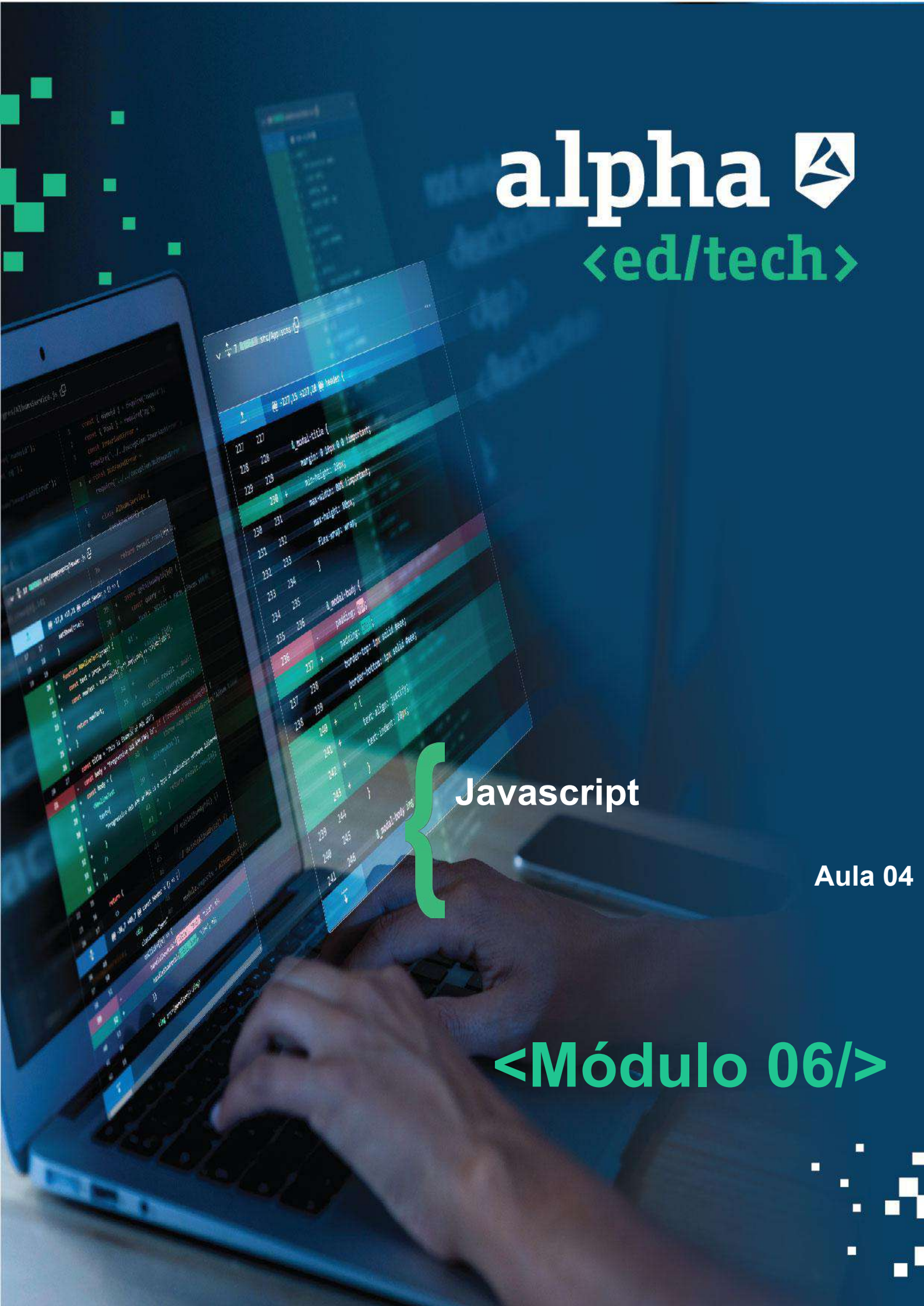

alpha
<ed/tech>
Javascript
Aula 04
<Módulo 06/>

Métodos funcionais de array

Sumário

Métodos funcionais de array

Introdução

Arrays possuem alguns métodos chamados "métodos funcionais", ou "métodos de alta ordem".

O nome é porque esses métodos exigem que você passe uma função para eles, parecido com o `addEventListener` onde você precisa especificar uma função.

A rigor, você não precisaria desses métodos.

Quando explicarmos cada um deles, mostraremos como seria um código equivalente (que faz a mesma coisa) *sem usar* o método.

Portanto eles não são obrigatórios, tem como fazer as coisas sem eles.

Mas esses métodos têm algumas vantagens:

- Comunicam claramente o que está sendo feito, é fácil de ler o código e entender o que está acontecendo.

  Se você evitar um desses métodos, fazendo um código equivalente sem eles, o código ficará mais complexo de escrever.

- O código fica mais curto com eles do que sem eles.

- Evitam bugs porque fazem loops implícitos, você não precisa programar o loop que vai acontecer.

  E aquilo que você não programa, não dá erro 😉

- São extremamente comuns em códigos javascript existentes.

  E mais comuns ainda ao usar bibliotecas de frontend como React, Angular ou Vue (veremos React no futuro).

Método forEach()

Introdução

O método `forEach` de arrays permite fazer iterações analogamente ao que um comando **`for`** consegue fazer.

Não tem uma vantagem clara de um em relação ao outro, na maior parte é uma questão de preferência.

Por exemplo, considere o seguinte loop com um **`for`** clássico para ter uma referência de algo já conhecido:

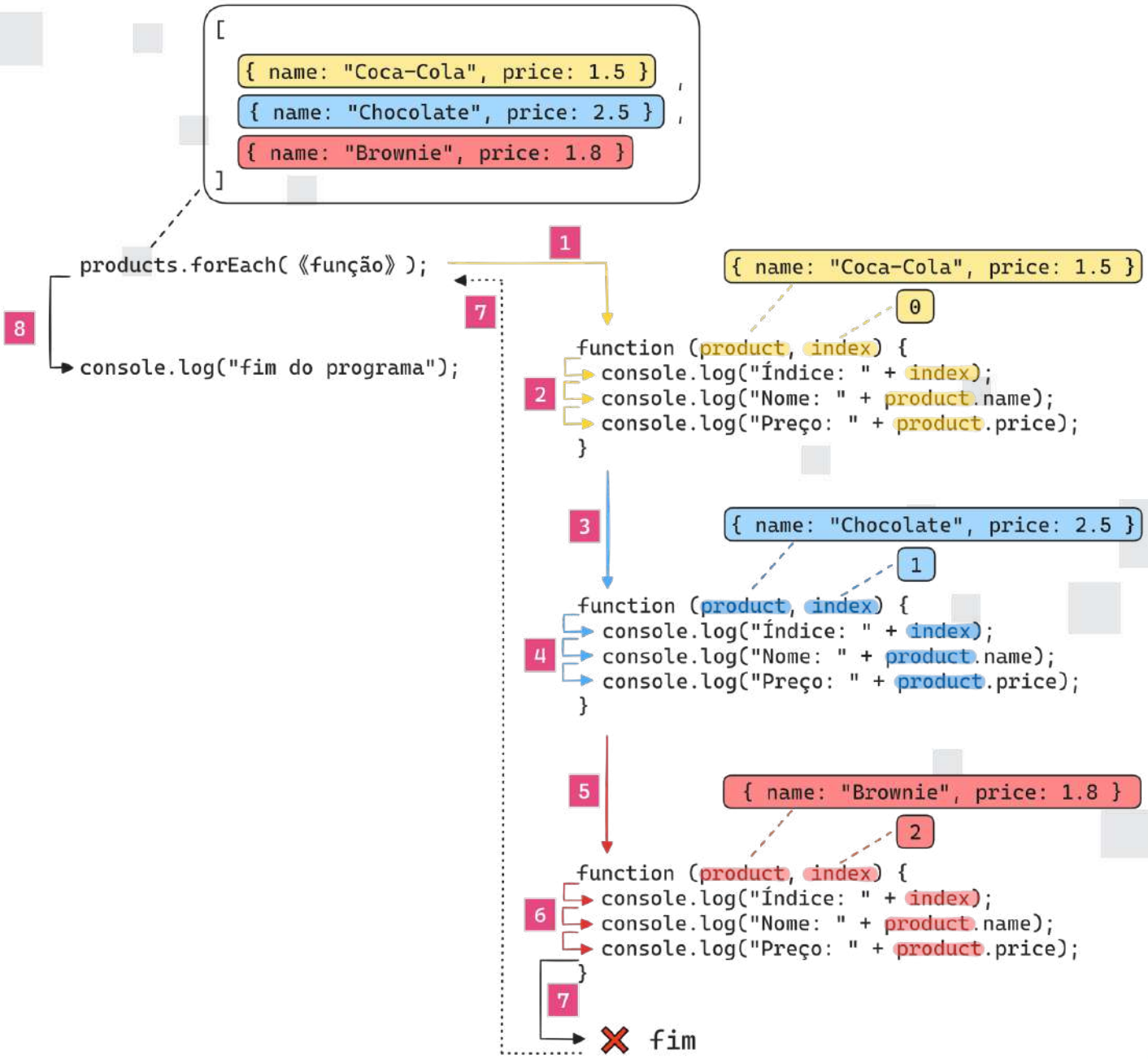
```
// array de exemplo
const products = [
  { name: "Coca-Cola", price: 1.5 },
  { name: "Chocolate", price: 2.5 },
  { name: "Brownie", price: 1.8 },
];

for (let i = 0; i < products.length; i++) {
  const product = products[i];

  // em vez de console.log, poderia ser outro
  // código qualquer
  console.log("Índice: " + i);
  console.log("Nome: " + product.name);
  console.log("Preço: " + product.price);
}

console.log("fim do programa");
```

O mesmo loop pode ser feito com o método `forEach`:

```
products.forEach(function (product, index) {
  console.log("Índice: " + index);
  console.log("Nome: " + product.name);
  console.log("Preço: " + product.price);
});
```

O diagrama da execução desse código é (números em rosa indicam a ordem da execução):

Método forEach()



A função anônima é executada 3 vezes (porque o array tem 3 elementos).

Método forEach()

Parâmetros

Em cada execução, a função recebe a informação de *qual é o elemento* e *qual é o índice dele* por meio dos dois **Parâmetros** da função (que, lembrando, são simplesmente duas variáveis locais da função).

O nome de cada parâmetro é arbitrário, nós os chamamos de `product` e `index` mas poderiam ter quaisquer outros nomes. Afinal são simplesmente duas variáveis locais, nós decidimos os nomes das variáveis.

Mas a ordem é fundamental: o primeiro dos parâmetros vai ser o elemento (nesse caso é um objeto porque temos um array de objetos), e o segundo vai ser o índice (um número).

O código `console.log` foi só para exemplificar.

Você pode ter qualquer código dentro da função, da mesma maneira que podia ter qualquer código dentro das chaves do `for`.

Se no seu código você não precisa saber qual é o *índice* do elemento, pode declarar a função sem o segundo parâmetro, ou seja:

```
products.forEach(function(product) {
  ...
});
```

Mas não é possível o contrário: declarar a função somente com `index` sem o `product`.

Se você tentar:

```
products.forEach(function(index) {
  ...
});
```

Vai funcionar, mas não da forma como você acha.

A variável `index` será o *elemento*, não o *índice*.

Isso acontece porque o javascript não se importa com o nome dos parâmetros da função, somente com a ordem deles (como já dito):

- O primeiro parâmetro é o elemento
- O segundo parâmetro (se houver) é o índice

Então ao declarar `function(index)`, como a regra é que "o primeiro parâmetro é o elemento", a variável `index` será o elemento (não o índice).

Método forEach()

Na verdade a função pode ter também um terceiro parâmetro , mas geralmente não é útil, você pode ler sobre ele na documentação da MDN sobre o `forEach`.

Sincronicidade

O `forEach` lembra um pouco o `addEventListener`, veja como a forma de escrever é parecida:

- `...addEventListener("click", 《função》);`
- `...forEach(《função》);`

Não estamos dizendo que as funcionalidades deles são parecidas, porque nisso eles são completamente diferentes: um é para eventos de clique (e outros eventos), o outro é para fazer repetições.

Estamos comparando a forma de escrever, a sintaxe deles é parecida.

Apesar de a sintaxe ser parecida, tem uma diferença muito importante na execução:

- O `addEventListener` não faz a função ser executada *agora*.
  Ele somente registra a função para ser executada no futuro, quando o clique acontecer.
  Isso é chamado de "código assíncrono" porque a função só vai ser executada no futuro.

- Já o `forEach` faz a função ser executada *agora*, imediatamente (e várias vezes).
  Isso é chamado de "código síncrono" porque a função é executada imediatamente.
  No diagrama, veja que a ordem dos passos (números em rosa) é:
  Primeiro executar a função 3 vezes (passos nº 1 a nº 6).
  Só depois disso , no passo nº 7 (linha pontilhada) volta para a linha de código inicial e prossegue para o `console.log("fim do programa")` que é o próximo comando (nº 8).

Método forEach()

Função nomeada ou função anônima

Mostramos como exemplo o uso do `forEach` com uma função anônima.

Mas não precisa ser uma função anônima, poderia ser uma função nomeada:

```
function printProduct(product, index) {
  console.log("Índice: " + index);
  console.log("Nome: " + product.name);
  console.log("Preço: " + product.price);
}

products.forEach(printProduct);
console.log("fim do programa");
```

O diagrama de execução é o mesmo: o `forEach` vai executar 3 vezes a função, e depois disso será executado o `console.log("fim do programa")`.

Se na sua aplicação existe somente 1 lugar onde você precisa do `forEach`, pode ser mais fácil usar uma função anônima.

Assim pelo menos você não precisa inventar um nome para a função.

Por outro lado, se há vários lugares na aplicação onde o mesmo `forEach` é usado, então vale a pena ter uma função nomeada.

Uma função nomeada permite economizar código, pois em todos os lugares você poderá usar o mesmo comando `products.forEach(printProduct)` em vez de repetir (copiar e colar) o código inteiro da função (se fosse anônima) em todos os lugares.

Aliás, esse é um preceito da *engenharia de software* chamado DRY, sigla para "don't repeat yourself" / "não seja repetitivo".

Ou seja, evite repetir o mesmo código em vários lugares da aplicação.

Método forEach()

Arrow function

Existe uma segunda sintaxe para funções anônimas que é um pouco mais curta para escrever.
Retire a palavra "function" antes dos parâmetros e adicione "=>" depois deles, ou seja:

```
function (element, index) {
    ...
}
```

→

```
(element, index) => {
    ...
}
```

Essa sintaxe da direita é uma função anônima também, chamada de "arrow function" por causa do símbolo da seta `=>`.
Ela pode ser usada como qualquer função anônima.
Por exemplo, em vez de escrever assim:

```
products.forEach(function (product, index) {
  console.log("Índice: " + index);
  console.log("Nome: " + product.name);
  console.log("Preço: " + product.price);
});
```

Você pode escrever assim:

```
products.forEach((product, index) => {
  console.log("Índice: " + index);
  console.log("Nome: " + product.name);
  console.log("Preço: " + product.price);
});
```

Quando a arrow function só tem 1 parâmetro, não precisa dos parênteses em volta do parâmetro (não precisa, mas pode colocar se quiser):

```
...
// só tem 1 parâmetro, não precisa dos parênteses
products.forEach(product => {
  console.log("Nome: " + product.name);
  console.log("Preço: " + product.price);
});
```

E se o código da função consistir de 1 único comando, não precisa das chaves:

```
// o código da função é só 1 comando (console.log),
// então não precisa das chaves (mas pode colocar se quiser).
// Note que ao omitir as chaves é *obrigatório*
// omitir também o ponto-e-vírgula depois do comando.
products.forEach((product) => console.log("Nome: " + product.name));
```

Método forEach()

Como dito, você pode usar arrow function em qualquer lugar onde usaria uma função anônima "convencional", inclusive num ouvinte de evento:

```
button.addEventListener("click", () => {
  // código do ouvinte de evento
});
```

Porém, dentro de uma arrow function, a palavra **`this`** *não funciona da mesma forma* que numa função convencional (não vamos entrar em detalhes).

Então se seu ouvinte de evento faz uso do **`this`**, *não use arrow function*.

Método map()

Introdução

O método `map()` de array permite transformar um array em outro array.

Mantendo o exemplo do array de produtos:

```
[
  { name: "Coca-Cola", price: 1.5 },
  { name: "Chocolate", price: 2.5 },
  { name: "Brownie", price: 1.8 },
];
```

O `map()` pode transformá-lo num array dos preços dos produtos (array de números):

```
[1.5, 2.5, 1.8];
```

Ou num array dos nomes dos produtos:

```
["Coca-Cola", "Chocolate", "Brownie"];
```

Isso seria útil por exemplo se cada objeto produto tivesse muitas propriedades (nome, preço, imposto, código de barras, estoque, etc.) mas você quisesse mostrar ao usuário somente a lista de nomes (omitindo todo o restante desnecessário das informações do produto).

