

Estrutura de dados

<https://docs.google.com/presentation/d/1UdwvLJEcpB2SsGZ1y4FfwFywpVxBaC3d/edit>

Aqui começamos a falar de verdade sobre Javascript, sem ser algo que "deixa o HTML dinâmico", mas sim como uma linguagem escalável e cheia de recursos.

JavaScript é uma linguagem leve, interpretada e baseada em objetos, mais conhecida como a linguagem de script para páginas Web, mas usada também em vários outros ambientes sem browser, tais como NodeJS, Apache CouchDB e Adobe Acrobat. O JavaScript é uma linguagem baseada em protótipos, multi-paradigma e dinâmica, suportando estilos de orientação a objetos, imperativos e declarativos (como por exemplo a programação funcional que falaremos mais para frente).

Sintaxe básica

JavaScript é **case-sensitive** e usa o conjunto de caracteres **Unicode**. Por exemplo, a palavra Früh (que significa "cedo" em Alemão) pode ser usada como nome de variável.

```
var Früh = "foobar";
```

Mas a variável `früh` não é a mesma que `Früh` porque JavaScript é case sensitive. Logo letras minúsculas e maiúsculas fazem uma grande diferença e com isso criamos **nosso primeiro padrão**:



Variáveis primitivas, **objetos** e **arrays** serão sempre escritas com camelCase

`camelCase` é uma convenção de nomenclatura que deve começar com a primeira letra minúscula e a primeira letra de cada nova palavra subsequente maiúscula:

```
coisasParaFazer  
idadeDoAmigo  
valorFinal
```

Declarações e comentários

A sintaxe dos comentários em JavaScript é simples e pode ser feito em uma linha ou mais:

```
// comentário de uma linha

/*
    isto é um comentário longo
    de múltiplas linhas.
*/

/* Você não pode /* alinhar comentários */ SyntaxError */
```

Comentários são boas formas de lembrar o que precisa ser feito no código pois durante o desenvolvimento podem ter casos em que vamos deixar algo preparado para o futuro ou para um colega implementar, isso acaba sendo uma boa forma de comunicar as formas que você espera que a sua equipe fará.

Também são utilizados para explicar o que uma porção de código faz. Embora seja desencorajado, nem todas as lógicas serão tão simples ao ponto de serem auto-explicáveis.

Existem três tipos de declarações de variáveis em JavaScript:

- `var` Declara uma variável, que pode ser iniciada com um valor.
- `let` Declara uma variável local de escopo do bloco, opcionalmente, inicializando-a com um valor.
- `const` Declara uma constante de escopo de bloco, apenas de leitura.

Mas o que é "escopo de bloco"?

Escopo de bloco é basicamente o lugar onde você declara essa variável. Quando criamos uma variável com `var` estamos colocando essa variável na `window`, logo teremos acesso a essa variável em qualquer parte da nossa aplicação. Usando `let` e `const` as variáveis serão criadas em seu escopo atual. Calma...

*Vamos falar de **escopos** daqui a pouco ;)*

```
var a = 1
let b = 2
const c = 3

console.log(window.a) // 1
console.log(window.b) // undefined
console.log(window.c) // undefined
console.log(b) // 2
console.log(c) // 3
```

Variáveis

Um valor primitivo é representado diretamente através do mais baixo nível da implementação de uma linguagem e em JavaScript, existem 6 tipos primitivos:

- String
- Number
- Boolean
- Null
- undefined
- Symbol (*)

Com exceção do `null` e do `undefined`, todos os primitivos tem um objeto wrapper equivalente. Wrappers nesse caso são objetos globais que possuem métodos para manipular seus respectivos tipos.

- `String` para o primitivo string.
- `Number` para o primitivo number.
- `Boolean` para o primitivo boolean.
- `Symbol` para o primitivo Symbol.

O método `valueOf()` do objeto wrapper retorna o valor primitivo e também é possível verificar o tipo da variável com `typeof nomedavariavel`.

