributos da tag param.
- axis e abbr como atributos da tag td e th.
- scope como atributo da tag td.

Elementos modificados ou ausentes

Existiam no HTML alguns elementos que traziam apenas características visuais e não semântica para o conteúdo da página. Esses elementos anteriormente foram descontinuados porque atrapalhavam o código e também porque sua função era facilmente suprida pelo CSS. Contudo, alguns destes elementos voltaram à tona com novos significados semânticos. Outros elementos que não descontinuados, mas seus significados foram modificados.

Elementos modificados

O elemento **b** passa a ter o mesmo nível semântico que um **span**, mas ainda mantém o estilo de negrito no texto. Contudo, ele não dá nenhuma importância para o texto marcado com ele.

O elemento **i** também passa a ser um **span**. O texto continua sendo itálico e para usuários de leitores de tela, a voz utilizada é modificada para indicar ênfase. Isso pode ser útil para marcar frases em outros idiomas, termos técnicos e etc.

O interessante é que nestes dois casos houve apenas uma mudança semântica.

Provavelmente você não precisará modificar códigos onde estes dois elementos são utilizados.

O elemento **a** sem o atributo **href** agora representa um placeholder no exato lugar que este link se encontra.

O elemento **address** agora é tratado como uma seção no documento.

O elemento **hr** agora tem o mesmo nível que um parágrafo, mas é utilizado para quebrar linhas e fazer separações.

O elemento **strong** ganhou mais importância.

O elemento **head** não aceita mais elementos child como seu filho.

6. Introdução ao CSS

A linguagem CSS (*Cascading Style Sheets*) foi criada em 10 de outubro de 1994 por Håkon Wium Lie e Bert Bos cujo a necessidade era de formatar documentos HTML.

Basicamente o browser executa o carregamento/parseamento do HTML e do CSS e posteriormente apresenta o documento formato ao usuário.

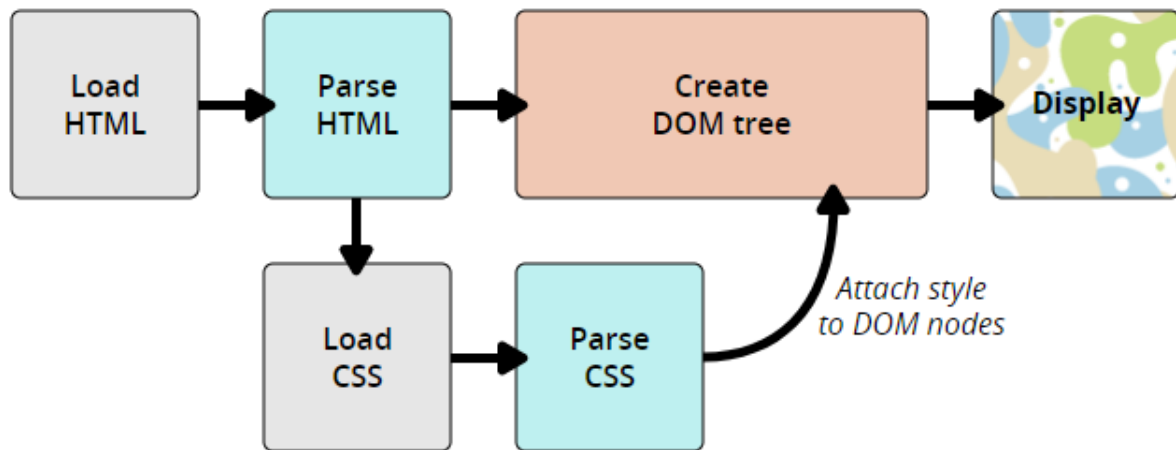


Figura 1 -Carregamento CSS

Fonte: <https://developer.mozilla.org>

Considerando o exemplo que desenvolvemos na seção anterior, se você observar não existe nenhum tipo de formatação (texto, cores, posicionamento) e é justamente para formatar o documento que iremos usar o CSS.

6.1. Sintaxe CSS

6.1.1. Vinculando CSS ao documento HTML

É possível vincular CSS ao document HTML de três maneiras:

Inline CSS

Aplicado diretamente na tag HTML.

Exemplo:

```
<p style="color: red;"> Este parágrafo ficou vermelho.</p>
```

Internal Style Sheet

Aplicado diretamente na página HTML por uma tag chamada <style></style>, entre essas tags ficam as declarações css.

Exemplo:

```
<head>
```

```
<title>my page</title>
<style type="text/css">
    p{color:red}</style>
</head>
<body>
<p>
    Isto é um parágrafo.
</p>
</body>
```

Externo CSS

Arquivo externo com extensão .css. O arquivo é “linkado” à página HTML.

Exemplo:

```
<head>
<title>Linkando folha de estilo</title>
<link rel="stylesheet" type="text/css" href="estilo.css">
<body>
    <Isto é um parágrafo.
</p>
</body>
```

6.1.2. Visão geral dos seletores e propriedades do CSS

O CSS é uma linguagem baseada em seleção e formada por um conjunto de declaração de regras. Por exemplo:

```
seletor {
    propriedade: valor;
}
```

O **seletor** representa uma estrutura que é utilizada como condicional para determinar os elementos que serão formatados.

A **propriedade** é característica que se deseja aplicar para o seletor.

O **valor** é o valor da propriedade.

Seletores de ID e Classe

id - Especifica um unico elemento.

É um atributo html e é definido no css com símbolo (#).

Exemplo:

```
#campo1 {text-align:center;color:red;}
```

O elemento HTML com **id = #campo1** irá ser formatado com a cor vermelha e o texto será centralizado.

Class - utilizado para especificar um grupo de elementos.

É um atributo html e é definido no css com símbolo (.)

```
.campo {text-align:center;}
```

O elemento HTML com a **class = .campo** irá ser formatado com o texto centralizado.

Seletores complexos

Os seletores mais básicos são de tipo (h1, div, entre outros) que são os elementos HTML, seletores de ID (#campo1) e classe (.campo), conforme visto anteriormente. Existem outros não tão comuns como os seletores de pseudoclasses (:hover) e seletores CSS3 mais complexos, incluindo [class^="content-"], :first-child.

Não está no escopo deste documento abordar detalhes de performance e eficiência em CSS, porém, é importante saber que cada tipo de seletor pode ser mais ou menos eficiente.

Considerando os mais eficientes para os menos eficientes temos os seguintes seletores:

- ID (#campo1)
- Classe (.campo)
- Tipo (h3)
- Irmão adjacente (h3 + p)
- Filho (li > ul)
- Descendente (ul a)
- Universal (*)
- Atributo ([type="text"])
- Pseudo-classe/Pseudo-elemento (li:hover)

Medidas em CSS

As principais unidades são as duas seguintes:

px (píxel) — Os píxeis representam unidades absolutas, um pixel, corresponde a um pixel no ecrã do utilizador. Exemplos: 10px, 20px, 1px.

% (percentagem) — As unidades percentuais representam unidades relativas, e são calculadas em relação ao contexto do elemento. O contexto varia em função do elemento e da propriedade. Exemplos: 1%, 100%, 5.5%.

em (unidade de medida tipográfica) essa é uma unidade relativa ao tamanho da fonte, **1em** equivale aproximadamente a **16px** (valor padrão do font-size dos navegadores).

rem (unidade de medida tipográfica) semelhante ao **em** porém terá a unidade relativa ao elemento root (o elemento body).

vw (relativo ao width do viewport). É 1/100 da largura do viewport. Se o navegador está em 800px de largura, 1vw equivale a 8px.

vh (relativo ao height do viewport). É 1/100 da altura do viewport. Se o navegador está 600px de altura, 1vh equivale a 6px.

Cores em CSS

Existe as alternativas para definição de cores por exemplo:

Palavra-chave: black, yellow, red, blue, etc.

Código RGB: rgb(vermelho, verde, azul)

Exemplos: rgb(100%, 40%, 0%); rgb(255, 102, 0).

Código Hexadecimal: #código hexadecimal

Exemplos: #326432, #000000, #0088ff.

Formatação de texto em CSS

Basicamente a formatação de texto pode ser dividida em duas categorias: Estilo da fonte e Estilo do texto.

Estilo da fonte

Para alterar o estilo de fonte usamos o **font-family**, por exemplo:

```
h1{
  font-family: sans-serif
}

p{
  font-family: Arial, Helvetica, sans-serif
}
```

Título

Lorem ipsum dolor sit amet consectetur adipisicing elit. Sapiente molestias aut nisi quasi nesciunt. Molestiae eos consequuntur atque! Asperiores nulla qui quo fugiat repellendus quae ipsum totam magni ab odit.

A propriedade **font-family** permite definir uma fonte específica ou uma lista de fontes. Esta propriedade tem algumas características importantes:

- Somente aplica as fontes que estão disponíveis, ou seja, fontes que não estão instaladas não são aplicadas.
- Quando não existe a fonte desejada o valor do font-family é o valor da fonte padrão do navegador.

O [CSS Font Stack](#) mantém uma lista atualizada das fontes que são suportadas em alguns sistemas operacionais.

A propriedade **font-size** permite definir o tamanho da fonte.

```
h1{
  font-size: 70px
}

p{
  font-size: 25px
}
```

Título

Lorem ipsum dolor sit amet consectetur adipisicing elit. Sapiente molestias aut nisi quasi nesciunt. Molestiae eos consequuntur atque! Asperiores nulla qui quo fugiat repellendus quae ipsum totam magni ab odit.

A propriedade **font-style** permite formatar o estilo da fonte.

```
h1{
  font-style: italic
}

p{
  font-style: normal
}
```


Título

Lorem ipsum dolor sit amet consectetur adipisicing elit. Sapiente molestias aut nisi quasi nesciunt. Molestiae eos consequuntur atque! Asperiores nulla qui quo fugiat repellendus quae ipsum totam magni ab odit.

A propriedade **font-weight** permite formatar a densidade da fonte.

```
h1{
  font-weight: lighter
}

p{
  font-weight: bold
}
```

Título

Lorem ipsum dolor sit amet consectetur adipisicing elit. Sapiente molestias aut nisi quasi nesciunt. Molestiae eos consequuntur atque! Asperiores nulla qui quo fugiat repellendus quae ipsum totam magni ab odit.

Estilo do texto

A propriedade **text-align** permite alinhar o texto.

```
h1{
  text-align: center
}

p{
  text-align: justify
}
```

Título

Lorem ipsum dolor sit amet consectetur adipisicing elit. Sapiente molestias aut nisi quasi nesciunt. Molestiae eos consequuntur atque! Asperiores nulla qui quo fugiat repellendus quae ipsum totam magni ab odit.

A propriedade **line-height** permite formatar a altura de cada linha de texto.

```
h1{
```

```
    line-height: 10px
}

p{
    line-height: 50px
}
```

Título

Lorem ipsum dolor sit amet consectetur adipisicing elit. Sapiente molestias aut nisi quasi nesciunt. Molestiae eos consequuntur atque! Asperiores nulla qui quo fugiat repellendus quae ipsum totam magni ab odit.

A propriedade **letter-spacing** e **word-spacing** respectivamente permite formatar o espaço entre as letras e palavras do texto.

```
h1{
    letter-spacing: 10px
}

p{
    word-spacing: 15px
}
```

T í t u l o

Lorem ipsum dolor sit amet consectetur adipisicing elit. Sapiente molestias aut nisi quasi nesciunt. Molestiae eos consequuntur atque! Asperiores nulla qui quo fugiat repellendus quae ipsum totam magni ab odit.

Formatação de lista no CSS

É possível utilizar HTML para estruturar conteúdos que representam listas, o exemplo básico de lista em HTML é:

```
<ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
    <li>Item 4</li>
</ul>
<ol>
    <li>Item A</li>
    <li>Item B</li>
    <li>Item C</li>
    <li>Item D</li>
</ol>
```

- Item 1
- Item 2
- Item 3
- Item 4

1. Item A
2. Item B
3. Item C
4. Item D

A linguagem CSS possibilita formatar estilos específicos de lista, por exemplo, a propriedade **list-style-type** permite formatar o tipo específico de bullets.

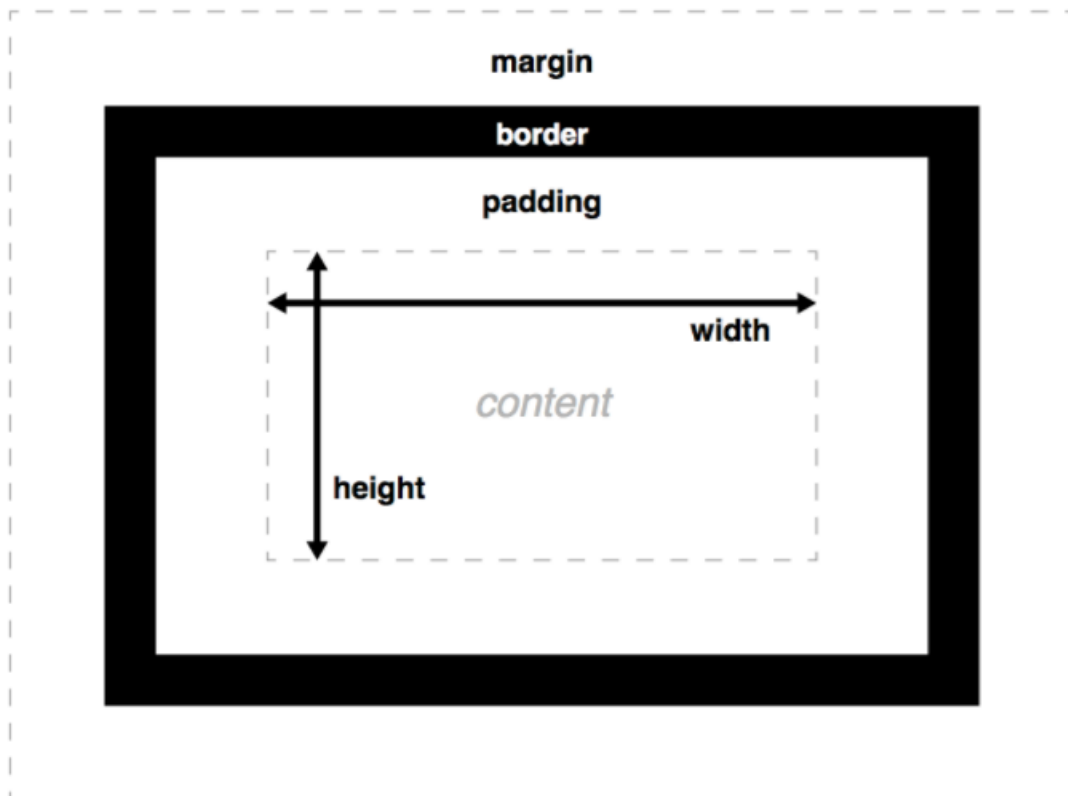
```
ul{  
  list-style-type: square  
}  
  
ol{  
  list-style-type: disc;  
}
```

- Item 1
- Item 2
- Item 3
- Item 4

- Item A
- Item B
- Item C
- Item D

Conhecendo o *Box Model*

Cada elemento dentro do documento é estruturado como uma caixa retangular (o famoso Box Model), com CSS é possível alterar algumas dessas características a fim de conseguir atingir a formatação desejada.



content: É o conteúdo do próprio elemento.

height: É a altura do próprio elemento.

width: É a largura do próprio elemento.

padding: É a largura de preenchimento do elemento.

border: É a borda do elemento, podemos definir tamanho, cores, o tipo da borda.

margin: Define a largura externa do elemento, ou seja, é como se fosse a distância entre cada box model dos elementos.

Observe abaixo um exemplo do box model do elemento **H1**

HTML:

```
<h1>Título</h1>
<p>Lorem ipsum dolor sit amet consectetur adipisicing elit.
Sapiente molestias aut nisi quasi nesciunt. Molestiae eos
consequuntur atque! Asperiores nulla qui quo fugiat repellendus quae
ipsum totam magni ab odit.
</p>
```

CSS:

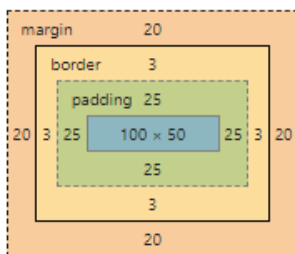
```
h1{
  height: 50px;
  width: 100px;
  padding: 25px;
```

```
border: 3px solid red;
margin: 20px
}
```

Resultado:



CSS no navegador



Abaixo algumas informações importantes que precisamos saber quando se está trabalhando com Box Model.

Overflow: Quando o tamanho do Box Model está fixo (largura e altura) o conteúdo pode não caber na caixa e neste caso ocorre o efeito de overflow, para esses casos o CSS oferece algumas opções para tratarmos esse tipo de comportamento. Na propriedade overflow existe uma variedade de valores que podemos configurar, no entanto os mais comuns são:

auto: Adiciona rolagem quando o conteúdo ultrapassa os limites do Box Model.

hidden: Oculta o conteúdo que ultrapassa os limites do Box Model.

visible: Não oculta o conteúdo que ultrapassa os limites do Box Model (Comportamento padrão dos navegadores).

HTML:

```
<p>Lorem ipsum dolor sit amet consectetur adipisicing elit.  
Laboriosam quos voluptas autem quisquam quia ea optio rem quasi inventore  
mollitia eos libero unde, sed ducimus numquam sunt quis officiis! Doloribus  
eius reiciendis in voluptate fugiat saepe voluptatem, enim,  
porro ducimus deserunt repellat repellendus quidem maxime, ad tempore  
perspiciatis eligendi cumque neque?  
Voluptatibus nulla neque corporis quae. Nobis, iure. Blanditiis, nisi quam?  
Vitae, aspernatur aperiam qui assumenda tempore cupiditate illo totam  
quibusdam beatae, corporis voluptas incidunt nisi quo tenetur quod maiores
```

iure. Accusantium repellat ex est id illo accusamus repudiandae omnis non sed quisquam ipsa soluta, perspiciatis voluptatum nostrum, autem excepturi?

CSS overflow auto:

```
p{  
  border: 1px dotted black;  
  height: 50px;  
  width: 300px;  
  overflow:auto  
}
```

accusamus repudiandae omnis non sed
quisquam ipsa soluta, perspiciatis
voluptatum nostrum, autem excepturi?

