Fundamentos

Bootcamp de Desenvolvimento Front End

Danilo Ferreira e Silva

2021

Fundamentos

Bootcamp de Desenvolvimento Front End

Danilo Ferreira e Silva

Sumário

Capítulo 1. Preparação

Este capítulo dá uma introdução ao módulo e detalha as ferramentas necessárias para acompanhar o curso. É importante que todas elas sejam corretamente instaladas e configuradas, tanto para acompanhar as aulas, quanto para desenvolver os trabalhos práticos.

Visão geral do módulo

Aplicações web são acessadas pelos usuários por meio do navegador, que oferece uma pilha de tecnologias que permitem a exibição de conteúdo das mais variadas formas, bem como a implementação de interatividade por meio de programação. Este módulo destina-se a estudar os três pilares desta pilha tecnológica: HTML, CSS e JavaScript. Ainda que se utilize frameworks modernos de desenvolvimento *front end* (como React, Angular ou Vue), é impossível estar totalmente alheio a estas três tecnologias. Sem compreendê-las, não é possível compreender o funcionamento de uma aplicação web, tampouco *frameworks* mais sofisticados. Daí a importância de estudá-las.

Ao longo deste módulo, e também do curso, a abordagem será bastante prática, portanto, é importante preparar o ferramental necessário. Para começar a desenvolver, precisaremos de três elementos: (i) um servidor web para hospedar nossa aplicação, (ii) um navegador para testar a aplicação e (iii) um editor de código fonte capaz de lidar com HTML, CSS e JavaScript. Utilizaremos como navegador o **Google Chrome**, o qual assumiremos que já esteja instalado. Os outros dois elementos (servidor web e editor de código) serão discutidos nas próximas seções, incluindo orientações para instalação.

Instalação do Node.js e live-server

Para cumprir o papel de servidor web, utilizaremos o Node.js em conjunto com o pacote *live-server*. Node.js é uma plataforma de desenvolvimento em

JavaScript que se tornou muito popular para o desenvolvimento web *full stack* nos últimos anos.

Para instalá-lo, baixe o instalador da versão estável (LTS) disponível no site, e execute a instalação até o final, conforme orientações da videoaula:

- https://nodejs.org/en/

Note que este curso foi produzido com a versão 14.15.5 do Node.js, mas não é um problema se você usar uma versão mais recente. Após a instalação, execute o comando `node -v` no prompt de comando para garantir que está tudo certo.

live-server

Após a instalação do Node.js, instalaremos o pacote *live-server*, que implementa um servidor web simples, que serve para arquivos estáticos de um diretório especificado. Isso será suficiente para servirmos HTML, CSS e JavaScript.

Para instalar o live-server, usaremos o *npm*, o gerenciador de pacotes do Node.js. Execute o comando abaixo no prompt de comando:

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

Feito isso, o executável live-server estará instalado globalmente e disponível no prompt de comando. Por exemplo, supondo que você execute:

```
C:\meu\diretorio>live-server
```

O servidor iniciará no diretório corrente, servindo os arquivos presentes no mesmo ou em subdiretórios. Além disso, ao fazer alterações nos arquivos, a página aberta no navegador se recarrega automaticamente, o que ajuda a tornar o desenvolvimento mais fluido (esse recurso é conhecido como *live reload*). Mais detalhes sobre tal pacote estão disponíveis em:

- https://www.npmjs.com/package/live-server

Instalação e configuração do VSCode

Como editor de código, utilizaremos o VSCode, IDE desenvolvida pela Microsoft com excelente suporte para HTML, CSS e JavaScript. Baixe o instalador do site oficial:

- https://code.visualstudio.com/

A instalação pode ser feita via executável, seguindo os passos com o valor padrão, conforme detalhado em videoaula. Uma outra possibilidade é fazer a instalação portável, via opção *Other Downloads*. Após descompactar o zip, é necessário criar a pasta `data` dentro do diretório de instalação para ativar o modo portável. Após instalar o VSCode, adicionaremos duas extensões:

- Debugger for Chrome: adiciona a capacidade de "debugar" o código JavaScript de uma página que está sendo exibida no Google Chrome.
- Prettier: Adiciona a capacidade de formatar automaticamente código HTML, CSS e JavaScript.

O VS possui uma vasta gama de preferências que podem ser ajustadas. Como sugestão, as seguintes configurações podem ser definidas em *File* > *Preferences* > *Settings*:

```
{
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "editor.tabSize": 2,
  "editor.detectIndentation": false,
  "editor.renderWhitespace": "all",
  "workbench.editor.enablePreview": false,
  "editor.formatOnSave": true
}
```

Dentre essas configurações, a mais importante está na primeira linha, definindo o *Prettier* como formatador padrão. As demais configurações são preferências pessoais. Consulte a videoaula para mais detalhes e explicações.

Criando um primeiro projeto

Para criar um primeiro projeto “Hello world”, siga os passos:

1. Crie um diretório chamado `hello`.
2. Abra o diretório no VSCode em *File* > *Open Folder*.
3. Crie um arquivo `index.html` com o conteúdo abaixo.

```
<!DOCTYPE html>
<body>
  <h1>Hello World!</h1>
</body>
```

Para executar o projeto, basta executar o comando *live-server* na pasta `hello`. Com isso, o navegador será aberto automaticamente, e recarregará a página a cada alteração em arquivos do projeto.

Capítulo 2. HTML

Hypertext Markup Language, ou HTML, é a linguagem para definição do conteúdo de uma página web. Um dos conceitos básicos que o HTML possibilita é o hipertexto, ou seja, texto não linear, com diversos conteúdos interconectados via links.

Estrutura básica e sintaxe

Um documento HTML possui a seguinte estrutura básica:

```
<!DOCTYPE html>
<html lang="pt-BR">
  <head>
    <title>Meu documento</title>
  </head>
  <body>
    <p>Este é um documento HTML.</p>
  </body>
</html>
```

Conforme exemplo acima, o documento deve iniciar com a marcação `<!DOCTYPE html>`, para que seja interpretado corretamente pelo navegador como HTML 5. Abaixo, normalmente temos o elemento *html*, que engloba toda o documento, bem como os elementos *head* e *body*. Enquanto o *head* contém metadados da página, *body* contém o conteúdo em si.