*** Ao final do módulo leia mais sobre Symbol para entender melhor sobre esse tipo**

© Curso Online de React do Zero ao Pro
Desenvolvido por Gustavo Vasconcellos e EBAC Online

Arrays e Objetos

Arrays e Objetos regem de longe o mundo de desenvolvimento. A partir dessas duas estruturas temos a criação de diversos tipos de aplicações e por isso saber usar eles da forma certa acaba nos poupando MUITO tempo!

Arrays

Vamos começar com Arrays, eles são geralmente descritas como "lista de coisas" podendo ser múltiplos valores armazenados em uma lista. Um objeto array pode ser armazenado em variáveis e ser tratado de forma muito similar a qualquer outro tipo de valor,a sacada é que podemos acessar cada valor dentro da lista individualmente.

Se nós não tivessemos arrays, teríamos que armazenar cada item em uma variável separada, então chamar o código para mostrar e adicionar separadamente cada item. Isto seria muito mais longo de escrever, menos eficiente e mais suscetível a erros. Se nós temos 10 itens para adicionar na fatura, isto é ruim o bastante, mas e se fosse 1000 itens ou 1000? Loucura né?

Partindo pro código, para criar um array devemos usar colchetes, os quais contém uma lista de itens separada por vírgulas. Podendo ser *strings*, *numbers*, *objects* e até mesmo outros arrays!

```
var shopping = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];
var sequence = [1, 1, 2, 3, 5, 8, 13];
var objects = [{ foo: 'bar' }, { foo: 'bar' }];
var random = ['tree', 795, [0, 1, 2]];
```

A partir disso temos que seguir um detalhe importante para o funcionamento constante do código: **Arrays devem ter sempre o mesmo conteúdo.** Exemplos como a variável `random` são péssimos num código pois sempre que usamos arrays é mais fácil se eles seguem o mesmo padrão para não precisarmos nos preocupar em criar mil e uma condições.

Para acessar os itens de um array precisamos mais uma vez usar a notação de colchetes. E assim como variaveis comuns, podemos alterar seu valor:

```
const shoppingList = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];
shoppingList[2] // 'cheese'
```

```
shoppingList[2] = 'sauce'  
shoppingList // ['bread', 'milk', 'sauce', 'hummus', 'noodles'];
```

E a partir disso espero que você tenha percebido que ao enviar o valor 2 dentro dos colchetes **não está retornando "milk"**. Isso acontece porque os arrays estão **eçam na posição 0**. E então como podemos fazer para obter o último valor de um array? Com a propriedade `length`

```
const shoppingList = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];  
const listLength = shopping.length // 5  
  
shoppingList[listLength - 1] // 'noodles'
```

Pois é, podemos até colocar variáveis dentro dos colchetes de um array!



Com isso criamos um padrão de entendimento! Como arrays sempre indicam um **grupo de coisas** e não apenas uma coisa, vamos sempre declarar arrays **no plural** ou com um **sufixo “List”**, dessa forma deixaremos mais simples o entendimento do que o código está fazendo.

Objetos

Diferente de Arrays, objetos vão ter as características de algo. Podemos entender objetos Javascript da mesma forma que na vida real, como por exemplo um Notebook, eles possuem diversas características que os diferem mas acabam sendo do mesmo tipo. Tela, memória, processador e etc, isso nós chamamos de **propriedades (ou campos)** e elas são iguais a variáveis a única diferença é que estão ligadas em um objeto:

```
const shopping = { name: 'noodles', price: 0.9, quantity: 2 }  
shopping.quantity // 2
```

Nesse exemplo temos um objeto `shopping` com três propriedades: `name`, `price` e `quantity`. Mas isso não nos impede de adicionar mais propriedades nela:

```
shopping.urgent = true  
shopping.description = "Meet flavor please."
```

E a partir disso também tem como obter propriedades que não existem, elas são `undefined` (e não `null`).



Com isso criamos um padrão de entendimento! Assim como propriedades de objetos que não existem são `undefined`, em situações que nossas variáveis forem mudar conforme o código executar vamos tentar iniciar variáveis que serão objetos como `null`.

