

EFEITOS NO DOM

Dataset

HTMLElement

Todo elemento HTML do DOM herda propriedades e métodos do construtor `HTMLElement`.

```
const h1 = document.querySelector('h1');
Object.prototype.toString.call(h1); // [object
HTMLHeadingElement]
// HTMLHeadingElement > HTMLElement > Element > Node >
EventTarget > Object
```

Dataset

`dataset` é uma propriedade de `HTMLElement`, essa propriedade é um objeto do tipo `DOMStringMap`. Dentro desse objeto existe uma coleção de chave / valor, com todos os atributos do elemento html que começarem com `data-`.

```
<div data-cor="azul" data-width="500">Uma Div</div>
<span data-anime="left" data-tempo="2000">Um Span</span>
```

```
// Ambos os valores selecionam a mesma div acima.
let div = document.querySelector('div');
div = document.querySelector('[data-cor]');
div = document.querySelector('[data-cor="azul"]');

div.dataset;
// DOMStringMap {cor: "azul", width: "500"}
div.dataset.cor; // 'azul'
div.dataset.width; // '500'
div.dataset.tempo = 1000;
// DOMStringMap {cor: "azul", width: "500", tempo: "1000"}
```

ORIGAMID

Data Atributes

Os atributos e valores que começarem com data- poderão ser utilizados como forma de configuração de plugins e interações DOM / JS.

```
<div data-anima="left" data-tempo="1000">Div 1</div>
<div data-anima="right" data-tempo="2000">Div 2</div>
```

```
const divs = document.querySelectorAll('[data-anima]');
divs.forEach((div) => {
  div.classList.add(div.dataset.anima);
});

// adiciona em cada div
// uma classe com o mesmo nome
// que o valor de data
```

Data vs Class

A vantagem de se utilizar data attributes é que torna mais fácil evitarmos conflitos com estilos do CSS. Além de deixar a estrutura da tag mais organizada.

```
<div data-anima="left" data-tempo="1000">Div 1</div>
<div class="anima-left tempo-1000">Div 2</div>
```

Nomenclatura

Por padrão o JavaScript não aceita - (traço) como caracter válido para nomear propriedades. Então qualquer traço será removido e a letra seguinte transformada em maiúscula.

```
<div data-anima-scroll="left">Div 1</div>
```

```
const div = document.querySelector('[data-anima-scroll]');
div.dataset;
// {animaScroll: 'left'}
div.dataset.animaScroll; // left
div.dataset.tempoTotal = 1000;
// Na div vira data-tempo-total="1000"
```

```
delete div.dataset.animaScroll; // remove o atributo
```

Exercícios

// Adicione um atributo `data-anime="show-down"` e
// `data-anime="show-right"` a todos as `section's`
// com descrição dos animais.

// Utilizando estes atributos, adicione a classe
// `show-down` ou `show-right` a sua respectiva `section`
// assim que a mesma aparecer na tela (animacao tab)

// No CSS faça com que `show-down` anime de cima para baixo
// e `show-right` continue com a mesma animação da esquerda
// para a direita

// Substitua todas as classes js- por data attributes.

EFEITOS NO DOM

Modules Introdução

Módulos

- **Manutenção**

Dividir o código em diferentes arquivos com funções específicas (módulos) facilita a manutenção.

- **Compartilhamento**

O compartilhamento de código com outros projetos é facilitado, pois basta você importar um módulo específico.

- **Nativo no ES6+**

Ferramentas que permitem dividirmos o código em módulos já existem a bastante tempo. Grunt, Gulp, Webpack, Browserify, Parcel e outras. Mas agora os módulos são nativos.

Modules ES6

Basta adicionar `type="module"` na tag script do HTML. Utilize a palavra chave `export` na frente do valor que deseja exportar (use `default` se for único). E `import nome from arquivo.js` para importar.

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

```
// arquivo scroll-suave.js
export default function scrollSuave() {
    ...
};
```

```
// arquivo script.js
import scrollSuave from './scroll-suave.js';

scrollSuave();
```

Geralmente um valor por módulo.

Named Exports

Você pode exportar mais de um valor. Quando for importar utilize as chaves para especificar cada valor. O nome importado deve ser igual ao exportado.

```
// arquivo scroll.js
export function scrollSuave() {
    ...
};

export function scrollAnimacao() {
    ...
};
```

```
// arquivo script.js
import { scrollSuave, scrollAnimacao } from './scroll.js';
scrollSuave();
scrollAnimacao();
```

```
// Importe todos os valores em um objeto
import * as scroll from './scroll.js';
```

```
scroll.scrollAnimacao();
```

Valores do Export

Podemos exportar objetos, funções, classes, números, strings e mais.

```
// arquivo configuracao.js
export function scrollSuave() {};
export const ano = 2000;
export const obj = {nome: 'Ford'};
export const str = 'Frase';
export class Carro {};
```

```
// arquivo script.js
import * as conf from './configuracao.js';
conf.str;
conf.obj;
conf.ano;
```

Características

- Strict mode

'use strict' por padrão em todos os arquivos.