Elementos em HTML são declarados por *tags* de abertura (`<html>`) e de fechamento `</html>`, que englobam outros elementos ou texto. Alguns elementos não podem ter um conteúdo, portanto, não possuem tag de fechamento (por exemplo a tag `<img>`). Além disso, elementos podem ter atributos e seus respectivos valores, por exemplo `lang="pt-BR"`. Nomes de elementos e de atributos não são sensíveis a caixa, mas valores de atributos são.

Dois dos atributos mais comuns, que podem ser usados qualquer elemento HTML, são *id* e *class*. O tributo *id* permite atribuir um identificador a um elemento,

facilitando que ele seja referenciado em JavaScript ou outros elementos. Já o atributo *class* permite estilizar um elemento mais facilmente, conforme veremos no capítulo seguinte.

Nas seções seguintes, veremos exemplos de vários elementos HTML comumente utilizados em páginas/aplicações web.

Texto, imagens e links

Os seguintes elementos podem ser usados para definir conteúdo textual básico em um documento.

h1-h6	Indicam até 6 níveis de títulos na página. Ou seja, *h1* é um Título principal (nível 1).
p	Parágrafo.
em	Enfatiza uma palavra ou trecho do texto (normalmente exibido em itálico).
strong	Enfatiza fortemente uma palavra ou trecho do texto (normalmente exibido em negrito).
code	Demarca código fonte.
kbd	Demarca entrada de teclado.
br	Introduz uma quebra de linha. Note que em HTML quebras de linha no código fonte não se refletem em quebras de linha no texto exibido.
sub	Subscrito, como em H_2O.
sup	Sobrescrito, como em x^2.

img	Uma imagem. O arquivo da imagem a ser exibida é especificada pelo atributo *src*, que pode ser um endereço absoluto ou relativo.
a	Um link para outra página (ou arquivo). O endereço é especificado pelo atributo *href*.

Outros Elementos

Listas e tabelas

Os seguintes elementos permitem a exibição de dados em listas.

ul	Define uma lista não ordenada (bullets).
ol	Define uma lista ordenada (com numeração).
li	Um item da lista. Deve estar dentro de ul ou ol.
dl	Define uma lista de definições.
dt	Um termo da lista de definições. Deve estar dentro de dl.
dd	A descrição de um termo da lista de definições. Deve vir após um dt.

Os seguintes elementos permitem a exibição de dados em tabelas.

table	Define uma tabela.
thead	Demarca o cabeçalho da tabela.
tbody	Demarca o corpo da tabela.
tr	Define uma linha da tabela.
td	Define uma célula da tabela. Deve estar dentro de tr.
th	Define uma célula de cabeçalho da tabela. Deve estar dentro de tr.

Formulários

Os seguintes elementos são usados para construir formulários, que recebem entrada do usuário.

form	Engloba o formulário.
input	Um campo de entrada. O atributo type define o tipo de campo, que pode ser textual, numérico, de data, checkbox, radio, entre outros.
label	Provê um rótulo para um campo.
select	Define um campo de seleção de opções fechadas. Pode ser de seleção múltipla se existir o atributo multiple.
option	Define uma opção. Deve estar dentro de select.
textarea	Define uma área de texto, ou seja, um campo com várias linhas de texto.

Contêineres

Muitas vezes precisamos agrupar conteúdo relacionado em um documento para fins de estruturação ou estilização, ou mesmo demarcar uma região da página com alguma semântica específica (um menu de navegação, o conteúdo principal, etc.). Para isso, temos dois agrupadores genéricos, span e div, bem como contêineres semânticos.

span	Agrupa um conjunto de elementos relacionados dentro de uma linha.
div	Agrupa elementos relacionados como um bloco (força uma quebra de linha).
header	Define uma região como cabeçalho da página.
footer	Define uma região como rodapé da página.

nav	Define um menu de navegação, contendo vários links.
main	Demarca o conteúdo principal da página.
aside	Demarca um conteúdo separado do conteúdo principal.
article	Demarca um artigo, ou seja, um conteúdo autocontido que possa ser lido fora do contexto da página.
section	Demarca uma seção.

Existem muitos outros elementos HTML. Neste capítulo apenas exemplificamos alguns dos mais comuns. Para uma lista completa consulte o artigo da MDN Web Docs:

- https://developer.mozilla.org/en-US/docs/Web/HTML/Element

Capítulo 3. CSS

Cascading Style Sheets, ou apenas CSS, é uma linguagem para definir estilização de documentos HTML. Enquanto o HTML lida com estrutura e conteúdo, CSS lida com cores, posicionamento, estilos de fonte, e qualquer outro aspecto visual da página.

Introdução e sintaxe básica

Incluindo CSS no documento

O CSS pode ser incluído no documento HTML de duas formas, como arquivo externo por meio do elemento link:

```
<link rel="stylesheet" href="styles.css">
```

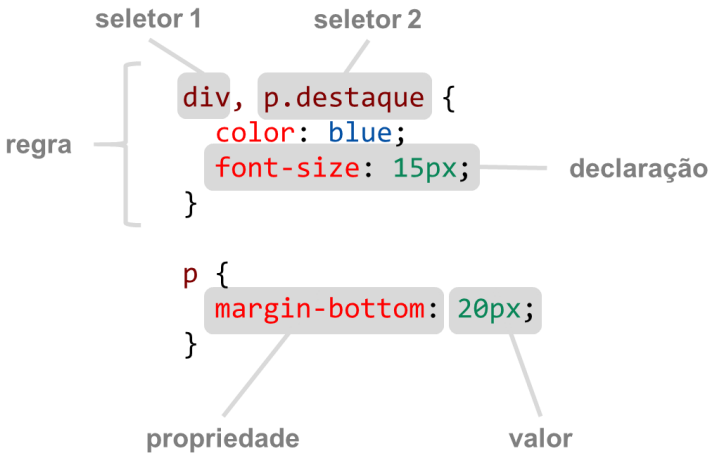
ou como estilos *inline* na página:

```
<style>
  div {
    color: blue;
  }
</style>
```

Por fim, podemos definir estilos diretamente em Sintaxe Básica.

Sintaxe básica

A sintaxe básica do CSS segue o seguinte padrão:



Regras CSS são compostas por *seletores*, que especificam para quais elementos elas se aplicam, e um bloco de declarações de propriedades e valores.

Propriedades CSS

Propriedades CSS podem ser herdadas de elementos pai para seus filhos, ou não. Por exemplo, ao definir a cor da fonte (*color*) em *body*, os elementos filhos herdam a mesma cor. Por outro lado a largura de um elemento (propriedade *width*) não é herdada.