Em JavaScript, quase tudo é um objeto. Todos os tipos primitivos - com exceção de `null` e `undefined` - são tratados como objetos. Eles podem receber propriedades, e possuem todas as características de objetos, inclusive Arrays:

```
const str = "minhaString",
      num = Math.random(), // 0.4415445342216153
      obj = new Object(); // {}

str.length // 11
num.toFixed(2) // "0.44"
obj.toString() // "[object Object]"
```

© Curso Online de React do Zero ao Pro
Desenvolvido por Gustavo Vasconcellos e EBAC Online

Uso de Escopos

Finalmente, depois de falar tanto sobre Escopos essa aula chegou! 🎉🎉🎉

A grosso modo, escopo é o contexto atual de execução, em que variáveis e expressões são criadas ou podem ser chamadas. Se uma variável ou outra expressão não estiver "no escopo atual", então não está disponível para uso. Os escopos também podem ser em camadas em uma **hierarquia** em que escopos filhos possuem acesso aos escopos pais, mas não vice-versa.

Basicamente podemos considerar que **qualquer** uso de chaves (com excessão dos objetos) num código cria um escopo novo. Então analisando o código abaixo podemos identificar diversos escopos:

```
var x, y, z;

(function foo() {
    const a = 1

    function bar() {
        let x = 2

        console.log(a) // 1
    }

    function bar2() {
        let y = 3
        var z = 1

        console.log(x) // undefined
        console.log(window.z) // undefined
        console.log(z) // 1
    }

    bar()
    bar2()

    console.log(x) // undefined
    console.log(y) // undefined
    console.log(z) // undefined
}());
```

Calma, foque apenas nas chaves {}, conseguimos ver 3 escopos definidos nesse código e as variáveis a, x, y e z sendo declaradas nesses escopos, esse exemplo mostra de forma clara como escopos são únicos e definem seus próprios valores e compartilham apenas com seus escopos filhos.

Condicionais também servem para criar escopos novos:

```
let x = 1, y = 2, z = 3;

if (x == 1) {
  let a = 0
  x = x + 1
} else {
  y = 1
}

console.log(x) // 2
console.log(y) // 2
console.log(a) // undefined
```

Falando mais de funções, eu quero criar um entendimento mais básico nesse momento, mas no futuro falaremos bem mais sobre elas.

```
function nomeDaFuncao() {
  // conteúdo da função

  return 1 // valor de retorno da função
}
```

Um função pode retornar qualquer coisa para o escopo em que for chamada, logo se eu quiser chamar e obter o valor do exemplo acima em uma variável eu devo:

```
const variavel = nomeDaFuncao()
console.log(variavel) // 1
```

A partir disso podemos chamar e obter o valor da função em qualquer lugar do nosso código.

© Curso Online de React ao Zero ao Pro
Desenvolvido por Gustavo Vasconcellos e EBAC Online

Operadores

Javascript tem um monte de operadores, e vai por mim, são muitos mesmo. Alguns sequer são usados de forma rotineira e por isso eu vou assinalá-los e também falar mais sobre. Basicamente, temos os seguintes operadores:

- Operadores de atribuição*
- Operadores de comparação*
- Operadores aritméticos*
- Operadores bit a bit
- Operadores lógicos*
- Operadores de string
- Operador condicional (ternário)*
- Operador vírgula
- Operadores unário
- Operadores relacionais

Não se esqueça, são apenas referências e é saber que elas existem que importa, por isso, mesmo que eu não tenha assinalado, leia a documentação e entenda o seu uso, algum dia você irá usar algum desses carinhos.

Antes de realmente começar, vamos só criar uma nomeação a partir dessas expressões para ficar mais claro "o que é o que":

- **Operando** são as expressões que faremos para que sejam operadas, exemplo `1`
- **Operador** é a expressão que dará comportamento junto de **N operandos**, exemplo `1 + 1`

```
operando1 operador operando2 // 1 + 1
operador operando // delete x
operando operador // 1++
```

Agora estamos prontos.