Vamos fazer esse segundo exemplo em código:

Primeiro se não existisse o `map()`, você poderia fazer essa transformação da seguinte forma:

```
const products = [
  { name: "Coca-Cola", price: 1.5 },
  { name: "Chocolate", price: 2.5 },
  { name: "Brownie", price: 1.8 },
];

// array que terá os nomes dos produtos
const newArray = [];

for (let i = 0; i < products.length; i++) {
  const product = products[i];

  // colocar o nome do produto no novo array
  newArray.push(product.name);
}

console.log(newArray); // ["Coca-Cola", "Chocolate", "Brownie"];
```

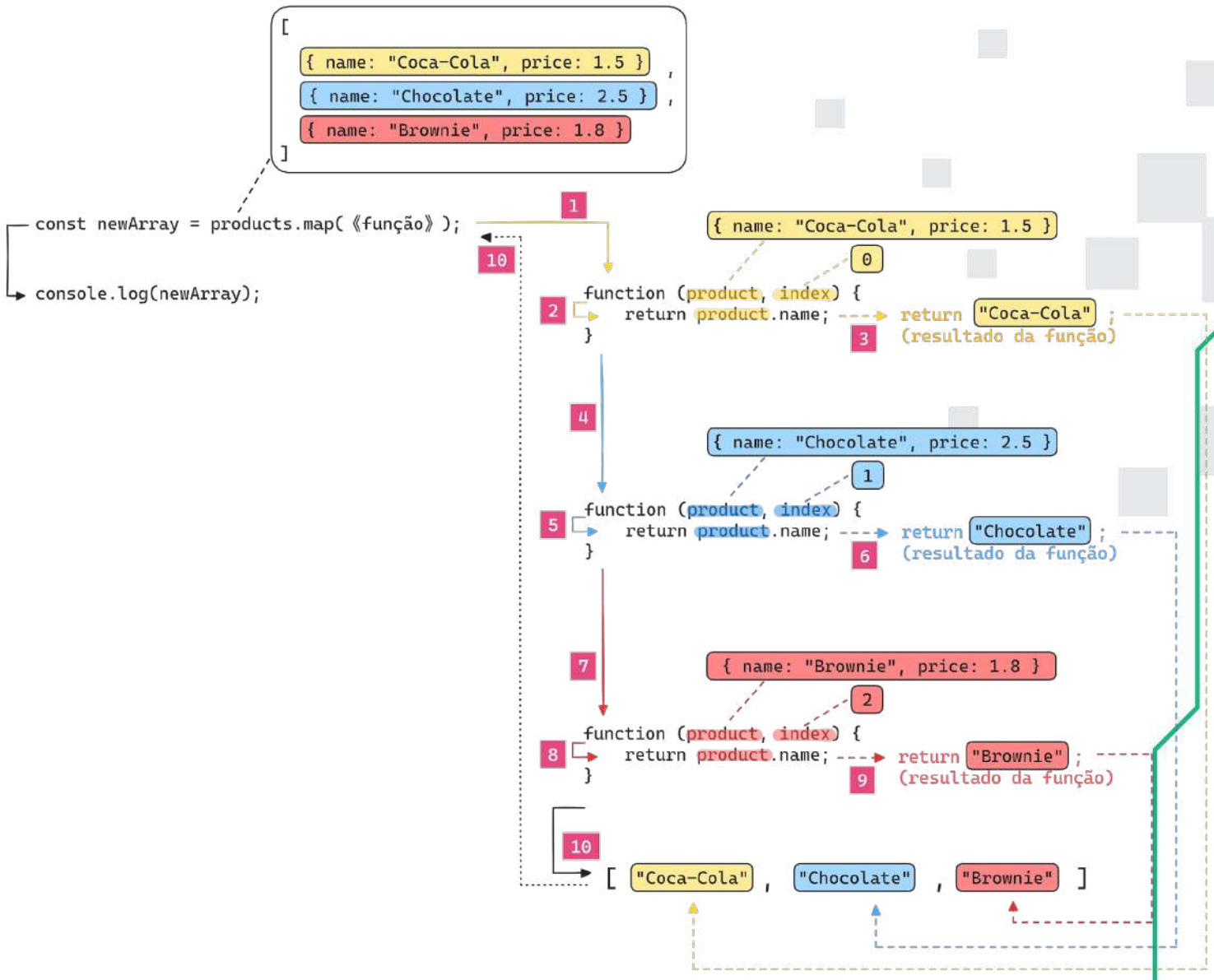
Esse código cria um novo array `newArray` que armazena os nomes dos produtos. Note que o array original `products` não é alterado (ele continua sendo um array de objetos).

Método map()

Com o método `map()`, a mesma transformação seria feita assim:

```
const newArray = products.map(function (product, index) {
  return product.name;
});

console.log(newArray); // ⇒ ["Coca-Cola", "Chocolate", "Brownie"];
```

O código é executado da seguinte forma (explicação na seções seguintes):



Antes de explicar o diagrama, tem um comando novo aí que é o **`return`**, vejamos o que ele faz primeiro.

Método map()

Comando return

Vamos deixar de lado o código do `map()` por um instante, para relembrar primeiro o que é o *resultado* (*valor de retorno*) de uma função, e como funciona o comando **`return`**.

1: Resultado de funções pré-definidas

Antes mesmo de *criar* suas próprias funções, você já *usava* funções pré-definidas do javascript, como `prompt`, `alert` e `Number`.

Ao chamar uma função, ela dá algum *resultado*, também chamado *valor de retorno*.

Por exemplo:

```
> Number("10")
<· 10
```

```
> prompt("Digite uma palavra")
<· 'cozinha'
```

O número `10` é o resultado (valor de retorno) da função `Number`, e a string `"cozinha"` é o resultado da função `prompt`.

Por outro lado, algumas funções não dão resultado, por exemplo `alert`, veja no Console:

```
> alert("oi mundo");
<· undefined
```

A função `alert` mostra uma tela com a mensagem, mas depois disso o resultado da função é **`undefined`** (quer dizer que não tem resultado).

Método map()

2: Resultado de funções criadas pelo programador

Essa discussão foi sobre o valor de retorno (resultado) de funções pré-definidas como `alert`, `Number`, etc.

Mas e as funções que você cria ? Por exemplo as funções ouvintes de evento, funções anônimas, arrow functions, etc ?

Elas *também podem dar algum resultado*, basta saber o comando necessário para isso, que é o **`return`**.

Esse comando pode ser usado de duas maneiras:

1. **`return`** 《Expressão》;
2. **`return`**;

No primeiro caso, o javascript vai calcular o resultado da 《Expressão》, o qual se tornará o resultado da função.

No segundo caso, o resultado da função é **`undefined`** (ou seja, função sem resultado).

Exemplo de uso do **`return`** numa função ouvinte de evento:

```
// exemplo: uma aplicação com um input e um botão
button.addEventListener("click", function () {
  const value = input.value;
  return Number(value);
});
```

Quando o usuário clica no botão, a função executa e seu resultado é o valor numérico digitado no input.

Por exemplo, se o usuário digitou 123 no input, fica assim:

- **`return`** `Number(value);` ⇒ **`return`** `Number("123");` ⇒ **`return`** `123;` ⇒ o resultado da função é o número `123`

3: Utilidade do return

Sobre o exemplo anterior:

```
button.addEventListener("click", function () {
  const value = input.value;
  return Number(value);
});
```

Você poderia se perguntar: *"onde vai ser usado o resultado da função ? Para quê ele serve ?"*

A resposta: esse resultado não será usado em lugar nenhum, não tem nenhuma utilidade, nem aparece em lugar algum (nem no HTML nem no Console).

E é por isso que quando estudamos sobre ouvintes de evento, nunca mencionamos sobre o **`return`**, porque ele simplesmente não serve para quase nada num ouvinte de evento.

Mas nesta aula não estamos analisando funções ouvintes de evento.

Estamos analisando uma função usada dentro do `map()`.

E nesse contexto existe utilidade para o resultado (valor de retorno) da função.

Veremos em seguida, na explicação sobre o diagrama de execução do `map()`.

Método map()

Na verdade existe uma utilidade possível para o `return` dentro de um ouvinte de evento.

Pois além de o `return` definir qual é o resultado da função, ele também faz outra coisa: *interrompe* a execução da função.

Por exemplo, no ouvinte de evento abaixo, as duas últimas linhas nunca são executadas:

```
button.addEventListener("click", function () {
  const value = input.value;
  // o return interrompe a função
  return Number(value);

  // então os comandos após o return
  // não são executados, como se não existissem
  console.log(1);
  console.log(2);
});
```

Isso pode ser aproveitado para simplificar alguns tipos de código.

Mas como não é importante para esta aula, não vamos entrar nesse detalhe.

Para ver como tirar proveito do `return` para simplificar o código de um ouvinte de evento, volte na aula sobre condicionais `if` e `else` onde mostramos exemplos disso.

Curiosidade: Quando uma função não tem o comando `return`, o resultado dela é `undefined`.

Método map()

Explicação do diagrama de execução do map()

Agora que você sabe que o `return` serve para definir qual o resultado da função, vamos voltar ao exemplo da Introdução.

A explicação daquele diagrama de execução é como segue:

- Nos passos 1 a 3 é a primeira iteração.

  O comando `return product.name` fica `return "Coca-Cola"`.

  Então a string `"Coca-Cola"` é o resultado da função anônima, o `map()` armazena esse resultado (no final veremos como ele é usado, por enquanto só lembre que o resultado fica armazenado).

- Nos passos 4 a 6 é a segunda iteração.

  O comando `return product.name` fica `return "Chocolate"`.

  Então a string `"Chocolate"` é o resultado da função anônima, o `map()` armazena esse resultado.

- Nos passos 7 a 9 é a última iteração.

  O comando `return product.name` fica `return "Browney"`.

  Então a string `"Browney"` é o resultado da função anônima, o `map()` armazena esse resultado.

- Finalmente no passo 10, o `map()` pega todos os resultados que ele armazenou, e constrói um novo array contendo cada resultado (na ordem das iterações).

  Esse novo array é o resultado do `map()`, esse resultado "sobe" (linha pontilhada) de volta para a linha de código inicial.

  Fica assim:

  ```
  const newArray = products.map(《função》); ⇒ const newArray = ["Coca-Cola", "Chocolate", "Brownie"];
  ```

- Agora que acabou a execução do `map()`, o código vai para o próximo comando, que é um `console.log` no exemplo (passo 11).

Então, em resumo, o método `map()` faz o seguinte:

- Executa a função para cada elemento do array.
- Cada execução gera um resultado por meio do comando `return`.
- No final o `map()` coleta todos os resultados num novo array.
- Esse novo array é o resultado do `map()`.

  Obs: o array `products` não é alterado, continua sendo o mesmo array de produtos.

  O `newArray` é um novo `array`.

  Eles são independentes, alterar algo em um não muda no outro.

Método map()

Não confunda os dois resultados:

- Existe o *resultado da iteração*, que é definido pelo `return` em cada iteração. Esses resultados são coletados pelo `map()`.
- E existe o *resultado do map*, que é um array construído a partir dos resultados coletados das iterações.

Mais exemplos

```
const list = [1, 2, 3, 4, 5];

// note que estamos omitindo o segundo parâmetro
// index porque não precisamos dele no código
console.log(
  list.map(function (item) {
    return item * 2;
  })
); // [2, 4, 6, 8, 10]

const list2 = ["ana", "bia", "carlos"];

console.log(
  list2.map((item) => {
    return item.toUpperCase();
  })
); // ["ANA", "BIA", "CARLOS"]

// Os mesmos dois exemplos, mas agora
// usando arrow function (funciona
// da mesma maneira):

console.log(
  list.map((item) => {
    return item * 2;
  })
); // [2, 4, 6, 8, 10]

console.log(
  list2.map((item) => {
    return item.toUpperCase();
  })
); // ["ANA", "BIA", "CARLOS"]
```

Método map()

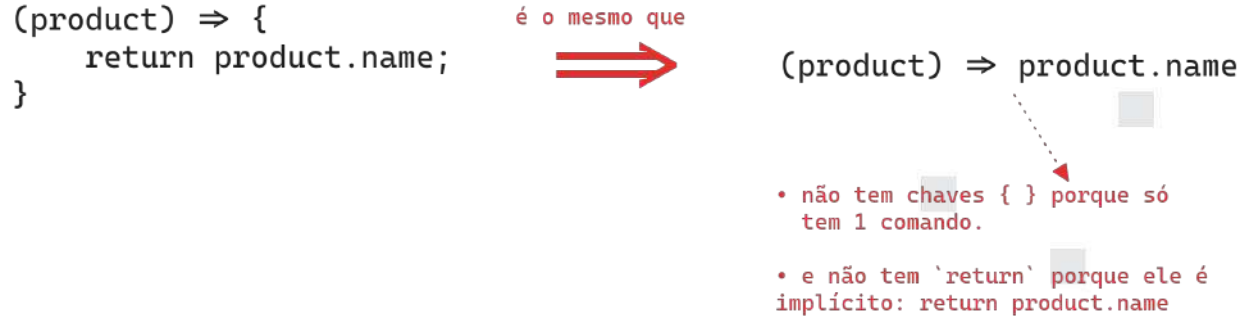
Arrow function com return implícito

No espírito de facilitar a escrita do código, escrevendo menos código para fazer a mesma coisa, podemos aproveitar uma características das arrow functions.

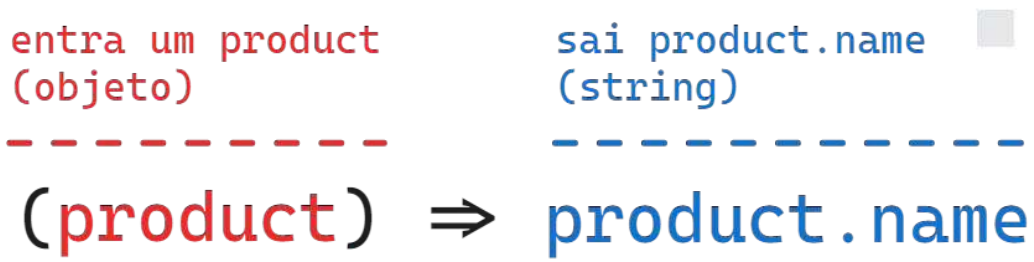
Lembre-se que, quando uma arrow function tem somente 1 comando, é desnecessário colocar as chaves { } da arrow function.

Mas tem mais uma simplificação:

Quando a arrow function não tem chaves { }, o único comando dela é implicitamente um comando de `return`:



Uma maneira de pensar nessa arrow function é que ela é uma "máquina de conversão" de algum dado de entrada para algum dado de saída:



Vamos ver mais exemplos:

```
const products = [
  { name: "Coca-Cola", price: 1.5 },
  { name: "Chocolate", price: 2.5 },
  { name: "Brownie", price: 1.8 },
];

console.log(products.map((product) => product.name)); // ["Coca-Cola", "Chocolate",
"Brownie"]

const list = [1, 2, 3, 4, 5];

console.log(list.map((item) => item * 2)); // [2, 4, 6, 8, 10]

const list2 = ["ana", "bia", "carlos"];

console.log(list2.map((item) => item.toUpperCase())); // ["ANA", "BIA", "CARLOS"]
```

Método map()

O comando `return` só é implícito numa arrow function sem chaves.

Em qualquer outro tipo de função, omitir o comando `return` não vai funcionar da mesma forma.

Por exemplo, você poderia pensar (incorretamente) que o seguinte código funcionaria:

```
const products = [
  { name: "Coca-Cola", price: 1.5 },
  { name: "Chocolate", price: 2.5 },
  { name: "Brownie", price: 1.8 },
];

console.log(
  products.map((product) => {
    product.name; // não colocamos o "return"
  })
);
// você acha que vai imprimir
// ["Coca-Cola", "Chocolate", "Brownie"] ?? Não vai !
```

Como a função tem chaves `{ }`, o `return` não é implícito, então o `product.name` não é considerado como resultado da função.

E quando uma função não tem um resultado explícito, o resultado dela é `undefined`.

Então esse código vai imprimir `[undefined, undefined, undefined]`, porque em cada uma das 3 iterações do `map()` o resultado da função é `undefined`.

Método findIndex()

Introdução

Reveja o exemplo da lista de tarefas (com prioridade e botão de deletar) da aula passada.

Naquela ocasião, tínhamos uma variável global `tasks` que era um array de tarefas `[{text: "lavar a louça", id: 7142, priority: "high"}, ...]`.

No HTML cada tarefa era um `<li id="7142">`, ou seja, o `<li>` no HTML possui o mesmo ID do objeto tarefa no array.

Ao clicar no botão de deletar, o ouvinte de evento acessava o HTML para ler o ID do `<li>`, e então procurava no array `tasks` o objeto com o mesmo ID.

Simplificadamente, essa busca no array era assim:

```
// exemplo do ID que estamos buscando no array
const taskId = 7142;

let index;

for (let i = 0; i < tasks.length; i++) {
  const task = tasks[i];

  if (task.id === taskId) {
    index = i;
  }
}

console.log("O índice da tarefa é: " + index);
```

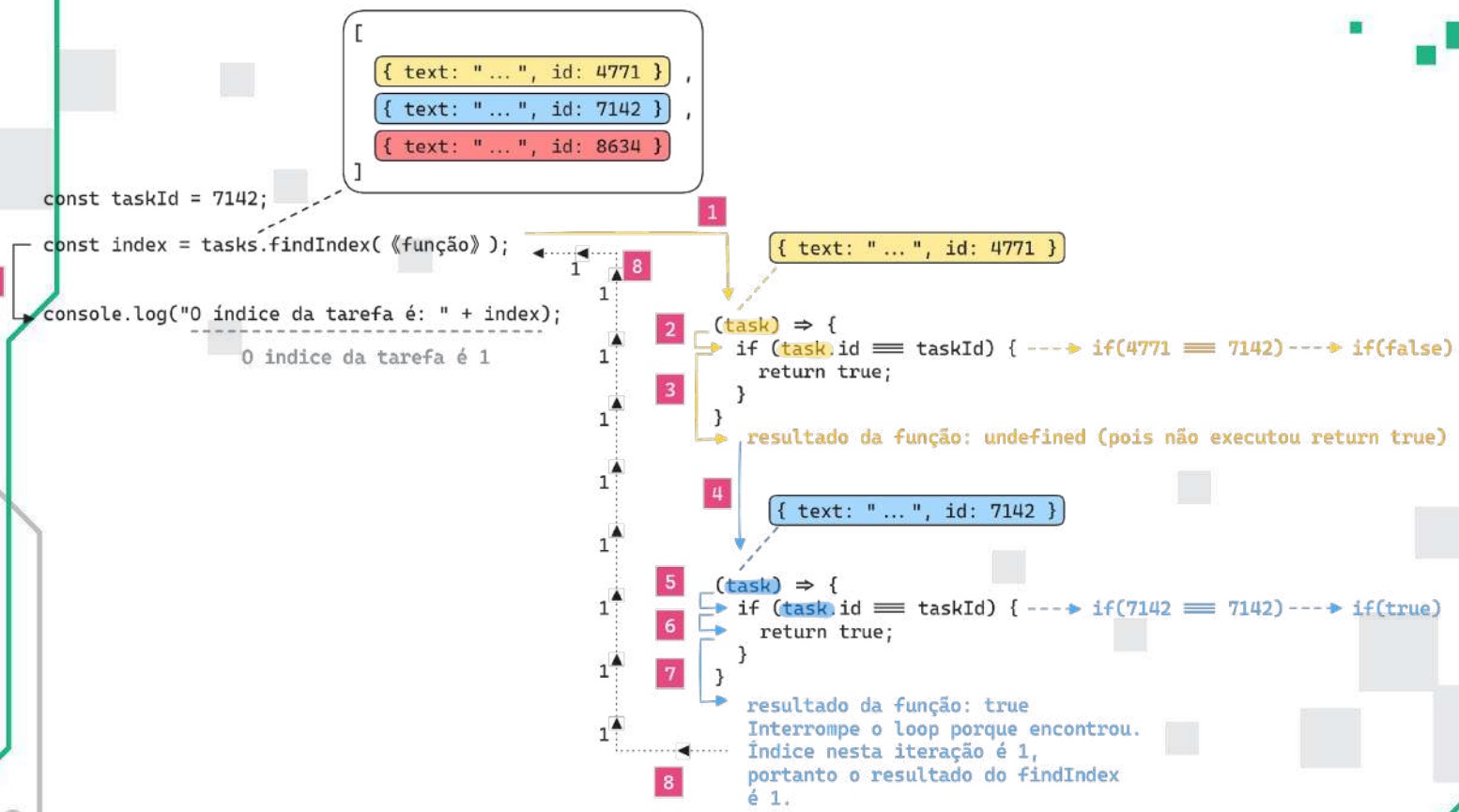
Existe um método de array que permite fazer a mesma coisa, encontrar o índice da tarefa buscada.