Propriedades herdadas	Propriedades não herdadas
color font-size font-family ...	margin width height border background-color ...

Seletores CSS

Podemos usar diferentes tipos de seletores e combinadores em CSS. Abaixo temos uma tabela com os tipos mais comuns.

E	Elemento do tipo **E**.
*	Todo elemento.
.classe	Elemento com a classe de nome **classe**.

#id	Elemento com atributo *id* igual a **id**.
E F	Elemento **F** descendente de **E**.
E > F	Elemento **F** filho de **E**.
E ~ F	Elemento **F** é irmão de **E** e precedido por **E**.
E + F	Elemento **F** irmão de **E** e *imediatamente* precedido por **E**.
E[atrib="x"]	Elemento **E** com atributo **atrib** de valor **x**.

Além de seletores por classes, temos também as pseudoclasses, que indicam características estruturais ou estado do elemento.

:disabled	Elemento de interface que esteja desabilitado.
:hover	Elemento cujo mouse está sobre.
:focus	Elemento que possui foco do cursor do teclado.
:first-child	Elemento que é o primeiro filho de seu pai.
:last-child	Elemento que é o último filho de seu pai.
:empty	Elemento que não possui filho.
:not(E)	Elemento *não* corresponde ao seletor **E**

Vale ressaltar que os diferentes tipos de seletores podem ser combinados em uma única expressão, por exemplo `a.destaque.grande:disabled` seleciona elementos A, com as classes *destaque* e *grande*, que está desabilitado.

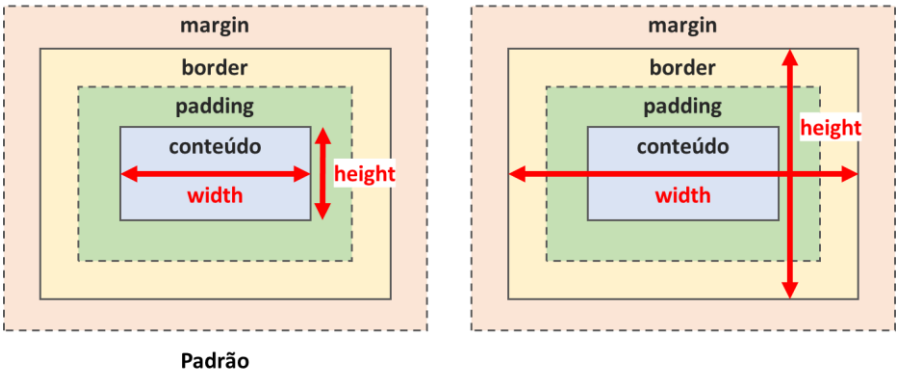
Dimensionamento e posicionamento de elementos

Box model

Elementos CSS são dimensionados seguindo o box model, que é representado pela figura a seguir.

```
div {
  box-sizing: content-box;
}
```

```
div {
  box-sizing: border-box;
}
```



Padrão

O espaço ocupado por um elemento é composto por seu conteúdo, o *padding* (distância do conteúdo para a borda), espessura da borda e por fim de sua margem. No formato *box-sizing: content-box* (comportamento padrão), a largura (propriedade **width**) e altura (propriedade **height**) consideram apenas a região de conteúdo. Por outro lado, no formato *box-sizing: border-box*, **width** e **height** consideram até a extremidade externa da borda.

As propriedades width, height, padding, margin, e várias outras, podem ser especificadas com diferentes unidades de medidas. Alguns exemplos são dados na tabela a seguir.

px	Um ponto da tela.
cm, mm, in	Centímetros, milímetros, polegadas, etc. (fazem mais sentido para impressão)
em	Tamanho da fonte corrente.
rem	Tamanho da fonte do elemento raiz da página.
vh	1% da altura do *viewport*.
vw	1% da largura do *viewport*.
%	Relativo a outra medida (normalmente do elemento pai).

Overflow

O comportamento quando um conteúdo não cabe dentro de um elemento é determinado pela propriedade **overflow**, que pode ter os valores:

- **visible** (padrão): conteúdo pode ser renderizado fora do box.
- **hidden**: conteúdo é cortado.
- **scroll**: exibe scroll bar sempre.
- **auto**: exibe scroll bar, se precisar.

Posicionamento padrão (normal flow)

Por padrão, os elementos são dispostos na página em linha (ao lado do anterior), ou em blocos (abaixo do anterior), dependendo do tipo de elemento. Por exemplo, *h1*, *div* e *p* são **block boxes**, enquanto *span*, *img* e *a* são **inline boxes**.

Podemos controlar o alinhamento horizontal de elementos dentro de uma linha com a propriedade **text-align**, cujos valores podem ser (left, right, center, justify, etc.). Já o alinhamento vertical dos elementos dentro de uma linha é controlado pela propriedade **vertical-align**, cujos valores podem ser (baseline, middle, top, bottom, etc.).

Posicionamento fora do normal flow

Usando a propriedade **position** podemos alterar o *normal flow*. O valor padrão dessa propriedade é *static*, mas podemos alterá-la para:

- **relative**: Posicionado de acordo com *normal flow*, mas deslocado de sua posição normal via propriedades top, right, bottom e left.

- **absolute**: Posicionado fora do normal flow (não ocupa espaço). Posição definida via propriedades top, right, bottom e left, relativas ao primeiro elemento pai posicionado (position ≠ static).

- **fixed**: Posicionado fora do normal flow (não ocupa espaço). Posição definida via propriedades top, right, bottom e left, relativas ao viewport (não faz scroll).