Atribuição

Um operador de atribuição atribui um valor ao operando à sua esquerda baseado na operação e no operando a direita. O operador de atribuição básico é o igual (`=`), que atribui o valor do operando à direita ao operando à esquerda. Ou melhor, `x = y` atribui o valor de `y` a `x`.

Operadores de Atribuição

Aa Nome	≡ Operador encurtado	≡ Operador padrão
<u>Atribuição padrão</u>	<code>x = y</code>	<code>x = y</code>
<u>Atribuição de adição</u>	<code>x += y</code>	<code>x = x + y</code>
<u>Atribuição de subtração</u>	<code>x -= y</code>	<code>x = x - y</code>
<u>Atribuição de multiplicação</u>	<code>x *= y</code>	<code>x = x * y</code>
<u>Atribuição de divisão</u>	<code>x /= y</code>	<code>x = x / y</code>
<u>Atribuição de resto (Mod)</u>	<code>x %= y</code>	<code>x = x % y</code>
<u>Atribuição exponencial</u>	<code>x **= y</code>	<code>x = x ** y</code>

Vale mencionar que existem outros operadores de atribuição, mas são bem menos utilizados, o que mais usamos no dia-a-dia é a atribuição padrão.

Comparação

Operadores de comparação fazem sempre com que a expressão retorne `true` ou `false`.

Operadores de comparação

Aa Operador	≡ Exemplos que retornam verdadeiro	≡ Descrição
<u>Igual (<code>==</code>)</u>	<code>3 == var1</code> <code>"3" == var1</code> <code>3 == '3'</code>	Retorna verdadeiro caso os operandos sejam iguais.
<u>Não igual (<code>!=</code>)</u>	<code>var1 != 4</code> <code>var2 != "3"</code>	Retorna verdadeiro caso os operandos não sejam iguais.
<u>Estritamente igual (<code>===</code>)</u>	<code>3 === var1</code>	Retorna verdadeiro caso os operandos sejam iguais e do mesmo tipo.
<u>Estritamente não igual (<code>!==</code>)</u>	<code>var1 !== "3"</code> <code>3 !== '3'</code>	Retorna verdadeiro caso os operandos não sejam iguais e/ou não sejam do mesmo tipo.
<u>Maior que (<code>></code>)</u>	<code>var2 > var1</code> <code>"12" > 2</code>	Retorna verdadeiro caso o operando da esquerda seja maior que o da direita.

Aa Operador	Exemplos que retornam verdadeiro	Descrição
<u>Maior que ou igual (\geq)</u>	<code>var2 >= var1 var1 >= 3</code>	Retorna verdadeiro caso o operando da esquerda seja maior ou igual ao da direita.
<u>Menor que (\leq)</u>	<code>var1 < var2 "12" < "2"</code>	Retorna verdadeiro caso o operando da esquerda seja menor que o da direita.
<u>Menor que ou igual (\leq)</u>	<code>var1 <= var2 var2 <= 5</code>	Retorna verdadeiro caso o operando da esquerda seja menor ou igual ao da direita.

E aqui vamos criar um novo padrão de entendimento:



Afim de garantir que nossas comparações serão sempre exatamente da forma que esperamos, utilizaremos `==` para estarmos sempre certos de que os valores são completamente **iguais**.

E para poder argumentar sobre isso, quero que você veja a [tabela de comparações do MDN](#), e a partir disso espero que tenha o entendimento que usar `==` não vai te entregar exatamente o que você espera nas suas comparações.

Na próxima aula vamos ter mais uma noção sobre comparações, mas até lá, siga com o resto dos operadores que devemos ver:

Aritméticos

Estes operadores usam valores numéricos (sejam literais ou variáveis) como seus operandos e retornam um valor numérico e trabalham da mesma forma como na maioria das linguagens de programação quando utilizados com números de ponto flutuante.

Repare que expressões que não existem retornam **NaN (Not a Number)**, um exemplo é a divisão por zero.