CSS overflow hidden

```
p{  
  border: 1px dotted black;  
  height: 50px;  
  width: 300px;  
  overflow:hidden  
}
```

Lorem ipsum dolor sit amet consectetur
adipiscing elit. Laboriosam quos voluptas
autem quisquam quia ea optio rem quasi

CSS overflow visible

```
p{  
  border: 1px dotted black;  
  height: 50px;  
  width: 300px;  
  overflow:visible  
}
```

Lorem ipsum dolor sit amet consectetur
adipiscing elit. Laboriosam quos voluptas
autem quisquam quia ea optio rem quasi
inventore mollitia eos libero unde, sed
ducimus numquam sunt quis officiis!
Doloribus eius reiciendis in voluptate fugiat
saepe voluptatem, enim, porro ducimus
deserunt repellat repellendus quidem maxime,
ad tempore perspiciatis eligendi cumque
neque? Voluptatibus nulla neque corporis
quae. Nobis, iure. Blanditiis, nisi quam? Vitae,
aspernatur aperiam qui assumenda tempore
cupiditate illo totam quibusdam beatae,
corporis voluptas incidunt nisi quo tenetur
quod maiores iure. Accusantium repellat ex
est id illo accusamus repudiandae omnis non
sed quisquam ipsa soluta, perspiciatis
voluptatum nostrum, autem excepturi?

Propriedades avançadas: É possível formatar o tamanho do Box Model utilizando as seguintes propriedades avançadas:

min-width, max-width, min-height, max-height.

min-width: Restringe a largura mínima do box model

max-width: Restringe a largura máxima do box model

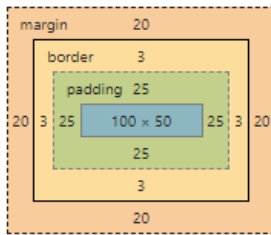
min-height: Restringe a altura máxima do box model

max-height: Restringe a altura máxima do box model

Entendendo o tamanho real do Box Model: A largura total do Box Model é a soma:

Largura total = **width + padding-right + padding-left + border-right + border-left**

```
h1{  
  height: 50px;  
  width: 100px;  
  padding: 25px;  
  border: 3px solid red;  
  margin: 20px  
}
```



Baseado no exemplo acima o calculo ficaria:

Largura total = $100 + 25 + 25 + 3 + 3$

Largura total = **156px**

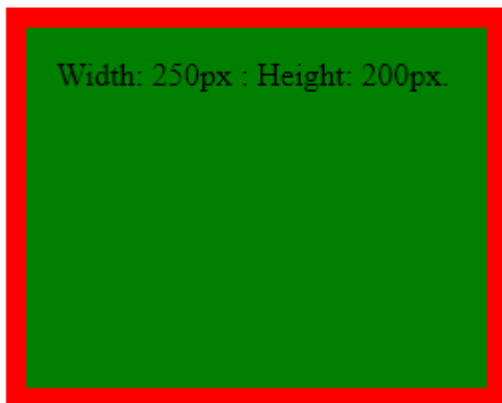
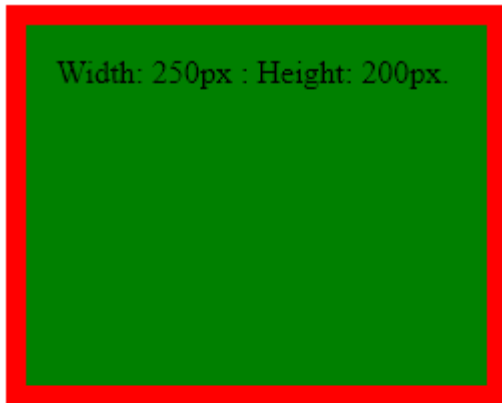
Por que saber o tamanho real do Box Model é importante? Pelo simples fato que se não considerarmos essa característica podemos obter resultados indesejados, uma vez que o tamanho real do conteúdo não é o **width** do conteúdo em si e sim a soma dos valores de **padding-right + padding-left + border-right + border-left**.

É possível considerar somente o tamanho do conteúdo declarado no CSS, ou seja, sem considerar o tamanho de padding e border? Sim, no CSS existe a propriedade **box-sizing** que permite configurar comportamentos diferentes para o Box Model.

Considere o seguinte HTML e CSS

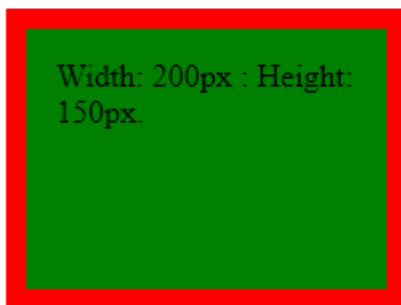
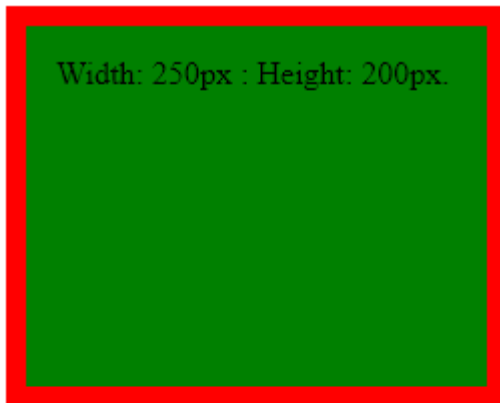
```
<div class="box-default"> Box Model default</div>
<div class="box-sizing"> Box Model Sizing</div>
```

```
div{
  width: 200px;
  height: 150px;
  padding: 15px;
  border: 10px solid red;
  margin: 30px;
  background-color: green
}
```

Aplicando a propriedade **box-sizing** ao elemento box-sizing.

```
.box-sizing{  
  box-sizing: border-box  
}
```



Observe que os valores de **padding** e **border** não foram adicionados ao conteúdo, ao invés disso, o conteúdo em si ficou menor, mas o **padding** e **border** ainda estão presentes, o que houve foi que eles ocuparam parte do conteúdo, porém, respeitando os 200px de **width**.

Box Model display

Todos os conceitos que descrevemos relacionados ao Box Model são aplicados em elementos cujo o comportamento padrão são chamados de nível de blocos (*block level elements*), porém no CSS existem outros elementos que são considerados Box Model mas possuem comportamentos diferentes.

A propriedade que define esse tipo de comportamento é o **display**. Existe uma variedade de valores disponível para essa propriedade, porém as mais comuns são:

block: Quando definido como **block** ocupa toda a linha.

inline: Quando definido como **inline** permite estar na mesma linha que outros elementos **inline**, porém as configurações de **width** e **height** não tem efeito.

inline-block: Quando definido como **inline-block** permite estar na mesma linha que outros elementos **inline**, porém ainda possui a integridade de elementos de bloco, ou seja não quebra o conteúdo e configurações de **width** e **height** tem efeito.

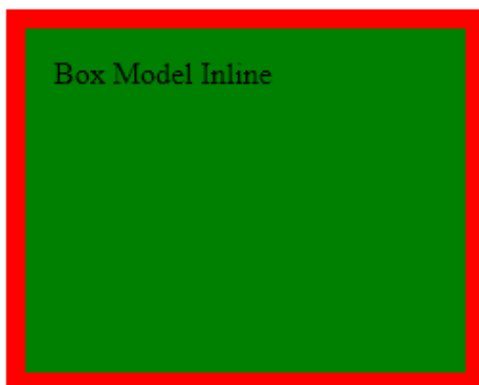
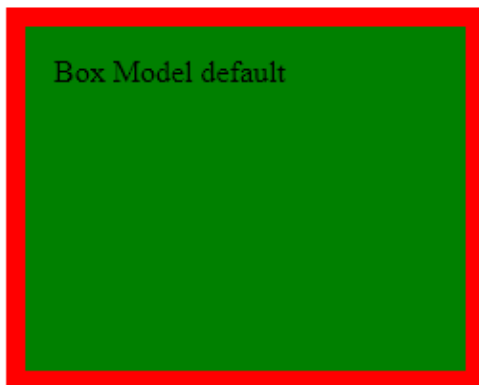
Existem outras possibilidades interessantes que podemos utilizar para configurar a propriedade display, você pode encontrar maiores detalhes [aqui](#).

Considere o seguinte HTML e CSS

```
<div class="box-default"> Box Model default</div>
```

```
<div class="box-inline"> Box Model Inline</div>
```

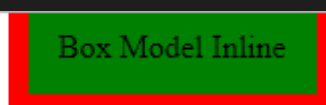
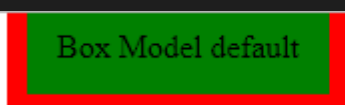
```
div{  
  width: 200px;  
  height: 150px;  
  padding: 15px;  
  border: 10px solid red;  
  margin: 30px;  
  background-color: green  
}
```



Observe que os dois elementos **div** cada um ocupou uma linha, ou seja um ficou abaixo do outro, esse comportamento se dá pelo fato do elemento **div** ser um elemento em nível de **block** por padrão.

CSS display **inline**

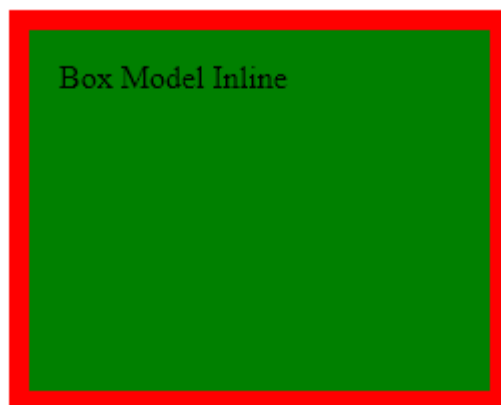
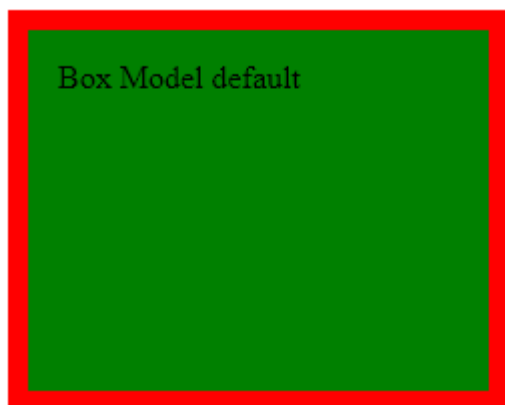
```
div{  
  display: inline  
}
```



Observe que os dois elementos **div** ocuparam a mesma linha e suas propriedades de **width** e **height** perderam o efeito.

CSS display **inline-block**

```
div{  
  display: inline-block  
}
```



Observe que os dois elementos **div** ocuparam a mesma linha, porém seus valores de **width** e **height** foram respeitados.

Layout com CSS

Uma página web segue certos padrões estruturais, ou seja, os elementos HTML geralmente são adicionados na página seguindo algum padrão, a este padrão damos o nome de layout. Existem várias possibilidades de layout web e o que diferencia um do outro é o objetivo do site/aplicação web, em outras palavras, para cada tipo de site/aplicação web nós podemos ter um tipo de layout diferente.

Por exemplo:

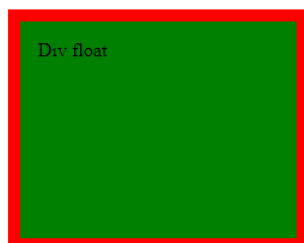


No CSS para implementar layouts é necessário conhecer propriedades específicas para **posicionar** elementos, ou seja, haverá situações em que será necessário alinhar elementos HTML de acordo com o layout desejado. A propriedade CSS cujo a finalidade é permitir o posicionamento dos elementos é o **float**.

Considerar o seguinte HTML e CSS

```
<div class="div-float"> Div float</div>
  <p>Lorem ipsum dolor sit amet consectetur adipisicing elit.
    Laboriosam quos voluptas autem quisquam quia ea optio rem quasi
    inventore mollitia eos libero unde, sed ducimus numquam sunt quis officiis!
    Doloribus eius reiciendis in voluptate fugiat saepe voluptatem, enim, porro
    ducimus deserunt repellat repellendus quidem maxime, ad tempore perspiciatis
    eligendi cumque neque?
    Voluptatibus nulla neque corporis quae. Nobis, iure. Blanditiis, nisi
    quam? Vitae, aspernatur aperiam qui assumenda tempore cupiditate illo totam
    quibusdam beatae, corporis voluptas incidunt nisi quo tenetur quod maiores
    iure. Accusantium repellat ex est id illo accusamus repudiandae omnis non sed
    quisquam ipsa soluta, perspiciatis voluptatum nostrum, autem excepturi?
  </p>
```

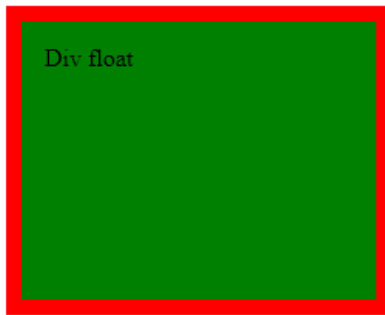
```
div{
  width: 200px;
  height: 150px;
  padding: 15px;
  border: 10px solid red;
  background-color: green
}
```



Lorem ipsum dolor sit amet consectetur adipisicing elit. Laboriosam quos voluptas autem quisquam qui
Doloribus eius reiciendis in voluptate fugiat saepe voluptatem, enim, porro ducimus deserunt repellat re
corporis quae. Nobis, iure. Blanditiis, nisi quam? Vitae, aspernatur aperiam qui assumenda tempore cupi
Accusantium repellat ex est id illo accusamus repudiandae omnis non sed quisquam ipsa soluta, perspici

Observe que o elemento **div** está acima do elemento **p** ok? Como vimos anteriormente **div** e **p** são considerados elementos bloco, ou seja, eles ocupam a linha toda e consequentemente irão ficar um abaixo do outro. Imagine que nós queremos que o parágrafo fique ao lado da div, o que devemos fazer? Simples, vamos declarar no CSS que a “div float” irá flutuar a esquerda do parágrafo.

```
.div-float{
  float:left
}
```



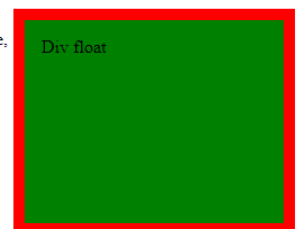
Lorem ipsum dolor sit amet consectetur adipisicing elit. Laboriosam quos voluptas autem quisquam quia ea optio rem quasi inventore mollitia eos libero unde, sed ducimus numquam sunt quis officiis! Doloribus eius reiciendis in voluptate fugiat saepe voluptatem, enim, porro ducimus deserunt repellat repellendus quidem maxime, ad tempore perspiciatis eligendi cumque neque? Voluptatibus nulla neque corporis quae. Nobis, iure. Blanditiis, nisi quam? Vitae, aspernatur aperiam qui assumenda tempore cupiditate illo totam quibusdam beatae, corporis voluptas incidunt nisi quo tenetur quod maiores iure. Accusantium repellat ex est id illo accusamus ipsa soluta, perspiciatis voluptatum nostrum, autem excepturi?

Basicamente o comportamento da propriedade **float** é este, permite flutuar os elementos na página.

E caso seja necessário que a div flutue a direita do parágrafo? Simples vamos declarar no CSS que desejamos que a div flutue a direita do parágrafo.

```
.div-float{  
    float:right  
}
```

r adipisicing elit. Laboriosam quos voluptas autem quisquam quia ea optio rem quasi inventore mollitia eos libero unde, sed ducimus numquam sunt quis officiis! Doloribus eius reiciendis in voluptate fugiat saepe voluptatem, enim, porro ducimus deserunt repellat repellendus quidem maxime, ad tempore perspiciatis eligendi cumque neque? Voluptatibus nulla neque corporis quae. Nobis, iure. Blanditiis, nisi quam? Vitae, aspernatur aperiam qui assumenda tempore cupiditate illo totam quibusdam beatae, corporis voluptas incidunt nisi quo tenetur quod maiores iure. Accusantium repellat ex est id illo accusamus ipsa soluta, perspiciatis voluptatum nostrum, autem excepturi?



A propriedade float aceita quatro valores:

right: Permitir flutuar o elemento para direita

left: Permitir flutuar o elemento para esquerda

inherit: Permitir flutuar o elemento baseado no valor do float do elemento pai

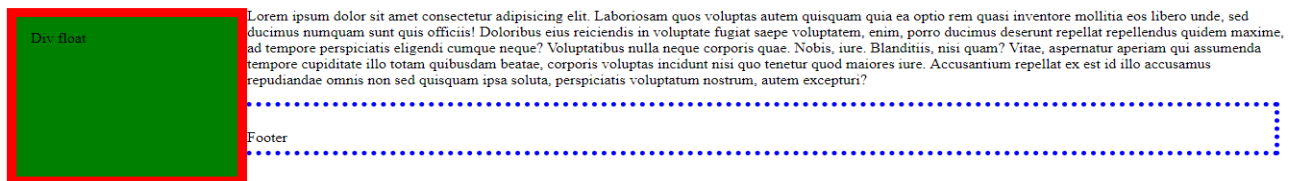
none: Não fluta o elemento (Esse é o comportamento default nos navegadores)

A propriedade **float** em algumas vezes pode inserir comportamentos de posicionamento indesejados, observe o seguinte HTML e CSS.

```
<div class="div-float"> Div float</div>  
<p>Lorem ipsum dolor sit amet consectetur adipisicing elit.  
    Laboriosam quos voluptas autem quisquam quia ea optio rem quasi  
    inventore mollitia eos libero unde, sed ducimus numquam sunt quis officiis!  
    Doloribus eius reiciendis in voluptate fugiat saepe voluptatem, enim, porro  
    ducimus deserunt repellat repellendus quidem maxime, ad tempore perspiciatis  
    eligendi cumque neque? Voluptatibus nulla neque corporis quae. Nobis, iure.  
    Blanditiis, nisi quam? Vitae, aspernatur aperiam qui assumenda tempore  
    cupiditate illo totam quibusdam beatae, corporis voluptas incidunt nisi quo  
    tenetur quod maiores iure. Accusantium repellat ex est id illo
```

```
accusamusrepudiandae omnis non sed quisquam ipsa soluta, perspiciatis voluptatum nostrum, autem excepturi?</p>
<footer><p>Footer</p></footer>
```

```
div{
  width: 200px;
  height: 150px;
  padding: 15px;
  border: 10px solid red;
  background-color: green
}
footer{
  height: 30px;
  border: 5px dotted blue;
  padding: 10px;
  margin: 10px
}
.div-float{
  float:left
}
```

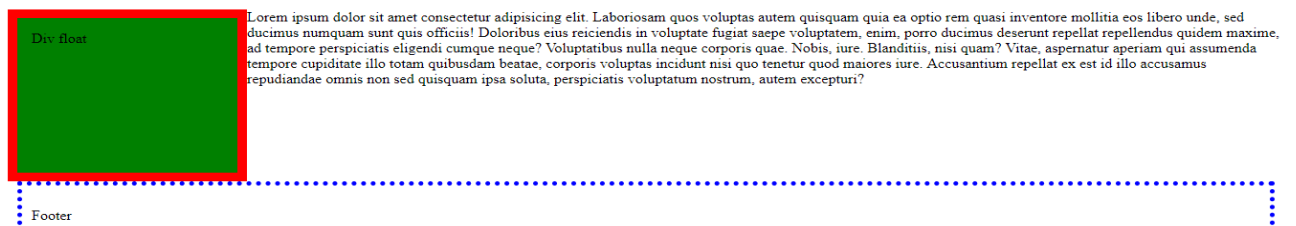


Observe que o “footer” deveria ficar abaixo de todo o conteúdo, como resolver isso?