O código é o seguinte:

```
// exemplo do ID que estamos buscando no array
const taskId = 7142;

// omitimos o segundo parâmetro da função
// porque não precisamos dele no código
const index = tasks.findIndex((task) => {
  if (task.id === taskId) {
    return true;
  }
});

console.log("O índice da tarefa é: " + index);
```

Segue um diagrama de como esse código funciona:

Método findIndex()



O `findIndex` executa a função várias vezes da mesma maneira que o `forEach` e o `map`, cada iteração é um elemento diferente do array.

E em cada iteração, o objetivo da função é dizer se o elemento que está sendo visto nesta iteração (parâmetro `task`) corresponde ou não ao elemento buscado.

- Na primeira iteração, a tarefa no array tem ID 4771, portanto ela não é a tarefa desejada.

  A função sinaliza ao `findIndex` que o elemento atual não é o desejado, essa sinalização é feita por meio do resultado da função:

  Como o **`if`** (passo 2) não foi executado, o **`return true`** foi pulado (passo 3), então a função ficou com resultado **`undefined`**.

  O `findIndex` vê esse resultado **`undefined`** e entende que o elemento atual não é o desejado, e continua o loop (passo 4).

- Na segunda iteração, a tarefa no array tem o ID procurado (7142).

  A função deve sinalizar que a tarefa foi encontrada, isso é feito por meio do **`return true`** (o **`if`** no passo 5 é executado, e a função dá resultado **`true`** no passo 6).

  Quando o `findIndex` vê que a função teve resultado **`true`**, ele entende que foi encontrado o elemento buscado.

  Então ele interrompe o loop nesta iteração.

Finalmente, a iteração que encontrou o elemento foi a iteração `1` (`1` é o índice da iteração).

Esse índice `1` é o resultado do `findIndex`, que "sobe" (linha pontilhada no passo 8) de volta para a linha de código inicial, que fica assim:

- **`const`** `index = products.findIndex(《função》);` ⇒ **`const`** `index = 1;`

Pois `1` é o resultado do `findIndex`.

Método findIndex()

Em resumo:

- O `findIndex` executa a função várias vezes igual ao `forEach`.
- Nas iterações em que a função não dá resultado `true`, o `findIndex` continua o loop.
- Quando numa certa iteração a função dá resultado `true`, o `findIndex` interrompe o loop nesta iteração.
- O índice desta iteração se torna o resultado do `findIndex`.

Num cenário hipotético onde o ID procurado não existe no array (nenhuma tarefa do array tem o ID buscado), nenhuma iteração vai executar o `return true`.

Nesse caso o `findIndex` faz o loop inteiro, entende que o elemento buscado não foi encontrado, e o resultado do `findIndex` é `-1`.

Existe outro método parecido chamado `find()`, veja a documentação da MDN para saber mais.

Método findIndex()

Mais exemplos

```
const list = ["a", "b", "c", "d"];

// imprime 2
// iteração 0: "a" === "c" --> false --> resultado undefined
// iteração 1:  "b" === "c" --> false --> resultado undefined
// iteração 2:  "c" === "c" --> true --> return true --> resultado 2
console.log(
  list.findIndex((element) => {
    if (element === "c") {
      return true;
    }
  })
);

// imprime -1
// iteração 0: "a" === "z" --> false --> resultado undefined
// iteração 1:  "b" === "z" --> false --> resultado undefined
// iteração 2:  "c" === "z" --> false --> resultado undefined
// iteração 3:  "d" === "z" --> false --> resultado undefined
// acabou o array e nenhuma iteração retornou true --> resultado -1
console.log(
  list.findIndex((element) => {
    if (element === "z") {
      return true;
    }
  })
);

const list2 = [100, 50, 45, 60];

// imprime 1
console.log(
  list2.findIndex((element) => {
    if (element === 50) {
      return true;
    }
  })
);

const list3 = [];

// imprime -1 porque tem 0 elementos,
// logo não realiza nenhuma iteração,
// entende-se que não encontrou
console.log(
  list3.findIndex(《qualquer função》);
)
```

Método findIndex()

Resultados truthy e falsy

Explicamos que a função anônima sinaliza ao `findIndex` se ela encontrou ou não o elemento desejado, e essa sinalização é feita por meio do valor de retorno:

- **`undefined`**: não encontrou o elemento desejado.
- **`true`**: encontrou o elemento desejado.

Na verdade para sinalizar "não encontrou", não precisa necessariamente ser **`undefined`**, pode ser qualquer valor "falsy" (qualquer valor que é interpretado como **`false`**).

Por exemplo o código abaixo também funcionaria, porque a função retorna **`false`** quando quer sinalizar que não encontrou:

```
const taskId = 7142;

const index = tasks.findIndex((task) => {
  if (task.id === taskId) {
    return true;
  } else {
    return false;
  }
});

console.log("O índice da tarefa é: " + index);
```

Outra maneira mais curta é:

```
const taskId = 7142;

const index = tasks.findIndex((task) => {
  // 1º passo: faz a comparação task.id === taskId.
  // Se ela deu true, o comando fica: `return true;`
  // Se ela deu false, o comando fica: `return false;`
  // De todo modo funciona, veja:
  // Tarefa não encontrada -> comparação dá false -> return false
  // Tarefa encontrada -> comparação dá true -> return true
  return task.id === taskId;
});

console.log("O índice da tarefa é: " + index);
```

Método findIndex()

Arrow function com return implícito

O exemplo das tarefas pode ser escrito ainda mais curto usando uma arrow function sem as chaves { }, da seguinte forma:

```
const taskId = 7142;

// supondo que tasks é
// [
//    { text: "...", id: 4771 },
//    { text: "...", id: 7142 },
//    { text: "...", id: 8634 }
// ],
// o `index` será 1
const index = tasks.findIndex((task) => task.id === taskId);

console.log("O índice da tarefa é: " + index);
```

Como você sabe, numa arrow function sem chaves { }, o return é implícito (então aquele código é realmente **return** task.id === taskId).

Portanto na primeira iteração do findIndex, task é o primeiro objeto do array e a comparação fica 4771 === 7142 ⇒ **false**.

Esse **false** é o resultado da arrow function, que o findIndex interpreta da mesma maneira que **undefined**, ou seja, não encontrou o elemento procurado.

Na segunda iteração, task é o segundo objeto do array e a comparação fica 7142 === 7142 ⇒ **true**.

Esse **true** é o resultado da arrow function, e como você sabe, o findIndex entende que o elemento foi encontrado, portanto ele interrompe o loop e o resultado do findIndex é 1.

Outros exemplos:

```
const list = ["a", "b", "c", "d"];

// imprime 2
// iteração 0: "a" === "c" --> false --> resultado false
// iteração 1:  "b" === "c" --> false --> resultado false
// iteração 2:  "c" === "c" --> true --> resultado (da arrow function) true -->
resultado (do findIndex) 2
console.log(list.findIndex((element) => element === "c")); // 2

// imprime -1
// iteração 0: "a" === "z" --> false --> resultado false
// iteração 1:  "b" === "z" --> false --> resultado false
// iteração 2:  "c" === "z" --> false --> resultado false
// iteração 3:  "d" === "z" --> false --> resultado false
// acabou o array e nenhuma iteração retornou true --> resultado (do findIndex) -1
console.log(list.findIndex((element) => element === "z")); // -1

const list2 = [100, 50, 45, 60];

console.log(list2.findIndex((element) => element === 50)); // 1
```

Método filter()

Introdução

Este método serve para construir um novo array contendo somente elementos selecionados.
Por exemplo, digamos que você tem um array de produtos:

```
const products = [
  { name: "Coca-Cola", price: 1.5 },
  { name: "Chocolate", price: 2.5 },
  { name: "Brownie", price: 1.8 },
];
```

A aplicação tem um filtro de preço, o usuário quer ver somente os produtos com preço abaixo de R$2,00.
E você quer construir um novo array contendo somente esses produtos.
Sem o método `filter()`, você poderia fazer isso com um loop com `if` dentro:

```
// array que vai armazenar somente os produtos filtrados
// (produtos que passam no filtro)
const filteredProducts = [];

for (let i = 0; i < products.length; i++) {
  const product = products[i];

  // se o preço é menor que R$2,00
  // incluir esse produto no novo array
  if (product.price < 2) {
    filteredProducts.push(product);
  }
}

// [ { name: 'Coca-Cola', price: 1.5 }, { name: 'Brownie', price: 1.8 } ]
console.log(filteredProducts);
```

Já com o `filter()`, fica assim:

```
// omitimos o segundo parâmetro `index` em
// function(product, index) porque
// não precisamos dele no código
const filteredProducts = products.filter(function (product) {
  if (product.price < 2) {
    return true;
  }
});

// filteredProducts será:
// [ { name: 'Coca-Cola', price: 1.5 }, { name: 'Brownie', price: 1.8 } ]

// Esse é um novo array, o array products fica intacto, inalterado
```

Método filter()

O mecanismo do `filter()` é uma mistura do `findIndex()` com o `map()`:

- Como o `findIndex()`, em cada iteração a função deve retornar **`true`** ou um valor "falsy" (**`undefined`**, **`false`**, etc.)

  Ela deve retornar **`true`** para sinalizar que o elemento atual passou no filtro, caso contrário o `filter()` entende que o elemento não passou no filtro.

  Mas o `findIndex()` interrompe o loop na primeira iteração que retorna **`true`**, já o `filter()` executa o loop sempre até o final.

- E como o `map()`, o `filter()` coleta os resultados da função em cada iteração.

  Mas o `map()` monta um novo array contendo esses resultados diretamente, já o `filter()` usa esses resultados meramente como um seletor: nas iterações com resultado **`true`**, o *elemento* correspondente deve ser incluído no array final.

  Ao final do loop, o `filter()` monta um novo array contendo os elementos onde a função retornou **`true`**.

  Esse array é o resultado do `filter()`.

Mais exemplos

O código do filtro dos produtos pode ser reescrito de forma mais curta:

```
const products = [
  { name: "Coca-Cola", price: 1.5 },
  { name: "Chocolate", price: 2.5 },
  { name: "Brownie", price: 1.8 },
];

// iteração 0: 1.5 < 2 --> true --> resultado true
// iteração 1: 2.5 < 2  --> false --> resultado false
// iteração 2: 1.8 < 2  --> true --> resultado true
// no final, o `filter()` monta um novo array contendo
// os elementos onde a função retornou true:
// [{ name: "Coca-Cola", price: 1.5 }, { name: "Brownie", price: 1.8 }]
const filteredProducts = products.filter((product) => product.price < 2);
```

Método filter()

Outros exemplos:

```
const list = ["a", "b", "c", "d", "c"];

// iteração 0: "a" === "c" --> false --> resultado false
// iteração 1:  "b" === "c" --> false --> resultado false
// iteração 2:  "c" === "c" --> true --> resultado true
// iteração 3: "d" === "c" --> false --> resultado false
// iteração 4: "c" === "c" --> true --> resultado true
// resultado do filter: ["c", "c"]
console.log(list.filter((element) => element === "c")); // ["c"]

// iteração 0: "a" === "z" --> false --> resultado false
// iteração 1:  "b" === "z" --> false --> resultado false
// iteração 2:  "c" === "z" --> false --> resultado false
// iteração 3:  "d" === "z" --> false --> resultado false
// resultado do filter: []
console.log(list.findIndex((element) => element === "z")); // []

const list2 = [100, 50, 45, 60];

console.log(list2.findIndex((element) => element >= 50)); // [100, 50, 60]
```

Método sort()

Introdução

Este método serve para ordenar um array ("sort" significa "ordenar", não "sortear").

Mas ele é razoavelmente diferente dos outros métodos vistos até agora, em pelos menos dois pontos:

1. Nos outros métodos, você escrevia uma função `function (element, index)`.

   Onde o primeiro parâmetro era o elemento do array e o segundo era o índice desse elemento (segundo parâmetro opcional).

   No `sort` os parâmetros não são desse jeito.

2. Nos outros métodos, você sabia que a função seria invocada 1 vez para cada elemento do array.

   No `sort` não é assim.

Lógica geral de ordenação

Antes de pensar em como é o uso do `sort()` em código, vamos pensar só na lógica (visão geral) que ele usa para fazer a ordenação.

Considere o array de produtos:

```
const products = [
  { name: "Coca-Cola", price: 1.5 },
  { name: "Chocolate", price: 2.5 },
  { name: "Brownie", price: 1.8 },
];
```

Se queremos ordenar em ordem *decrescente* de preço (por exemplo), podemos começar comparando o primeiro e o segundo produto, procurando responder a seguinte pergunta:

- *Qual dos dois deve vir antes ?* (Coca-Cola, Chocolate)

Resposta: como a Coca-Cola custa menos que o Chocolate, é o Chocolate que deve vir antes.

Então já sabemos a ordem relativa desses dois produtos:

- Chocolate –> Coca-Cola

Agora podemos comparar a Coca-Cola com o Brownie, procurando responder a mesma pergunta:

- *Qual dos dois deve vir antes ?* (Coca-Cola, Brownie)

Resposta: como a Coca-Cola custa menos que o Brownie, é o Brownie que deve vir antes.

Então já sabemos a ordem relativa desses dois produtos:

- Brownie –> Coca-Cola

Até agora temos duas ordens relativas já conhecidas:

- Chocolate –> Coca-Cola
- Brownie –> Coca-Cola

Note que isso ainda não é suficiente ! Porque não sabemos se o Chocolate ou o Brownie deve vir antes.

Então ainda precisamos comparar eles dois, procurando responder a mesma pergunta:

- *Qual dos dois deve vir antes ?* (Chocolate, Brownie)

Método sort()

Resposta: como o Chocolate custa mais que o Brownie, é o Chocolate que deve vir antes.
Agora sim temos a ordem completa:

- Chocolate –> Brownie –> Coca-Cola

Em outras palavras, o array ordenado com esse critério (ordem decrescente de preço) é:

```
[
  { name: "Chocolate", price: 2.5 },
  { name: "Brownie", price: 1.8 },
  { name: "Coca-Cola", price: 1.5 },
];
```

Código do método sort()

O código real, usando o método `sort()`, para fazer a ordenação do array de produtos em ordem decrescente de preço é:

```
const sortedArray = products.sort((product1, product2) => {
  // se o produto 1 deve vir antes
  if (product1.price > product2.price) {
    return -1;
  }
  // se o produto 2 deve vir antes
  else if (product1.price < product2.price) {
    return +1;
  }
  // se os preços são iguais
  else {
    return 0;
  }
});

// [
//   { name: "Chocolate", price: 2.5 },
//   { name: "Brownie", price: 1.8 },
//   { name: "Coca-Cola", price: 1.5 }
// ]
console.log(sortedArray);
```

Método sort()

O `sort()` imita a lógica que descrevemos na seção anterior:

1. Pega dois produtos.
2. Compara os dois para ver qual deve vir antes.

   Esta é a parte que a sua função faz !

   Ela faz a comparação e sinaliza qual dos dois deve vir antes (explicação adiante).
3. De acordo com o resultado, pode já ser suficiente para determinar a ordem de todos os produtos.

   Se for esse o caso, acabou o `sort()`, e o resultado é o array ordenado.

   Mas se ainda não há informação suficiente para saber a ordem relativa de todos os produtos, o `sort()` volta ao passo 1.

   Ou seja, escolhe outros dois produtos onde ainda há dúvida (qual deve vir antes) e repete o processo, até que ele tenha feito comparações suficientes para determinar a ordem de todos os produtos.

Os passos 1 e 3 não aparecem no código, porque isso é o mecanismo subjacente ("por trás das cortinas") que o `sort()` realiza.

O único passo visível no código é o passo 2, que corresponde a sua função `(product1, product2) => ...`

Funciona assim:

Para começar, o `sort()` escolhe dois elementos do array, digamos que sejam Coca-Cola e Chocolate (como na seção anterior), e invoca a sua função.

O primeiro parâmetro da função é o objeto `{name: "Coca-Cola", price: 1.5}` e o segundo parâmetro é o objeto `{name: "Chocolate", price: 2.5}`.

O objetivo da função é sinalizar para o `sort()` qual dos dois elementos deve vir antes. Isso é feito por meio do resultado (valor de retorno) da função:

- Se o `product1` deve vir *antes* do `product2`, a função deve retornar um número negativo.

  Qualquer negativo serve, só precisa ser negativo. Geralmente padronizamos o `-1`.
- Se o `product1` deve vir *depois* do `product2`, a função deve retornar um número positivo.

  Qualquer positivo serve, só precisa ser positivo. Geralmente padronizamos o `1`.
- Última possibilidade: se o `product1` e o `product2` são equivalentes (têm o mesmo preço), a função deve retornar `0`.

  No nosso array de exemplo isso não acontece, já que não há dois produtos com o mesmo preço.

Na primeira iteração, a função retorna `1`, o que significa que a Coca-Cola deve vir depois do Chocolate.

Agora o `sort()` pega outros dois produtos (objetos) do array e invoca a sua função de novo, com novos valores para `product1` e `product2`.

A função faz o mesmo processo: decide qual deve vir antes e retorna um número negativo/zero/positivo para sinalizar.

E assim por diante, sua função é invocada várias vezes até que o `sort()` tenha coletado informação suficiente para ser capaz de determinar a ordem que você queria para o array.

Finalmente, o resultado do `sort()` é o array ordenado.

Método sort()

Como você sabe, os nomes dos parâmetros `product1` e `product2` são arbitrários, pode dar o nome que quiser.

Não sabemos exatamente *quais* pares de produtos o `sort()` vai pegar para comparar.

Isso fica a critério dos navegadores web que implementaram o código interno do `sort()`.

Para nós que estamos programando em javascript, não faz diferença saber isso.

Pois nos basta que a função de ordenação esteja correta, ou seja, retorne um número negativo/zero/positivo para sinalizar corretamente qual dos dois produtos deve vir primeiro.

Método sort()

Mais exemplos

```
const list = [30, 1, -3, 0, 40, 100];

// [-3, 0, 1, 30, 40, 100]
console.log(
  list.sort((num1, num2) => {
    if (num1 < num2) {
      return -1;
    } else if (num1 > num2) {
      return 1;
    } else {
      return 0;
    }
  })
);

// [100, 40, 30, 1, 0, -3]
console.log(
  list.sort((num1, num2) => {
    if (num1 < num2) {
      return 1;
    } else if (num1 > num2) {
      return -1;
    } else {
      return 0;
    }
  })
);

const list2 = [10];

// resultado: [10]
// nem invoca a função, porque só tem 1 elemento,
// então qualquer ordenação é igual: [10]
console.log(list2.sort(《função》));
```

Método sort()

O sort() ordena “in-place”

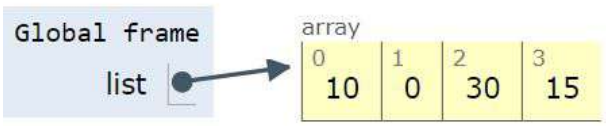
Um detalhe sobre o resultado do `sort()` criar bugs no seu código, caso você não saiba sobre esse detalhe.