Os operadores aritméticos padrão são os de soma (+), subtração (-), multiplicação (*) e divisão (/) e a partir disso podemos criar expressões como:

```
console.log(1 / 2); // 0.5
console.log(1 / 2 == 1.0 / 2.0); // true
```

Além das quatro operações comuns de matemática, temos também:

- Módulo (ou mod) → Retorna o inteiro restante da divisão dos dois operandos, acaba sendo mais utilizado quando precisamos descobrir se um número é par ou ímpar:

```
14 % 2 // 0 -> é par
21 & 2 // 1 -> é ímpar
```

- Exponenciação → Calcula a base elevada á potência do expoente, que é: `base`
`** expoente`

```
2 ** 3 // 8
10 ** -1 // 0.1
```

- Incremento → Adiciona 1 ao seu operando e pode ser utilizado de duas formas:
 - Se usado como operador prefixado (`++x`), retorna o valor de seu operando após a adição.
 - Se usado como operador pós-fixado (`x++`), retorna o valor de seu operando antes da adição.

```
var x = 1
var y = x++ // retornou 1 e depois incrementou para 2
var z = ++x // incrementou para 3 e depois retornou 3

console.log(x, y, z) // 3, 1, 3
```

- Decremento → Diminui 1 ao seu operando e pode ser utilizado das mesmas formas que o incremento:
 - Se usado como operador prefixado (`--x`), retorna o valor de seu operando após a operação.
 - Se usado como operador pós-fixado (`x--`), retorna o valor de seu operando antes da operação.

```
var x = 3
var y = x-- // retornou 3 e depois reduziu para 2
var z = --x // reduziu para 1 e depois retornou 1

console.log(x, y, z) // 1, 3, 1
```

- Negação Unária → Retorna a negação de seu operando.

```
var x = 3
-x // -3
!true // -1
```

- Adição Unária → Tenta converter o operando em um número, sempre que possível.

```
+"3" // 3
+true // 1
```

Lógicos

  Opa opa opa! Parado ai   Segura a emoção que operadores lógicos e lógicas booleanas nós iremos falar **na próxima aula**. Então pode continuar e segura vontade porque a próxima aula é muito boa!

Condicional Ternário

O operador condicional é o único operador JavaScript que utiliza três operandos. O operador pode ter um de dois valores baseados em uma condição. A sintaxe é:

```
condicao ?valor1 :valor2
```

Se `condicao` for verdadeira, o operador terá o valor de `valor1`. Caso contrário, terá o valor de `valor2`. Você pode utilizar o operador condicional em qualquer lugar onde utilizaria um operador padrão.

Por exemplo,

```
var status = (idade >= 18) ? "adulta" : "menor de idade";
```

Esta declaração atribui o valor "adulta" à variável `status` caso `idade` seja dezoito ou mais. Caso contrário, atribui o valor "menor de idade".

Unários

Operadores unários são bem raros de encontrar, mas existem dois que garantem que serão úteis:

- `delete` → Apaga um objeto, propriedade de um objeto ou um elemento no índice especificado de uma matriz

```
delete nomeObjeto;
delete nomeObjeto.propriedade;
delete nomeObjeto[indice];
```

- `typeof` → Retorna uma string indicando o tipo do operando. `operando` é uma string, variável, palavra-chave ou objeto cujo tipo deve ser retornado

```
var meuLazer = new Function("5 + 2");
var forma = "redondo";
var tamanho = 1;
var hoje = new Date();

typeof meuLazer; // "function"
typeof forma; // "string"
typeof tamanho; // "number"
typeof hoje; // "object"
typeof naoExiste; // "undefined"
```

© Curso Online de React do Zero ao Pro
Desenvolvido por Gustavo Vasconcellos e EBAC Online

Lógica booleana

Entender completamente esse tópico é muito importante para que um desenvolvedor possa, alem de desenvolver coisas mais complexas, ver e entender códigos com mais facilidade.

Operadores lógicos

Como pulamos esses operadores, começaremos falando deles. Existem 3 operadores lógicos em Javascript: **AND**, **OR** e **NOT**. Julguem para todos os casos que `isTrue` é uma variável com o valor `true` e `isFalse` é outra com o valor `false`.