No CSS temos a opção de limpar o efeito de flutuação nos elementos, o comportamento acima está correto pois se analisarmos a **div-float** está flutuando a esquerda do conteúdo conforme declaramos, ou seja, o que devemos fazer é limpar esse comportamento para determinado elemento.

Observe o seguinte CSS

```
footer{
  clear: left
}
```



Para resolver o problema utilizamos a propriedade **clear:left** no elemento **footer**, desta

forma, atingimos o comportamento esperado, ou seja, o **footer** ficou no final do conteúdo. A propriedade **clear** aceita quatro valores:

right: Permitir limpar o comportamento de flutuação a direita do elemento.

left: Permitir limpar o comportamento de flutuação a esquerda do elemento.

both: Permitir limpar o comportamento de flutuação para ambos os entidos (esquerda e direita).

none: Não fluta o elemento (Esse é o comportamento default nos navegadores).

Importante:

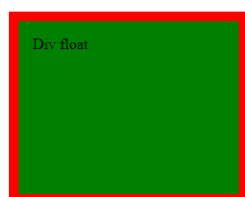
Limpar o comportamento de flutuação é necessário pois evitamos efeitos indesejados no layout, existem várias técnicas para limpar comportamentos de flutuação indesejados, uma das mais famosas é o **clearfix**, observe o seguinte HTML e CSS:

```
<div class="div-float"> Div float</div>

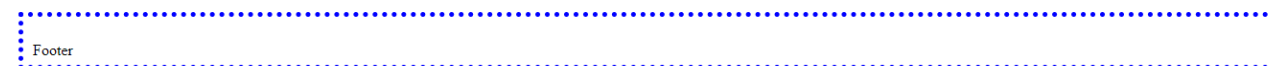
<p class="clearfix">Lorem ipsum dolor sit amet consectetur adipisicing elit.
Laboriosam quos voluptas autem quisquam quia ea optio rem quasi inventore
mollitia eos libero unde, sed ducimus numquam sunt quis officiis! Doloribus
eius reiciendis in voluptate fugiat saepe voluptatem, enim, porro ducimus
deserunt repellat repellendus quidem maxime, ad tempore perspiciatis eligendi
cumque neque? Voluptatibus nulla neque corporis quae. Nobis, iure.
Blanditiis, nisi quam? Vitae, aspernatur aperiam qui assumenda tempore
cupiditate illo totam quibusdam beatae, corporis voluptas incidunt nisi quo
tenetur quod maiores iure. Accusantium repellat ex est id illo accusamus
repudiandae omnis non sed quisquam ipsa soluta, perspiciatis voluptatum
nostrum, autem excepturi?

</p>
<footer><p>Footer</p></footer>
```

```
.clearfix:after {
  content: ".";
  visibility: hidden;
  display: block;
  height: 0;
  clear: both;
}
```



Lorem ipsum dolor sit amet consectetur adipisicing elit. Laboriosam quos voluptas autem quisquam quia ea optio rem quasi inventore mollitia eos libero unde, sed ducimus numquam sunt quis officiis! Doloribus eius reiciendis in voluptate fugiat saepe voluptatem, enim, porro ducimus deserunt repellat repellendus quidem maxime, ad tempore perspiciatis eligendi cumque neque? Voluptatibus nulla neque corporis quae. Nobis, iure. Blanditiis, nisi quam? Vitae, aspernatur aperiam qui assumenda tempore cupiditate illo totam quibusdam beatae, corporis voluptas incidunt nisi quo tenetur quod maiores iure. Accusantium repellat ex est id illo accusamus repudiandae omnis non sed quisquam ipsa soluta, perspiciatis voluptatum nostrum, autem excepturi?



Basicamente a declaração CSS acima funciona da seguinte maneira:

- Aplica-se em todo elemento que tenha a classe **clearfix**
- Cria-se um conteúdo depois do elemento que contém a classe **clearfix**, esse conteúdo é um caracter “.” invisível e com display block para ocupar a linha toda.
- Aplica-se o **clear:both** para limpar a flutuação em ambos os sentidos (esquerda e direita).

Essa abordagem é útil pois você pode aplicar essa classe em todos os elementos que você achar necessário.

Conhecendo a propriedade “position” em CSS

Outra propriedade que auxilia na implementação de layout em CSS é a propriedade **position**, basicamente essa propriedade define em qual contexto o elemento deve ser posicionado, esse contexto pode ser em relação a outro elemento ou em relação a página.

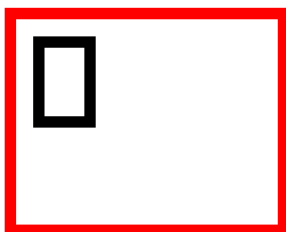
A propriedade **position** aceita quatro valores:

static: É o posicionamento padrão, posiciona o elemento no fluxo normal dos elementos na página.

Considere o seguinte HTML e CSS com **position:static**

```
<div class="div-parent">
  <div class="div-position"></div>
</div>
```

```
.div-parent{
  width: 200px;
  height: 150px;
  padding: 15px;
  border: 10px solid red;
}
.div-position{
  border: 10px solid black;
  height: 50px;
  width: 25px;
  padding: 5px;
  position:static;
}
```



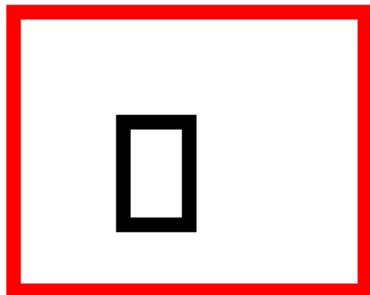
relative: Tem comportamento semelhante ao **static** a diferença é que permite alterar a posição final do elemento, para isso é preciso usar as propriedades **top**, **bottom**, **left**, and **right**.

Considere o seguinte HTML

```
<div class="div-parent">
  <div class="div-position"></div>
</div>
```

Posicionando elemento utilizando **position:relative top:50px left:50px**

```
.div-parent{
  width: 200px;
  height: 150px;
  padding: 15px;
  border: 10px solid red;
}
.div-position{
  border: 10px solid black;
  height: 50px;
  width: 25px;
  padding: 5px;
  position:relative;
  top:50px;
  left: 50px;
}
```



absolute: O elemento fica separado dos demais ou seja, é como se o elemento não existisse no fluxo normal de renderização dos elementos. No posicionamento absolute a referência é a página e também é possível especificar a distância dos lados do elemento utilizando as propriedades **top**, **bottom**, **left** e **right** em relação a página.

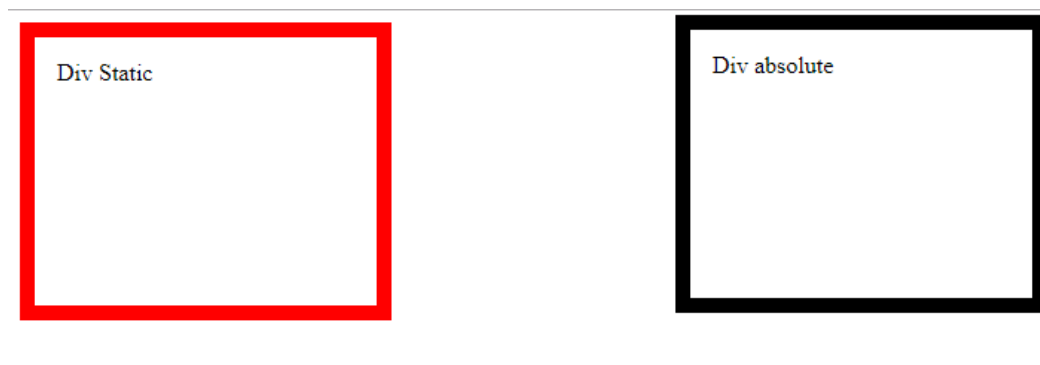
Considere o seguinte HTML

```
<div class="div-static">Div Static</div>
<div class="div-absolute">Div absolute</div>
```

Posicionando elemento utilizando **position:absolute top:3px right:0px**

```
.div-static{
  width: 200px;
  height: 150px;
  padding: 15px;
  border: 10px solid red
}

.div-absolute{
  width: 200px;
  height: 150px;
  padding: 15px;
  border: 10px solid black;
  position: absolute;
  top: 3px;
  right: 0px
}
```



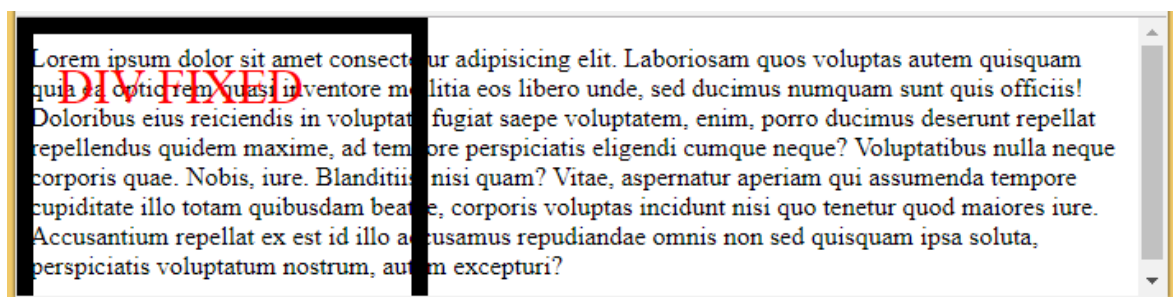
fixed: Possui comportamento semelhante ao **absolute** a diferença é que neste o elemento é fixado em sua posição e sua referência é a tag **html** ou seu ancestral mais próximo.

Considere o seguinte HTML

```
<p>Lorem ipsum dolor sit amet consectetur adipisicing elit.
Laboriosam quos voluptas autem quisquam quia ea optio rem quasi inventore
mollitia eos libero unde, sed ducimus numquam sunt quis officiis! Doloribus
eius reiciendis in voluptate fugiat saepe voluptatem, enim,
porro ducimus deserunt repellat repellendus quidem maxime, ad tempore
perspiciatis eligendi cumque neque? Voluptatibus nulla neque corporis quae.
Nobis, iure. Blanditiis, nisi quam? Vitae, aspernatur aperiam qui assumenda
tempore cupiditate illo totam quibusdam beatae, corporis voluptas incidunt
nisi quo tenetur quod maiores iure. Accusantium repellat ex est id illo
accusamus repudiandae omnis non sed quisquam ipsa soluta, perspiciatis
voluptatum nostrum, autem excepturi?
</p>
<div class="div-fixed">Div fixed</div>
```

Posicionando elemento utilizando **position: fixed top:0px left:0px**

```
.div-fixed{
  width: 200px;
  height: 150px;
  padding: 15px;
  border: 10px solid black;
  color: red;
  position: fixed;
  text-transform: uppercase;
  font-size: 30px;
  top: 0px;
  left: 0px
}
```



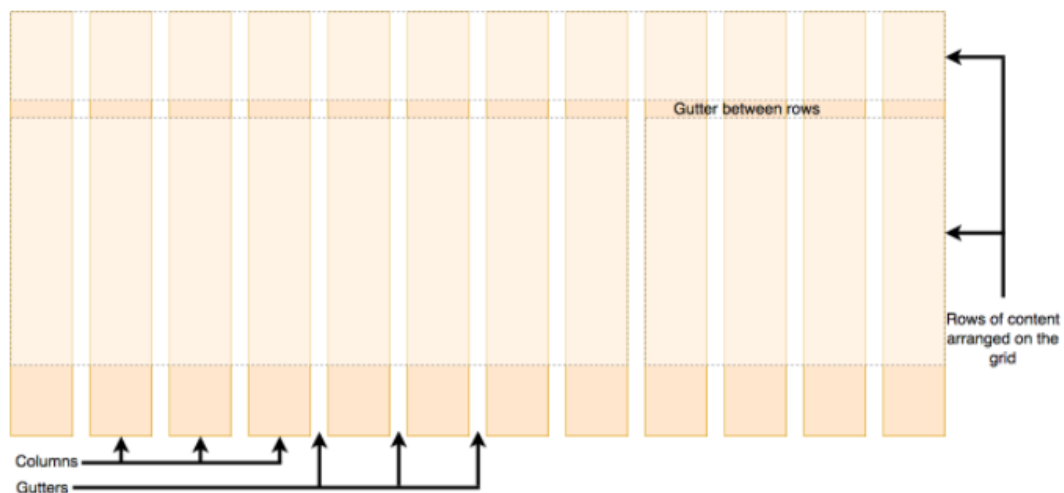
A pesar da rolagem na página o elemento div-fixed ficará posicionado no mesmo lugar da página pois ele está fixado.

Não iremos abordar as propriedades CSS de maneira isolada, ou seja, iremos aplicar as propriedades em um projeto prático e sob demanda serão apresentadas as propriedades de formatação de texto, fontes, cores, bordas, efeitos, etc. Você pode conferir a lista de todas as propriedades CSS [aqui](#).

Conhecendo Grid Layout

Com a base adquirida sobre posicionamentos em CSS iremos conhecer o sistema de Grid para páginas web.

Grid é um conjunto de linhas horizontais e verticais que auxiliam no alinhamento dos elementos na página tornando o layout web mais consistente.



Existem alguns frameworks e bibliotecas que já fazem esse trabalho, porém, nosso objetivo é aplicar o conceito de grid utilizando puramente CSS. Nossa primeira grid em CSS irá ter as seguintes características:

- 12 Colunas e será fixa na largura de 980px;
- Centralizada na página;
- Grid poderá se adaptar para 2, 3, 4 e 6 colunas.

Considere o seguinte HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Grid</title>
    <style>
      * {
        box-sizing: border-box;
      }

      body {
        width: 980px;
        margin: 0 auto;
      }

      .container {
        padding-right: 20px;
      }
      .row {
        clear: both;
      }
      .col {
        float: left;
        margin-left: 20px;
        width: 60px;
      }
    </style>
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col"></div>
        <div class="col"></div>
        <div class="col"></div>
        <div class="col"></div>
        <div class="col"></div>
        <div class="col"></div>
        <div class="col"></div>
        <div class="col"></div>
        <div class="col"></div>
        <div class="col"></div>
        <div class="col"></div>
        <div class="col"></div>
      </div>
    </div>
  </body>
</html>
```

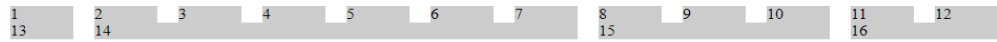
```

        background: #cccccc;
    }
    /* Duas colunas de largura de 120px e uma adicional com largura de
20px */
    .col.span2 { width: 140px; }
    /* Tres colunas de largura de 180px e duas adicionais com largura de
40px */
    .col.span3 { width: 220px; }
    /* Assim por diante */
    .col.span4 { width: 300px; }
    .col.span5 { width: 380px; }
    .col.span6 { width: 460px; }
    .col.span7 { width: 540px; }
    .col.span8 { width: 620px; }
    .col.span9 { width: 700px; }
    .col.span10 { width: 780px; }
    .col.span11 { width: 860px; }
    .col.span12 { width: 940px; }
</style>
</head>

<body>
    <div class="container">
        <div class="row">
            <div class="col">1</div>
            <div class="col">2</div>
            <div class="col">3</div>
            <div class="col">4</div>
            <div class="col">5</div>
            <div class="col">6</div>
            <div class="col">7</div>
            <div class="col">8</div>
            <div class="col">9</div>
            <div class="col">10</div>
            <div class="col">11</div>
            <div class="col">12</div>
        </div>
        <div class="row">
            <div class="col span1">13</div>
            <div class="col span6">14</div>
            <div class="col span3">15</div>
            <div class="col span2">16</div>
        </div>
    </div>
</body>
</html>

```

Basicamente a disposição dos elementos na grid baseado no HTML e CSS acima ficaria como:



Imagine que você precise que um elemento ocupe metade da largura do grid, ou seja que ocupe 6 colunas.

Adicione o seguinte HTML

```
<div class="col span6">6</div>
```

Observe que a div 06 ocupou exatamente a metade da largura do layout.



6.2. CSS3 Grid e Flexbox

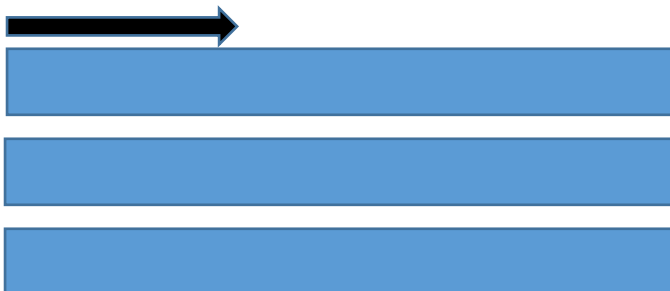
Na seção anterior foi apresentado uma alternativa para construção de layout baseado em grid utilizando conceitos conhecidos do CSS, essa abordagem foi amplamente utilizada por um tempo, porém com a chegada do [flexbox](#) facilitou a maneira de posicionar elementos e posteriormente os navegadores começaram a oferecer suporte a [grid](#) que facilitou ainda mais o trabalho de posicionar elementos com CSS. Todo iniciante em CSS ou até mesmo pessoas com mais experiência ainda fazem a seguinte pergunta: **Flexbox ou Grid?**

A resposta para essa pergunta é: Um não substitui o outro e podem ser utilizados juntos. O flexbox faz algumas coisas que o grid não faz e o grid faz algumas coisas que o flexbox não faz.

Não está em nosso escopo abordar detalhes técnicos profundos de cada um, o objetivo aqui é aplicá-los de maneira correta e objetiva.

Quando usar Flexbox?

Flexbox se comporta melhor quando se lida com uma dimensão, seja em linha ou em coluna. Pense em um menu fluído na vertical ou horizontal, para esses casos o flexbox é a melhor escolha.



Conhecendo o Flexbox

No HTML