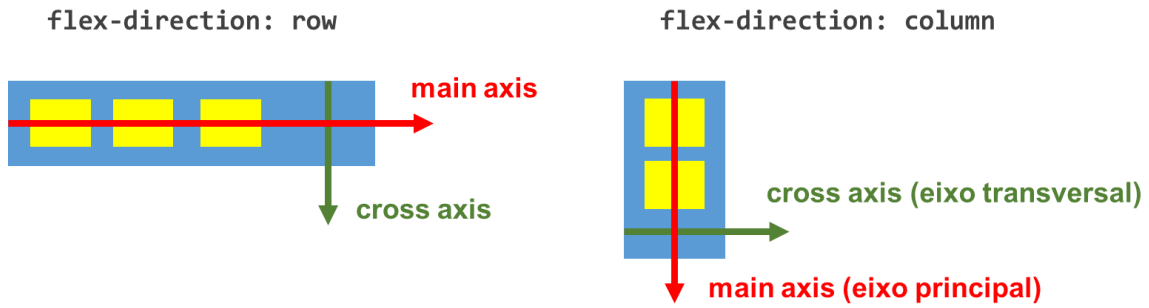
Flexbox layout

O modelo de layout *flexbox* foi introduzido nas especificações mais recentes do CSS e oferece grande poder e versatilidade para posicionar elementos. Para ativá-lo, definimos a propriedade **display** como **flex** ou **inline-flex**:

- **flex**: comporta como *block box* externamente, e *flexbox layout* internamente;

- **inline-flex**: comporta como *inline box* externamente, e *flexbox layout* internamente.

A direção dos filhos de um contêiner *flexbox* é definida pela propriedade **flex-direction**, que pode ter valor *row* ou *column*. No caso de *row*, o eixo principal é horizontal, ou seja, os elementos são dispostos ao lado dos outros. No caso de *column*, o eixo principal é vertical e os elementos são dispostos abaixo um do outro.



Além disso, é possível controlar várias propriedades para determinar os alinhamentos dos filhos.

- Alinhamento no eixo principal (justify-content);

- Alinhamento no eixo transversal (align-items, align-self);
- Tamanho de cada item (flex, flex-grow, flex-shrink, flex-basis);
- Quebras de linha/colunas (flex-wrap).

Capítulo 4. JavaScript básico

Introdução

JavaScript é uma linguagem dinâmica e interpretada, que pode ser executada no navegador. Essa linguagem surgiu com o antigo navegador Netscape, com o intuito de permitir interações simples. Porém, JavaScript passou por um processo de padronização e evolução e hoje é amplamente utilizada para desenvolvimento de aplicações, tanto no *front end*, quanto no *back end*, em plataformas como o Node.js.

Neste capítulo estudaremos conceitos básicos da linguagem, voltados principalmente para quem não conhece JavaScript.

Adicionando scripts na página

Para incluir JavaScript em um documento HTML, utilizamos o elemento *script*, que pode ser usado de duas formas: (i) script externo, cujo caminho é especificado por meio da propriedade *src* ou (ii) script inline, cujo código é incluído diretamente como conteúdo do elemento *script*, conforme exemplo abaixo.

```
<script src="./js/meu-script-externo.js"></script>

<script>
  console.log('Este é um script inline');
</script>
```

Sintaxe básica

Abaixo temos um primeiro exemplo de código JavaScript, que demonstra algumas características básicas da sintaxe.

```
var mensagem = 'Olá';
mensagem = 'Olá Danilo';

console.log(mensagem); // comentário
// comentário de uma linha
/*
comentário de várias linhas
```

```
console.log(mensagem);
*/
```

Note que a primeira linha declara uma variável e a inicializa, enquanto na segunda linha atribuímos um novo valor à variável `mensagem`. Em seguida, imprimimos o valor da variável no console, chamando a função `console.log`.

Tipos e valores

Tipos primitivos

JavaScript possui os tipos primitivos number (número inteiro ou fracionário), string (sequência de caracteres) e boolean (verdadeiro ou falso). Valores primitivos são declarados como no código a seguir.

```
var vNumber = 5.78;
var vString = 'abacaxi';
var vBoolean = true;
```

Objetos

Podemos definir objetos contendo as propriedades que desejarmos, sejam elas primitivas ou não. Note que valores do tipo objeto são definidos entre chaves, conforme exemplo abaixo. Propriedades também podem ser adicionadas ou removidas de um objeto a qualquer momento.

```
var aluno1 = {
  matricula: 7627364,
  nome: 'Danilo Ferreira',
  curso: 'Bootcamp Front End',
  ativo: true
};
```

Arrays

Arrays armazenam uma lista de valores, primitivos ou não. Valores do tipo array são definidos usando colchetes. Podemos adicionar elementos em um array existente com a função *push,* e também obter seu tamanho com a propriedade *length*.

```
var frutas = ['Banana', 'Laranja', 'Maçã'];
frutas.push('Melancia');
```

```
console.log(frutas.length); // imprime 4
```

Undefined e null

Variáveis não inicializadas possuem o valor especial *undefined*. Vale ressaltar que JavaScript também possui o valor *null*, que representa uma referência para nenhum objeto, comum em outras linguagens de programação. No entanto *null* e undefined, embora sejam conceitualmente semelhantes, não são exatamente o mesmo valor.

Operadores e expressões

JavaScript fornece um conjunto de operadores, que podem ser utilizados para montar expressões lógicas, aritméticas, entre outras. Nesta seção veremos alguns dos operadores mais comuns.

Operadores lógicos

!x	Negação
x && y	E lógico
x || y	OU lógico

Operadores aritméticos

x + y	Adição, ou concatenação, no caso de *strings*
x - y	Subtração
+x	Converte para número (se já não for)
-x	Inversão de sinal
x / y	Divisão
x * y	Multiplicação
x % y	Resto da divisão de x por y
x ** y	Exponenciação

Operadores aritméticos com atribuição

`x += y`	Adição, ou concatenação, no caso de *strings*
`x -= y`	Subtração
`x /= y`	Divisão
`x *= y`	Multiplicação
`x %= y`	Resto da divisão de x por y
`x **= y`	Exponenciação

Operadores aritméticos com atribuição

`x++`	Retorna x e depois incrementa
`x--`	Retorna x e depois decrementa
`++x`	Incrementa e depois retorna x
`--x`	Decrementa e depois retorna x

Operadores de comparação

`x < y`	Menor
`x > y`	Maior
`x <= y`	Menor ou igual
`x >= y`	Maior ou igual

Operadores de igualdade

`x == y`	Igual
`x != y`	Diferente
`x === y`	Exatamente igual (mesmo tipo)
`x !== y`	Não exatamente igual

Precedência de operadores

Ao avaliar expressões com vários operadores, serão avaliados primeiro aqueles com maior precedência, conforme tabela abaixo (da maior precedência para a menor). No entanto, podemos modificar a ordem de avaliação com o uso de parênteses.