AND (&&)

O operador **AND** serve para lógicas em que queremos que **ambos os valores sejam verdadeiros**.

```
isTrue && isTrue // true  
isTrue && isFalse // false  
isFalse && isTrue // false
```

Note que independente da posição do valor `false` ele nunca dará que a condição é verdadeira

OR (||)

O operador **OR** serve para lógicas em que queremos que **pelo menos um dos valores seja verdadeiro**.

```
isTrue || isTrue // true  
isFalse || isTrue // true  
isTrue || isFalse // true  
isFalse || isFalse // false
```

Note que independente da posição do valor `false`, se tiver apenas um valor `true` a expressão resultará em um valor verdadeiro.

NOT (!)

O operador **NOT** serve para negar o valor.

```
!isTrue // false  
!isFalse // true  
!"Cão" // false  
!!isFalse // false
```

Perceba que ele transforma **valores verdadeiros em falsos** e vice-versa, e o mais legal é a transformação da String `"Cão"` em `false`.

Short-Circuit Evaluation

E com esse comportamento, entro nessa funcionalidade maravilhosa do Javascript, *short-circuit* é uma mão na roda quando queremos escrever códigos curtos e simples, vejamos mais como ele funciona:

O comportamento com o operador **AND** faz com que o JS verifique os **valores da esquerda para a direita**, a partir disso o último valor a esquerda será retornado dessa expressão se todos os valores a esquerda forem verdadeiros.

```
isFalse && 'Cão' // false  
isTrue && 'Cão' // 'Cão'
```

O comportamento com o operador **OR** já é bem diferente, se o seu valor **mais a direita for verdadeiro, ele sempre vai retornar aquele valor**, mesmo se os valores mais a esquerda forem falsos!

```
isFalse || 'Cão' // 'Cão'  
isTrue || 'Cão' // 'Cão'
```

Dito isso, segue algumas práticas boas para melhorar um pouco sua performance:

- Em lógicas com **AND** deixe os valores que seriam possivelmente `false` primeiro
- Em lógicas com **OR** deixe os valores que tendem a ser `true` primeiro

Em ambos os casos o JS não **precisará perder tempo** entrando em outras expressões para **validar seu valor**, pois com o **AND** a expressão **ira parar se tiver um valor `false`** e com **OR** apenas um valor `true` é **suficiente para aceitar a expressão**.

Beleza, mas como assim "se o valor for verdadeiro" se a expressão é `isFalse || 'Cão'` e "Cão" não é `true`?

Falsy e Truthy

JS é cheio de pegadinhas, e isso acontece pois ele entende valores como verdadeiros e falsos, isso cria uma série de comportamentos estranhos mas também facilita a escrever menos código.

Verdadeiros	Falsos
true	false
“Cão”	””
1	0
-1	null
[]	undefined
{}	1 == 2
1 == “1”	[1] == [1]
1 === 1	{foo: "bar"} == {foo: "bar"}

E isso complica mais ainda quando juntamos isso com a conversão automática dos valores que Javascript faz. Vamos falar mais sobre a tirinha do Bob Esponja e sobre expressões.

Expressões lógicas



- `0 == "0" // true` → Essa expressão é verdadeira pois o valor em String é convertido para Number `0 == 0`
- `0 == [] // true` → O valor primitivo de um array vazio é **igual a 0** quando **comparado com um numero**, então isso vira `0 == 0`.
- `[] == '0' // false` → Para explicar esse valor, vamos pensar na tabela mostrada acima e também em outra expressão `[] == ''` que é **VERDADEIRA**

Basicamente, um Array vazio é um **valor verdadeiro** de acordo com a tabela e uma string com conteúdo também, já uma string vazia é um **valor falso**. O que acontece é a conversão do Array vazio que ao ser comparado com `==` sempre converte para `false`.

```
var array = []

if (array) console.log('truthy')

if (array == false) console.log('wtf??')
truthy
wtf??
```

A partir disso, vamos criar um **padrão de entendimento!**



SEMPRE usaremos expressões e condições lógicas utilizando `==` para garantir que valores e seus tipos sejam os mesmos na hora da comparação.