```
<div class="flex-container">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
</div>
```

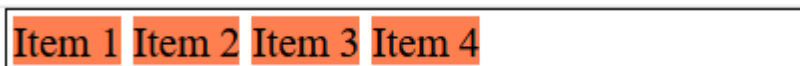
No CSS

```
.flex-container{
  border: 1px solid #000;
  max-width: 400px;
  margin: 0 auto;

  display: flex
}

.item{
  margin: 3px;
  background: coral;
  text-align: center;
  font-size: 1.3em;
}
```

Resultado



É importante destacar que:

- Deve ser definido o elemento container **display: flex**
- Automaticamente os seus filhos diretos, em nosso exemplo as tags **div** serão automaticamente transformadas em **flex itens**.

Observe o seguinte exemplo:

No HTML

```
<div class="flex-container">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
  <div class="item">Item 6</div>
```



```
<div class="item">Item 7</div>
<div class="item">Item 8</div>
<div class="item">Item 9</div>
<div class="item">Item 10</div>
</div>
```

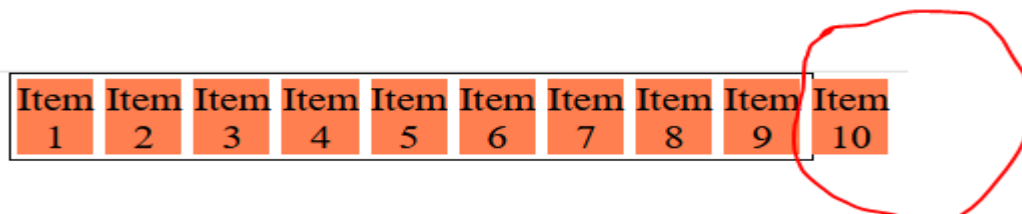
No CSS

```
.flex-container{
  border: 1px solid #000;
  max-width: 400px;
  margin: 0 auto;

  display: flex
}

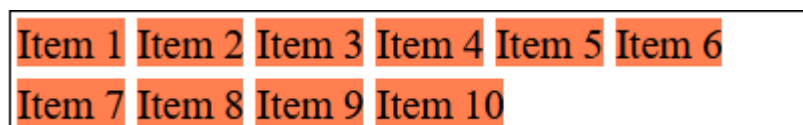
.item{
  margin: 3px;
  background: coral;
  text-align: center;
  font-size: 1.3em;
}
```

Resultado



Observe que o elemento ultrapassou o limite do container, como estamos utilizando flexbox esse tipo de situação deve ser resolvido de maneira mais limpa possível, ou seja usando recursos de flexbox. Indo direto ao ponto, com flex box é possível definir que os flex itens pode ou não quebrar linha, isso pode feito configurando a propriedade **flex-wrap** no elemento container assim:

```
flex-wrap: wrap;
```



Alterando a direção do Flexbox

Se você observar nos exemplos anteriores a direção dos itens está em linhas, ou seja, um ao lado do outro. Esse comportamento pode ser modificado utilizando a propriedade **flex-direction**, por exemplo:

No HTML

```
<div class="flex-container column">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
</div>
```

No CSS

```
.flex-container{
  border: 1px solid #000;
  max-width: 400px;
  margin: 0 auto;

  display: flex;
  flex-wrap: wrap;
}

.item{
  margin: 3px;
  background: coral;
  text-align: center;
  font-size: 1.3em;
}

.column{
  flex-direction: column;
}
```

Resultado

Item 1
Item 2
Item 3
Item 4
Item 5

Essa propriedade é bastante útil para visualizar a página no mobile pois faz todo sentido exibir o conteúdo em uma única coluna.

Alinhando itens dentro do container

Outra propriedade importante é o **justify-content** que é utilizada para alinhar os itens dentro de um flex container. Observe o seguinte exemplo:

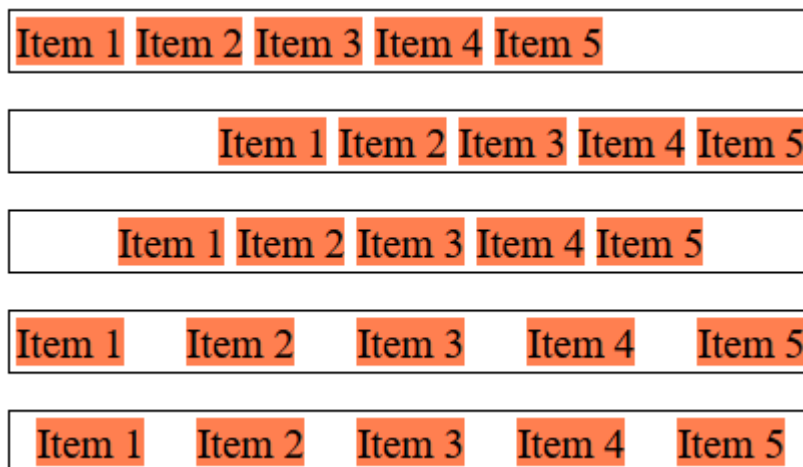
No HTML

```
<div class="flex-container align-start">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
</div>
<br>
<div class="flex-container align-end">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
</div>
<br>
<div class="flex-container align-center">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
</div>
<br>
<div class="flex-container space-between">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
</div>
<br>
<div class="flex-container space-around">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
</div>
```

No CSS

```
.align-start{
  justify-content: flex-start;
}
.align-end{
  justify-content: flex-end;
}
.align-center{
  justify-content: center
}
.space-between{
  justify-content: space-between
}
.space-around{
  justify-content: space-around;
}
```

Resultado



***Importante:** Caso os itens ocupem todo o container essa propriedade não funciona.

Alinhando itens de acordo com o container

O **align-items**, permite alinhar os itens baseado no eixo do container, por exemplo:

No HTML

```
<div class="flex-container align-items align-items-stretch">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
</div>
<br>
<div class="flex-container align-items align-items-start">
```

```

    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
  </div>
  <br>
  <div class="flex-container align-items align-items-end">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
  </div>
  <br>
  <div class="flex-container align-items align-items-center">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
  </div>
  <br>
  <div class="flex-container align-items align-items-baseline">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
  </div>

```

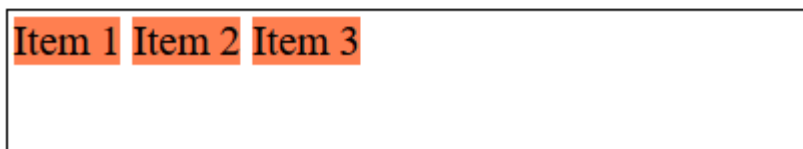
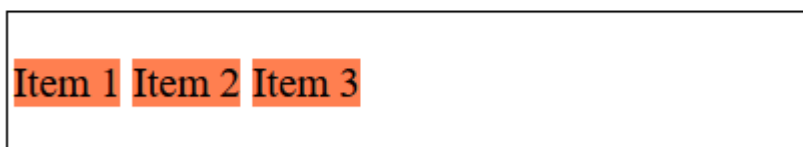
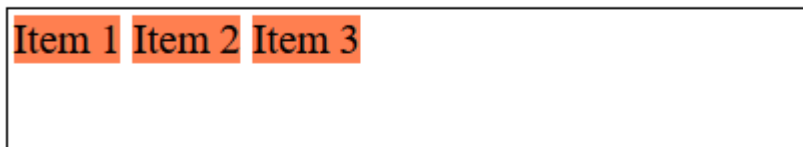
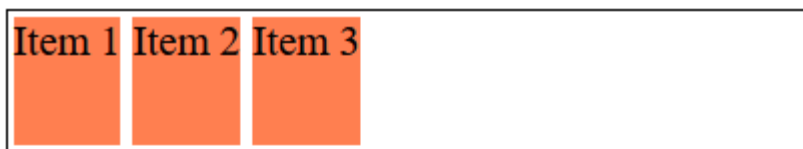
No CSS

```

.align-items{
  height: 70px
}
.align-items-stretch{
  align-items: stretch;
}
.align-items-start{
  align-items: flex-start;
}
.align-items-end{
  align-items: flex-end;
}
.align-items-center{
  align-items: center
}
.align-items-baseline{
  align-items: baseline;
}

```

Resultado



align-items: stretch - Default, permite que os flex itens cresçam igualmente.

align-items: flex-start - Alinha os flex itens ao início.

align-items: flex-end - Alinha os flex itens ao final.

align-items: center - Alinha os flex itens ao centro.

align-items: baseline - Alinha os flex itens de acordo com a linha base da tipografia.

Alinhando itens em relação ao eixo vertical

O **align-content**, permite o alinhamento das linhas do container em relação ao eixo vertical. Essa propriedade possui algumas particularidades:

- Apenas funciona se existir mais de uma linha
- O container deve ser maior que a soma das linhas dos flex itens, ou seja, deve ser definido uma altura, caso contrário não terá efeito.

Observe o seguinte exemplo:

No HTML

```
<div class="flex-container align-content align-content-stretch">  
  <div class="item">Item 1</div>
```

```

    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
  </div>
  <div class="flex-container align-content align-content-start">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
  </div>
  <div class="flex-container align-content align-content-end">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
  </div>
  <div class="flex-container align-content align-content-center">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
  </div>
  <div class="flex-container align-content align-content-between">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
  </div>
  <div class="flex-container align-content align-content-around">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
  </div>

```

No CSS

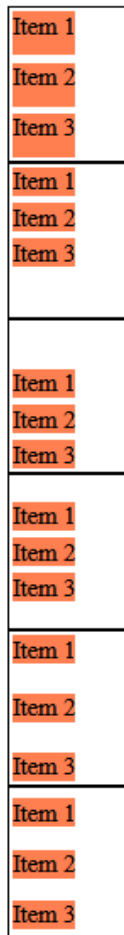
```

.align-content{
  max-width: 100px;
  height: 130px;
  flex-wrap: wrap;
}
.align-content-stretch{
  align-content: stretch;
}
.align-content-start{
  align-content: flex-start;
}
.align-content-end{
  align-content: flex-end;
}
.align-content-center{
  align-content: center;
}

```

```
.align-content-between{
  align-content: space-between;
}
.align-content-around{
  align-content: space-around
}
```

Resultado



Controlando o tamanho dos flex itens

É possível configurar o tamanho que os flex itens devem ocupar, basicamente é preciso definir as seguintes propriedades:

flex-grow: Define o quanto um flex item deve crescer. Por padrão o valor é **0**, isso significa que irá ocupar o tamanho máximo baseado no conteúdo interno ou irá respeitar a propriedade **width** caso seja definida. Um detalhe é que o **justify-content** não funciona se o **flex-grow** estiver definido.

flex-shrink: Define a redução de tamanho do flex item. Por padrão o valor é **1**, isso significa que os flex itens terão os tamanhos reduzidos para caber no container.

- Caso seja definido o valor **0** não será permitido a diminuição do flex item.

- Caso seja definido por exemplo o valor **2**, significa que irá diminuir duas vezes mais que o flex item definido com 1.

flex-basis: Define o tamanho inicial do flex item. Por padrão o valor é **auto**, isso significa que se não há tamanho especificado o tamanho será de acordo com o conteúdo.

A forma mais comum de usar as propriedades flex-grow, flex-shrink e flex-basis é utilizar a propriedade **flex** que é um atalho para essas propriedades. A recomendação é utilizar a propriedade **flex** ao invés de utilizar as propriedades flex-grow, flex-shrink e flex-basis individualmente.

Caso não defina nenhum valor o padrão da propriedade **flex** é:

flex: 0 1 auto //flex-grow: 0 flex-shrink: 1 flex-basis: auto

Vejamos alguns exemplos utilizando flex-grow:

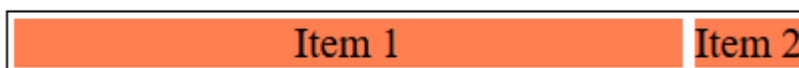
No HTML

```
<div class="flex-container flex-items-size">
  <div class="item grow-1">Item 1</div>
  <div class="item grow-default">Item 2</div>
</div>
```

No CSS

```
.grow-1{
  flex: 1;
}
```

Resultado



No HTML

```
<div class="flex-container flex-items-size">
  <div class="item grow-1">Item 1</div>
  <div class="item grow-2">Item 2</div>
  <div class="item grow-default">Item 3</div>
</div>
```

No CSS

```
.grow-1{
  flex: 1;
}
.grow-2{
```

```
flex: 2;  
}
```

Resultado



Vejamos um exemplo combinando as propriedades:

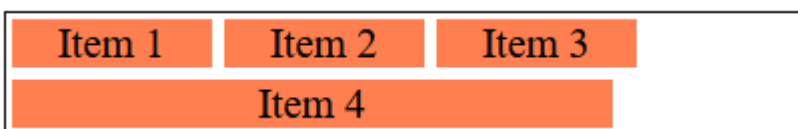
No HTML

```
<div class="flex-container flex-items-size">  
  <div class="item with-basis-100">Item 1</div>  
  <div class="item with-basis-100">Item 2</div>  
  <div class="item with-basis-100">Item 3</div>  
  <div class="item with-basis-300">Item 4</div>  
</div>
```

No CSS

```
.with-basis-100{  
  flex: 0 1 100px;  
}  
.with-basis-300{  
  flex: 0 1 300px;  
}
```

Resultado



Alterando a ordem dos flex-item

A propriedade **order** permite alterar a ordem dos flex itens. Por padrão o valor é **0**, significa que a ordem dos itens será a ordem definida no HTML.

Vejamos um exemplo:

Em HTML

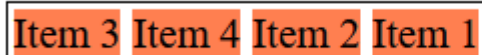
```
<div class="flex-container order-item">  
  <div class="item order-4">Item 1</div>  
  <div class="item order-3">Item 2</div>  
  <div class="item">Item 3</div>
```

```
<div class="item">Item 4</div>
</div>
```

Em CSS

```
.order-4{
  order: 4;
}
.order-3{
  order: 3;
}
```

Resultado



Definindo alinhamento específico de um flex item

A propriedade **align-self** permite configurar um item especificamente. O valor padrão é **auto**, significa que será respeitado o alinhamento definido no flex container.

Vejamos um exemplo:

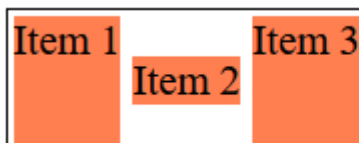
No HTML

```
<div class="flex-container align-items">
  <div class="item">Item 1</div>
  <div class="item align-self-center">Item 2</div>
  <div class="item">Item 3</div>
</div>
```

No CSS

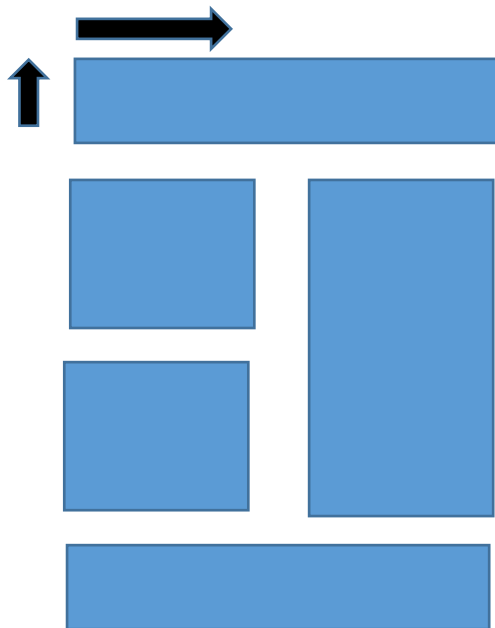
```
.align-self-center{
  align-self: center;
}
```

Resultado



Quando usar Grid?

Grid se comporta melhor quando utilizado em duas dimensões e quando há necessidade de definir estruturas.



Conhecendo o Grid

No HTML

```
<div class="grid-container">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
</div>
```

No CSS

```
.grid-container{
  display: grid;
}
```

Resultado

	Item 1
	Item 2
	Item 3
	Item 4
	Item 5

É importante destacar que:

- Deve ser definido o elemento container
display: grid

Grids são compostos por linhas e colunas, o mecanismo de grid em CSS permite configurar tanto o número de colunas quanto número de linhas. Essas configurações podem ser feitas utilizando as propriedades: **grid-template-columns**, **grid-template-rows** e **grid-template-areas**. Ao invés de utilizar essas propriedades separadamente podemos utilizar a propriedade **grid-template** que é um atalho para essas propriedades.

O formato da propriedade é a seguinte:

grid-template: grid-template-rows / grid-template-columns

Por exemplo, deseja-se criar um grid com duas linhas e três colunas:

No HTML

```
<div class="grid-container grid-exemplo-2r-3c">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
  <div class="item">Item 6</div>
</div>
```

No CSS