Operadores	Associatividade
`f(x)  x.y  x[y]`	→
`!x  +x  -x`	←
`x**y`	←
`x * y  x / y  x % y`	→
`x + y  x - y`	→
`x < y  x <= y  x > y  x >= y`	→
`x == y  x != y  x === y  x !== y`	→
`x && y`	→
`x || y`	→

Funções e escopo

Podemos definir funções com a palavra-chave *function*, conforme exemplo a seguir. Variáveis definidas dentro da função possuem escopo local (apenas na função), enquanto variáveis declaradas fora da função possuem escopo global, podendo ser acessadas inclusive em outros scripts.

```
var m = 'Olá ';

function imprimeOla() {
  var nome = 'Danilo'; // variável local
  console.log(m); // pode ler variável global
  console.log(nome);
}
```

Vale ressaltar que funções podem receber parâmetros e retornar valores, como a função `soma` definida a seguir.

```
function soma(a, b) {
  return a + b;
}
```

Comandos de decisão

if/else

Executa um bloco de código condicionalmente, baseado em uma expressão de decisão. Opcionalmente pode ter a cláusula *else*.

```
function maior(a, b) {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

Operador ternário

Avalia uma expressão condicionalmente, baseado em uma expressão de decisão. Tem o mesmo comportamento do comando if/else, mas trata-se de uma expressão, não um comando.

```
function maior(a, b) {
  return (a > b) ? a : b;
}
```

switch/case

Executa código condicionalmente, baseado na comparação de uma expressão de controle com um conjunto de valores possíveis. Pode ter uma cláusula *default* para capturar outros casos não tratados.

```
function formataUnidade(un) {
  switch (un) {
    case "s":
      return "segundo";
    case "m":
```

```
      return "metro";
    default:
      console.log("Não implementei ainda");
  }
}
```

Comandos de repetição

while

Verifica uma condição de controle e executa o bloco de código enquanto ela for verdadeira.

```
var contador1 = 1;
while (contador1 <= 5) {
  console.log(contador1); // imprime de 1 a 5
  contador1++;
}
```

do while

Executa o bloco de código e verifica uma condição de controle. Se for verdadeira, repete.

```
var contador2 = 1;
do {
  console.log(contador2); // imprime de 1 a 5
  contador2++;
} while (contador2 <= 5);
```

for

Executa comandos de inicialização e em seguida verifica uma condição de controle e executa o bloco de código enquanto ela for verdadeira. Ao final de cada repetição, executa um comando fornecido (por exemplo, incrementar um contador).

```
for (var contador3 = 1; contador3 <= 5; contador3++) {
  console.log(contador3); // imprime de 1 a 5
}
```

Capítulo 5. Interação com o DOM

Por meio de JavaScript é possível interagir com os elementos da página, utilizado a API conhecida como DOM (Document Object Model). O DOM nada mais é do que uma estrutura de dados que representa todos os elementos da página como objetos que podem ser acessados e modificados via JavaScript. Entre outras tarefas, podemos modificar o conteúdo da página e reagir a eventos.

Interação básica

O primeiro passo para interagir com o DOM é obter uma referência para um elemento de interesse, o que normalmente é feito com uma das seguintes funções:

```
// obtém o elemento pelo id
var el1 = document.getElementById("idElemento");

// obtém o primeiro elemento capturado pelo seletor
var el2 = document.querySelector("div.destaque");

// obtém lista de todos os elementos capturados pelo seletor
var elList = document.querySelectorAll("li.selecionado");
```

Adicionalmente, podemos navegar nos elementos e seus filhos a partir da raiz. Por exemplo, `document.body.children[0]` obtém o primeiro filho do elemento *body*.

Com uma referência de elemento em mãos, é possível alterar propriedades. No exemplo a seguir alteramos o conteúdo textual do elemento.

```
var el1 = document.getElementById("idElemento");
el1.textContent = "novo texto"; // muda o conteúdo de el1
```

Por fim, podemos reagir a eventos na página. Uma forma de fazer isso é utilizando atributos no HTML, como no código a seguir, onde a função `soma` é chamada ao clicar no botão.

```
<button onclick="soma()">Calcular</button>
```

Eventos

Além de reagir a eventos por meio do atributo *onclick* e outros, podemos registrar *listeners* de eventos via JavaScript.

```
var el = document.getElementById("mybtn");
// registra listener do evento click
el.addEventListener("click", myHandler);

// se precisar, remove o listener
el.removeEventoListener("click", myHandler)
```

Essa forma nos permite ter maior controle sobre o comportamento sobre como o *listener* de eventos será registrado. Também podemos desfazer a operação, se necessário.

Propagação de eventos

Ao ocorrer uma ação que dispara evento, inicialmente o evento se propaga, partindo da raiz até o elemento alvo da ação, executando qualquer *listener* de eventos que esteja registrado na fase de *capturing* em algum destes elementos. Em seguida, o evento se propaga do elemento alvo da ação até a raiz, executando qualquer *listener* de eventos que esteja registrado na fase de *bubbling*. A função `addEventListener` registra por padrão na fase de bubbling, mas muda para a fase de *capturing* se passarmos o valor `true` como terceiro parâmetro.

```
// registra na fase de capturing
el.addEventListener("click", myHandler, true);

 // registra na fase de bubbling
el.addEventListener("click", myHandler, false);
```

O objeto evento

Quando um listener de eventos é chamado, ele recebe como parâmetro um objeto que representa o evento e possui diversas informações sobre ele. No

exemplo abaixo obtemos as coordenadas do mouse no momento do *click*. Cada tipo de evento pode possuir propriedades específicas.

```
function myHandler(event) {
  console.log("clicou na posição " + event.pageX + ", " + event.pageY);
}
```

O objeto evento também nos permite chamar as funções `stopPropagation` e `preventDefault`. Enquanto a primeira interrompe a propagação do evento, a segunda impede o tratamento padrão do navegador (por exemplo, seguir um link ao clicar no mesmo).

Tipos de eventos usados frequentemente

`focus`	Elemento recebe foco.
`blur`	Elemento perde foco.
`input`	Valor de um elemento muda (input, select, textarea).
`submit`	Formulário submetido.
`keydown`	Tecla do teclado pressionada.
`keyup`	Tecla do teclado liberada.
`click`	Botão do mouse pressionado e liberado.
`mousemove`	Mouse movimentado sobre o elemento.

Criando elementos dinamicamente

Podemos utilizar as funções `document.createElement` e `elemento.appendChild` para criar elementos dinamicamente e inseri-los no DOM. Também podemos inserir elementos antes de outro elemento informado usando `elemento.insertBefore`. Por fim, podemos remover elementos do DOM com a função `elemento.remove`.