Exceto pelo `sort()`, os outros métodos que retornam um array (`map` e `filter`) não têm o mesmo problema.

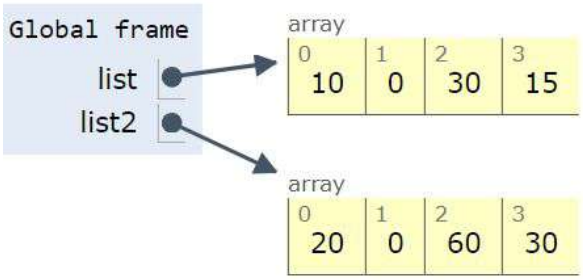
Considere um código hipotético com `map`:

```
const list = [10, 0, 30, 15];

const list2 = list.map((num) => num * 2);
```

Antes do `map`, os dados estão assim:



Depois do `map`, ficam assim:



Nenhuma surpresa: o array original é um, e o array produzido pelo `map` é outro.

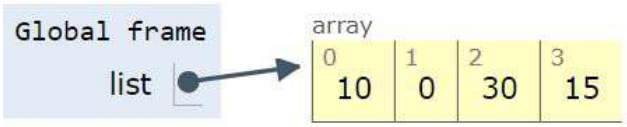
O `filter` funciona similarmente: ele cria um novo array.

Agora e se o código usar o `sort` ?

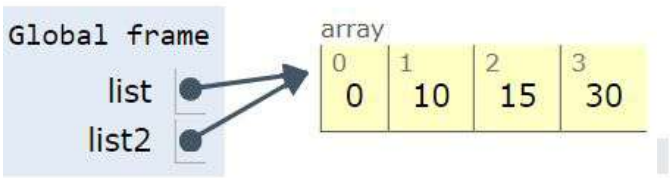
```
const list = [10, 0, 30, 15];

// ordem crescente
const list2 = list.sort((num1, num2) => {
  if (num1 < num2) {
    return -1;
  } else if (num1 > num2) {
    return 1;
  } else {
    return 0;
  }
});
```

Método sort()

Antes do `sort`, os dados estão assim:



Depois do `sort`, ficam assim:



Claramente diferente do `map` e do `filter` !

O array produzido pelo `sort` é o mesmo array original, que agora está ordenado (perceba que a ordem dos elementos mudou, como era para ser).

Isso se chama modificação "in-place" / "no lugar".

Significa que, em vez de o `sort` criar outro array independente para dispor os elementos ordenados, ele usa *o mesmo* array original.

Quando o `sort` retorna o array ordenado (que vai parar na variável `list2`), esse array é o mesmo array original.

Consequentemente, ambas as variáveis `list` (array original) e `list2` (que recebeu o resultado do `sort`) se referem ao mesmo array.

Esse fato poderia gerar um bug na sua aplicação se você achasse que `list` estaria intacto após o `sort`, quando na verdade claramente o `list` foi impactado, porque o algoritmo de ordenação operou de forma in-place.

Outros métodos de alta ordem

Introdução

Esses não são todos os métodos de alta ordem que arrays possuem.
Existem ainda:

- `flatMap()`
- `find()`
- `findLast()`
- `findLastIndex()`
- `toSorted()`
- `reduce()`
- `reduceRight()`
- `every()`
- `some()`

Você pode ler mais sobre cada um deles na documentação da MDN.

Exemplo de aplicação

Introdução: dados de filmes

Encontramos no site Kaggle uma base de dados sobre filmes, contendo informação sobre ~5000 filmes (título, gêneros, ano, orçamento, etc.):

https://www.kaggle.com/datasets/carolzhangdc/imdb-5000-movie-dataset?resource=download

Vamos fazer uma aplicação web que lista todos esses filmes e permite ao usuário ordená-los segundo algum critério (ano, título) e filtrá-los segundo algum critério (somente filmes após o ano 2000, por exemplo).

A página terá um campo para importar os dados dos filmes, e uma vez importados, haverá controles para filtro/ordenação, e uma tabela que mostra os filmes:

Exemplo de aplicação

Vamos usar o seguinte HTML e CSS:

```
<!-- 1: exibido quando a página inicia -->
<div id="import-container">
    Dados: <textarea id="import-input"></textarea>
    <br />
    <button id="import-button">Importar</button>
</div>

<!-- 2: escondido, só é exibido após a importação dos dados acima -->
<div id="presentation-container" class="hidden">
    <!-- 3: controles de filtro -->
    <strong>Filtros</strong>
    <br />
    Duração: de <input id="min-duration" /> até <input id="max-duration" />
    <br />
    Pontuação IMDB: de <input id="min-score" /> até <input id="max-score" />
    <br />
    <hr />

    <!-- 4: controles de ordenação -->
    <strong>Ordenação</strong>
    <br />
    <select id="sort-property">
        <option>Título</option>
        <option>Ano</option>
    </select>
    <select id="sort-order">
        <option>Crescente</option>
        <option>Decrescente</option>
    </select>
    <hr />

    <button id="apply-button">Aplicar</button>
    <br />

    <!-- 5: tabela de filmes -->
    <strong>Filmes</strong>
    <table>
        <thead>
            <tr>
                <th>Título</th>
                <th>Ano</th>
                <th>Pontuação</th>
                <th>Duração</th>
            </tr>
        </thead>
        <tbody id="table-body"></tbody>
    </table>
</div>
```

```
/* style.css */

.hidden {
  display: none;
}
```

Exemplo de aplicação

Formato dos dados

A base de dados no link mencionado é um arquivo no format CSV, mas, para facilitar, já fizemos a conversão desses dados para JSON.

O arquivo em formato JSON está disponível como um anexo desta aula chamado **Dados de filmes (JSON)**.

Esse arquivo contém um array de objetos, onde cada objeto é um filme.

Por exemplo, o primeiro objeto da lista é o seguinte:


```
{
  "color": "Color",
  "director_name": "James Cameron",
  "num_critic_for_reviews": "723",
  "duration": "178",
  "director_facebook_likes": "0",
  "actor_3_facebook_likes": "855",
  "actor_2_name": "Joel David Moore",
  "actor_1_facebook_likes": "1000",
  "gross": "760505847",
  "genres": "Action|Adventure|Fantasy|Sci-Fi",
  "actor_1_name": "CCH Pounder",
  "movie_title": "Avatar",
  "num_voted_users": "886204",
  "cast_total_facebook_likes": "4834",
  "actor_3_name": "Wes Studi",
  "facenumber_in_poster": "0",
  "plot_keywords": "avatar|future|marine|native|paraplegic",
  "movie_imdb_link": "http://www.imdb.com/title/tt0499549/?ref_=fn_tt_tt_1",
  "num_user_for_reviews": "3054",
  "language": "English",
  "country": "USA",
  "content_rating": "PG-13",
  "budget": "237000000",
  "title_year": "2009",
  "actor_2_facebook_likes": "936",
  "imdb_score": "7.9",
  "aspect_ratio": "1.78",
  "movie_facebook_likes": "33000"
}
```


São várias informações, mas só vamos usar `"movie_title"` (nome do filme), `"title_year"` (ano do filme), `"duration"` (duração) e `"imdb_score"` (pontuação de 0 a 10).

Exemplo de aplicação

Importação dos dados, parte 1: JSON.parse()

O usuário deve colar o conteúdo do arquivo JSON na `<textarea>` e clicar em "Importar".
A página vai:

1. Guardar o array de filmes numa variável global `moviesArray`.
2. Esconder a `<div>` de importação de dados e mostrar a `<div>` de exibição dos filmes.
3. Popular a tabela com todos os filmes da lista, sem nenhum filtro ou ordenação. Posteriormente o usuário poderá aplicar algum filtro/ordenação.

Para os passos 1 e 2, basta o seguinte código:

```
let moviesArray;
const tbody = document.getElementById("table-body");

document.getElementById("import-button").addEventListener("click", function () {
  try {
    moviesArray = JSON.parse(document.getElementById("import-input").value);
  } catch (err) {
    alert("Erro ao importar JSON");
    return;
  }

  document.getElementById("import-container").classList.add("hidden");
  document.getElementById("presentation-container").classList.remove("hidden");
});
```

Com isso, abra a página, cole os dados (demora alguns segundos porque são muitos dados) e teste a variável `moviesArray` pelo Console para verificar que deu certo:

Exemplo de aplicação

Importação dos dados, parte 2: eliminar defeitos

Falta o passo 3: ao importar os filmes, exibí-los na tabela.

Mas antes de exibir na tabela, vamos corrigir alguns problemas dos dados:

Primeiro, se você olhar bem para os dados, perceberá que alguns filmes estão com ano faltando (string vazia), por exemplo:


```
{
  "color": "Color",
  "director_name": "",
  "num_critic_for_reviews": "6",
  "duration": "24",
  "director_facebook_likes": "",
  "actor_3_facebook_likes": "",
  "actor_2_name": "",
  "actor_1_facebook_likes": "0",
  "gross": "",
  "genres": "Action|Adventure|Animation|Family|Fantasy",
  "actor_1_name": "Pablo Sevilla",
  "movie_title": "Yu-Gi-Oh! Duel Monsters            ",
  "num_voted_users": "12417",
  "cast_total_facebook_likes": "0",
  "actor_3_name": "",
  "facenumber_in_poster": "0",
  "plot_keywords": "anime|based on manga|hero|surrealism|zen",
  "movie_imdb_link": "http://www.imdb.com/title/tt0249327/?ref_=fn_tt_tt_1",
  "num_user_for_reviews": "51",
  "language": "Japanese",
  "country": "Japan",
  "content_rating": "",
  "budget": "",
  "title_year": "",
  "actor_2_facebook_likes": "",
  "imdb_score": "7.0",
  "aspect_ratio": "",
  "movie_facebook_likes": "124"
}
```

Exemplo de aplicação

E outros estão com a duração faltando (string vazia), por exemplo:


```
{
  "color": "Color",
  "director_name": "",
  "num_critic_for_reviews": "10",
  "duration": "",
  "director_facebook_likes": "",
  "actor_3_facebook_likes": "502",
  "actor_2_name": "Tuppence Middleton",
  "actor_1_facebook_likes": "1000",
  "gross": "",
  "genres": "Drama|History|Romance|War",
  "actor_1_name": "Jim Broadbent",
  "movie_title": "War & Peace            ",
  "num_voted_users": "9277",
  "cast_total_facebook_likes": "4528",
  "actor_3_name": "Lily James",
  "facenumber_in_poster": "1",
  "plot_keywords": "male frontal nudity|male nudity|tv mini series",
  "movie_imdb_link": "http://www.imdb.com/title/tt3910804/?ref_=fn_tt_tt_1",
  "num_user_for_reviews": "44",
  "language": "English",
  "country": "UK",
  "content_rating": "TV-14",
  "budget": "",
  "title_year": "",
  "actor_2_facebook_likes": "888",
  "imdb_score": "8.2",
  "aspect_ratio": "16.0",
  "movie_facebook_likes": "11000"
}
```


Para não atrapalhar, vamos jogar fora esses filmes com "defeito" nos dados, mudando o ouvinte de evento (comentários indicam as mudanças):

```
document.getElementById("import-button").addEventListener("click", function () {
  try {
    moviesArray = JSON.parse(document.getElementById("import-input").value);
  } catch (err) {
    alert("Erro ao importar JSON");
    return;
  }

  document.getElementById("import-container").classList.add("hidden");
  document.getElementById("presentation-container").classList.remove("hidden");

  // usamos o método filter duas vezes para
  // manter somente os filmes onde o ano está
  // presente e a duração está presente.
  // O resultado é guardado de volta na variável
  // moviesArray
  moviesArray = moviesArray
    .filter((movie) => movie.title_year !== "")
    .filter((movie) => movie.duration !== "");
});
```

Exemplo de aplicação

Nesse código, note que usamos dois `filter` consecutivos.
A capacidade de "concatenar" métodos dessa forma se chama **Chaining** (encadeamento).
Funciona como o diagrama abaixo sugere:

Ou seja, o primeiro `filter` produz um novo array, e esse novo array é submetido ao segundo `filter`, que produz mais um array.
Esse último array é o resultado final.
Em código, o passo a passo seria algo como:

- `moviesArray = moviesArray.filter(《 função 1 》).filter《 função 2 》)`
- `⇒ moviesArray = [《todos os filmes》].filter(《 função 1 》).filter(《 função 2 》)`
- O primeiro filter é aplicado, ficando: `⇒ moviesArray = [《somente filmes que passaram no filtro 1》].filter(《 função 2 》)`
- O segundo filter é aplicado, ficando: `⇒ moviesArray = [《somente filmes que já haviam passado no filtro 1, e que passaram no filtro 2》]`

Exemplo de aplicação

Finalmente a variável `moviesArray` recebe o array resultante dos dois filtros.
Para testar, recarregue a página e importe os dados, depois veja no Console:

```
> moviesArray
<· (4923) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
 ▼{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, …] i
  ▼ [0 … 99]
    ▶ 0: {color: 'Color', director_name: 'James Cameror
    ▶ 1: {color: 'Color', director_name: 'Gore Verbinsk
    ▶ 2: {color: 'Color', director_name: 'Sam Mendes',
    ▶ 3: {color: 'Color', director_name: 'Christopher N
```

Agora há somente 4923 filmes, originalmente havia 5043 (veja o último print do Console).

Poderíamos ter feito os dois filtros com uma única função `filter`, assim:

```
filter((movie) => movie.title_year !== "" && movie.duration !== "");
```

Mas fica mais legível separado em duas funções, é mais rápido de olhar para o código e perceber o que cada filtro está fazendo.

Exemplo de aplicação

Importação dos dados, parte 3: converter tipos de dados

Ao fazer a importação, tem mais um passo que precisamos fazer: converter os tipos de dados das propriedades que nos interessam (título, duração, ano, pontuação).

A propriedade `"movie_title"` é uma string como deveria, mas você já deve ter percebido que `"duration"`, `"title_year"` e `"imdb_score"` também são strings, deveriam ser números.

Se as deixarmos como strings, podemos ter problemas na hora de ordenar os dados.

Por exemplo, se dois filmes têm durações `"90"` e `"180"` (minutos), então claramente o primeiro filme tem duração menor, mas no javascript a comparação `"90" < "180"` dá **`false`** !

Isso acontece porque, como você já sabe, comparação entre strings é feita com base na ordem alfabética:

Como o primeiro caractere de cada string é `"9"` e `"1"`, e o 9 vem depois do 1 na ordem alfabética, o javascript considera que `"90"` é "maior" que `"180"` , não menor.

Isso não teria acontecido se tivéssemos convertido para números, pois aí sim `90 < 180` dá **`true`** como faz sentido.

Para converter essas 3 propriedades em números, vamos inserir mais uma etapa de "chaining":

Um `map` que retorna um novo objeto contendo somente as 4 propriedades que nos interessam (porque as outras não serão usadas), com tipos convertidos.

A mudança no código é a seguinte:

```
// novidade: map no final
moviesArray = moviesArray
  .filter((movie) => movie.title_year !== "")
  .filter((movie) => movie.duration !== "")
  .map((movie) => {
    return {
      title: movie.movie_title,
      duration: Number(movie.duration),
      year: Number(movie.title_year),
      score: Number(movie.imdb_score),
    };
  });
```

Ou seja:

Exemplo de aplicação

```
[
   {
       color: "Color",
       director_name: "James Cameron",
       num_critic_for_reviews: "723",
       duration: "178",
       director_facebook_likes: "0",
       actor_3_facebook_likes: "855",
       actor_2_name: "Joel David Moore",
       actor_1_facebook_likes: "1000",
       gross: "760505847",
       genres: "Action|Adventure|Fantasy|Sci-Fi",
       actor_1_name: "CCH Pounder",
       movie_title: "Avatar",
       title_year: "2009",
       imdb_score: "7.9",
       aspect_ratio: "1.78",
       movie_facebook_likes: "33000",
        ... muitas outras propriedades ...
   },
   {
       color: "Color",
       director_name: "Gore Verbinski",
       num_critic_for_reviews: "302",
       duration: "169",
       director_facebook_likes: "563",
       actor_3_facebook_likes: "1000",
       actor_2_name: "Orlando Bloom",
       actor_1_facebook_likes: "40000",
       gross: "309404152",
       genres: "Action|Adventure|Fantasy",
       actor_1_name: "Johnny Depp",
       movie_title: "Pirates of the Caribbean: At World's End",
       title_year: "2007",
       imdb_score: "7.1",
       aspect_ratio: "2.35",
       movie_facebook_likes: "0"
   },
    ...
]
```

map

```
[
   {
       title: "Avatar",
       duration: 178,
       year: 2009,
       score: 7.9
   },
   {
       title: "Pirates of the Caribbean: At World's End",
       duration: 169,
       year: 2007,
       score: 7.1
   },
    ...
]
```

Para testar que deu certo, recarregue a página, reimporte os dados, e veja no Console:

```
> moviesArray
<- (4923) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  ▼{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
   {…}, …] i
    ▼ [0 … 99]
      ▼ 0:
          duration: 178
          score: 7.9
          title: "Avatar"
          year: 2009
        ▶ [[Prototype]]: Object
      ▼ 1:
          duration: 169
          score: 7.1
          title: "Pirates of the Caribbean: At World's E
          year: 2007
        ▶ [[Prototype]]: Object
```

Exemplo de aplicação

Nos dois `filter`, usamos o `return` implícito da arrow function sem chaves.
Ou seja, em vez de:

```
// com chaves
filter((movie) => {
  return movie.title_year !== ""; // com return
});
```

Fizemos:

```
filter((movie) => movie.title_year !== ""); // sem chaves, sem return !
```

Aproveitando o fato de que a arrow function só tinha 1 comando (o `return movie.title_year !== ""`), pudemos retirar as chaves `{ }`, e ao fazer isso o `return` fica implícito.
Mas não fizemos a mesma coisa no `map`, apesar de que ele só tem 1 comando também: `return { title: ..., duration: ..., ... }`

Para fazer a mesma coisa no `map`, escreva-o da seguinte forma:

```
map((movie) => ({
  title: movie.movie_title,
  duration: Number(movie.duration),
  year: Number(movie.title_year),
  score: Number(movie.imdb_score),
}));
```

A única novidade é que precisa dos parênteses em volta do objeto: `({ title: ..., duration: ..., ... })`
Porque sem os parênteses, o javascript acha que as chaves `{ }` são chaves da função, quando na verdade são chaves do objeto.

Exemplo de aplicação

Importação dos dados, parte 4: exibir os filmes na tabela

Após importar, filtrar e mapear os dados, só falta exibí-los na tabela.

Aqui a lógica não tem nenhuma novidade, é um loop onde cada iteração cria um `<tr>` (linha da tabela).