```
.grid-exemplo-2r-3c{
  grid-template: 100px 100px / 200px 200px 200px;
}
```

Resultado é: A primeira e segunda linha com 100px. A primeira, segunda e terceira coluna com 200px.

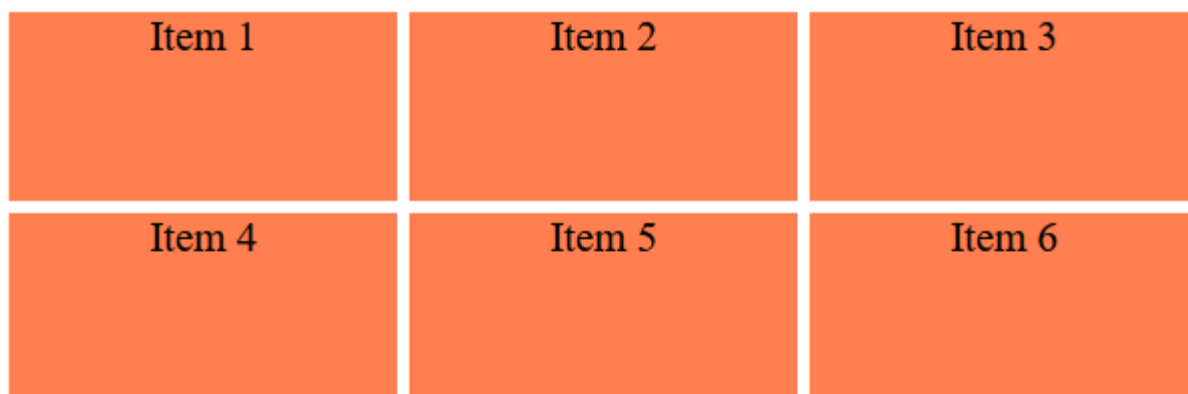
Item 1	Item 2	Item 3
Item 4	Item 5	Item 6

Em nosso exemplo foi definido em pixel os valores de linha e coluna e também definimos explicitamente a quantidade de linhas e colunas. É possível utilizar outras notações para definir linhas e colunas no CSS grid.

Utilizando a função **repeat()**

```
.grid-exemplo-repeat{
  grid-template: repeat(2, 100px) / repeat(3, 200px);
}
```

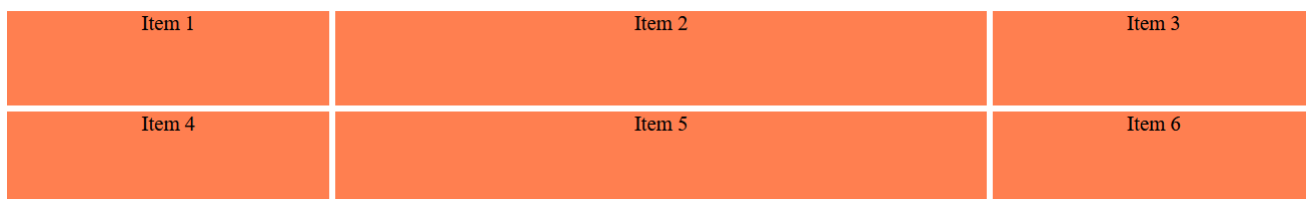
Resultado



Utilizando unidades **fracionais**

```
.grid-exemplo-fracional{
  grid-template: repeat(2, 100px) / 1fr 2fr 1fr;
}
```

Resultado



Definindo espaçamentos

A propriedade **gap** permite definir o espaçamento entre linha e coluna. É possível definir o espaçamento da coluna utilizando **column-gap** ou definir o espaçamento da linha utilizando **row-gap**.

Vejamos o seguinte exemplo:

No HTML

```
<div class="grid-container gap">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
```

```
<div class="item">Item 3</div>
<div class="item">Item 4</div>
</div>
```

No CSS

```
.gap{
  grid-template: repeat(2, 1fr) / repeat(2, 1fr);
  gap: 10px;
}
```

Resultado

Item 1	Item 2
Item 3	Item 4

Definindo o tamanho de colunas que são geradas automaticamente

A propriedade **grid-auto-columns** permite configurar o tamanho de colunas que são geradas automaticamente. Por exemplo:

No HTML

```
<div class="grid-container auto-columns-size">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
</div>
```

No CSS estamos definindo duas colunas e as colunas geradas automaticamente terão 100px de largura.

```
.auto-columns-size{
  grid-template: "a a";
  grid-auto-columns: 100px;
}
```

Resultado

1	2
3	

Definindo o tamanho de linhas que são geradas automaticamente

A propriedade **grid-auto-rows** permite configurar o tamanho de linhas que são geradas automaticamente. Por exemplo:

No HTML

```
<div class="grid-container auto-rows-size">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
</div>
```

No CSS estamos definindo duas colunas e a linha gerada automaticamente terá 50px de altura.

```
.auto-rows-size{
  grid-template:
    "a a";
  grid-auto-rows: 50px;
}
```

Resultado



Definindo o fluxo dos itens

A propriedade **grid-auto-flow** permite definir o fluxo dos itens. É possível configurar as seguintes propriedades:

grid-auto-flow: row – Gera novas linhas automaticamente

grid-auto-flow: column – Gera novas colunas automaticamente

grid-auto-flow: dense – Se possível posiciona o máximo de elementos nas primeiras posições do grid, essa propriedade pode bagunçar o conteúdo.

Com **grid-auto-flow: row**

No HTML

```
<div class="grid-container grid-auto-flow-row">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
  <div class="item">Item 6</div>
</div>
```


No CSS

```
.grid-auto-flow-row{
  grid-template: repeat(3, 1fr) / repeat(3, 1fr);
  grid-auto-flow: row;
}
```

Resultado

Item 1	Item 2	Item 3
Item 4	Item 5	Item 6

Com **grid-auto-flow: column**

No HTML

```
<div class="grid-container grid-auto-flow-column">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
  <div class="item">Item 6</div>
</div>
```

No CSS

```
.grid-auto-flow-column{
  grid-template: repeat(3, 1fr) / repeat(3, 1fr);
  grid-auto-flow: column;
}
```

Resultado

Item 1	Item 4
Item 2	Item 5
Item 3	Item 6

Com **grid-auto-flow: dense**

No HTML

```
<div class="grid-container grid-auto-flow-dense">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item span3">Item 5</div>
  <div class="item">Item 6</div>
```

```

<div class="item">Item 7</div>
<div class="item">Item 8</div>
<div class="item">Item 9</div>
<div class="item">Item 10</div>
<div class="item">Item 11</div>
</div>

```

No CSS

```

.grid-auto-flow-dense{
  grid-template: repeat(3, 1fr) / repeat(3, 1fr);
  grid-auto-flow: dense;
}

```

Resultado

Item 1	Item 2	Item 3
Item 4	Item 6	Item 7
Item 5		
Item 8	Item 9	Item 10
Item 11		

Conhecendo a propriedade **grid**

Essa propriedade é um atalho para as propriedades: grid-template-rows, grid-template-columns, grid-template-areas, grid-auto-rows, grid-auto-columns e grid-auto-flow.

Exemplo: Considerando que seja necessário definir um grid com 3 colunas e 4 linhas onde a primeira e quarta linha devm ter 100px de altura e o auto flow deve ser column.

Em HTML

```

<div class="grid-container grid-shortcut">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
  <div class="item">Item 6</div>
  <div class="item">Item 7</div>
  <div class="item">Item 8</div>
  <div class="item">Item 9</div>
  <div class="item">Item 10</div>
  <div class="item">Item 11</div>
  <div class="item">Item 12</div>
</div>

```

Em CSS

```
.grid-shortcut{  
  grid: 100px 1fr 1fr 100px / auto-flow repeat(3, 1fr)  
}
```

Resultado

Item 1	Item 5	Item 9
Item 2	Item 6	Item 10
Item 3	Item 7	Item 11
Item 4	Item 8	Item 12

Alinhando grid itens em relação ao eixo X

A propriedade **justify-content** permite alinhar os itens do grid em relação ao eixo X. Vejamos um exemplo de como centralizar itens do grid no eixo X.

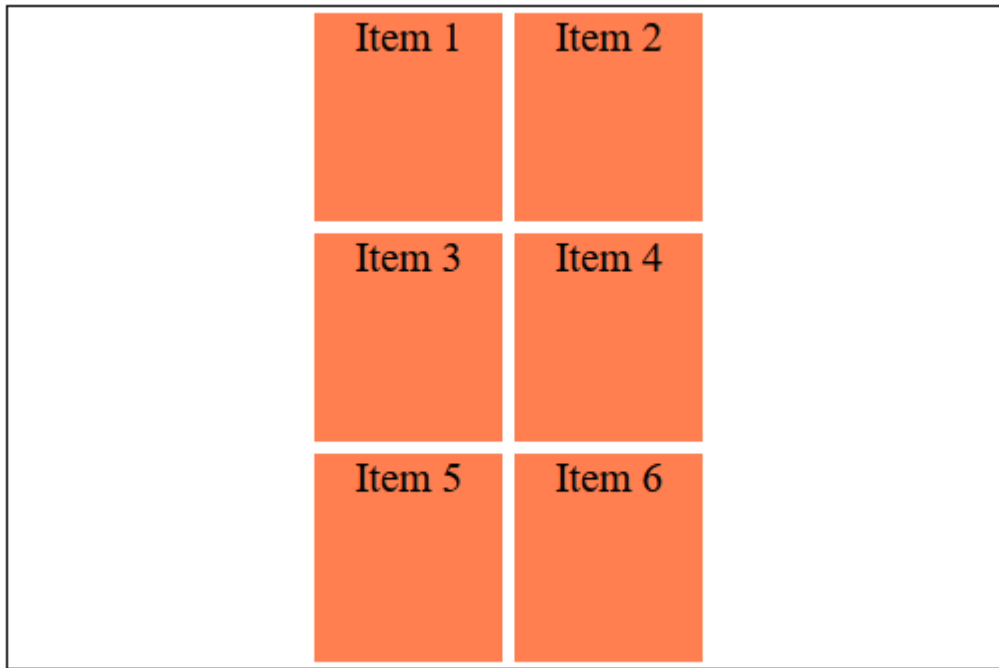
No HTML

```
<div class="grid-container grid-justify-content-center">  
  <div class="item">Item 1</div>  
  <div class="item">Item 2</div>  
  <div class="item">Item 3</div>  
  <div class="item">Item 4</div>  
  <div class="item">Item 5</div>  
  <div class="item">Item 6</div>  
</div>
```

No CSS

```
.grid-justify-content-center{  
  border: 1px solid #000;  
  max-width: 500px;  
  grid-template: repeat(3, 110px) / 100px 100px;  
  justify-content: center;  
}
```

Resultado



Os valores possíveis para a propriedade **justify-content** são: start, end, stretch, space-around, space-between, space-evenly, center.

Alinhando grid itens em relação ao eixo Y

A propriedade **align-content** permite alinhar os itens em relação ao eixo Y. Vejamos um exemplo de como centralizar elementos do grid no eixo Y:

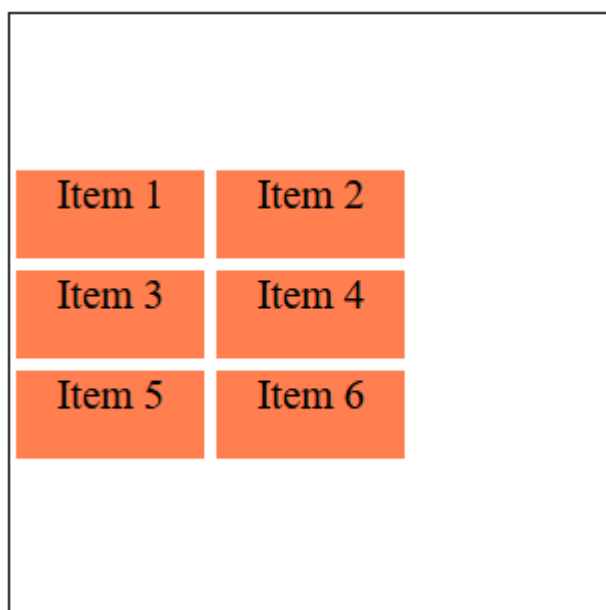
No HTML

```
<div class="grid-container grid-align-content-center">
  <div class="item">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
  <div class="item">Item 6</div>
</div>
```

No CSS

```
.grid-align-content-center{
  border: 1px solid #000;
  max-width: 300px;
  height: 300px;
  grid-template: repeat(3, 50px) / 100px 100px;
  align-content: center;
}
```

Resultado



Os valores possíveis para o **align-content** são: start, end, stretch, space-around, space-between, space-evenly, center.

Existem propriedades que alinham o conteúdo dos itens do grid como o **justify-items** e **align-items**.

Manipulando limites de grid itens

A propriedade **grid-column** permite delimitar o item dentro do grid container. Observe o exemplo e que o item ocupa a segunda coluna:

No HTML

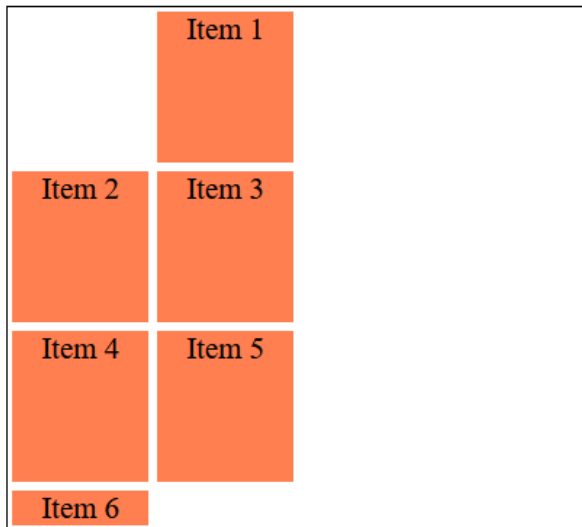
```
<div class="grid-container grid-column">
  <div class="item grid-column-2">Item 1</div>
  <div class="item">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
  <div class="item">Item 6</div>
</div>
```

No CSS

```
.grid-column{
  border: 1px solid #000;
  max-width: 400px;
  grid-template: repeat(3, 110px) / 100px 100px;
}
.grid-column-2{
  grid-column: 2;
```

```
}
```

Resultado



Outro recurso bastante útil é que essa propriedade permite definir a posição inicial e final do elemento dentro do grid. Observe o exemplo a seguir que define que o item 2 deve ocupar a coluna 1 e a coluna 2.

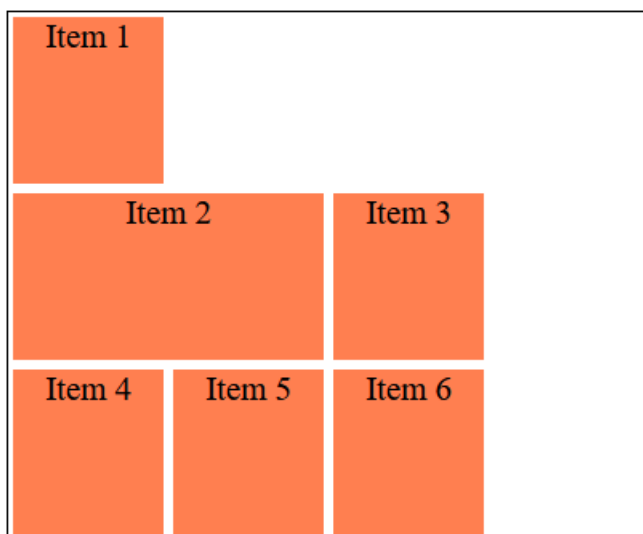
No HTML

```
<div class="grid-container grid-column">
  <div class="item">Item 1</div>
  <div class="item grid-column-1-2">Item 2</div>
  <div class="item">Item 3</div>
  <div class="item">Item 4</div>
  <div class="item">Item 5</div>
  <div class="item">Item 6</div>
</div>
```

No CSS

```
.grid-column{
  border: 1px solid #000;
  max-width: 400px;
  grid-template: repeat(3, 110px) / 100px 100px 100px;
}
.grid-column-1-2{
  grid-column: 1 / 3; /* Ocupa a coluna 1 a coluna 2 */
  grid-column-start: 1; /* Inicia na coluna 1 */
  grid-column-end: 3; /* Ocupa até a coluna 2 */
}
```

Resultado



Existe também a propriedade **grid-row** que funciona da mesma maneira para linhas. Assim como é possível definir quais colunas ocupar com o **grid-column** é possível definir quais linhas ocupar com **grid-row**.

A propriedade **grid-area** é um atalho para **grid-row-start**, **grid-column-start**, **grid-row-end**, **grid-column-end**.

A propriedade **justify-self** justifica o item em relação ao eixo X e o **align-self** justifica o item em relação ao eixo Y.

7. Introdução ao Javascript

Criado por Brendan Eich da Netscape o Javascript é uma linguagem leve, interpretada e com recursos de orientação a objetos e funções de primeira classe. Inicialmente o contexto de execução do javascript era o navegador web onde a principal finalidade era adicionar comportamento em páginas web. Atualmente o javascript tem sido utilizado também fora do navegador pelo fato de ser uma linguagem de script de fácil aprendizado e multiparadigma (orientação a objetos, imperativa e funcional) tem ganhado espaço inclusive do lado servidor.

O javascript padrão é o [ECMA-262](#) que desde 2012 os navegadores mais modernos suportam ao menos a versão [ECMAScript 2011 \(ES5.1\)](#), em navegadores mais antigos a especificação de referência é o ECMAScript 3. Em 2015 foi publicado o [ECMAScript 2015 \(ES6\)](#) que trouxe várias novidades e melhorias, geralmente o ciclo de lançamento das especificações ECMAScript tem sido anual, e o último lançamento até a escrita desse documento foi o [ECMAScript 2019](#).

É importante ter em mente que antes de escrever código javascript baseado em alguma versão do ECMAScript esteja certo que os navegadores, engine javascript que deseja utilizar suportam a especificação, nem sempre são suportados todos os itens da especificação então se atente a isso. Para detalhes de suporte e compatibilidade consulte o site kangax.github.io que possui informações atualizadas sobre o suporte a novas features.

7.1. Sintaxe Javascript

No javascript existe o seguinte conjunto de palavras reservadas:

break	do	instanceof	typeof
case	else	new	var
catch	finally	return	void
continue	for	switch	while
debugger	function	this	with
default	if	throw	
delete	in	try	

Estas palavras reservadas não podem ser utilizadas como identificadores no javascript.

Operadores existem para nos auxiliar a escrever expressões, no javascript nós temos o seguinte conjunto de operadores:

{	}	()	[]
.	;	,	<	>	<=
>=	==	!=	===	!==	
+	-	*	%	++	--
<<	>>	>>>	&	 	^
!	~	&&	 	?	:
=	+=	-=	*=	%=	<<=
>>=	>>>=	&=	 =	^=	

Os tipos básicos no javascript são:

Undefined Type: Variáveis que não tem valor definido são consideradas **undefined**

Null Type: É exatamente o valor **null**

Boolean Type: Representa valores lógicos **true** ou **false**

String Type: Representa dados textuais

Sequências de caracteres literais (Strings) são envolvidas por “” ou “” por exemplo:

```
"Minha String"  
'Minha String'
```

No caso de string o operador + assume a função de concatenação, por exemplo:

```
var myString = "DM" + "121";
```

O resultado desse código será:

```
DM121
```

Abaixo segue o conjunto de funções que podem ser utilizadas para manipulação de String, não iremos abordar todas as funções disponíveis para manipulação de strings, você pode encontrar maiores detalhes [aqui](#).

String.prototype.toString ()

String.prototype.valueOf ()

String.prototype.charAt (pos)

String.prototype.charCodeAt (pos)

String.prototype.concat ([string1 [, string2 [, ...]]])

String.prototype.indexOf (searchString, position)

String.prototype.lastIndexOf (searchString, position)

String.prototype.localeCompare (that)

String.prototype.match (regexp)

String.prototype.replace (searchValue, replaceValue)

String.prototype.search (regexp)

String.prototype.slice (start, end)

String.prototype.split (separator, limit)

String.prototype.substring (start, end)

String.prototype.toLowerCase ()

String.prototype.toLocaleLowerCase ()

String.prototype.toUpperCase ()

String.prototype.toLocaleUpperCase ()

String.prototype.trim ()

length()

Seque alguns exemplos de manipulação de String em javascript.

Substituindo String

String.prototype.replace (searchValue, replaceValue)

```
var myString = "DM121";  
var myStringReplace = myString.replace("DM", "DM:");
```

O resultado desse código será que o trecho **DM** será substituído por **DM:**

```
DM:121
```

Separando String

String.prototype.split (separator, limit)

```
var myString = "Item1,Item2,Item3";  
var stringSplit = myString.split(",");  
console.log(stringSplit[0]);  
console.log(stringSplit[1]);  
console.log(stringSplit[2]);
```

Basicamente a função **Split** retorna um array de string até o ponto do separador. Neste exemplo o resultado esperado é um array com três posições:

```
Item1  
Item2  
Item3
```

Transformando strings para maiúsculas ou minúsculas

String.prototype.toUpperCase () e **String.prototype.toLowerCase ()**

```
var myString = "DM121";  
console.log(myString.toLowerCase());  
console.log(myString.toUpperCase());
```

O resultado esperado neste exemplo é:

```
dm121  
DM121
```

Procurando substring em string

```
var myString = "DM121";  
console.log(myString.substring(1,3));
```

Basicamente a função **substring** irá retornar parte da string dado um índice inicial e índice final, neste exemplo o resultado esperado é:

```
M1
```

Para usar scape em string em javascript pode ser utilizado o caracter \ por exemplo:

```
scape = "Minha String \"com scape\""
```

O resultado seria: "Minha String "com scape""

Comentários de código podem ser feitos da seguinte maneira:

```
/* Comentário mais de uma linha */  
//comentário única linha
```

Number Type: Pode ter 18437736874454810627 ($2^{64} - 2^{53} + 3$) valores e isso representa um double precision de 64bits. Se uma variável possui um único valor não número esta é considerada como um **NaN**. Utilizando a função isNaN é possível identificar essa característica.

```
isNaN('2005/12/12') //verdadeiro  
isNaN(5-2) //falso
```

Abaixo segue o conjunto de funções que podem ser utilizadas para manipulação de números, não iremos abordar todas as funções disponíveis para manipulação de números, você pode encontrar maiores detalhes [aqui](#).

<Adicionar Number.prototype>

Object Type: É uma coleção de propriedades onde cada propriedade tem um nome que irá servir para associar ao dado.

```
object = {nome:"DM121", titulo:"Introdução ao Desenvolvimento Web"}  
object.nome; //guarda a string "DM121"  
object.titulo; //guarda a string "Introdução ao Desenvolvimento Web"
```

Declarando variáveis

Variáveis no javascript são declaradas da seguinte forma:

```
var minhaVariavel = 10;
```

Onde **var** é a palavra reservada e **minhaVariavel** o identificador e o sinal de = é o operador de atribuição, neste caso dizemos que **minhaVariavel** recebe o valor **10**.

If Statement

A sintaxe de uma afirmação **if** pode ser escrita da seguinte maneira:

```
var minhaVariavel = 10;

if (minhaVariavel == 10) {
    console.log("Igual a 10");
} else {
    console.log("Diferente que 10");
}
```

Neste exemplo verifica se o valor da variável é igual a 10 e baseado nessa condição é impresso a mensagem específica.

Estruturas de repetição

No javascript as estruturas de repetição são baseadas nos comandos: do, while, for.

Do

```
var minhaVariavel = 10;

do{
    console.log("Iteração" + minhaVariavel);
    minhaVariavel--;
}while(minhaVariavel != 0);
```

Neste exemplo é impresso mensagens “Iteração10”, “Iteração9”...“Iteração 1”.

While

```
var minhaVariavel = 10;

while(minhaVariavel != 0){
    console.log("Iteração" + minhaVariavel);
    minhaVariavel--;
}
```

Neste exemplo é impresso mensagens “Iteração10”, “Iteração9”...“Iteração 1”.

For

```
for (var minhaVariavel = 10; minhaVariavel > 0; minhaVariavel--) {  
    console.log("Iteração" + minhaVariavel);  
}
```

Neste exemplo é impresso mensagens “Iteração10”, “Iteração9”.....“Iteração 1”.

No caso do **For** temos outras formas de escrever iterações.

For...in iterando em objetos.

```
var anyObject = {nome:"DM121", titulo:"Introdução ao Desenvolvimento Web"};  
  
for (var i in anyObject) {  
    console.log(i + " - " + anyObject[i]);  
}
```

O resultado será:

```
nome - DM121  
titulo - Introdução ao Desenvolvimento Web
```

Controlando a fluxo de iterações

Quando escrevemos estruturas de repetição em alguns casos temos que considerar interrupções de laços ou até mesmo pular algum item específico de uma iteração, quando estamos utilizando **do**, **while** ou **for** é possível interromper a iteração utilizando a palavra reservada **break** ou ir para o próximo elemento utilizando a palavra reservada **continue**.

Interrompendo iterações com **break**

```
for (var i = 0; i < 10; i++) {  
    console.log(i);  
    if (i == 4) {  
        break;  
    }  
}
```

Neste exemplo será impresso via console.log os valores de 0 a 4 pois quando a variável **i** for igual a **4** a interação será interrompida com **break**.

Pulando interações com **continue**

```
for (var i = 0; i < 10; i++) {  
    if (i == 4) {  
        continue;  
    }  
    console.log(i);  
}
```

```
}
```

Neste exemplo será impresso via `console.log` os valores de 0 a 9 exceto o **valor 4**, pois quando a variável **i** for igual a **4** a iteração será interrompida imediatamente, ou seja, não irá executar o código **`console.log(i)`** e irá pular pra próxima iteração.

Instrução `switch/case`

```
var mes = "Março";

switch(mes){
  case "Janeiro":
    console.log("O mês é: " + mes);
    break;

  case "Fevereiro":
    console.log("O mês é: " + mes);
    break;

  case "Março":
    console.log("O mês é: " + mes);
    break;

  case "Abrir":
    console.log("O mês é: " + mes);
    break;

  default:
    console.log("Não encontrado o mês");
    break;
}
```

Neste exemplo o resultado será:

O mês é: Março

A instrução **`switch`** recebe a expressão que será comparada a cada clausula **`case`**, o **`default`** é chamado toda vez que nenhuma das clausulas são satisfeitas.

Array em Javascript

O primeiro elemento de um array inicia da posição 0.

```
var myArray = [1, 10, 60];
var meses = ["Janeiro", "Fevereiro", "Março", "Abril"];
```

Neste exemplo criamos dois arrays, um array de números inteiros e outro array de meses do ano. O acesso ao array é através de índices, por exemplo:

```
myArray[1];
```

O resultado será: **10**

```
meses[3];
```

O resultado será: **"Abril"**

A outra maneira de instanciar um array é utilizando a palavra chave **Array**, por exemplo:

```
var myArray = new Array(1, 10, 60);
```

Quando declaramos um array usando colchetes [] significa que estamos utilizando a forma literal e ao menos será possível especificar os elementos iniciais do array. Quando é declarado um array via o construtor **new Array** isso permite especificar o **length** do array, as duas maneiras produzem basicamente os mesmos resultados.

Abaixo segue o conjunto de funções que podem ser utilizadas para manipulação de array, não iremos abordar todas as funções disponíveis para manipulação de arrays, você pode encontrar maiores detalhes [aqui](#).

Array.isArray (arg)

Array.prototype.toString ()

Array.prototype.toLocaleString ()

Array.prototype.concat ([item1 [, item2 [, ...]]])

Array.prototype.join (separator)

Array.prototype.pop ()

Array.prototype.push ([item1 [, item2 [, ...]]])

Array.prototype.reverse ()

Array.prototype.shift ()

Array.prototype.slice (start, end)

Array.prototype.sort (comparefn)

Array.prototype.splice (start, deleteCount [, item1 [, item2 [, ...]]])

Array.prototype.unshift ([item1 [, item2 [, ...]]])

Array.prototype.indexOf (searchElement [, fromIndex])

Array.prototype.lastIndexOf (searchElement [, fromIndex])

Array.prototype.every (callbackfn [, thisArg])

Array.prototype.some (callbackfn [, thisArg])

Array.prototype.forEach (callbackfn [, thisArg])

Array.prototype.map (callbackfn [, thisArg])

Array.prototype.filter (callbackfn [, thisArg])

Array.prototype.reduce (callbackfn [, initialValue])

Array.prototype.reduceRight (callbackfn [, initialValue])

Adicionando item ao array

Para adicionar itens ao array é utilizado o método:

Array.prototype.push ([item1 [, item2 [, ...]]])

```
var myArray = new Array();
var anyObject = {nome:"DM121", titulo:"Introdução ao Desenvolvimento Web"};
myArray.push(anyObject);
```

Neste exemplo foi adicionado o objeto **anyObject** ao array.

Acessando o myArray[0] temos:

```
{nome: "DM121", titulo: "Introdução ao Desenvolvimento Web"}
```

Removendo o último item ao array

Para remover o último elemento do array é utilizado o método:

Array.prototype.pop ()

```
var myArray = new Array();
var anyObject = {nome:"DM121", titulo:"Introdução ao Desenvolvimento Web"};
var anyObject2 = {nome:"DM122", titulo:"Desenvolvimento Híbrido"};
myArray.push(anyObject);
myArray.push(anyObject2);
var lastElement = myArray.pop();
console.log(lastElement);
```

Neste exemplo é adicionado dois objetos ao array e com o método pop() é removido o último item do array, o retorno do método pop() é o elemento removido.

Procurando um item específico no array

Para procurar um item específico no array podemos utilizar o método:

Array.prototype.indexOf (searchElement [, fromIndex])

```
var myArray = ["Janeiro", "Fevereiro", "Março", "Abril", "Maio"];
var index = myArray.indexOf("Março");
if (index >= 0) {
```



```
    console.log(index);  
} else {  
    console.log("Não encontrado!");  
}
```

Neste exemplo caso o elemento seja encontrado será retornado o índice do elemento, caso contrário é retornado -1.

Iterando nos itens do array

O objeto array possui um método específico para iterarmos nos itens do array.

Array.prototype.forEach (callbackfn [, thisArg])

```
var myArray = new Array("Javascript", "Java", "C#", "C", "C++", "GO", "Python",  
"Groovy");  
myArray.forEach(function(elemento){  
    console.log(elemento);  
});
```

Neste exemplo será impresso todos os elementos do **myArray**.

```
Javascript  
Java  
C#  
C  
C++  
GO  
Python  
Groovy
```

Invertando elementos do array

O objeto array possui um método específico para reverter as posições do array, o que ocorre é que o primeiro elemento do array vai para a última posição e o último elemento do array vai para a primeira posição.

Array.prototype.reverse ()

```
var myArray = [5, 9, 0, 2, 3];  
var reverted = myArray.reverse();  
console.log(reverted);
```

O resultado esperado é:

```
[ 3, 2, 0, 9, 5 ]
```

Concatenando arrays

O objeto array possui um método específico para concatenar arrays.

Array.prototype.concat ([item1 [, item2 [, ...]]])

```
var myArray1 = ["Janeiro", "Fevereiro", "Março", "Abril", "Maio", "Junho"];
var myArray2 = ["Julho", "Agosto", "Setembro", "Outubro", "Novembro",
"Dezembro"];
var mayArrayConcat = myArray1.concat(myArray2);
console.log(mayArrayConcat);
```

O resultado esperado é:

```
[ 'Janeiro',
  'Fevereiro',
  'Março',
  'Abril',
  'Maio',
  'Junho',
  'Julho',
  'Agosto',
  'Setembro',
  'Outubro',
  'Novembro',
  'Dezembro' ]
```

Nós exploramos alguns dos métodos disponíveis para manipulação de array. Conforme apresentado existem outros métodos que não estão exemplificados neste documento, a recomendação é que você explore os outros métodos para manipulação de arrays.

Funções em Javascript

Em javascript as funções são definidas pela palavra chave **function** cujo o escopo da função é delimitado por chaves {...}. Toda função pode receber um ou mais parâmetros e retornar ou não algum valor por exemplo:

```
var list = [1,2,9, 0, -8, 10];
```

```
console.log(removeItemFromArray(list, -8));

function removeItemFromArray(list, element){

    list.splice(list.indexOf(element),1);

    return list;

}
```

Neste exemplo é criado uma função chamada **removeItemFromArray** que recebe dois parâmetros e retorna a variável **list**. Após a chamada dessa função o resultado esperado é a própria list com o elemento -8 removido da lista [1, 2, 9, 0, 10].

Em javascript as funções podem ser passadas como parâmetros para outras funções (callbacks), isso é útil quando temos operações assíncronas envolvidas.

Escopo de variáveis em Javascript

Observe o seguinte código:

```
var list = [1,2,9, 0, -8, 10];

console.log(removeItemFromArray(list, -8));

function removeItemFromArray(list, element){

    list.splice(list.indexOf(element),1);

    console.log("Exibe o valor da variável a: " + a);

    return list;

};

var a = "A";
```

Basicamente declaramos um array chamado **list** e chamamos a função **removeItemFromArray** e passamos dois parâmetros, a variável **list** e o número **-8**, a função irá remover o item **-8** da lista, imprimir a variável **a** e retornar a lista atualizada.

Observe que após a declaração da função temos a variável **a** declarada e recebendo o valor “A”, minha pergunta é:

Esse código funciona? Sim funciona. No javascript as declarações de variáveis são processadas antes de executar qualquer código, ou seja, independente se a variável está dentro ou não de algum bloco (if, for, while, function) ela terá visibilidade.

Em versões mais novas da especificação ECMAScript existem outras maneiras de declarar variáveis com escopo melhor definido, por enquanto não iremos tratar destas características.

O que são closures Javascript?

O conceito de closure permite a escrita de códigos javascript melhores e nos dá uma visão melhor de escopo no javascript.

Observe o código abaixo:

```
function funcaoPrincipal(param){
    var nome = "Olá " + param;

    function funcaoInterna(){
        console.log(nome);
    }

    return funcaoInterna;
}

var f1 = funcaoPrincipal("DM121");

f1();
f1();
f1();
```

Observe que temos funções aninhadas, a função **funcaoInterna** está dentro da função **funcaoPrincipal**, outro detalhe é que a variável **nome** é acessível normalmente dentro da função interna.

O resultado esperado para esse código é:

```
Olá DM121
Olá DM121
Olá DM121
```

Na linha:

```
var f1 = funcaoPrincipal("DM121");
```

Se chamarmos a função **funcaoPrincipal** neste primeiro momento nada acontece, no entanto se invocarmos **f1()** percebe-se que temos a mesma saída.

```
f1();
f1();
f1();

Olá DM121
Olá DM121
Olá DM121
```

Esse é o comportamento de closure, a **funcaoPrincipal** travou todo o escopo e a consequência disso é que pode-se utilizar em momentos diferentes, ou seja, o valor da variável **nome** está fechado dentro de um escopo definido.

Funções de objeto globais

As funções abaixo podem ser chamadas diretamente sem a necessidade de um objeto.

eval (x)

parseInt (string , radix)

parseFloat (string)

isNaN (number)

isFinite (number)

Abaixo alguns exemplos de uso de funções globais do javascript.

parseInt (string , radix)

```
var myString = "45";  
console.log(parseInt(myString));
```

A função **parseInt** converte uma string em número inteiro.

O parâmetro **radix** define a base numérica do número a ser convertido.

```
var hexString = "0xF";  
console.log(parseInt(hexString, 16));
```

O resultado esperado é:

```
15
```

parseFloat (string)

```
var myString = "3.14";  
console.log(parseFloat(myString));  
console.log(parseFloat('314e-2'));  
console.log(parseFloat('0.0314E+2'));  
console.log(parseFloat('3.14com caracteres'));
```

A função **parseFloat** converte o argumento em um número real.

Em todos os casos acima a conversão será possível, em casos onde a conversão não funcione é retornado um **NaN**.

Manipulação de data e hora

No javascript existem funções específicas para manipulação de datas.

A partir do Date prototype é possível utilizar as seguintes funções para manipular datas:

Date.parse (string)

```
console.log(Date.parse("March 21, 2018"));
```

Neste exemplo o **Date.parse** retorna um número que representa a data.

```
1521601200000
```

Date.UTC (year, month [, date [, hours [, minutes [, seconds [, ms]]]]])

```
console.log(Date.UTC(2018, 03, 09, 19, 20));
```

Neste exemplo o **Date.UTC** retorna um número que representa a data é possível enviar vários parâmetros para configurar a data. A data mínima é de 1º de janeiro de 1970, 00:00:00, e a hora acompanha o padrão universal.

Date.now ()

```
console.log(Date.now());
```

O **Date.now()** retorna um número que representa a data e hora corrente.

```
1523312656666
```

Funções acessíveis através de um objeto Date

Abaixo segue o conjunto de funções que podem ser utilizadas para manipulação de datas, não iremos abordar todas as funções disponíveis para manipulação de datas, você pode encontrar maiores detalhes [aqui](#).

Instanciando um objeto date.