```
// Cria um elemento
var myDiv = document.createElement("div");
// Insere o elemento na página, no final do body
```

```
document.body.appendChild(myDiv);

// Cria outro elemento
var myH2 = document.createElement("h2");
myH2.textContent = "Dynamic H2";
// Insere ele antes do div
document.body.insertBefore(myH2, myDiv);

// Remove da página
myDiv.remove();
```

Uma outra maneira de criar elementos dinamicamente é por meio da propriedade *innerHTML*, que recebe uma *string* com o código HTML a ser avaliado como conteúdo do elemento. Note que essa abordagem pode dar margem a ataques conhecidos como *HTML Injection* se os devidos cuidados não forem tomados, pois a string passada é interpretada como HTML.

Alterando estilos de elementos

Podemos alterar os estilos de um elemento via propriedade *style* (que é o correspondente no DOM ao atributo *style* do elemento). Isso pode ser feito com todo o *style*, ou alterando propriedades CSS individuais.

```
// Define style como string
el.style = "color: red; margin: 4px";

// Define propriedade individuais do style
el.style.color = "red";
```

Além disso, é possível adicionar ou remover classes em elementos dinamicamente.

```
// Define className (que corresponde ao atributo class)
el.className = "classe1 classe2";

// Adiciona ou remove classes individualmente
el.classList.add("classe1");
el.classList.remove("classe1");
```

Capítulo 6. Orientação a objetos em JavaScript

JavaScript possui mecanismos que possibilitam a programação orientada a objetos, porém de forma um pouco diferente do que linguagens como Java ou C#. Neste capítulo exploraremos as opções oferecidas pela linguagem.

Instanciando objetos

Em JavaScript, qualquer função pode ser usada para instanciar um objeto usando a palavra-chave **new**.

```
function Retangulo(altura, largura) {
  this.altura = altura;
  this.largura = largura;
  this.area = function () {
    return this.altura * this.largura;
  };
}

var r1 = new Retangulo(3, 4);
```

No exemplo acima, a variável r1 referencia uma instância de objeto criada pela função Retangulo, que foi usada como um construtor. Nesta função adicionamos as propriedades altura, largura e área (que é uma função). Note que a palavra-chave **this** corresponde ao objeto recém-instanciado.

Prototype chain

Internamente, acessos a propriedades em objetos JavaScript utilizam o mecanismo de *Prototype chain*. Todo objeto possui uma referência interna a outro objeto que é seu *prototype*. Ao buscar uma propriedade busca-se no objeto atual, depois em seu *prototype*, depois no *prototype* do *prototype*, até que não exista um *prototype*.

Objetos instanciados via **new** possuem *prototype* que estão ligados à sua função construtora. No exemplo abaixo adicionamos a função `area` ao *prototype* da função `RetanguloV2`, o que fará com que os objetos instanciados por ela herdem

essa função via *prototype chain*. Isso evita que criemos uma cópia da função para cada objeto instanciado, reduzindo o consumo de memória.

```
function RetanguloV2(altura, largura) {
  this.altura = altura;
  this.largura = largura;
}
RetanguloV2.prototype.area = function () {
  return this.altura * this.largura;
};
```

Classes e herança

A partir da edição 6 da especificação do ECMAScript (padrão adotado pela linguagem JavaScript), foi introduzida a sintaxe para declaração de classes, facilitando a programação orientada a objetos. Internamente, classes são baseadas no mecanismo de *prototype chain* já existente, mas permitem um código mais claro. Com esse recurso, o exemplo anterior ficaria da seguinte forma:

```
class Retangulo {
  constructor(altura, largura) {
    this.altura = altura;
    this.largura = largura;
  }
  area() {
    return this.altura * this.largura;
  }
}
```

Além disso, podemos utilizar herança para herdar membros de uma classe base. Por exemplo, no caso abaixo, Quadrado herda os membros de Retangulo com a opção de sobrescrevê-los, se necessário.

```
class Quadrado extends Retangulo {
  constructor(dimensao) {
    super(dimensao, dimensao);
  }
}
```

Capítulo 7. JavaScript moderno

A linguagem JavaScript segue o padrão especificado pelo ECMA International conhecido como ECMAScript, que vem evoluindo ao longo dos últimos anos. A sexta edição da especificação ES6 (também conhecida como ES2015) trouxe inúmeras novidades para a linguagem. Neste capítulo falaremos de várias delas.

Let, const, desestruturação, spread, e template strings

Declarações de variáveis com let e const

Podemos usar a palavra-chave **let** para declarar variáveis e **const** para declarar constantes, que não podem ser atribuídas após a inicialização. A única diferença entre **var** e **let** é que enquanto o escopo da primeira é todo o corpo da função onde foi declarada, o escopo da segunda é apenas o bloco onde foi declarada. Com isso, **let** se comporta de forma mais segura, evitando situações que podem levar a bugs no código.

```
let a = 1;
const b = 1;
a = 2;
// b = 2; // erro, não pode alterar const
```

Atribuição via desestruturação

Com essa nova sintaxe, é possível automaticamente atribuir variáveis desmembrando elementos de um *array*, ou propriedades de um objeto.

```
let primos = [2, 3, 5, 7, 11, 13];
// Desestruturação de array
let [p1, p2, ...resto] = primos;
// Corresponde a let p1 = primos[0], p2 = primos[1]

let curso = {nome: "Bootcamp Front End", presencial: false, turma: 1};
// Desestruturação de objeto
let { nome, turma, ...outrosCampos } = curso;
// Corresponde a let nome = curso.nome, turma = curso.turma
```

Spread operator

O *spread operator* nos permite expandir os elementos de um *array*, ou propriedades de um objeto, ao criar um novo *array* ou objeto, como demonstrado abaixo.

```
let primos = [2, 3, 5, 7, 11, 13];
let primos2 = [...primos, 17];
// Equivalente a
// let primos2 = [2, 3, 5, 7, 11, 13, 17]

let curso = {nome: "Bootcamp Front End", presencial: false, turma: 1};
let curso2 = {...curso, descricao: "Bla bla"}
// Equivalente a
// let curso2 = {nome: "Bootcamp Front End", presencial: false, turma: 1, desc
ricao: "Bla bla"}
```

Template strings

Template string (ou *template literal*) oferece uma sintaxe conveniente para montar uma string intercalada com variáveis, evitando uma sequência de operações de concatenação. Tais *templates* são delimitados pelo caractere crase e podem interpolar expressões com a sintaxe `${}`. Outra diferença com relação a *strings* literais é que elas podem conter quebras de linha.