- Variáveis ficam no module apenas

Não vazam para o escopo globo.

- This fora de um objeto faz referência a undefined

Ao invés de fazer referência ao window.

- Assíncrono

use strict

O modo estrito previne que algumas ações consideradas erros. Basta adicionarmos `'use strict'` no topo de um arquivo, que ele entrará neste modo.

```
'use strict';

nome = 'Ford'; // erro, variável global
delete Array.prototype; // erro, não deletável
window.top = 200; // erro, não pode mudar
const arguments = 3.14; // escrever em palavra reservada
```

Por padrão todo module está no modo estrito

Exercício

// Divida o projeto em diferentes módulos

EFEITOS NO DOM

setTimeout e setInterval

setTimeout()

`setTimeout(callback, tempo, arg1, arg2, ...)` método assíncrono que ativa o callback após `tempo`. Não existe garantia de que o código será executado exatamente após o tempo, pois o callback entra na fila e espera pela Call Stack estar vazia.

```
function espera(texto) {  
    console.log(texto);  
}  
setTimeout(espera, 1000, 'Depois de 1s');
```

Imediato

Se não passarmos o argumento de tempo ele irá assumir o valor 0 e entrará na `fila` imediatamente para ser executado. Podemos passar uma função anônima diretamente com argumento.

```
setTimeout(() => {  
    console.log('Após 0s?');  
});
```

Exemplo de setTimeout

Loops e setTimeout

Um loop é executado rapidamente, em milissegundos. Se colocarmos um setTimeout dentro do loop, todos eles serão adicionados à Web Api praticamente no mesmo tempo. Um evento de setTimeout não espera o tempo do anterior acabar para iniciar.

```
for(let i = 0; i < 20; i++) {  
  setTimeout(() => {  
    console.log(i);  
  }, 300);  
}
```

Corrigindo o Loop

Agora ele está multiplicando o tempo por i. Assim o primeiro aparecerá em 0ms, o segundo em 300ms, o terceiro em 600ms e assim em diante.

```
for(let i = 0; i < 20; i++) {  
    setTimeout(() => {  
        console.log(i);  
    }, 300 * i);  
}
```

This e Window

setTimeout é um método do objeto Window. O valor de `this` dentro do mesmo callback é uma referência ao seu objeto no caso Window.

```
const btn = document.querySelector('button');
btn.addEventListener('click', handleClick);

function handleClick(event) {
  setTimeout(function() {
    this.classList.add('active');
  }, 1000)
}

// Erro pois window.classList não existe
```

Arrow Function

Quando utilizamos uma Arrow Function como callback, o contexto de `this` passa a ser do local onde o setTimeout foi iniciado.

```
const btn = document.querySelector('button');
btn.addEventListener('click', handleClick);

// this agora é btn.
function handleClick(event) {
  setTimeout(() => {
    this.classList.add('active');
  }, 1000)
}
```

setInterval

`setInterval(callback, tempo, arg1, arg2, ...)`, irá ativar o callback toda vez que a quantidade de tempo passar.

```
function loop(texto) {  
    console.log(texto);  
}  
setInterval(loop, 1000, 'Passou 1s');  
  
// loop a cada segundo  
let i = 0;  
setInterval(() => {  
    console.log(i++);  
}, 1000);
```

clearInterval

`clearInterval(var)` , podemos parar um intervalo com o clearInterval. Para isso precisamos atribuir o setInterval a uma variável.

```
const contarAte10 = setInterval(callback, 1000);

let i = 0;
function callback() {
  console.log(i++);
  if (i > 10) {
    clearInterval(contarAte10);
  }
}
```

Exercícios

```
// Mude a cor da tela para azul e depois para vermelho a cada  
2s.  
  
// Crie um cronometro utilizando o setInterval. Deve ser  
possível  
// iniciar, pausar e resetar (duplo clique no pausar).
```

EFEITOS NO DOM

Date Object

new Date()

O construtor `Date` cria um objeto contendo valores como mês, dia, ano, horário e mais. A data é baseada no relógio interno do computador.

```
const agora = new Date();
agora;
// Semana Mês Dia Ano HH:MM:SS GMT
agora.getDate() // Dia
agora.getDay() // Dia da Semana ex: 5 = Fri
agora.getMonth() // Número dia mês
agora.getFullYear() // Ano
agora.getHours() // Hora
agora.getMinutes() // Minutos
agora.getTime() // ms desde 1970
agora.getUTCHours() - 3 // Brasília
```

getTime()

O método `getTime()` mostra o tempo total em milissegundos desde o dia 1 de janeiro de 1970.

```
const agora = new Date();
agora.getTime(); //  
  
// total de dias desde 1 de janeiro de 1970
const diasPassados = agora.getTime() / (24 * 60 * 60 * 1000);
```

Dias até

Podemos criar um objeto com uma data no futuro, passando uma string com o valor da data.

```
const agora = new Date();
const promocao = new Date('December 24 2018 23:59');

function converterEmDias(time) {
    return time / (24 * 60 * 60 * 1000);
}

const diasAgora = converterEmDias(agora);
const diasPromocao = converterEmDias(promocao);

const faltam = diasPromocao - diasAgora;
```