```
var date = new Date();  
console.log(date);
```

Date.prototype.getDate()

Date.prototype.getDay()

Date.prototype.getFullYear()

Date.prototype.getHours()

Date.prototype.getMilliseconds()

Date.prototype.getMinutes()

Date.prototype.getMonth()

Date.prototype.getSeconds()

Date.prototype.getTime()

Date.prototype.getTimezoneOffset()

Date.prototype.getUTCDate()

Date.prototype.getUTCDay()

Date.prototype.getUTCFullYear()

Date.prototype.getUTCHours()

Date.prototype.getUTCMilliseconds()

Date.prototype.getUTCMinutes()
Date.prototype.getUTCMonth()
Date.prototype.getUTCSeconds()
Date.prototype.getYear()
Date.prototype.setDate()
Date.prototype.setFullYear()
Date.prototype.setHours()
Date.prototype.setMilliseconds()
Date.prototype.setMinutes()
Date.prototype.setMonth()
Date.prototype.setSeconds()
Date.prototype.setTime()
Date.prototype.setUTCDate()
Date.prototype.setUTCFullYear()
Date.prototype.setUTCHours()
Date.prototype.setUTCMilliseconds()
Date.prototype.setUTCMinutes()
Date.prototype.setUTCMonth()
Date.prototype.setUTCSeconds()
Date.prototype.setYear()
Date.prototype.toString()
Date.prototype.toGMTString()
Date.prototype.toISOString()
Date.prototype.toJSON()
Date.prototype.toLocaleDateString()
Date.prototype.toLocaleFormat()
Date.prototype.toLocaleString()
Date.prototype.toLocaleTimeString()
Date.prototype.toSource()
Date.prototype.toString()

Date.prototype.toString()

Date.prototype.toUTCString()

Date.prototype.valueOf()

Retornando o dia da semana do objeto date

Date.prototype.getDay()

```
var date = new Date();  
console.log(date.getDay());
```

O resultado esperado é um número inteiro que representa o dia da semana, onde **zero** corresponde a domingo, **um** segunda-feira e assim por diante.

Retornando os minutos do objeto date

Date.prototype.getMinutes()

```
var date = new Date("09/03/2018 14:32");  
console.log(date.getMinutes());
```

Neste exemplo a data acima é esperado retornar 32 que é justamente o número de minutos passado para o objeto date.

Considerações Importantes

- Carregamentos globais no mesmo objeto.

```
var a = 1;  
window.a = 2;  
console.log(a);
```

- O escopo é da função, blocos não tem escopo.
- Comportamento do typeof

```
console.log(typeof null); //object  
console.log(typeof 100); //number  
console.log(typeof ''); //string
```

- Comportamento do **null** e **undefined**

```
console.log(null === undefined); // false
```



```
console.log(null == undefined); // true
console.log(null === null);    // true
console.log(null == null);     // true
console.log(!null);            // true
console.log(isNaN(1 + null));   // false
console.log(isNaN(1 + undefined)); // true
```

- O **this** no javascript.
 - Cada linha de código Javascript é executado em um contexto de execução;
 - Esses contextos são empilhados, onde o contexto mais ativamente executado é colocado na parte superior da pilha;
 - Temos o código de contexto global (código fora de qualquer função);
 - Temos o código de contexto de função (código dentro da função);
 - Temos o código de contexto eval (código global via chamada da função eval);

Em resumo podemos considerar o funcionamento do **this** desta forma:

- Por padrão, **this** é referente ao objeto global;
- Quando se chama funções como propriedade de um objeto pai, o **this** é referente ao objeto pai dentro da função que foi chamada.
- Quando uma função é chamada com **new** o **this** é referente ao objeto recentemente criado dentro desta função.
- Quando uma função é chamada com **call** ou **apply** o **this** é referente ao primeiro argumento que foi passado, caso seja **null** ou não objeto o **this** irá se referir ao objeto global. O **call** e o **apply** estão presentes em todas as funções javascript e o funcionamento dos dois é idêntico (altera o valor do **this** explicitamente), a diferença é que o **apply** aceita um array como parâmetro enquanto o **call** aceita valores passados individualmente.

7.2. DOM API

DOM (Document Object *Model*) é uma representação em formato de árvore dos elementos da página e é baseado nesta estrutura que os navegadores conseguem renderizar páginas web.

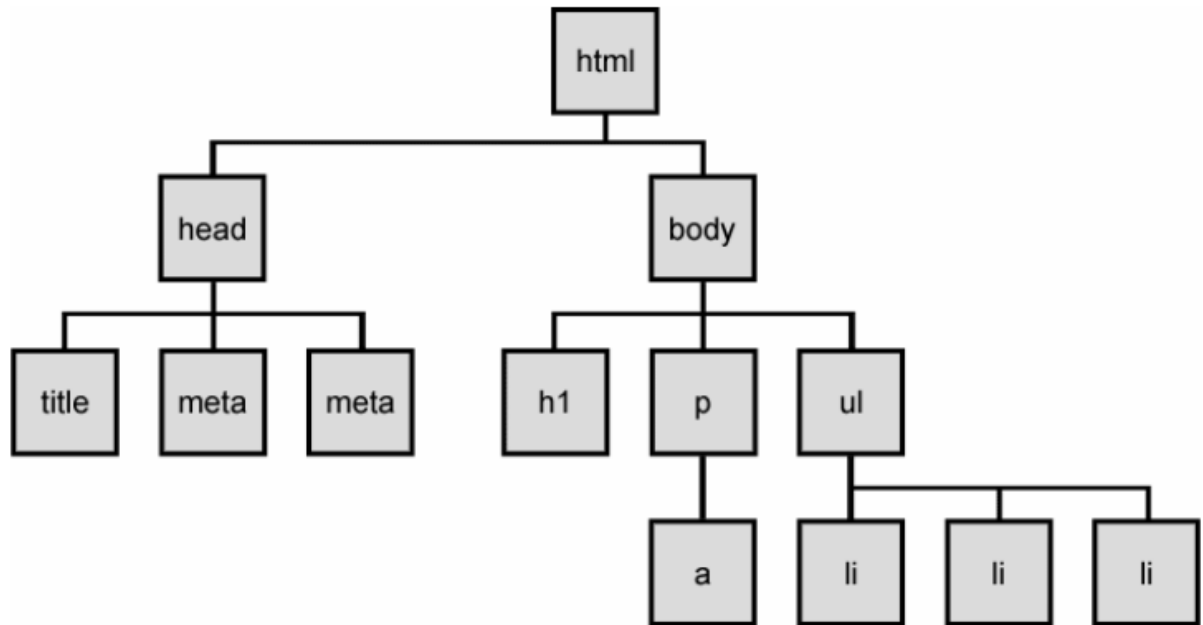


Figura 2 - DOM (Document Object Model)

Por que é preciso manipular o DOM? Para tornar páginas web dinâmicas.

Onde entra Javascript nisso? DOM não é uma linguagem e sim uma representação, a linguagem que irá manipular essa representação é o Javascript. Inicialmente o DOM e Javascript eram fortemente acoplados, mas atualmente ambos são entidades completamente separadas, ou seja, o conteúdo da página (documento, elementos, textos) são armazenados na página, porém o Javascript é capaz de acessar e manipular esse conteúdo via APIs.

Os navegadores implementam o DOM de maneira diferente, no entanto, para que o DOM seja acessível e possível de ser manipulado os navegadores devem seguir as especificações do DOM (<https://www.w3.org/DOM/> e <https://dom.spec.whatwg.org/>).

Considere o seguinte código HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>DOM</title>
```

```

</head>

<body>

  <h1 id="titulo">Manipulando DOM</h1>

  <p class="texto">Lorem ipsum dolor, sit amet consectetur adipisicing elit.</p>

  <p class="texto">Sint nam nostrum fugit! Modi hic repudiandae corporis
eveniet.</p>

  <button type="button">Enviar</button>

<script>

  var windowObj = window;

  var documentObj = document;

  console.log(windowObj);

  console.log(documentObj);

</script>

</body>

</html>

```

Se olharmos para o console do navegador temos as seguintes informações:

windowObj essa variável representa o objeto window (o navegador), observe que temos acesso aos métodos e atributos referentes ao objeto **window**.

```

▼ Window {postMessage: f, blur: f, focus: f, close: f, frames: Window, ...} ⓘ
  ▶ alert: f alert()
  ▶ applicationCache: ApplicationCache {status: 0, onchecking: null, onerror: null, onnoupdate: null, ondownloading: null, ...}
  ▶ atob: f atob()
  ▶ blur: f ()
  ▶ btoa: f btoa()
  ▶ caches: CacheStorage {}
  ▶ cancelAnimationFrame: f cancelAnimationFrame()
  ▶ cancelIdleCallback: f cancelIdleCallback()
  ▶ captureEvents: f captureEvents()
  ▶ chrome: {loadTimes: f, csi: f}
  ▶ clearInterval: f clearInterval()
  ▶ clearTimeout: f clearTimeout()
  ▶ clientInformation: Navigator {vendorSub: "", productSub: "20030107", vendor: "Google Inc.", maxTouchPoints: 0, hardwareConcurrency: 4, ...}
  ▶ close: f ()
  ▶ closed: false

```

documentObj essa variável representa o **document** HTML, observe que temos disponível todos os elementos HTML da página, em outras palavras, temos acesso ao documento da página.

```

▼ #document
  <!DOCTYPE html>
  <html lang="en">
    ▼ <head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <title>DOM</title>
    </head>
    ▼ <body>
      <h1 id="titulo">Manipulando DOM</h1>
      <p class="texto">Lorem ipsum dolor, sit amet consectetur adipisicing elit.</p>
      <p class="texto">Sint nam nostrum fugit! Modi hic repudiandae corporis eveniet.</p>
      <button type="button">Enviar</button>
      ▶ <script>...</script>
    </body>
  </html>

```

Podemos concluir que o objeto **window** e **document** são nossas interfaces para manipular o DOM via javascript.

A API DOM é extensa e não iremos abordar todas as possibilidades aqui, segue a lista das APIs mais comumente utilizadas para manipulação do DOM. Você pode encontrar mais detalhes [aqui](#).

document.getElementById(id)

document.getElementsByTagName(name)

document.createElement(name)

document.querySelector(String selector)

document.querySelectorAll(String selector)

parentNode.appendChild(node)

element.innerHTML

element.style.left

element.setAttribute()

element.getAttribute()

element.addEventListener()

window.content

window.onload

window.dump()

window.scrollTo()

Acessando um elemento específico do DOM

document.getElementById(id)

```

var elementoTitulo = document.getElementById("titulo");
console.log(elementoTitulo);

```

O elemento impresso no console é:

<h1 id="titulo">Manipulando DOM</h1>

Neste exemplo invocamos o método **getElementById** do objeto **document** que nos retorna o elemento que possui o ID titulo.

Acessando elementos pela tag name

document.getElementsByTagName(name)

```
var paragrafos = document.getElementsByTagName("p");
console.log(paragrafos);
```

```
▼ HTMLCollection(2) [p.texto, p.texto] ⓘ
  ▶ 0: p.texto
  ▶ 1: p.texto
  length: 2
  ▶ __proto__: HTMLCollection
```

Observe que foi retornado os dois parágrafos existentes na página.

Adicionando elementos ao DOM

document.createElement(name)

```
var newElement = document.createElement("h2");
var textElement = document.createTextNode("Titulo H2");
newElement.setAttribute("id", "titulo_h2");
newElement.appendChild(textElement);
elementoTitulo.insertAdjacentElement("afterend", newElement);
```

Observe o que ocorre em cada trecho de código:

Primeiramente é criado o elemento **h2**

```
var newElement = document.createElement("h2");
```

É criado o elemento **textNode**, nesse caso com conteúdo “Titulo H2”

```
var textElement = document.createTextNode("Titulo H2");
```

Ao elemento **newElement** é adicionado um id chamado de **titulo_h2**

```
newElement.setAttribute("id", "titulo_h2");
```

No elemento **newElement** é adicionado o elemento **textElement** como filho

```
newElement.appendChild(textElement);
```

E por fim, é adicionado ao **elementoTitulo** o **newElement**

```
elementoTitulo.insertAdjacentElement("afterend", newElement);
```

O **insertAdjacentElement** recebe dois parâmetros, o primeiro é o position que indica onde o elemento irá ser adicionado, em nosso caso iremos adicionar após o **elementoTitulo** e o segundo parâmetro é o elemento que será adicionado, neste caso **newElement**.

Acessando propriedades do objeto window

```
var w=window.innerWidth;
var h=window.innerHeight;
var newParagrafo = document.createElement("p");
var textElementParagrafo = document.createTextNode("Largura: "+w+" Altura: "+h);
newParagrafo.appendChild(textElementParagrafo);
document.body.appendChild(newParagrafo);
```

Neste exemplo é acessado via objeto **window** as propriedades **innerWidth** e **innerHeight** que respectivamente retornam a largura e a altura da janela do navegador.

Acessando elementos com querySelector

document.querySelector(String selector)

```
var paragrafo = document.querySelector(".texto");
console.log(paragrafo);
```

Neste exemplo o **querySelector** retorna o primeiro elemento do documento que contém o seletor de classe **texto**.

```
<p class="texto">Lorem ipsum dolor, sit amet consectetur adipisicing elit.</p>
```

document.querySelectorAll(String selector)

```
var paragrafos = document.querySelectorAll(".texto");
console.log(paragrafos);
```

Neste exemplo o **querySelectorAll** retorna todos os elementos do documento que contém o seletor de classe **texto**.

```
▼ NodeList(2) [p.texto, p.texto] ⓘ
  ► 0: p.texto
  ► 1: p.texto
  length: 2
  ► __proto__: NodeList
```

Manipulando estilos CSS

Acessando o DOM via Javascript é possível alterar/adicionar formatações CSS, observe o seguinte código:

```
var titulo = document.querySelector("#titulo");
titulo.style.color = "red";
```

Neste exemplo o primeiro elemento do documento que contém o seletor de ID **titulo** terá a cor vermelha.

Manipulando DOM

Lorem ipsum dolor, sit amet consectetur adipisicing elit.

Sint nam nostrum fugit! Modi hic repudiandae corporis eveniet.

Enviar

DOM Eventos

A API DOM nos permite manipular eventos, os estilos mais comuns são: **addEventListener** e **onEvent**.

Considere o seguinte HTML.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>DOM</title>
</head>
<body>
  <h1 id="titulo">Manipulando DOM</h1>
  <p class="texto">Lorem ipsum dolor, sit amet consectetur adipisicing elit.</p>
  <p class="texto">Sint nam nostrum fugit! Modi hic repudiandae corporis
eveniet.</p>
  <button type="button">Enviar</button>
<script>
</script>
</body>
</html>
```

Utilizando o **addEventListener**

```
var button = document.querySelector("button");
button.addEventListener('click', whenClicked);

function whenClicked(event){
  console.log("Clicou " + event);
}
```

Bom, nesse exemplo estamos registrando um listener para o elemento **button** que irá chamar a função **whenClicked** quando o evento **click** acontecer.

Como resultado temos:

```
3 Clicou [object MouseEvent]  
>
```

Os eventos mais comuns utilizados na interface MouseEvent são:

click, dblclick, mouseup, mousedown.

Observação: O **addEventListener** é a maneira mais comum de se trabalhar com eventos no DOM, no entanto no internet explorer com versão menor ou igual ao Internet Explorer 8 é usado o **attachEvent** ao invés do **addEventListener**.

A segunda opção, podemos adicionar o event listener diretamente no elemento HTML, por exemplo:

```
<h1 id="titulo" onclick="whenClicked(this)">Manipulando DOM</h1>
```

Neste caso adicionamos o evento **onclick** ao elemento **h1** e toda vez que acontecer o click será chamada a função **whenClicked** passando a referência do elemento que disparou o evento (**this**).

Como resultado temos:

```
Clicou [object HTMLHeadingElement]  
>
```


8. Tópicos Avançados

8.1. ECMA Script versões

O Javascript se tornou um padrão em 1997 sendo padronizado por uma associação chamada [ECMA International](https://www.ecma-international.org/). Desde então várias versões já foram lançadas, oficialmente o nome da linguagem Javascript é ECMAScript mas popularmente é abreviada como ES. Abaixo segue o histórico de versões do Javascript desde a sua padronização.

**A tabela abaixo é apenas o histórico de versões, ou seja, não lista e nem detalha todas as funcionalidades introduzidas em cada versão, você pode consultar a especificação oficial caso queira aprofundar seus estudos <https://www.ecma-international.org/ecma-262/>*

Versão	Nome Oficial	Fuincionalidade
1	ECMAScript 1 (1997)	First Edition.
2	ECMAScript 2 (1998)	Editorial changes only.
3	ECMAScript 3 (1999)	Added Regular Expressions. Added try/catch.
4	ECMAScript 4	Never released.
5	ECMAScript 5 (2009)	Added "strict mode". Added JSON support. Added String.trim(). Added Array.isArray(). Added Array Iteration Methods.
5.1	ECMAScript 5.1 (2011)	Editorial changes.
6	ECMAScript 2015	Added let and const. Added default parameter values. Added Array.find(). Added Array.findIndex().
7	ECMAScript 2016	Added exponential operator (**). Added Array.prototype.includes.
8	ECMAScript 2017	Added string padding. Added new Object properties. Added Async functions. Added Shared Memory.
9	ECMAScript 2018	Added rest / spread properties. Added Asynchronous iteration. Added Promise.finally(). Additions to RegExp.
10	ECMAScript 2019	Array.flat() Array.flatMap() String.trimStart() & String.trimEnd() Optional Catch Binding Object.fromEntries() Symbol.description Function.toString() Well Formed JSON.Stringify() Array.Sort Stability

8.2. ECMA Script 2015 – ES6

O ECMA Script 2015, popularmente conhecido como **ES6** foi um divisor de águas para muitos, ou seja, as mudanças mais significativas foram introduzidas nessa versão. Não será abordado em detalhes todas as features do ES6, a ideia aqui é listar as mais importantes e assim permitir trabalhar com javascript de uma maneira mais profissional e alinhado com o mercado e com as boas práticas.

Constants

Com a palavra reservada **const** é possível definir uma variável imutável, ou seja, não podemos fazer re-atribuir outro valor.

```
const PI = 3.141593
```

Block-Scoped Variables

Com a palavra reservada **let** é possível definir uma variável cujo escopo é definido pelo bloco onde essa variável se encontra. No exemplo abaixo a variável “x” está sendo declarado dentro do bloco “if”, isso significa que a variável “x” não pode ser referenciada fora desse bloco.