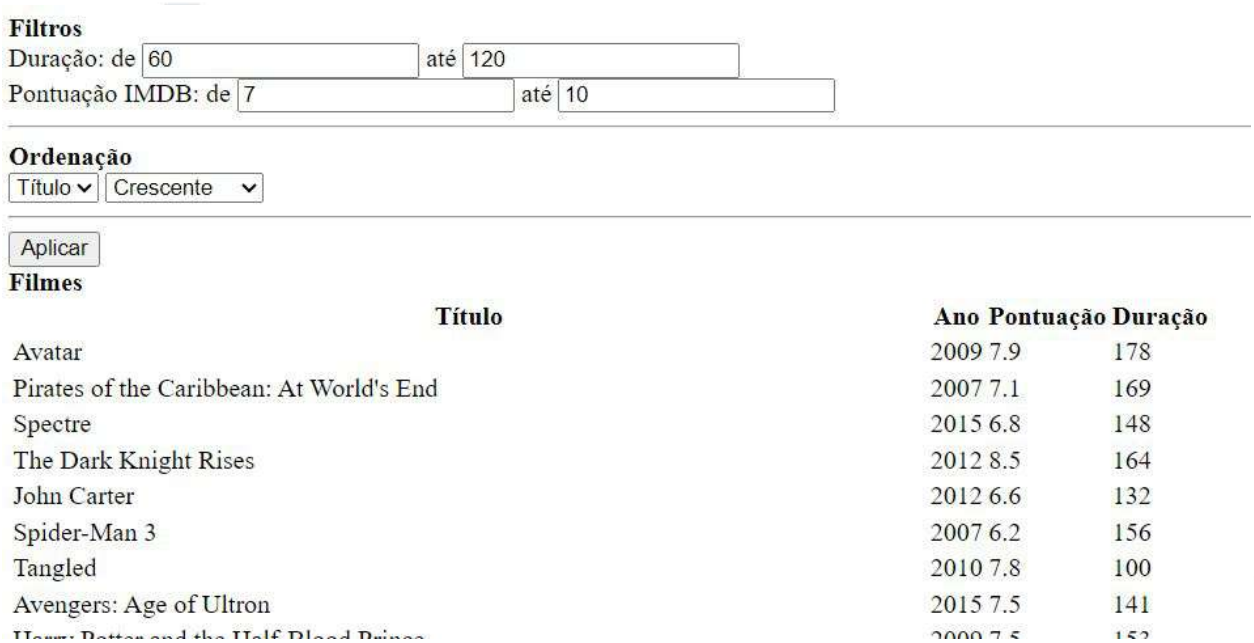
Usando o loop `forEach` para praticar, o código final completo da importação de dados fica assim:

```
let moviesArray;
const tbody = document.getElementById("table-body");

document.getElementById("import-button").addEventListener("click", function () {
  try {
    moviesArray = JSON.parse(document.getElementById("import-input").value);
  } catch (err) {
    alert("Erro ao importar JSON");
    return;
  }

  document.getElementById("import-container").classList.add("hidden");
  document.getElementById("presentation-container").classList.remove("hidden");

  moviesArray = moviesArray
    .filter((movie) => movie.title_year !== "")
    .filter((movie) => movie.duration !== "")
    .map((movie) => ({
      title: movie.movie_title,
      year: Number(movie.title_year),
      score: Number(movie.imdb_score),
      duration: Number(movie.duration),
    }));

  moviesArray.forEach((movie) => {
    const tr = document.createElement("tr");
    tr.innerHTML = `
      <td>${movie.title}</td>
      <td>${movie.year}</td>
      <td>${movie.score}</td>
      <td>${movie.duration}</td>
    `;
    tbody.appendChild(tr);
  });
});
```

Exemplo de aplicação

Filtragem, parte 1: filtro básico

Logo após importar os dados, o usuário vê a lista de filmes completa, e alguns inputs para filtrar e ordenar:



A filtragem e ordenação só acontecem ao clicar em "Aplicar".

Por enquanto, vamos fazer somente a filtragem (ordenação fica para depois).

No exemplo da imagem, ao clicar em "Aplicar", a página deve exibir somente filmes que atendam a ambas as restrições:

1. duração entre 60 e 120 minutos
2. pontuação entre 7 e 10.

Fazer a filtragem é "fácil": são dois `filter` .

Mas a tabela no HTML já tem uma lista de filmes, como faremos para exibir somente os desejados e esconder os outros ?

Uma tática simples é: apague todo o conteúdo da tabela, e refaça somente com os filmes desejados.

O código fica assim:

Exemplo de aplicação

```
document.getElementById("apply-button").addEventListener("click", function () {
  // 1: ler valores dos 4 inputs e converter para número.
  // Estamos assumindo que o usuário preencherá números
  // válidos em todos os inputs.
  // Na prática deveríamos checar que os valores de
  // minDuration, maxDuration, minScore, maxScore
  // são todos diferentes de NaN
  const minDurationInput = document.getElementById("min-duration").value.trim();
  const minDuration = Number(minDurationInput);

  const maxDurationInput = document.getElementById("max-duration").value.trim();
  const maxDuration = Number(maxDurationInput);

  const minScoreInput = document.getElementById("min-score").value.trim();
  const minScore = Number(minScoreInput);

  const maxScoreInput = document.getElementById("max-score").value.trim();
  const maxScore = Number(maxScoreInput);

  // 2: limpar a tabela
  tbody.innerHTML = "";

  // 3: fazer os dois filtros e logo em seguida o forEach
  // para preencher a tabela.
  moviesArray
    .filter(
      (movie) => movie.duration >= minDuration && movie.duration <= maxDuration
    )
    .filter((movie) => movie.score >= minScore && movie.score <= maxScore)
    .forEach((movie) => {
      const tr = document.createElement("tr");
      tr.innerHTML = `
        <td>${movie.title}</td>
        <td>${movie.year}</td>
        <td>${movie.score}</td>
        <td>${movie.duration}</td>
      `;
      tbody.appendChild(tr);
    });
});
```

Exemplo de aplicação

Quando discutimos sobre a importação de dados, fizemos uma atribuição à variável `moviesArray`:

- `moviesArray = moviesArray.filter(...).filter(...).map(...)`

Naquela ocasião, o valor original de `moviesArray` era o array de filmes "cru" (direto do textarea), e nós queríamos eliminar os filmes com informação faltante e depois converter os tipos de dados para número.

Por isso fizemos a atribuição: para trocar o valor de `moviesArray` para o array resultante da limpeza.

Agora que estamos falando da filtragem e ordenação feitas pelo usuário, nós *não queremos* trocar o valor de `moviesArray`, senão os filmes que não passam no filtro de duração/pontuação seriam perdidos para sempre.

Por isso fazemos o `filter()` *sem atribuir o resultado* à variável `moviesArray`, que continua contendo todos os filmes.

O array resultado dos dois `filter()` vai para o `forEach()`, que exibe esses filmes na tabela.

Filtragem, parte 2: inputs vazios

Fizemos a filtragem supondo que todos os inputs de filtro estavam preenchidos.

Mas na verdade queremos que seja possível que o usuário deixe alguns inputs vazios.

Por exemplo:

Ao deixar vazios os dois inputs de duração (mínimo e máximo), o usuário quer dizer que não quer filtrar por duração.

E ao deixar vazio o input de mínima pontuação, quer dizer que não há uma mínima pontuação desejada.

Em outras palavras, nessa última imagem o filtro é: "filmes ruins (pontuação ≤ 5) com qualquer duração".

Como você faria para permitir isso no código ?

Exemplo de aplicação

Um jeito fácil é pensar de outra maneira no input vazio:

- Se um input de máximo (pode ser duração ou pontuação) ficou vazio, quer dizer que não há um máximo.

  Mas outra maneira de pensar é: o máximo é `Infinity`.

  Lembramos que `Infinity` é um número especial no javascript, o "maior de todos os números".

  Ao considerar o máximo como `Infinity`, é como se não tivesse máximo, pois todos os números são menores que `Infinity`.
- E se um input de mínimo ficou vazio, consideramos que ele vale `-Infinity`, que é o menor de todos os números.

Se não se convenceu de que isso funciona, aplique essa lógica na imagem acima:

- O filtro de duração é: `duration >= -Infinity && duration <= Infinity`.

  Ou seja, qualquer duração, como desejado.
- E o filtro de pontuação é: `score >= -Infinity && score <= 5` .

  Ou seja, qualquer pontuação até 5 (não tem mínimo na prática, porque qualquer pontuação é ≥ -Infinity)

Já podemos modificar o código para implementar isso (comentários indicam as mudanças):

Exemplo de aplicação

```
document.getElementById("apply-button").addEventListener("click", function () {
  const minDurationInput = document.getElementById("min-duration").value.trim();
  // se input ficou vazio, o mínimo é -Infinity, senão é o que estiver no input
  const minDuration =
    minDurationInput === "" ? -Infinity : Number(minDurationInput);

  const maxDurationInput = document.getElementById("max-duration").value.trim();
  // se input ficou vazio, o máximo é Infinity, senão é o que estiver no input
  const maxDuration =
    maxDurationInput === "" ? Infinity : Number(maxDurationInput);

  const minScoreInput = document.getElementById("min-score").value.trim();
  // se input ficou vazio, o mínimo é -Infinity, senão é o que estiver no input
  const minScore = minScoreInput === "" ? -Infinity : Number(minScoreInput);

  const maxScoreInput = document.getElementById("max-score").value.trim();
  // se input ficou vazio, o máximo é Infinity, senão é o que estiver no input
  const maxScore = maxScoreInput === "" ? Infinity : Number(maxScoreInput);

  tbody.innerHTML = "";

  moviesArray
    .filter(
      (movie) => movie.duration >= minDuration && movie.duration <= maxDuration
    )
    .filter((movie) => movie.score >= minScore && movie.score <= maxScore)
    .forEach((movie) => {
      const tr = document.createElement("tr");
      tr.innerHTML = `
        <td>${movie.title}</td>
        <td>${movie.year}</td>
        <td>${movie.score}</td>
        <td>${movie.duration}</td>
      `;
      tbody.appendChild(tr);
    });
});
```

A filtragem está pronta, pode testar !

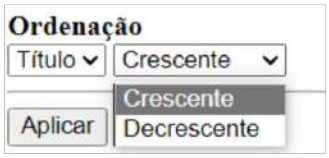
Exemplo de aplicação

Ordenação, parte 1: ordenar por título ou ano

Falta a ordenação.

Ao clicar em "Aplicar", a página vai primeiro filtrar os filmes, depois ordenar os que passaram no filtro, e então exibir na tabela.

No formulário de ordenação, o usuário escolhe se quer ordenar por Título ou Ano, e se quer em ordem Crescente ou Decrescente (de Título ou Ano conforme foi escolhido):



Vamos fazer agora a ordenação por título ou ano, deixando a escolha de crescente/decrescente para depois (por enquanto será sempre crescente).

O código fica assim (comentários indicam as mudanças):

```
document.getElementById("apply-button").addEventListener("click", function () {
  const minDurationInput = document.getElementById("min-duration").value.trim();
  const minDuration =
    minDurationInput === "" ? -Infinity : Number(minDurationInput);

  const maxDurationInput = document.getElementById("max-duration").value.trim();
  const maxDuration =
    maxDurationInput === "" ? Infinity : Number(maxDurationInput);

  const minScoreInput = document.getElementById("min-score").value.trim();
  const minScore = minScoreInput === "" ? -Infinity : Number(minScoreInput);

  const maxScoreInput = document.getElementById("max-score").value.trim();
  const maxScore = maxScoreInput === "" ? Infinity : Number(maxScoreInput);

  // 1: precisamos saber se ordenar por "Título" ou "Ano"
  const sortProperty = document.getElementById("sort-property").value;

  tbody.innerHTML = "";
```

(código continua na próxima página)

Exemplo de aplicação

(continuação do código da página anterior, ainda dentro do ouvinte de evento)

```
  // 2: colocamos um `sort()` após os `filter()`.
  // Checamos se a sortProperty é "Título" ou "Ano"
  // usando um switch.
  // No caso de "Título", ordenação crescente
  // significa ordem alfabética.
  moviesArray
    .filter(
      (movie) => movie.duration >= minDuration && movie.duration <= maxDuration
    )
    .filter((movie) => movie.score >= minScore && movie.score <= maxScore)
    .sort((movie1, movie2) => {
      switch (sortProperty) {
        case "Título":
          // lembrando o que significa o `<` para strings:
          // se o título do movie1 vem antes do título
          // do movie2 na ordem alfabética
          if (movie1.title < movie2.title) {
            return -1;
          } else if (movie1.title > movie2.title) {
            return 1;
          } else {
            return 0;
          }

        case "Ano":
          if (movie1.year < movie2.year) {
            return -1;
          } else if (movie1.year > movie2.year) {
            return 1;
          } else {
            return 0;
          }
      }
    })
    .forEach((movie) => {
      const tr = document.createElement("tr");
      tr.innerHTML = `
        <td>${movie.title}</td>
        <td>${movie.year}</td>
        <td>${movie.score}</td>
        <td>${movie.duration}</td>
      `;
      tbody.appendChild(tr);
    });
});
```

Exemplo de aplicação

Ordenação, parte 2: ordem crescente ou decrescente

Até aqui assumimos que a ordem seria crescente, agora vamos permitir que o usuário escolha.

O que muda se a ordem for decrescente ?

Olhando o código do `sort()`, onde ele tinha `return 1`, precisamos trocar para `return -1`, e vice-versa.

Podemos fazer isso com um operador ternário, o código final fica assim (comentário indicam as mudanças):

```
document.getElementById("apply-button").addEventListener("click", function () {
  const minDurationInput = document.getElementById("min-duration").value.trim();
  const minDuration =
    minDurationInput === "" ? -Infinity : Number(minDurationInput);

  const maxDurationInput = document.getElementById("max-duration").value.trim();
  const maxDuration =
    maxDurationInput === "" ? Infinity : Number(maxDurationInput);

  const minScoreInput = document.getElementById("min-score").value.trim();
  const minScore = minScoreInput === "" ? -Infinity : Number(minScoreInput);

  const maxScoreInput = document.getElementById("max-score").value.trim();
  const maxScore = maxScoreInput === "" ? Infinity : Number(maxScoreInput);

  const sortProperty = document.getElementById("sort-property").value;

  // 1: "Crescente" ou "Decrescente"
  const sortOrder = document.getElementById("sort-order").value;

  tbody.innerHTML = "";
```

(código continua na próxima página)

Exemplo de aplicação

(continuação do código da página anterior, ainda dentro do ouvinte de evento)

```
  moviesArray
    .filter(
      (movie) => movie.duration >= minDuration && movie.duration <= maxDuration
    )
    .filter((movie) => movie.score >= minScore && movie.score <= maxScore)
    .sort((movie1, movie2) => {
      switch (sortProperty) {
        case "Título":
          if (movie1.title < movie2.title) {
            // 2: se a ordem é crescente, então
            // o resultado é -1 (movie1 deve vir antes de movie2),
            // senão é 1 (movie1 deve vir depois de movie2).
            return sortOrder === "Crescente" ? -1 : 1;
          } else if (movie1.title > movie2.title) {
            // etc.
            return sortOrder === "Crescente" ? 1 : -1;
          } else {
            return 0;
          }

        case "Ano":
          if (movie1.year < movie2.year) {
            // etc.
            return sortOrder === "Crescente" ? -1 : 1;
          } else if (movie1.year > movie2.year) {
            // etc.
            return sortOrder === "Crescente" ? 1 : -1;
          } else {
            return 0;
          }
      }
    })
    .forEach((movie) => {
      const tr = document.createElement("tr");
      tr.innerHTML = `
        <td>${movie.title}</td>
        <td>${movie.year}</td>
        <td>${movie.score}</td>
        <td>${movie.duration}</td>
      `;
      tbody.appendChild(tr);
    });
});
```

Exemplo de aplicação

Evitar código repetido

A aplicação já funciona perfeitamente.

Mas vamos lembrar do princípio DRY: "don't repeat yourself" / "não se repita".

Atualmente tem um código idêntico repetido em dois lugares, veja se o encontra no código completo da aplicação:

```
let moviesArray;
const tbody = document.getElementById("table-body");

document.getElementById("import-button").addEventListener("click", function () {
  try {
    moviesArray = JSON.parse(document.getElementById("import-input").value);
  } catch (err) {
    alert("Erro ao importar JSON");
    return;
  }

  document.getElementById("import-container").classList.add("hidden");
  document.getElementById("presentation-container").classList.remove("hidden");

  moviesArray = moviesArray
    .filter((movie) => movie.title_year !== "")
    .filter((movie) => movie.duration !== "")
    .map((movie) => ({
      title: movie.movie_title,
      year: Number(movie.title_year),
      score: Number(movie.imdb_score),
      duration: Number(movie.duration),
    }));

  moviesArray.forEach((movie) => {
    const tr = document.createElement("tr");
    tr.innerHTML = `
      <td>${movie.title}</td>
      <td>${movie.year}</td>
      <td>${movie.score}</td>
      <td>${movie.duration}</td>
    `;
    tbody.appendChild(tr);
  });
});
```

(código continua na próxima página)

Exemplo de aplicação

(continuação do código da página anterior)

```
document.getElementById("apply-button").addEventListener("click", function () {
  const minDurationInput = document.getElementById("min-duration").value.trim();
  const minDuration =
    minDurationInput === "" ? -Infinity : Number(minDurationInput);

  const maxDurationInput = document.getElementById("max-duration").value.trim();
  const maxDuration =
    maxDurationInput === "" ? Infinity : Number(maxDurationInput);

  const minScoreInput = document.getElementById("min-score").value.trim();
  const minScore = minScoreInput === "" ? -Infinity : Number(minScoreInput);

  const maxScoreInput = document.getElementById("max-score").value.trim();
  const maxScore = maxScoreInput === "" ? Infinity : Number(maxScoreInput);

  const sortProperty = document.getElementById("sort-property").value;

  const sortOrder = document.getElementById("sort-order").value;

  tbody.innerHTML = "";
```

(código continua na próxima página)

Exemplo de aplicação

(continuação do código da página anterior, ainda dentro do ouvinte de evento)

```
  moviesArray
    .filter(
      (movie) => movie.duration >= minDuration && movie.duration <= maxDuration
    )
    .filter((movie) => movie.score >= minScore && movie.score <= maxScore)
    .sort((movie1, movie2) => {
      switch (sortProperty) {
        case "Título":
          if (movie1.title < movie2.title) {
            return sortOrder === "Crescente" ? -1 : 1;
          } else if (movie1.title > movie2.title) {
            return sortOrder === "Crescente" ? 1 : -1;
          } else {
            return 0;
          }

        case "Ano":
          if (movie1.year < movie2.year) {
            return sortOrder === "Crescente" ? -1 : 1;
          } else if (movie1.year > movie2.year) {
            return sortOrder === "Crescente" ? 1 : -1;
          } else {
            return 0;
          }
      }
    })
    .forEach((movie) => {
      const tr = document.createElement("tr");
      tr.innerHTML = `
        <td>${movie.title}</td>
        <td>${movie.year}</td>
        <td>${movie.score}</td>
        <td>${movie.duration}</td>
      `;
      tbody.appendChild(tr);
    });
});
```