EFEITOS NO DOM

Forms

Forms

É comum utilizarmos inputs de formulários para criarmos uma interface entre funções de JavaScript e o usuário final do site. Para isso precisamos aprender como pegar os valores dos formulários.

```
<form name="contato" id="contato">
  <label for="nome">Nome</label>
  <input type="text" name="nome" id="nome">
  <label for="email">Email</label>
  <input type="email" name="email" id="email">
  <label for="mensagem">Mensagem</label>
  <textarea name="mensagem" id="mensagem"></textarea>
</form>
```

```
document.forms; // lista com os formulários
document.forms.contato; // form com nome contato
document.forms.contato.elements; // elementos
document.forms[0].elements[0].value; // valor do primeiro
```

Values

A propriedade `value` retorna o valor do elemento no formulário. Se adicionarmos um callback ao `keyup` (tecla levantar), podemos ficar de olho no evento e puxar o valor sempre que ele mudar. `change` dispara quando houver mudanças.

```
const form = document.getElementById('contato');
function handleKeyUp(event) {
  console.log(event.target.value);
}
form.addEventListener('keyup', handleKeyUp);
```

Validação

O método `checkValidity` verifica se um input com o atributo `required`, é válido ou não. A propriedade `validationMessage` possui a mensagem padrão de erro do browser. É possível modicar com `setCustomValidity('')`

```
<input type="email" name="email" id="email" required>
<span class="erro"></span>
```

```
const form = document.getElementById('contato');
function handleChange(event) {
  const target = event.target;
  if(!target.checkValidity()) {
    target.classList.add('invalido');
    target.nextElementSibling.innerText =
    target.validationMessage;
  } else {
    target.classList.remove('invalido');
  }
}
form.addEventListener('change', handleChange);
```

ORIGAMID

Select

```
<select name="cores" id="cores">
  <option value="black">Preto</option>
  <option value="white">Branco</option>
  <option value="blue">Azul</option>
</select>
<input type="color">
```

```
const form = document.getElementById('contato');
function handleChange(event) {
  document.body.style.backgroundColor = event.target.value;
}
form.addEventListener('change', handleChange);
```

Diferentes Inputs

```
<input type="color">
<input type="date">
<input type="number">
<input type="range">
<input type="password">
```

```
const form = document.getElementById('contato');
function handleChange(event) {
  console.log(event.target.value)
}
form.addEventListener('change', handleChange);
```

Checkbox

Retorna o valor de value, se estiver checada ou não. `checked` retorna true ou false.

```
<label for="identidade">Possui identidade?</label>
<input type="checkbox" value="identidade" id="identidade">
<label for="casado">Casado?</label>
<input type="checkbox" value="casado" id="casado">
```

```
const form = document.getElementById('contato');
function handleChange(event) {
  if(event.target.checked)
    console.log(event.target.value);
}
form.addEventListener('change', handleChange);
```

Radio

A diferença entre Radio e Checkbox é que radio aceita apenas uma seleção por grupo. Radio é agrupado pelo atributo name.

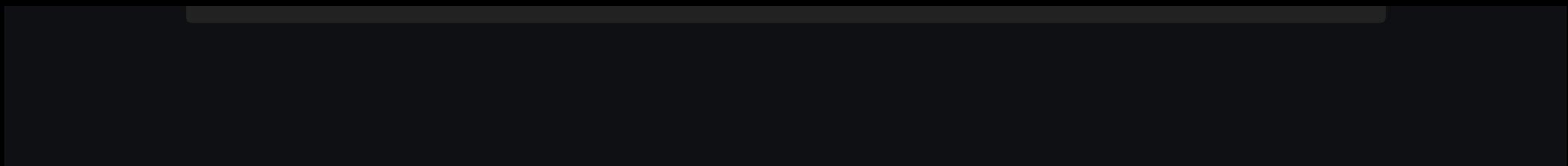
```
<input type="radio" id="guitarra" value="guitarra"
name="instrumento" />
<label for="guitarra">Guitarra</label>

<input type="radio" id="violao" value="violao"
name="instrumento" />
<label for="violao">Violão</label>

<input type="radio" id="baixo" value="baixo"
name="instrumento" />
<label for="baixo">Baixo</label>
```

```
const form = document.getElementById('contato');
function handleChange(event) {
  if(event.target.checked)
    console.log(event.target.value);
```

ORIGAMID



Pegando todos os valores

Ao invés de selecionarmos elemento por elemento, podemos utilizar um objeto para colocarmos todos os dados que o usuário colocar no formulário.

```
<form name="contato" id="contato">
  <label for="nome">Nome</label>
  <input type="text" name="nome" id="nome">
  <label for="email">Email</label>
  <input type="email" name="email" id="email">
  <label for="mensagem">Mensagem</label>
  <textarea name="mensagem" id="mensagem"></textarea>
</form>
```

```
const form = document.getElementById('contato');
const dados = {};
function handleChange(event) {
  dados[event.target.name] = event.target.value;
}
form.addEventListener('change', handleChange);
```