Em adicional, vamos também pensar em utilizar **valores verdadeiros e falsos** para escrevermos menos código e ocultar algumas comparações.

Por exemplo, podemos transformar `string.length === 0` em `!string`

© Curso Online de React do Zero ao Pro
Desenvolvido por Gustavo Vasconcellos e EBAC Online

Eventos do Navegador

Se você achou que Javascript tem muitos operadores, você vai ficar ainda mais impressionado com os Eventos que o navegador tem.

Mas fica tranquilo, vou deixar a documentação com as Categorias de Eventos para você conhecer alguns tipos e vou FILTRAR BEM eles para que você possa focar somente nos eventos nativos mais importantes ao final do material.

Mas por enquanto, vamos nos atentar em duas funções bem importantes para fazer isso tudo funcionar:

- `element.addEventListener` → Adiciona um evento
- `element.removeEventListener` → Remove um evento

Esse dois métodos que todo elemento HTML tem colocam e tiram um "observador" no elemento e sempre que algum evento for disparado ele vai executar a função que enviarmos para ele:

```
const button = document.getElementById('button')

button.addEventListener('click', () => {
  // do something
})
```

Nesse exemplo tem o evento `click` que é basicamente o evento nativo que mais usamos, de longe! Na sessão de "Eventos nativos do navegador" tem mais eventos que os elementos HTML já possuem e todos vão seguir o mesmo padrão, então vamos complicar um pouco e entrar em eventos customizados

Eventos customizados

Algumas vezes temos que customizar alguns eventos para poder transmitir ou executar algo específico no código que os eventos do navegador não são capazes de transmitir, são esses os casos que nos levam a customizar os eventos.

Para isso precisamos de 3 coisas:

- Criar um evento
- Atrelar ele em algum elemento
- Disparar o evento quando necessário

```
const openModalEvent = new Event('openModal');

document.addEventListener('openModal', (e) => { /* ... */ });

document.dispatchEvent(openModalEvent);
```

Na maioria das vezes que criamos eventos personalizados também queremos enviar dados de um lugar para o outro e a partir disso manipular os dados na função, para isso usamos o `CustomEvent`:

```
const openModalEvent = new CustomEvent(
  'openModal',
  {
    detail: {
      modalName: 'faq'
    }
  }
);

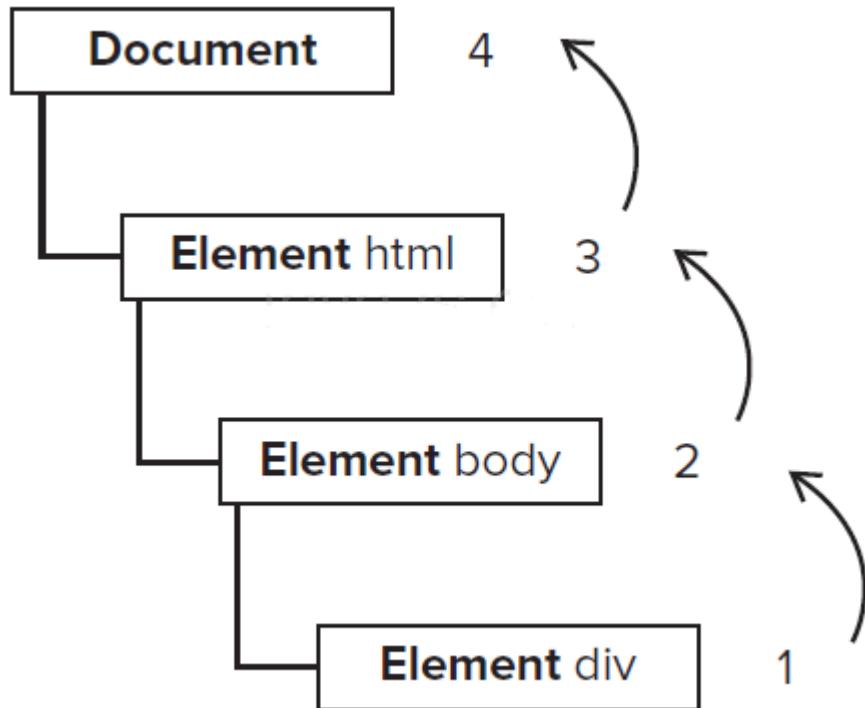
function openModalHandler(event) {
  console.log('The modal is: ' + event.detail.modalName);
}

document.addEventListener('openModal', openModalHandler);

document.dispatchEvent(openModalEvent);
```