Exemplo de aplicação

O código em questão é a função dentro dos dois `forEach`.
É o mesmo código, ele pega um `movie` (objeto filme) e o coloca na tabela do HTML.
Para não repetir esse código, vamos torná-lo uma função nomeada (em vez de anônima como está agora):

```
// "renderizar filme" (renderizar quer dizer "exibir").
function renderMovie(movie) {
  const tr = document.createElement("tr");
  tr.innerHTML = `
        <td>${movie.title}</td>
        <td>${movie.year}</td>
        <td>${movie.score}</td>
        <td>${movie.duration}</td>
      `;
  tbody.appendChild(tr);
}
```

Aí em cada lugar que usamos o `forEach`, ele será usado como `forEach(renderMovie)`.
Seguem os novos códigos dos dois ouvintes de evento:

```
document.getElementById("import-button").addEventListener("click", function () {
  try {
    moviesArray = JSON.parse(document.getElementById("import-input").value);
  } catch (err) {
    alert("Erro ao importar JSON");
    return;
  }

  document.getElementById("import-container").classList.add("hidden");
  document.getElementById("presentation-container").classList.remove("hidden");

  moviesArray = moviesArray
    .filter((movie) => movie.title_year !== "")
    .filter((movie) => movie.duration !== "")
    .map((movie) => ({
      title: movie.movie_title,
      year: Number(movie.title_year),
      score: Number(movie.imdb_score),
      duration: Number(movie.duration),
    }));

  moviesArray.forEach(renderMovie);
});
```

(código continua na próxima página)

Exemplo de aplicação

(continuação do código da página anterior)

```
document.getElementById("apply-button").addEventListener("click", function () {
  const minDurationInput = document.getElementById("min-duration").value.trim();
  const minDuration =
    minDurationInput === "" ? -Infinity : Number(minDurationInput);

  const maxDurationInput = document.getElementById("max-duration").value.trim();
  const maxDuration =
    maxDurationInput === "" ? Infinity : Number(maxDurationInput);

  const minScoreInput = document.getElementById("min-score").value.trim();
  const minScore = minScoreInput === "" ? -Infinity : Number(minScoreInput);

  const maxScoreInput = document.getElementById("max-score").value.trim();
  const maxScore = maxScoreInput === "" ? Infinity : Number(maxScoreInput);

  const sortProperty = document.getElementById("sort-property").value;

  const sortOrder = document.getElementById("sort-order").value;

  tbody.innerHTML = "";

  moviesArray
    .filter(
      (movie) => movie.duration >= minDuration && movie.duration <= maxDuration
    )
    .filter((movie) => movie.score >= minScore && movie.score <= maxScore)
    .sort((movie1, movie2) => {
      switch (sortProperty) {
        case "Título":
          if (movie1.title < movie2.title) {
            return sortOrder === "Crescente" ? -1 : 1;
          } else if (movie1.title > movie2.title) {
            return sortOrder === "Crescente" ? 1 : -1;
          } else {
            return 0;
          }

        case "Ano":
          if (movie1.year < movie2.year) {
            return sortOrder === "Crescente" ? -1 : 1;
          } else if (movie1.year > movie2.year) {
            return sortOrder === "Crescente" ? 1 : -1;
          } else {
            return 0;
          }
      }
    })
    .forEach(renderMovie);
});
```

Subprogramação

Sumário

Funções

Introdução

Dentro de uma página web, há várias situações onde você poderia querer notificar o usuário sobre o resultado de ações dele.

Por exemplo, quando ele adiciona uma tarefa numa lista de tarefas, uma notificação dizendo "tarefa adicionada com sucesso", ou "falha na criação da tarefa".

A função `alert()` pode ser usada para isso, mas a janela de alerta é feia ! E não pode ser customizada com CSS.

Seria legal se existisse alguma outra função que mostrasse um alerta mais bonito, mas não existe.

Se não existe, podemos criar essa função.

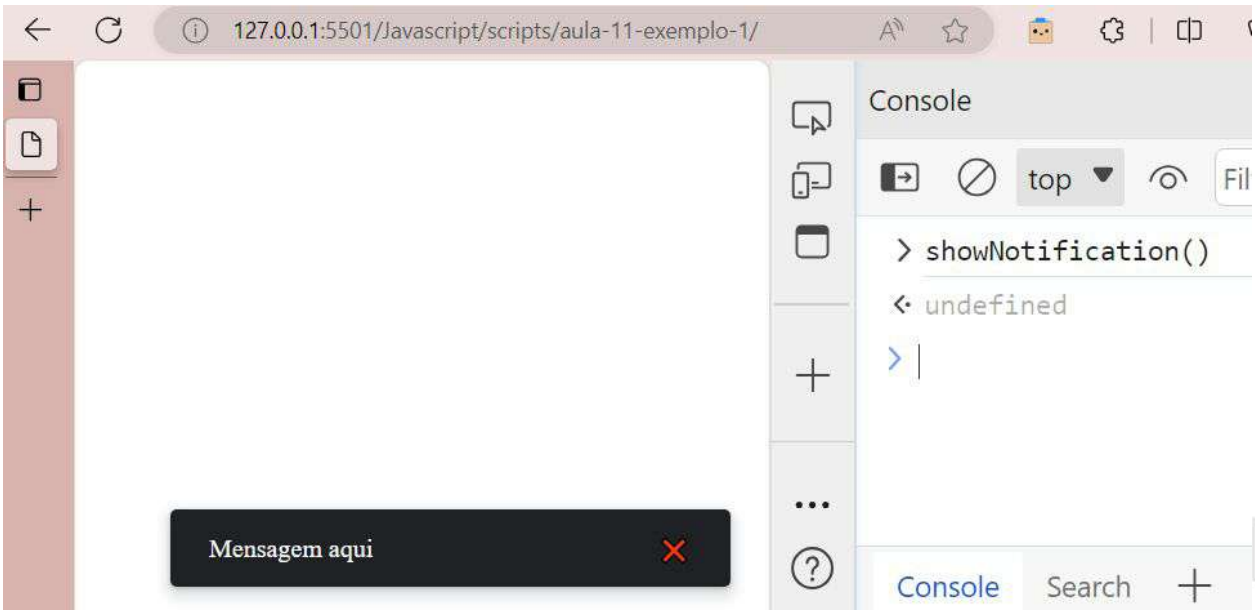
Por exemplo:

```
function showNotification() {
  // Estrutura da notificação:
  // <div><div>mensagem</div><div>❌</div></div>

  // uma div escura posicionada no canto direito inferior da tela.
  // O estilo está todo definido aqui por facilidade, mas
  // poderia estar num arquivo CSS
  const container = document.createElement("div");
  container.style.position = "absolute";
  container.style.right = "24px";
  container.style.bottom = "24px";
  container.style.minWidth = "300px";
  container.style.minHeight = "32px";
  container.style.padding = "8px 24px";
  container.style.display = "flex";
  container.style.alignItems = "center";
  container.style.justifyContent = "space-between";
  container.style.backgroundColor = "#202124";
  container.style.color = "white";
  container.style.borderRadius = "4px";
  container.style.boxShadow =
    "0 1px 3px 0 rgba(60, 64, 67, 0.302), 0 4px 8px 3px rgba(60, 64, 67, 0.149)";

  // div contendo uma mensagem
  const messageDiv = document.createElement("div");
  messageDiv.innerText = "Mensagem aqui";

  // botão de fechar (é uma div)
  const closeButton = document.createElement("div");
  closeButton.innerText = "❌";
  closeButton.style.cursor = "pointer";

  container.appendChild(messageDiv);
  container.appendChild(closeButton);

  // colocar o container no final do <body>
  // (não realmente importa onde, porque
  // a posição é "absolute")
  document.body.appendChild(container);

  closeButton.addEventListener("click", function () {
    container.remove();
  });
}
```

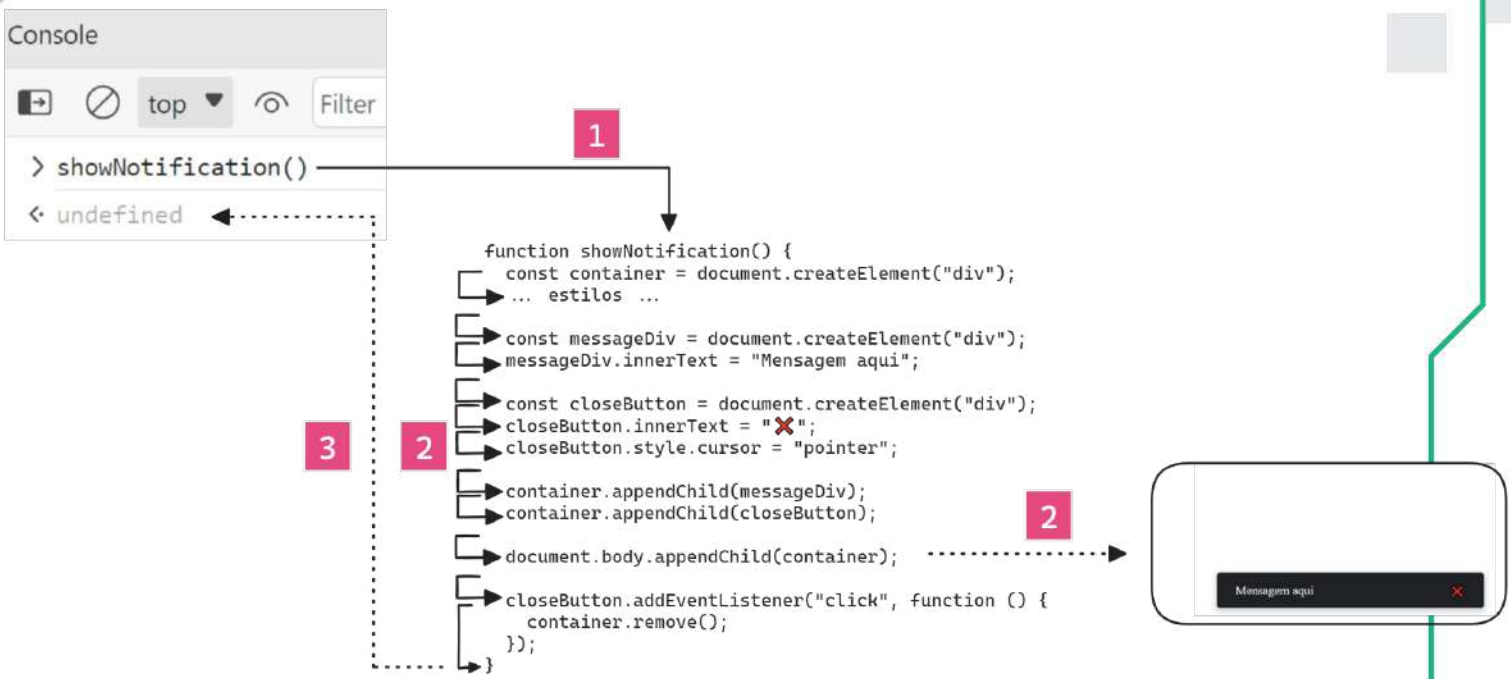
Funções

Invocação de função no Console (só para testar)

Coloque a função anterior dentro de um html vazio e abra no navegador para testar.
Ao abrir a página, evidentemente ela estará vazia.
Mas abra o Console e digite `showNotification()`, a notificação deverá aparecer:



O que aconteceu foi o seguinte (sequência de passos indicada pelos quadrados rosa):



Funções

Ao digitar no Console o nome da função com parênteses no final `showNotification()`, os parênteses têm um significado especial: é o comando de **Invocação** da função.

Significa que o navegador web vai imediatamente executar o código da função (passos 1 e 2).

No final, o código da função acaba, e o navegador volta para o Console (linha pontilhada no passo 3).

O **`undefined`** que aparece no Console é porque essa função não teve nenhum resultado (ainda veremos sobre resultado de funções).

Se você esquecer os parênteses, digitando somente `showNotification` no Console, o navegador *não vai* executar a função.

Em vez disso, ele vai mostrar o código dela:

```
> showNotification
< ƒ showNotification() {
    // Estrutura da notificação:
    // <div><div>mensagem</div><div>✕</div></div>

    // Container da notificação.
    // uma div escura posicionada no canto
    // direito inferio…
```

Não mostra o código inteiro porque é grande demais.

Isso que aparece no Console *não é uma string*, é uma representação do código da função.

Aquela letra "f" bem no início do resultado quer dizer que `showNotification` é uma função.

Funções

Invocação de função no meio do código da aplicação

Começamos ilustrando pelo Console porque é mais simples, mas não é realista.

Na realidade, você estaria programando uma página web que faz alguma coisa, e gostaria de usar a função `showNotification` no meio desse código.

Para ver como isso funciona, vamos fazer uma página de lista de tarefas simples. Coloque o seguinte HTML:

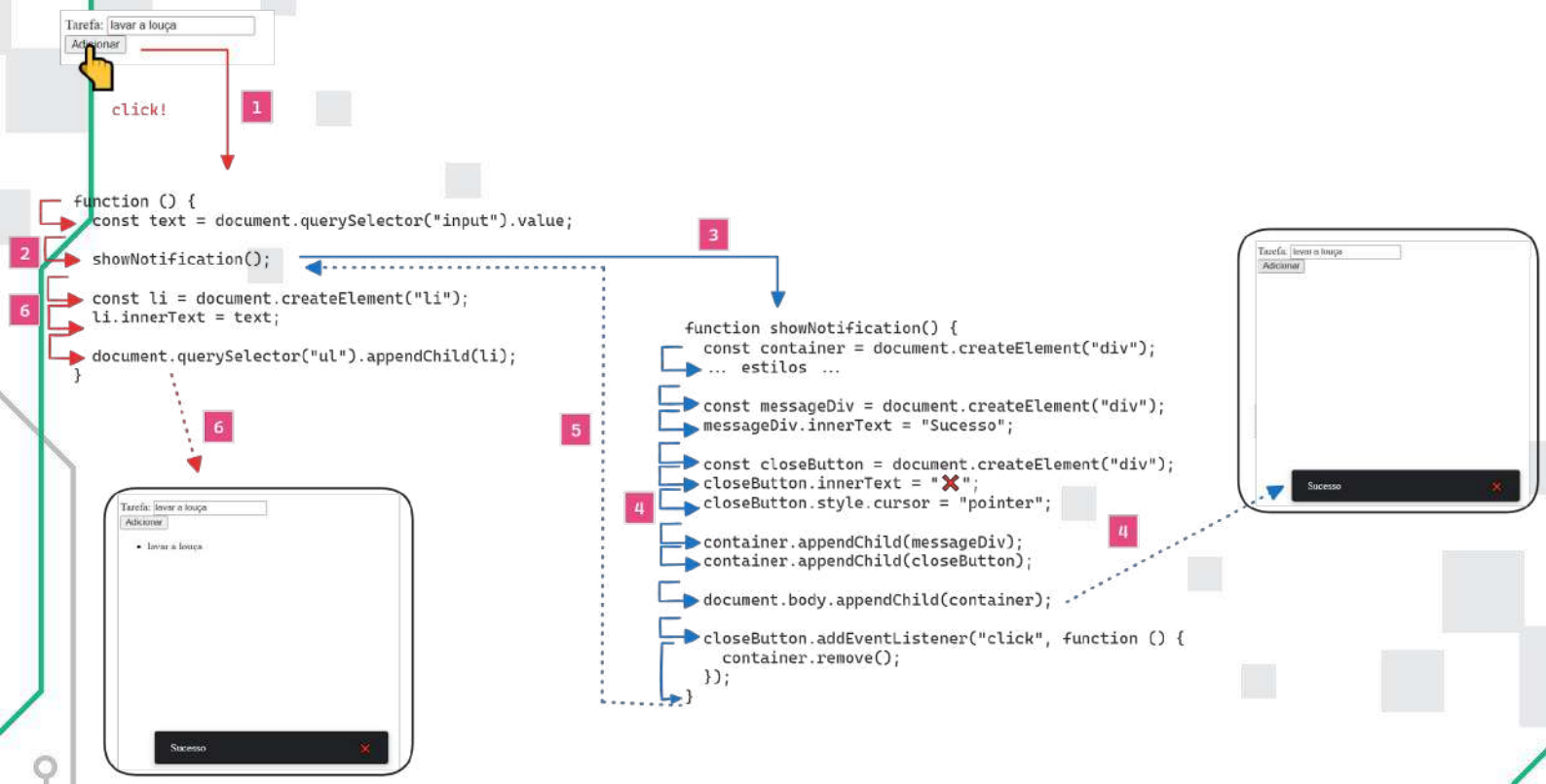
```
Tarefa: <input />
<br />
<button>Adicionar</button>
<ul></ul>
```

E o seguinte javascript:

```
document.querySelector("button").addEventListener("click", function () {
  const text = document.querySelector("input").value;

  showNotification();

  const li = document.createElement("li");
  li.innerText = text;

  document.querySelector("ul").appendChild(li);
});

function showNotification() {
  // ... código da showNotification,
  // trocando a mensagem "Mensagem Aqui"
  // por "Sucesso" para fazer mais sentido
}
```

Esse código funciona assim (explicação em seguida):

Funções



- No passo 1, o usuário clica no botão "Adicionar" e o código do ouvinte de evento começa a ser executado.

- No passo 2, a segunda linha do código invoca a função `showNotification`.
  Nesse momento, o navegador *congela* o ouvinte de evento onde estava, e passa a executar o código da `showNotification` (passo 3).

- O código da `showNotification` (passo 4) insere na página a `<div>` da notificação.

- No passo 5, o código da `showNotification` acaba.
  Então o navegador web volta para o código do ouvinte de evento, que tinha sido previamente congelado.
  Como o navegador está *voltando* da `showNotification` de volta para a segunda linha do ouvinte de evento, dizemos que a `showNotification` *retornou* (acabou).

- No passo 6, as 3 últimas linhas do ouvinte de evento são executadas, e elas inserem a `<li>` da tarefa na página.

Funções

Por isso que a técnica de invocar funções no meio do código é chamada de **Subprogramação**:

Se considerarmos que cada função é uma *tarefa*, podemos interpretar que a `showNotification` (azul) é uma *subtarefa* do ouvinte de evento (vermelho).

Quando a tarefa principal (ouvinte de evento vermelho) invoca a subtarefa (função `showNotification` em azul), a tarefa principal fica *congelada* esperando a subtarefa retornar (terminar).

E pode haver uma "subsubtarefa" também. Não é o caso do exemplo anterior, mas seria algo assim:

- Uma função (tarefa principal, por exemplo um ouvinte de evento) invoca outra função (subtarefa).
- Essa outra função invoca outra função ("subsubtarefa").

Quando a tarefa principal invoca a subtarefa, fica congelada esperando a subtarefa ser executada.

Quando a subtarefa invoca a "subsubtarefa", fica congelada esperando a "subsubtarefa" ser executada.

Aí a "subsubtarefa" termina e retoma a subtarefa.

Depois a subtarefa termina e retoma a tarefa principal.