```
function myFunction(flag) {  
  if (flag) {  
    let x = 10;  
  }  
  return x; // ReferenceError: x is not defined  
}  
console.log(myFunction(true));
```

Block-Scoped Functions

Anteriormente, vimos um exemplo de closure em javascript no estilo ES5, basicamente tínhamos funções aninhadas cujo objetivo era de obter um escopo bem definido. Em ES6 temos uma solução melhor para esse tipo de situação.

```
{  
  function privateFunction(){  
    console.log("Inacessível fora desse escopo")  
    {  
      console.log("Inacessível mesmo a partir de privateFunction")  
    }  
  }  
  privateFunction()  
}  
  
console.log(privateFunction())
```

Observando o código acima podemos chegar na seguinte conclusão: Tudo no ES6 que é envolvido por chaves {.....} tem seu próprio escopo.

Arrow Functions

Na versão ES6 foi introduzida uma nova maneira de escrever funções, as “arrow functions”.

```
const area = r => 3.14 * r * r;  
console.log(area(7))
```

O exemplo acima é uma função que dado o raio (r) calcula-se a área.

Não passar parâmetros para uma arrow function.

```
const message = () => console.log("Alguma Mensagem")  
console.log(message())
```

Passando mais de um parâmetro para uma arrow function.

```
const square = (l, w) => l * w;  
console.log(square(20, 30))
```

Nos exemplos acima a própria sentença é o retorno, por isso que não é utilizado a palavra reservada “return”. Observe abaixo um exemplo utilizando sentenças mais complexas dentro de uma arrow function.

```
const maiorDeIdade = (y) => {  
  if(isNaN(y)){  
    return "Parâmetro Inválido";  
  }  
  
  let today = new Date();  
  
  if(today.getFullYear() - y >= 18){  
    return "Maior de Idade";  
  } else {  
    return "Menor de Idade"  
  }  
}  
  
console.log(maiorDeIdade(2000))
```

Por que utilizar arrow functions?

Ao usar arrow functions é possível escrever funções mais curtas e também existe um comportamento do **this** dentro de arrow function que pode ser considerado um ponto positivo.

Antes de abordarmos o comportamento do **this** nas arrow functions, vamos relembrar o comportamento do **this** nas funções javascript, observe o seguinte exemplo:

```
const anyObject = {
  items : [],
  addAll: function(itemsToAdd) {
    var self = this;
    itemsToAdd.forEach(function(item){
      self.add(item)
    });
  },

  add: function(item){
    var self = this;
    self.items.push(item)
  }
}

anyObject.addAll(["Item 1", "Item 2", "Item 3"])
console.log(anyObject.items)
```

No exemplo acima a finalidade é apenas utilizar a função **add** dentro da função **addAll** do objeto literal **anyObject**, observe que estamos utilizando uma variável temporária **self** que serve para pegar o valor externo do **this** e utilizar dentro da função interna, esse hack é necessário pois a função interna não consegue utilizar o **this** do escopo externo. Outra solução seria utilizar o **bind(this)**. Perceba que essa solução não é nada elegante, mas se você já é mais experiente com Javascript certamente já utilizou esse tipo de hack.

No ES6, você não precisa usar esse tipo de abordagem se seguir as seguintes regras:

- Não use arrow functions para funções que serão chamadas como: **object.method()**. Esse tipo de função já irá receber a referência do **this** do objeto que a invoca.
- Não use arrow functions em callbacks com a necessidade de contextos dinâmicos. Por exemplo:

```
var myButton = document.getElementById('save');
myButton.addEventListener('click', () => {
  this.classList.toggle('on');
});
```

O **this** não se refere ao **myButton** e sim ao escopo pai, nesse caso irá ocorrer um **TypeError**.

- Utilize arrow functions para os demais casos.

Qual o comportamento do **this** na arrow function?

Arrow functions não tem seu próprio this. O valor do **this** dentro de uma arrow function é herdado do escopo que envolve essa arrow function.

```
const myLiteralObject = {
  items : [],
  addAll : function(itemsToAdd){
```

```

        itemsToAdd.forEach(item => this.add(item))
    },

    add : function(item){
        this.items.push(item)
    }
}

myLiteralObject.addAll(["Item 1", "Item 2", "Item 3"])
console.log(myLiteralObject.items)

```

No exemplo acima a função **addAll** recebe o **this** do chamador (**myLiteralObject**), a inner function dentro do **addAll** é uma arrow function (**item => this.add(item)**) e consequentemente irá herdar o **this** do escopo onde ela está anexada.

Aproveitando que estamos utilizando objeto literal, podemos reescrever os métodos do objeto literal no estilo ES6, por exemplo:

```

const myLiteralObject = {
    items : [],
    addAll(itemsToAdd){
        itemsToAdd.forEach(item => this.add(item))
    },

    add(item){
        this.items.push(item)
    }
}

myLiteralObject.addAll(["Item 1", "Item 2", "Item 3"])
console.log(myLiteralObject.items)

```

Template Literals

No ES6 é possível interpolar strings de maneira mais elegante, observe o exemplo abaixo.

```

let code = 'DM121';
let titulo = `Introdução ao desenvolvimento Web - ${code}`;
console.log(titulo);

```

Se você utilizar **`meu template`** isso será entendido como um template literal permitindo que você utilize o **`${expressão ou variáveis}`**

Destructuring Assignment

- Array Matching

```

let list = [ 1, 2, 3 ];
let [ a, , b ] = list;
console.log(a);

```

```
console.log(b);
```

- Parameter Context Matching

```
function fullName([name, lastName]){  
  console.log(name, lastName);  
}  
  
fullName(["Maria", "Silveira"]);
```

Observe o destructuring de objetos

```
function fullName({name, lastName}){  
  console.log(name, lastName)  
}  
  
fullName({name : 'Maria', lastName: 'Silveira'})
```

Modules

No ES6 você pode importar/exportar código sem a necessidade de existir um namespace global, observe o exemplo abaixo:

```
//lib.js  
export const PI = 3.14;  
  
export const sum = (a, b) => a + b;  
export const sub = (a, b) => a - b;  
export const area = r => PI * r * r;
```

```
//app.js  
import * as calc from "./lib";  
  
console.log(`PI: ${calc.PI}`)  
console.log(`SUM: ${calc.sum(2,2)}`)  
console.log(`SUB: ${calc.sub(10, 3)}`)  
console.log(`AREA: ${calc.area(7)}`)
```

No exemplo acima temos códigos javascripts separados em dois arquivos o **lib.js** e o **app.js**. O **lib.js** está exportando algumas funções e o **app.js** está importando essas funções a partir do **lib.js**. Esse conceito de modules já existia na versão anterior do Javascript mas com ajuda de algumas bibliotecas como o [commonJS](#). Até a escrita desse documento a versão do nodejs **12.16.1 LTS** ainda está em estado de “[Stability: 1 - Experimental](#)” para essa feature, o que significa que não é recomendada para ser utilizada em produção. Para conseguir executar o exemplo acima iremos utilizar Babel.

Você encontrar um template do projeto usando módulos e Babel aqui:

<https://drive.google.com/drive/folders/1wHtCtK7yfZQb8UKPtfoW-D6I5kcv0OKY>

Classes

No ES6 foi introduzido a sintaxe de classes cujo objetivo é de permitir que programadores vindo de outras linguagens/paradigmas consigam utilizar o Javascript de maneira mais familiar.

```
class Car {  
  
  constructor(color, year, model){  
    this.color = color;  
    this.year = year;  
    this.model = model;  
  }  
  start(){  
    console.log('Start...')  
  }  
  stop(){  
    console.log('Stop...')  
  }  
  accelerate(){  
    console.log('accelerate...')  
  }  
}  
  
let car = new Car('Black', 2020, 'Sedan');  
console.log(car.start())  
console.log(car.accelerate())  
console.log(car.stop())  
console.log(car)
```

Utilizando generalização, getter/setter no ES6

```
class Gol extends Car {  
  constructor(color, year, model, version){  
    super(color, year, model)  
    this.version = version  
  }  
  
  showVersion(){  
    console.log(this.version)  
  }  
  
  set iSystem(iSystem){  
    this._iSystem= iSystem;  
  }  
  
  get iSystem(){  
    return this._iSystem;  
  }  
}
```

```
let gol = new Gol('Red', 2010, 'hatch', 'trend')
gol.iSystem = true;
console.log(gol);
```

Utilizando membros estáticos

```
class OrderStatus {
  static get entregue(){
    return 'ENTREGUE'
  }

  static get transporte(){
    return 'TRANSPORTE'
  }

  static get processando(){
    return 'PROCESSANDO'
  }
}

console.log(OrderStatus.entregue)
console.log(OrderStatus.transporte)
console.log(OrderStatus.processando)
```

Symbol Type

É um novo tipo primitivo introduzido no ES6 que permite criar identificador único e imutável para propriedades de objetos.

```
console.log(typeof Symbol()) //symbol
console.log(typeof 10) //number
console.log(typeof '') //string
console.log(typeof []) //object
let mySym = Symbol('mySym')
let myObject = {[mySym] : '10'}
console.log(myObject[mySym])
console.log(myObject[Object(mySym)])
```

Onde isso é útil? Com **Symbol** podemos adicionar propriedades dinamicamente em objetos sem que ocorra colisões.

```
let existObject = {name: '', age: 0}
let isValidSymbol = Symbol('isValid')
console.log(existObject)
existObject[isValidSymbol] = true
```



```
console.log(existObject)
console.log(JSON.stringify(existObject))
console.log(existObject[isValidSymbol])
```

Iterators

Permite que objetos customizem comportamentos de iteração. No exemplo abaixo com a ajuda do **Iterator** iremos permitir que um dado objeto possa ser um **Iterable**.

```
class Status{

    static get DONE(){
        return 'DONE'
    }

    static get DOING(){
        return 'DOING'
    }

    static get TODO(){
        return 'TO DO'
    }

    [Symbol.iterator]() {

        let status = 'TO DO'

        return {
            next(){
                switch(status){
                    case 'TO DO':
                        status = 'DOING'
                        return { done: false, value : Status.TODO }

                    case 'DOING':
                        status = 'DONE'
                        return {done: false, value : Status.DOING}

                    case 'DONE':
                        status = undefined
                        return {done: false, value : Status.DONE}

                    default:
                        return {done: true, value: undefined}
                }
            }
        }
    }
}
```

```
status = new Status()

for(s of status){
  console.log(s)
}
```

Promises

No ES6 foi adicionado suporte a Promises. Um objeto Promise permite lidar com operações assíncronas, ao invés de enviar callback para uma função o próprio objeto Promise possui callbacks que permitem tomar ações em caso de sucesso ou erro. Observe o seguinte exemplo:

```
if(doSomething() == 'Done'){
  console.log('Finish')
} else {
  console.log('No Finish')
}

function doSomething(){
  setTimeout(() => {
    return 'Done'
  }, 1000)
}
```

Provavelmente o código acima não irá funcionar pois a função **doSomething** irá executar o processamento após 1 segundo, portanto a verificação do retorno dessa função irá executar antes que a função retorne o valor esperado, ou seja temos um clássico comportamento assíncrono.

Como resolver isso? Usando Promise. Observe o seguinte exemplo:

```
let myPromise = doSomethingPromise();

myPromise
  .then(result => {
    console.log(result)
  })
  .catch(err => {
    console.log(err)
  })

function doSomethingPromise(){
  return new Promise((resolve, reject) => {
    setTimeout((done) => {
      done === true ? resolve('Done') : reject('No finish...')
    }, 1000, true);
  });
}
```

A função **doSomethingPromise()** agora devolve um objeto Promise que possui duas callbacks anexadas, uma callback de sucesso e outra callback de erro, de posse dessas callbacks é possível fazer o tratamento adequado.

Proxy

É um recurso poderoso adicionado no ES6 que permite interceptar operações executadas em um objeto.

```
class Visitor{

  constructor(name, age){
    this._name = name;
    this._age = age;
  }

  set name(name){
    this._name = name;
  }

  set age(age){
    this._age = age
  }

  get name(){
    return this._name;
  }

  get age(){
    return this._age;
  }
}

const visitor = new Visitor('Zé', 0);
const handler = {
  set(target, prop, value, receiver){
    return prop === 'age' && isNaN(value) ? 0 : Reflect.set(target, prop, value, receiver);
  }
};

const proxyVisitor = new Proxy(visitor, handler);
proxyVisitor.age = 'a'
console.log(proxyVisitor);
```

O código acima configura um proxy para o o objeto **visitor**, esse proxy irá atuar da seguinte maneira:

Toda vez que for executado uma operação **set** no objeto **visitor** uma validação “not a number” será executada especificamente para a propriedade **age**, ou seja, se ocorrer isso:

```
proxyVisitor.age = 'a'
```

Nenhum valor será atribuído a variável **age** caso contrário se for atribuído um valor numérico para variável **age** esse valor será atribuído normalmente.

Considerações na utilização de Proxy

- No exemplo acima foi utilizado a operação **set** como “armadilha” para executar o proxy, no entanto você pode utilizar outras operações por exemplo: get, has, apply, construct, deleteProperty entre outras. Você pode verificar a lista completa de operações “armadilha” [aqui](#).
- Use reflection ao invés de Object. Existem situações que o Reflect retorna valores mais úteis que o Object e a consequência disso é que nos permite escrever um código mais sucinto.

Existem outras features legais que foram adicionados ao ES6, não iremos abordá-las aqui, você pode conferir a especificação completa do ES6 [aqui](#).

8.3. ECMA Script 2016 – ES7

Abaixo algumas features interessantes que foram introduzidas no ES7.

Exponentiation Operator (**)

```
console.log(4 ** 3)
```

Includes

```
console.log(numbers.includes(3))
```

8.4. ECMA Script 2017 – ES8

Abaixo algumas features interessantes que foram introduzidas no ES7.

Async/Await

No ES8 foi introduzido soluções para tratamento assíncrono, vimos anteriormente que no ES6 já foi introduzido Promises, porém, no ES8 é adicionado o mecanismo de async/await que também nos permite lidar com comportamento assíncronos de forma mais declarativa e com a ajuda de Promises. Observe o seguinte exemplo:

```
async function doSomethingAsync(){
  return await invoke();
}

function invoke(){
  return new Promise((resolve, reject) => {
```

```

        setTimeout((done) => {
            done === true ? resolve({status: 'done'}) : reject('Error')
        }, 1000, true)
    })
}

doSomethingAsync()
    .then(result => console.log(result))
    .catch(err => console.log(err))

```

8.5. ECMA Script 2018 – ES9

Async Interaction

Anteriormente vimos a configuração do Iterable no estilo ES6, no entanto esse Iterable é síncrono. No ES9 é possível configurar Iterables cujo a fonte de dados é assíncrona, observe o código abaixo:

```

const myIterable = {
    [Symbol.asyncIterator]() {
        let i = 0
        return {
            next() {
                if(i < 10) {
                    return Promise.resolve({done: false, value: i++})
                }

                return Promise.resolve({done: true})
            }
        }
    }
};

(async function () {
    for await(let item of myIterable){
        console.log(item)
    }
})();

```

8.6. ECMA Script 2019 – ES10

Método Flat

Cria-se um novo vetor recursivo a partir dos elementos de um sub-vetor até a profundidade especificada.

```
let flatProfundidade1 = [0, 1, 2, 3, [4, 5, 6, [7, 8, 9, [10, 11]]]];
let flatProfundidade2 = [0, 1, 2, 3, [4, 5, 6, [7, 8, 9, [10, 11]]]];
let flatProfundidadeInfinity = [0, 1, 2, 3, [4, 5, 6, [7, 8, 9, [10, 11]]]];
console.log(flatProfundidade1.flat()); //profundidade 1 [ 1, 2, 3, 4, 5, 6, [ 7, 8, 9, [ 10, 11, 12 ] ] ]
console.log(flatProfundidade2.flat().flat()) //profundidade 2
console.log(flatProfundidadeInfinity.flat(Infinity)) //Infinity
```

Método FlatMap

Dado uma função, mapeia cada elemento do array e ajusta o resultado usando o método flat de profundidade 1.

```
let myArrayFlatMap = [0, 1, 2, 3, [4, 5, 6, [7, 8, 9, [10, 11]]]];
console.log(myArrayFlatMap.flat(Infinity).flatMap(item => item * 2)); // Flat Map
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22 ]
```

Object.fromEntries

Transforma uma lista de chave/valor de objetos em objeto.

```
const data = { joao: '123456', maria: '654321', eduarda: '98765' }
const entries = Object.entries(data)
const myObject = Object.fromEntries(entries)
console.log(entries) // [ [ 'joao', '123456' ], [ 'maria', '654321' ], [ 'eduarda', '98765' ] ]
console.log(myObject) // { joao: '123456', maria: '654321', eduarda: '98765' }
```

Array.sort

Agora o sort é uma função [estável](#), ou seja, objetos que possuem a mesma chave permanece na mesma ordem mesmo que estejam ordenados ou não ordenados.

```
const myObject = [
  { name: 'joão', idade: 18 },
  { name: 'eduarda', idade: 6 },
  { name: 'maria', idade: 43 },
  { name: 'roberta', idade: 18 },
  { name: 'patricia', idade: 77 }
]

const sortFn = (a, b) => a.idade - b.idade
console.log(myObject)
console.log(myObject.sort(sortFn))
```

Bom, esse foi um breve resumo do que tem acontecido com as versões de javascript, lembrando que não foram apresentadas todas as features de cada versão a idéia foi de apresentar um resumo das features mais relevantes. A recomendação é, sempre que for utilizar alguma feature

verifique se a engine javascript ou o navegador que vai executar seu código suporta, para obter essa informação consulte a documentação oficial de sua engine Javascript ou do navegador. Na web existe uma tabela detalhada das features do ES e seu suporte em cada navegador/engine, você pode consultá-la [aqui](#).

9. REFERÊNCIAS

ECMAScript. Disponível em < <http://www.ecma-international.org/>>. Acesso em 14 Jan. 2020.

Mozilla Developer. Disponível em < <https://developer.mozilla.org> />. Acesso em 05 Mar. 2019.

Tableless. Disponível em <<https://tableless.com.br>>. Acesso em 05 Mar. 2019.

Caniuse. Disponível em <<https://caniuse.com> >. Acesso em 03 Abr. 2019.