Event bubbling

O Event bubbling ocorre quando um usuário interage com um elemento no HTML e o evento se propaga como “bolhas” por todos os elementos que estão aninhados a ele.



A grosso modo, se clicamos na div também clicamos na tag `body`, na `html` e no `document`, mas isso fica mais fácil com o exemplo a seguir:

```
<div id="pai">pai
  <div id="filho">filho
    <div id="neto">neto</div>
  </div>
</div>
```

```
const pai = document.getElementById('pai');
const filho = document.getElementById('filho');
const neto = document.getElementById("neto");

pai.addEventListener('click', () => console.log("pai foi clicado"));
filho.addEventListener('click', () => console.log("filho foi clicado"));
neto.addEventListener('click', () => console.log("neto foi clicado"));
```

Ao clicar no elemento *neto*, a ação vinculada a ele será disparada e depois os elementos acima dele também serão disparados, no caso o elemento *filho* e depois o elemento *pai*.

Forçando disparos de eventos

Do mesmo jeito que disparamos eventos customizados e próprios, podemos disparar os eventos padrões do navegador:

```
function simulateClick() {
    // Cria um evento que imita o click
    const event = new MouseEvent('click', {
        view: window,
        bubbles: true,
        cancelable: true
    });
    const cb = document.getElementById('checkbox');

    // dispara o evento
    const cancelled = !cb.dispatchEvent(event);

    /*
        Se o elemento ja tiver um evento atrelado a ele e chamar a
        função event.preventDefault() simular eventos nao funcionará
    */
    if (cancelled) {
        alert("cancelled");
    } else {
        alert("not cancelled");
    }
}
```

Eventos nativos do navegador

Paginas

- load → Dispara quando a pagina inteira terminou de carregar
- DOMContentLoaded → Dispara quando o documento HTML terminou de carregar
- resize → Dispara sempre que mudamos o tamanho da janela do navegador
- scroll → Dispara sempre que rolamos a página verticalmente ou horizontalmente

Focus

- focus → Dispara quando um elemento tem o foco do mouse - por exemplo, quando clicamos em um campo de texto.
- blur → Dispara quando um elemento perdeu o foco - por exemplo, ao clicar em outro elemento

Mouse

- click → Dispara ao clicar em um elemento
- mouseenter → Dispara ao entrar na area de um elemento
- mouseleave → Dispara ao sair na area de um elemento
- mousemove → Dispara ao se mover na area de um elemento ou seus filhos
- mouseover → Dispara ao ser movido para a area de um elemento ou seus filhos
- mouseout → Dispara ao ser removido da area de um elemento ou seus filhos

Teclado

Teclar é dividido em 4 eventos:

- keydown → que é quando apertamos uma tecla
- keypress → quando o computador identifica que uma tecla foi pressionada
- input → o texto esta sendo inserido e alterando o campo de texto
- keyup → a tecla é levantada

Usualmente utilizamos o evento `input` para fazer a identificação de alteração de texto em elementos.

Formulários

- reset → Dispara se algum elemento pai for um `<form>` e um `<button type="reset">` for clicado
- submit → Dispara em algumas ocasiões caso tenha algum pai que seja um `<form>`:
 - `<input type="submit">` → Dispara apenas em inputs com tipo "submit"
 - `<button>` → Dispara por padrão ao clique
- change → É disparado em `<input>`, `<select>` e `<textarea>` quando é identificada alguma alteração no valor inserido nesses elementos.
 - [Leia mais](#) sobre `change` e `input`

© Curso Online de React do Zero ao Pro
Desenvolvido por Gustavo Vasconcellos e EBAC Online