Quando você invoca funções pré-definidas, como `Number("123")`, `prompt("Digite uma palavra")`, etc, é isso que está acontecendo:

O seu código fica congelado e o código da função é executado até acabar, depois seu código é retomado de onde parou.

A diferença é que o código das funções pré-definidas fica "oculto" dentro do sistema do navegador web.

Já o código de funções "customizadas" como a `showNotification` é você mesmo quem define.

O código do ouvinte de evento invocou a função `showNotification`.

Mas isso *não tem nenhuma influência* sobre o que você já sabe com relação a escopo de variáveis.

Ou seja, a `showNotification` não pode acessar as variáveis locais `text` e `li` do ouvinte de evento.

E o ouvinte de evento não pode acessar as variáveis locais `container`, `messageDiv` e `closeButton` da função `showNotification`.

Funções

Parâmetro

Da maneira como está, a mensagem de notificação é sempre `"Sucesso"`, essa string está definida dentro da `showNotification`.

Mas, e se no ouvinte de evento quisermos fazer uma validação do input, mostrando a mensagem "Tarefa não pode estar vazia" se o input está vazio, ou então "Tarefa adicionada com sucesso" caso contrário ?

Em outras palavras, queremos poder decidir a mensagem a ser exibida na notificação, como fazíamos com o `alert(《mensagem》)`.

O código que queremos é o seguinte (ainda não funcional):

```
document.querySelector("button").addEventListener("click", function () {
  const text = document.querySelector("input").value;

  if (text === "") {
    showNotification("Tarefa não pode ser vazia");
  } else {
    const li = document.createElement("li");
    li.innerText = text;

    document.querySelector("ul").appendChild(li);

    showNotification("Tarefa adicionada com sucesso");
  }
});
```

Se você testar esse código, verá que ele não dá nenhum erro, mas também não mostra as mensagens desejadas, e sim `"Sucesso"` sempre.

Para funcionar como esperado, a `showNotification` precisa de um tipo especial de variável chamado de **Parâmetro**.

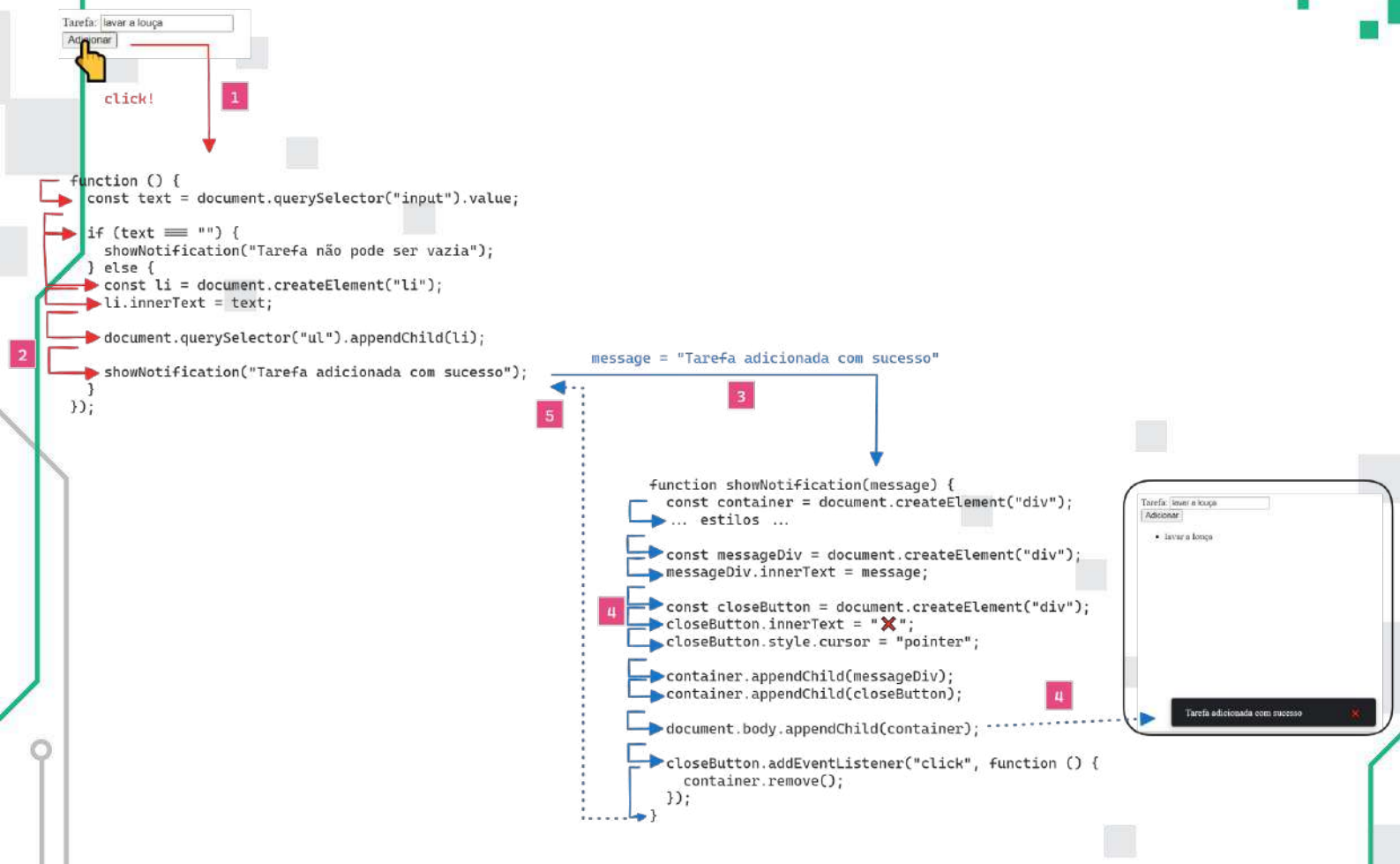
Começando pela solução, o código é o seguinte (comentários indicam as mudanças):

Funções

```
// Parâmetro `message` no meio dos parênteses.
// É uma variável que vai conter uma string.
function showNotification(message) {
  const container = document.createElement("div");
  container.style.position = "absolute";
  container.style.right = "24px";
  container.style.bottom = "24px";
  container.style.minWidth = "300px";
  container.style.minHeight = "32px";
  container.style.padding = "8px 24px";
  container.style.display = "flex";
  container.style.alignItems = "center";
  container.style.justifyContent = "space-between";
  container.style.backgroundColor = "#202124";
  container.style.color = "white";
  container.style.borderRadius = "4px";
  container.style.boxShadow =
    "0 1px 3px 0 rgba(60, 64, 67, 0.302), 0 4px 8px 3px rgba(60, 64, 67, 0.149)";

  const messageDiv = document.createElement("div");
  // Colocar a string na div
  messageDiv.innerText = message;

  const closeButton = document.createElement("div");
  closeButton.innerText = "❌";
  closeButton.style.cursor = "pointer";

  container.appendChild(messageDiv);
  container.appendChild(closeButton);

  document.body.appendChild(container);

  closeButton.addEventListener("click", function () {
    container.remove();
  });
}
```

E o código funciona da seguinte forma:
Se o usuário preencher algo corretamente no input:

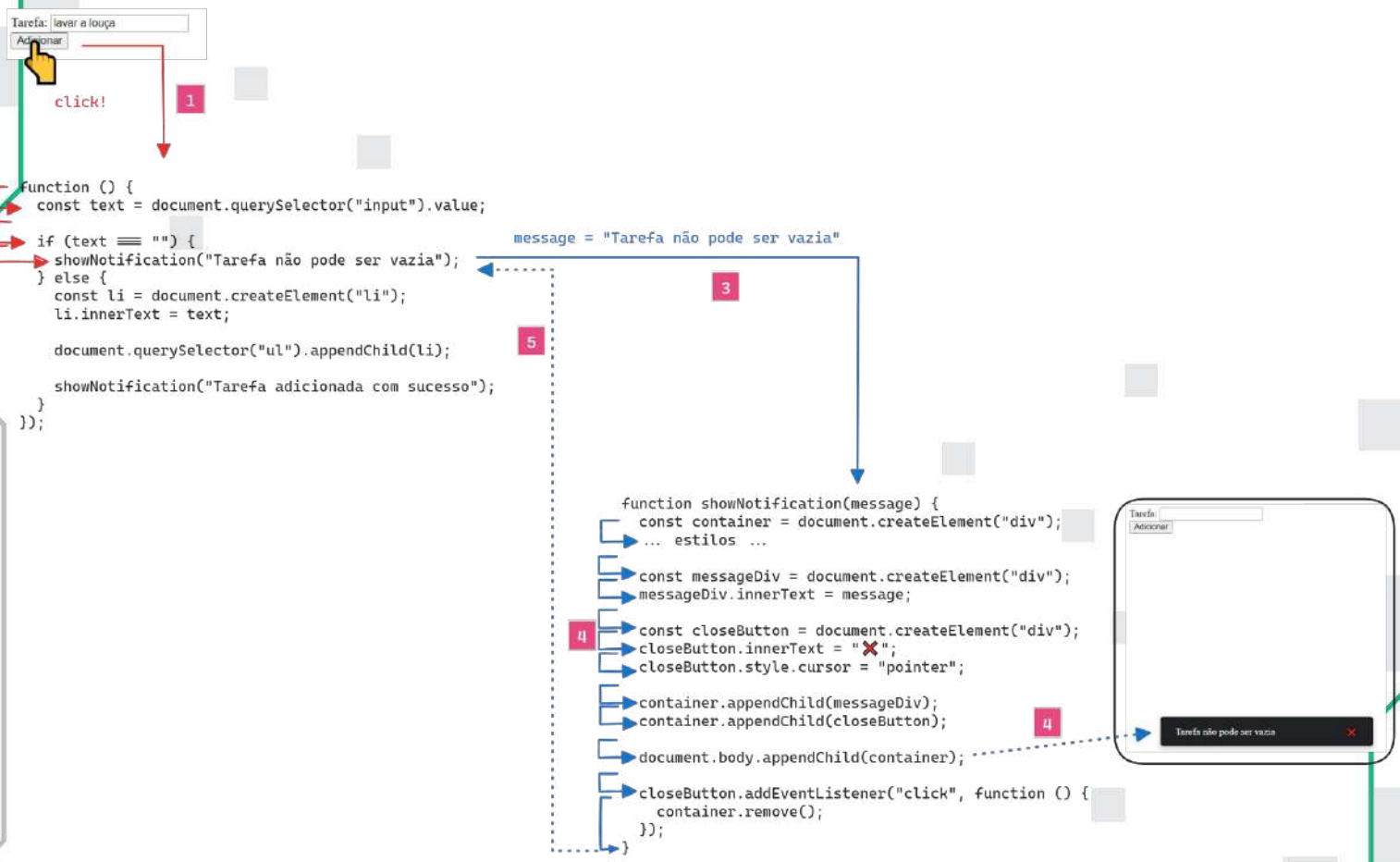
Funções



- No passo 1, o evento de clique é disparado e começa a execução do ouvinte de evento
- Passo 2: na última linha do ouvinte de evento, ele invoca a `showNotification`.
- Passo 3: a string `"Tarefa adicionada com sucesso"` é atribuída à variável `message` dentro da função.
  Esse é um código "implícito", é assim que um parâmetro funciona.
  A variável `message` (parâmetro) é uma variável local normal da função `showNotification`.
  A única característica especial é que seu valor é definido por quem invocou a função (ouvinte de evento vermelho).
- Passo 4: a `showNotification` constrói a notificação.
- Passo 5: a `showNotification` retorna, o ouvinte de evento é retomado.
  E ele também termina, já que a invocação da `showNotification` foi seu último comando.

Agora o outro cenário, se o usuário deixar o input vazio:

Funções



É praticamente a mesma coisa, mas agora a string é `"Tarefa não pode ser vazia"`.

Em resumo: um parâmetro é uma variável declarada no meio dos parênteses da função: `function showNotification(message) { ... }`

Apesar de ser tecnicamente a declaração da variável, você não pode colocar `let` nem `const`:

- ❌ `function showNotification(let message)` . Erro de sintaxe !
- ❌ `function showNotification(const message)` . Erro de sintaxe !

Implicitamente é um `let` (portanto você pode trocar o valor da variável dentro da função).

Ao invocar a função, você coloca no meio dos parênteses o valor que deseja atribuir à variável: `showNotification("Tarefa adicionada com sucesso")`.

O propósito de um parâmetro é oferecer, para quem invoca a função, uma oportunidade de passar informações para dentro dela (nesse caso a mensagem).

Obs: o parâmetro é variável local da função `showNotification`, portanto somente a função pode acessar o parâmetro.

Funções

Múltiplos parâmetros

A caixa de notificação está muito monocromática ! É preto sempre !

Vamos permitir alguma variabilidade, da seguinte forma:

- `showNotification(《mensagem》, "success")` vai mostrar uma notificação com fundo verde claro (indicando sucesso)
- `showNotification(《mensagem》, "error")` vai mostrar uma notificação com fundo vermelho (indicando erro)

Em outras palavras, serão *dois* parâmetros agora.

O primeiro ainda é a mensagem como antes, e o segundo é uma string indicando a cor de notificação desejada.

O código necessário é o seguinte (mudanças indicadas por comentários):

```
document.querySelector("button").addEventListener("click", function () {
  const text = document.querySelector("input").value;

  if (text === "") {
    // colocar o valor do segundo parâmetro
    showNotification("Tarefa não pode ser vazia", "error");
  } else {
    const li = document.createElement("li");
    li.innerText = text;

    document.querySelector("ul").appendChild(li);

    // colocar o valor do segundo parâmetro
    showNotification("Tarefa adicionada com sucesso", "success");
  }
});
```

(código continua na próxima página)

Funções

(continuação do código da página anterior)

```
// mais um parâmetro. Parâmetros são
// separados por vírgulas
function showNotification(message, type) {
  const container = document.createElement("div");
  container.style.position = "absolute";
  container.style.right = "24px";
  container.style.bottom = "24px";
  container.style.minWidth = "300px";
  container.style.minHeight = "32px";
  container.style.padding = "8px 24px";
  container.style.display = "flex";
  container.style.alignItems = "center";
  container.style.justifyContent = "space-between";
  container.style.backgroundColor = "#202124";
  container.style.color = "white";
  container.style.borderRadius = "4px";
  container.style.boxShadow =
    "0 1px 3px 0 rgba(60, 64, 67, 0.302), 0 4px 8px 3px rgba(60, 64, 67, 0.149)";

  // mudar a cor do fundo a depender do valor da variável
  // `type`
  if (type === "success") {
    container.style.backgroundColor = "seagreen";
  } else if (type === "error") {
    container.style.backgroundColor = "tomato";
  }

  const messageDiv = document.createElement("div");
  messageDiv.innerText = message;

  const closeButton = document.createElement("div");
  closeButton.innerText = "❌";
  closeButton.style.cursor = "pointer";

  container.appendChild(messageDiv);
  container.appendChild(closeButton);

  document.body.appendChild(container);

  closeButton.addEventListener("click", function () {
    container.remove();
  });
}
```

Funções

Agora temos 3 tipos de notificação:

```
showNotification("Tarefa adicionada com sucesso", "success");
```


```
showNotification("Tarefa não pode ser vazia", "error");
```



```
showNotification("Mensagem aqui");
    :
    :..▶ equivalente a:
         showNotification("Mensagem aqui", undefined);
```


Note que a invocação da função pode omitir o segundo parâmetro, o que é entendido como equivalente a **`undefined`**.

Nesse caso, dentro da `showNotification` a variável `type` tem valor **`undefined`**, por isso não é executado nem o **`if`** `(type === "success")` nem o **`else if`** `(type === "error")`.

Na verdade é possível também omitir ambos os parâmetros, invocando a função como `showNotification()`.

Nesse caso tanto `message` quanto `type` terão valor **`undefined`**.

Apesar de não dar nenhum erro, já não faz mais sentido, porque a notificação vai aparecer na tela com texto "undefined".

Funções

Cuidado ! Passagem por valor e por referência

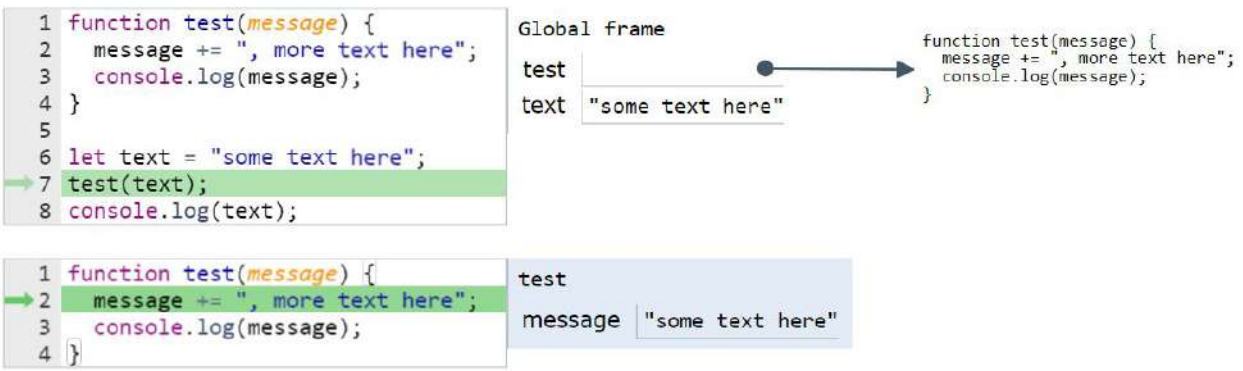
Esta seção não é essencial, se quiser pode pular e voltar se tiver problemas com o que será descrito aqui.
Nos exemplos anteriores, os parâmetros da função eram do tipo string.
Ao passar valores string para a função, o que ela recebe é uma *cópia* do valor original, vamos explicar:
Considere o seguinte código hipotético (não tem nenhuma utilidade prática):

```
function test(message) {
  message += ", more text here";
  console.log(message);
}

let text = "some text here";
test(text);
console.log(text);
```

O passo a passo desse código é o seguinte:

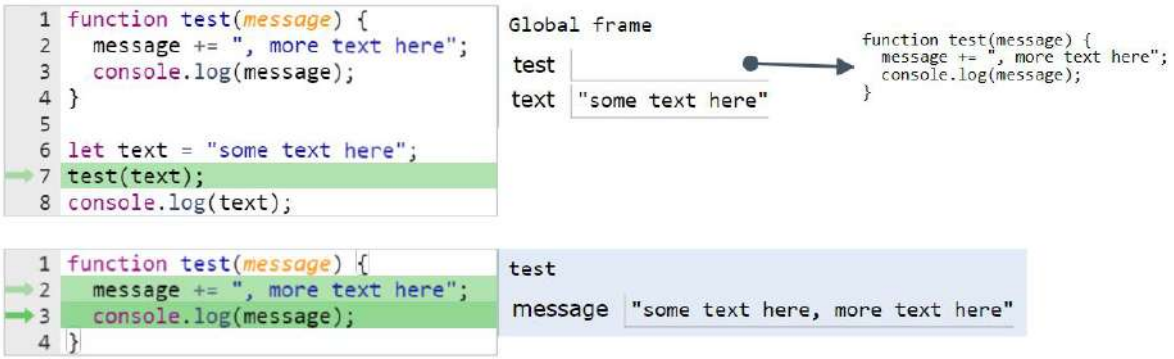
- Passo 1: quando a função é invocada, o valor da variável `text` é calculado primeiro.
  O comando que era `test(text)` se torna `test("some text here")`.
  Essa string é uma cópia do valor da variável `text`, não é a *mesma string* (mesmo dado na memória do computador) que a variável `text` possui.