```
let a = 2, b = 3;
let soma = a + b;
// Template string
console.log(`${a} + ${b} = ${soma}`);
// Equivale a
console.log(a + " + " + b + " = " + soma);
```

Arrow functions

Com essa nova sintaxe, é possível declarar funções de forma mais compacta, especialmente em casos em que precisamos passar uma função como parâmetro, sem atribuir um nome a ela. A declaração é feita com a sintaxe abaixo.

```
// Função convencional
function soma(a, b) {
```

```
  return a + b;
}

// Equivalente a arrow function
const soma = (a, b) => {
  return a + b;
}

// Também equivalente a
const soma = (a, b) => a + b;
```

Vale ressaltar que *arrow functions* possuem um comportamento diferente com relação ao uso da palavra-chave **this** em seu corpo. Enquanto numa função convencional o **this** é sempre o objeto alvo da chamada (ou o objeto global, se a chamada não for a partir de um objeto), o **this** da *arrow function* é a mesma referência apontada pelo **this** da função na qual a *arrow function* foi declarada.

Manipulação de arrays

Iteração com for of

Com a sintaxe **for of**, podemos iterar em um *array*, percorrendo cada elemento dele, sem precisar lidar com contadores e condições da repetição, como demonstrado a seguir.

```
for (let item of array) {
  console.log(item);
}
```

Outra opção é utilizar a função *forEach*, que recebe uma função como parâmetro e a chama para cada item do *array*.

```
array.forEach((item, index) => {
  console.log(`${index}: ${item.name}`);
});
```

Funções map, filter e find

A função *map* transforma os elementos de um *array* com base em uma função de transformação recebida, devolvendo um novo *array* ao final do processo.

```
// Transforma array de objetos para array de strings
let names = usPresidents.map((item) => item.name);
```

A função *filter* filtra um *array* com base em uma função que é avaliada para cada elemento. Caso a função retorne *true*, o elemento é mantido, caso contrário o elemento é filtrado. Um novo array é devolvido ao final do processo.

```
// Obtém apenas presidentes do partido republicano
let republicans = usPresidents.filter((item) => item.party == "Republican");
```

A função *find* procura o primeiro elemento do *array* para o qual a função fornecida retorna *true*. O elemento encontrado é devolvido como resultado, ou *null* caso não encontre.

```
// Encontra primeiro presidente do partido democrata
let firstDemocratic = usPresidents.find((item) => item.party == "Democratic");
```

Função sort

Essa função ordena um *array* com base em uma função de comparação estipulada. Ela deve retornar um número negativo, zero ou positivo para indicar, respectivamente, se o primeiro parâmetro é menor, igual ou maior que o segundo.

```
usPresidents.sort((i1, i2) => {
  if (i1.birth_year < i2.birth_year) {
    return -1;
  } else if (i1.birth_year > i2.birth_year) {
    return 1;
  } else {
    return 0;
  }
});
```

Módulos

Quando uma aplicação fica muito grande, torna-se necessário modularizar, ou seja, dividir o código em diferentes arquivos. No entanto, traz ao menos duas dificuldades:

1. Dependências entre scripts são implícitas. É necessário incluir todas as dependências de um script na página, na ordem correta.
2. Não há encapsulamento adequado (tudo que é declarado no escopo global pode ser lido/escrito de qualquer script).

A partir da edição 6, a especificação de ECMAScript inclui o conceito de módulos. Tal recurso teve como objetivo resolver essas e outras questões. No exemplo abaixo, temos a declaração de um módulo `modulo_a.js` e a utilização do mesmo em outro módulo `modulo_b.js`. Para isso, precisamos exportar as declarações no primeiro, e importá-las no segundo. Qualquer declaração não exportada torna-se privada, visível apenas dentro do arquivo.

modulo_a.js

```
function fazAlgo() {
  // ...
}

export { fazAlgo };
```

modulo_b.js

```
import { fazAlgo } from "./modulo_a.js";

fazAlgo();
```

É importante ressaltar que para que um arquivo seja interpretado como um módulo, devemos incluí-lo na página usando o atributo type="module" no elemento script.

```
<script type="module" src="./modulo_b.js"></script>
```

Feito isso, o módulo A será carregado automaticamente, pois é uma dependência de B. Além do mais, ainda que vários módulos dependam de A, ele será incluído apenas uma vez, tornando a gestão de dependências automática.

Por fim, também podemos importar todos as declarações de um módulo por meio da sintaxe `import * as ma from "./modulo_a.js"`. Assim, podemos usar qualquer membro exportado de A, prefixando-o com o objeto `ma`, por exemplo `ma.fazAlgo()`.

Capítulo 8. Requisições HTTP em JavaScript

Neste capítulo veremos como carregar dados dinamicamente via JavaScript (normalmente JSON), sem a necessidade de carregar outra página completa. Esse tipo de técnica, que inicialmente era feita via o objeto `XMLHttpRequest`, hoje pode ser feita com uma API mais moderna, que estudaremos neste capítulo.

A API fetch

A API fetch nos permite disparar requisições HTTP conforme desejado e tratar a resposta, tudo isso em "segundo plano" sem recarregar a página. O ponto de entrada da API é a função fetch, que tem o seguinte formato:

```
let p = fetch("http://minha.api/endpoint", { method: "GET" });
```

O primeiro parâmetro da função é a URL a ser carregada, e o segundo parâmetro é um objeto opcional, com opções da requisição. Entre essas opções, as mais comumente usadas são:

- **method**: GET, POST, PUT, DELETE, HEAD, etc.
- **headers**: cabeçalhos HTTP.
- **body**: corpo da requisição (para postar dados codificados em JSON, por exemplo).

Promises

Chamadas à função fetch, não retornam a resposta da requisição diretamente, mas sim um objeto *Promise*, que representa um resultado futuro a ser resolvido. Isso ocorre porque a requisição é feita de forma assíncrona, em segundo plano. Quando a resposta chegar, um evento é disparado. Devemos portanto registrar uma função *handler* para tratar a resposta quando ela chegar por meio da função *then*, como exemplificado abaixo.

```
let employeesPromise = fetch("http://localhost:3000/employees");

employeesPromise.then((resp) => {
  // Handler executado ao receber a resposta,
  // converte a resposta JSON para objeto
```

```
  resp.json().then((employees) => {
    // Handler executado após converter JSON para objeto,
    // exibe o resultado no console
    console.log(employees);
  });
});
```

Note que o objeto de resposta possui a função *json*, que devolve outra *promise* com os dados JSON convertidos em objeto.

Dominando promises: carregamento sequencial e paralelo

Encadeamento de promises

Podemos simplificar o exemplo anterior utilizando o encadeamento de *promises*. A função *then*, na verdade, retorna uma nova *promise*, que encapsula o valor retornado dentro da função *handler* fornecida. Além do mais, o *handler* pode retornar um outro objeto *promise*, que sua resposta será capturada.

```
fetch("http://localhost:3000/employees").then((resp) => {
  return resp.json();
}).then((employees) => {
  console.log(employees);
});
```

Carregamento de dados sequencial

Podemos forçar o carregamento sequencial de dados alinhando uma chamada à função *fetch* dentro de um *handler* da resposta anterior.

```
fetch("http://localhost:3000/employees").then((resp) => {
  return resp.json();
}).then((employees) => {
  fetch("http://localhost:3000/roles").then((resp2) => {
    return resp2.json();
  }).then((roles) => {
    console.log(employees);
    console.log(roles);
  });
});
```

Carregamento de dados em paralelo

Se retirarmos o alinhamento do código anterior o carregamento será feito em paralelo, apesar do código parecer sequencial (lembre-se que a chama *fetch* é assíncrona).

```
fetch("http://localhost:3000/employees").then((resp) => {
  return resp.json();
}).then((employees) => {
  console.log(employees);
});

fetch("http://localhost:3000/roles").then((resp2) => {
  return resp2.json();
}).then((roles) => {
  console.log(roles);
});
```

Nesse formato, não conseguimos ler as variáveis `employees` e `roles` na mesma função, pois as respostas chegam em momentos diferentes. Para essas situações, podemos usar a função `Promise.all` que cria uma nova *promise* que aguarda um *array* de *promises*, e encapsula seus resultados em um *array*, do mesmo tamanho.

```
Promise.all([
  fetch("http://localhost:3000/employees").then(r => r.json()),
  fetch("http://localhost:3000/roles").then(r => r.json())
]).then(([employees, roles]) => {
  console.log(employees);
  console.log(roles);
});
```

Async/await

As palavras-chave **async** e **await** foram introduzidas na ES2017 para simplificar o código assíncrono, escrevendo-o como se fosse síncrono. Quando uma função é marcada como **async**, podemos usar o comando **await**, que aguarda uma *promise* e devolve seu resultado.

```
// Sem async/await
fetch("http://localhost:3000/employees")
  .then(r => r.json())
  .then(employees => {
    console.log(employees);
    fetch("http://localhost:3000/roles")
      .then(r => r.json())
      .then(roles => {
        console.log(roles);
      });
  });

// Com async/await
let employees = await fetch("http://localhost:3000/employees")
  .then(r => r.json());
console.log(employees);
let employees = await fetch("http://localhost:3000/roles")
  .then(r => r.json());
console.log(roles);
```

Tratamento de erros

Uma requisição HTTP pode falhar, tanto por erros de rede quanto por erros no *back end*. Portanto é importante tratá-los com a função *catch*, como exemplificado.

```
fetch("http://localhost:3000/employees").then(r => {
  if (r.ok) {
    return r.json();
  } else {
    // O fetch falha com erros de rede, mas por padrão
    // respostas HTTP em status de erro não falham. Por
    // isso lançamos o erro manualmente usando throw.
    throw new Error("A resposta veio com status de erro");
  }
})
.then(employees => {
  // Tudo certo, faz algo com os dados
})
.catch(erro => {
  // Ocorreu um erro, dá um tratamento adequado
});
```

Quando utilizamos async/await, o tratamento pode ser feito por meio do bloco try/catch, como se o código fosse síncrono.

```
try {
  let employees = fetch("http://localhost:3000/employees").then(r => {
    if (r.ok) {
      return r.json();
    } else {
      throw new Error("A resposta veio com status de erro");
    }
  });
  // Tudo certo, faz algo com os dados
} catch (erro) {
  // Ocorreu um erro, dá um tratamento adequado
}
```

Capítulo 9. Tarefas temporizadas ou periódicas em JavaScript

Pode ser necessário agendar a execução de código em uma aplicação, tanto para executar tarefas periódicas, quanto para atrasar a execução de alguma tarefa.

setTimeout

A função `setTimeout` agenda a execução de uma função após um atraso especificado em milissegundos.

```
let timeout = setTimeout(() => {
  // executa após 500 ms
}, 500);

// Se precisar, cancelar o agendamento
clearTimeout(timeout);
```

setInterval

A função `setInterval` agenda a execução repetida de uma função com um intervalo de tempo especificado em milissegundos.

```
let interval = setInterval(() => {
  // executa repetidamente uma vez por segundo
}, 1000);

// Se precisar, interrompe a execução
clearInterval(interval);
```

requestAnimationFrame

A função `requestAnimationFrame` agenda a execução de uma tarefa antes da renderização do próximo quadro pelo navegador. É possível chamar `requestAnimationFrame` para executar animações, jogos, e qualquer tarefa em tempo real na taxa de atualização da tela do dispositivo de forma eficiente.

```
function draw(time) {
  // Atualiza algum elemento

  // Agenda uma nova execução para o quadro seguinte
  requestAnimationFrame(draw);
```

```
}
requestAnimationFrame(draw);
```

Referências

W3C. HTML Standard, 2021. Especificação oficial da linguagem HTML. Disponível em: <https://www.w3.org/html/>. Acesso em: 04 mar. 2021.

MOZILLA. MDN Web Docs, 2021. Disponível em: <https://developer.mozilla.org/>. Acesso em: 04 mar. 2021

W3C. CSS Snapshot, 2020. Especificação oficial da linguagem CSS. Disponível em: <https://www.w3.org/TR/CSS/>. Acesso em: 04 mar. 2021

ECMA INTERNATIONAL. ECMAScript 2020 language specification, 11th edition, 2020. Especificação oficial de ECMSScript. Disponível em: <https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>. Acesso em: 04 mar. 2021