- Passo 2: A variável `message` recebe a string do passo anterior, portanto `text` e `message` cada uma tem uma cópia diferente da string.
  Ou seja, quando a função `test` inicia a executar, estamos assim:



Funções

- Passo 3: a variável `message` é alterada, e o console.log mostra `"some text here, more text here"`.
  Ficamos assim:



Note como a variável `text` não foi impactada.

- Passo 4: depois de a função retornar e o código original ser retomado, o console.log mostra `"some text here"`, comprovando que a variável `text` não foi impactada pelas manipulações realizadas dentro da função, pois a variável `message` na função sempre trabalhou com uma *cópia* do valor de `text`.

Ao fato de que o parâmetro `message` recebe uma *cópia* do valor fornecido, damos o nome de **Passagem por Valor**.

Isso é o que acontece ao trabalhar com os chamados "tipos primitivos", que são número, string, booleano, undefined e null.

Porém, ao trabalhar com objetos (e arrays, ou qualquer tipo de objeto), que não são um tipo primitivo, não é assim que funciona.

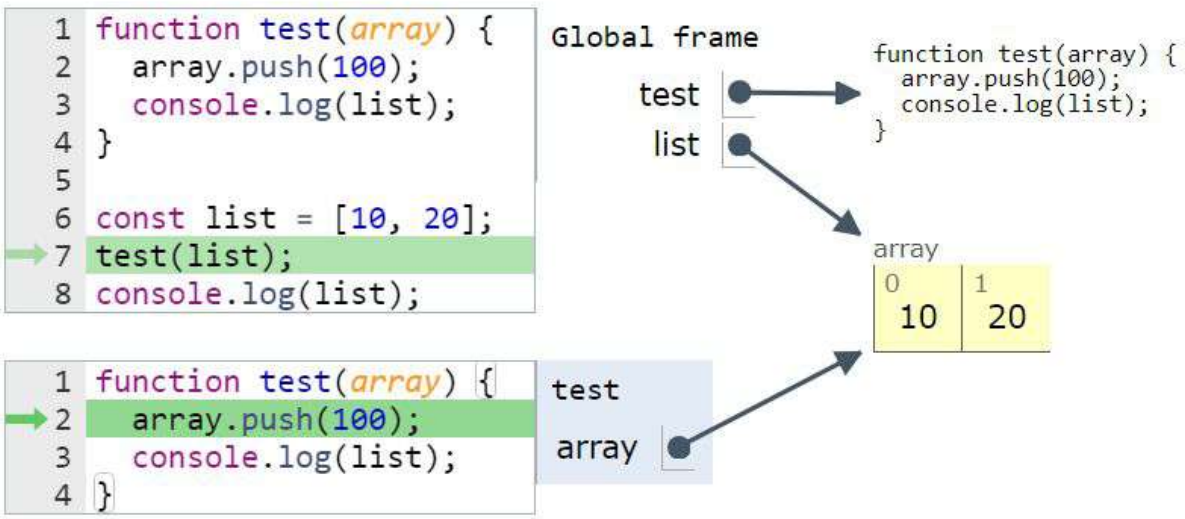
Por exemplo, considere o código seguinte (também meramente ilustrativo):

```
function test(array) {
  array.push(100);
  console.log(list);
}

const list = [10, 20];
test(list);
console.log(list);
```

Neste código, `list` é inicialmente o array `[10, 20]`, e logo em seguida a função é executada.

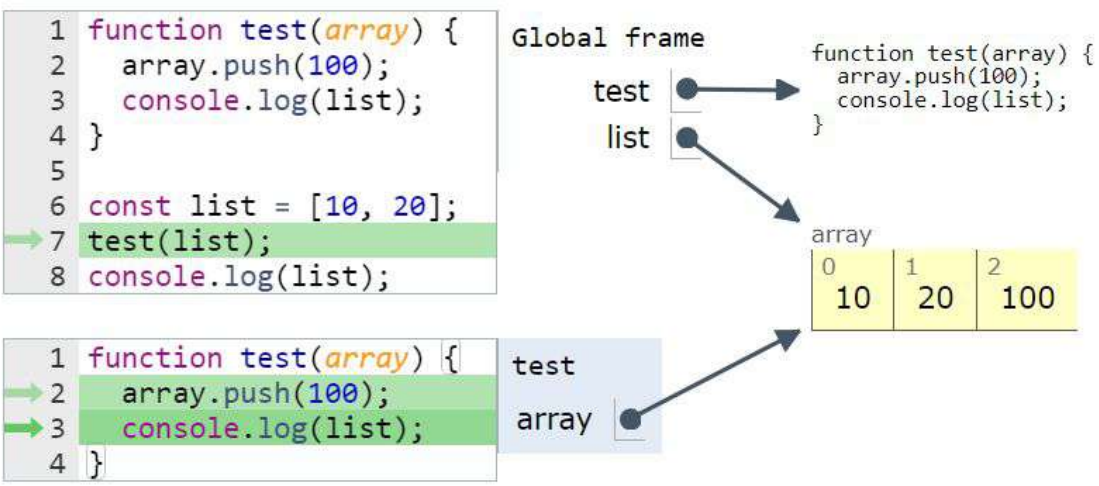
Neste momento (função iniciou), estamos assim:

Funções



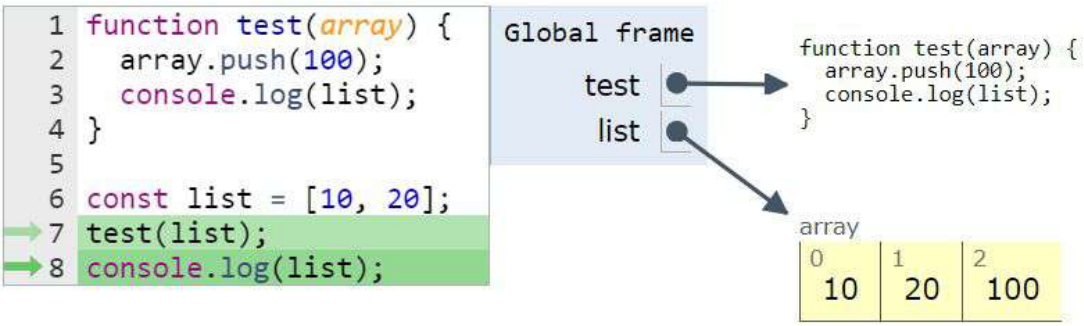
Note que a variável `array` *não é um array independente*, mas sim referencia o mesmo array da variável `list`.

Por isso ao fazer o `array.push(100)`, ficamos assim:



Funções

E então o `console.log` dentro de `test` imprime `[10, 20, 100]`.
Agora a `test` retorna e o código original é retomado:



Diferente do que acontecia com strings, a variável `list` foi sim impactada pelo `push` realizado dentro da função `test`.
O `console.log` vai imprimir de novo `[10, 20, 100]`.
Ao fato de que o parâmetro `array` recebeu não uma cópia de `list`, mas sim *os mesmos dados*, damos o nome de **Passagem por Referência**.
Isso acontece para o tipo de dados objeto, qualquer tipo de objeto:

- objetos "avulsos", como `{name: "joão", idade: "20"}`
- objetos do DOM
- objetos `Date`
- arrays
- etc.

Funções

Cuidado ! Shadowing

Esta seção não é essencial, se quiser pode pular e voltar se tiver problemas com o que será descrito aqui.

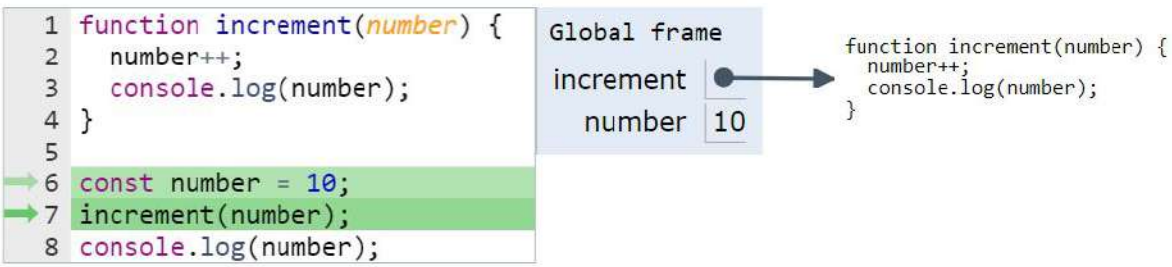
Shadowing (sombreamento) acontece quando uma variável local (seja parâmetro ou não) de uma função tem o mesmo nome que uma variável global.

Por exemplo no seguinte código hipotético (sem utilidade prática):

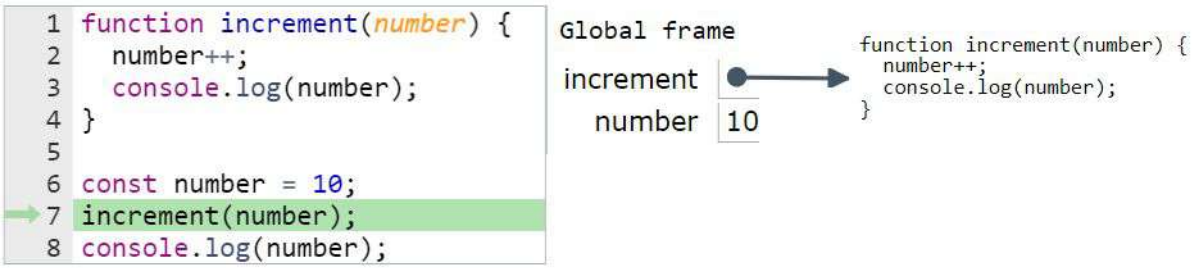
```
function increment(number) {
  number++;
  console.log(number);
}

const number = 10;
increment(number);
console.log(number);
```

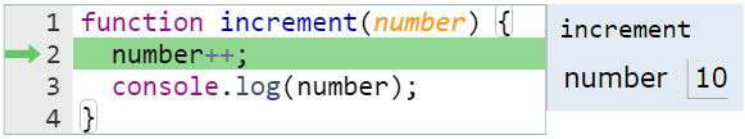
Note que há uma variável global chamada `number`, e a função tem uma variável local (nesse caso, um parâmetro) também chamado `number`.

Quando o código inicia (antes de começar a função), estamos assim:



Quando a função inicia (antes de executar o `number++`), estamos assim:





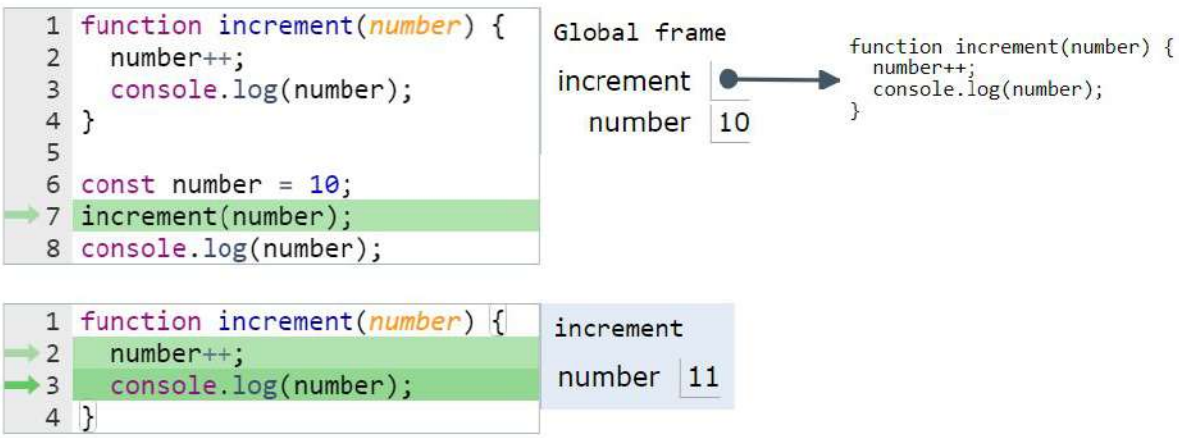
Funções

Note que agora há *duas* variáveis `number`:

- Existe a variável global
- E existe a variável local (parâmetro)

Dentro da função, o nome `number` sempre vai se referir à variável local, por isso dizemos que ela "sombreia" a variável global (torna a variável global inacessível dentro da função).

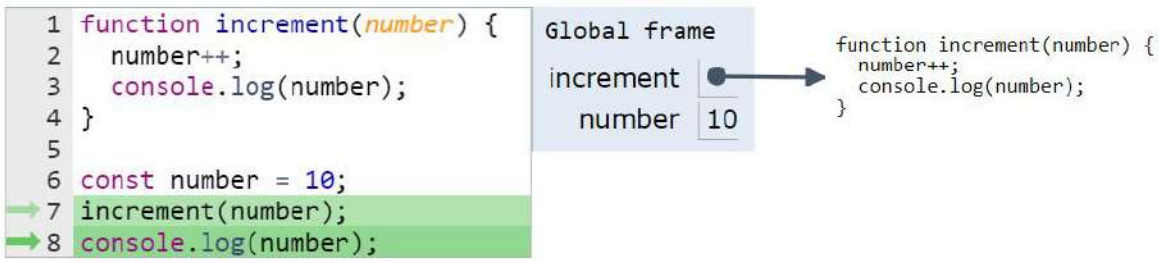
Daí o comando `number++` incrementa a variável local, a global não é impactada:



E o `console.log` imprime 11.

Depois a função retorna e o código original é retomado.

Nesse momento, estamos assim:



Funções

E o `console.log` imprime 10.
Se o código tivesse sido:

```
function increment() {
  number++;
  console.log(number);
}

const number = 10;
increment(number);
console.log(number);
```

A única diferença é que a função não tem o parâmetro `number`.
Nesse código, a palavra `number` dentro da função se refere à variável global, como você já sabia.
Não tem nenhuma variável local com o mesmo nome da global, por isso não há shadowing.
Logo os `console.log` imprimem `11` e `11`.

Funções

Valor de retorno

Talvez você já tenha visto, em plataformas de rede social ou blogs, que a data em que uma postagem foi feita costuma ser exibida de forma "amigável".

Por exemplo, se hoje é 29 de janeiro de 2024, então uma postagem feita:

- no dia 28 seria exibida como "ontem"
- no dia 27 seria exibida como "há 2 dias"
- no dia 22 seria exibida como "há 1 semana"
- no dia 29 de dezembro de 2023 seria exibida como "há 1 mês"

Suponha que você está programando uma plataforma de blog, onde cada artigo é guardado com a data na forma ano-mês-dia (como `"2024-01-29"`).

Ao exibir os artigos na página, você gostaria de mostrar a versão "amigável" da data.

Então seria bom ter uma função chamada por exemplo `getFriendlyDate` ("pegar/calcular data amigável"), que fizesse a conversão para você.

Ou seja:

- `getFriendlyDate("2024-01-29")` ⇒ `"hoje"`
- `getFriendlyDate("2024-01-28")` ⇒ `"ontem"`
- `getFriendlyDate("2024-01-27")` ⇒ `"há 2 dias"`
- etc.

Em outras palavras, é uma função que faz algo análogo a `Number`: `Number("100")` resulta em `100`, ou seja, ela converte o valor pedido para outro valor (string em número).

Já a nossa `getFriendlyDate` deve converter uma string em outra string.

Para um protótipo, vamos supor que você tem uma `<ul>` no HTML onde vai exibir os artigos, e seu array de artigos é:

```
const articles = [
  { title: "Artigo 1", date: "2024-01-29" }, // Data de hoje
  { title: "Artigo 2", date: "2024-01-28" }, // Data de ontem
  { title: "Artigo 3", date: "2024-01-22" }, // Uma data há mais de uma semana
  { title: "Artigo 4", date: "2024-01-15" }, // Duas semanas atrás
  { title: "Artigo 5", date: "2024-01-01" }, // Quase um mês atrás
  { title: "Artigo 6", date: "2023-12-29" }, // Mais de um mês atrás
  { title: "Artigo 7", date: "2023-12-01" }, // Dois meses atrás
  { title: "Artigo 8", date: "2023-10-29" }, // Três meses atrás
];
```

Funções

Se a função `getFriendlyDate` já tivesse sido definida (mas não foi, é justamente isso que vamos fazer daqui a pouco), o código para exibir esses artigos seria o seguinte:

```
const ul = document.querySelector("ul");

for (const article of articles) {
  const li = document.createElement("li");
  const friendlyDate = getFriendlyDate(article.date);
  li.innerText = `${article.title} - ${friendlyDate}`;
  ul.appendChild(li);
}
```

A página ficaria assim:

- Artigo 1 - hoje
- Artigo 2 - ontem
- Artigo 3 - há 1 semana
- Artigo 4 - há 2 semanas
- Artigo 5 - há 1 mês
- Artigo 6 - há 1 mês
- Artigo 7 - há 2 meses
- Artigo 8 - há 3 meses

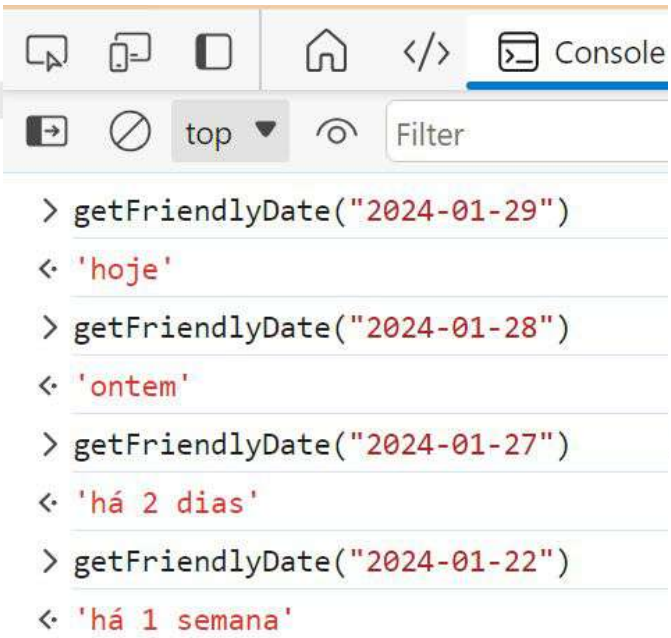
Agora vamos ao código da `getFriendlyDate`.
Primeiro, ela precisa de um parâmetro para a string de entrada, pode se chamar `date`.
Segundo, ela deve usar alguma lógica para construir a string de resposta.
O código fica assim:

Funções

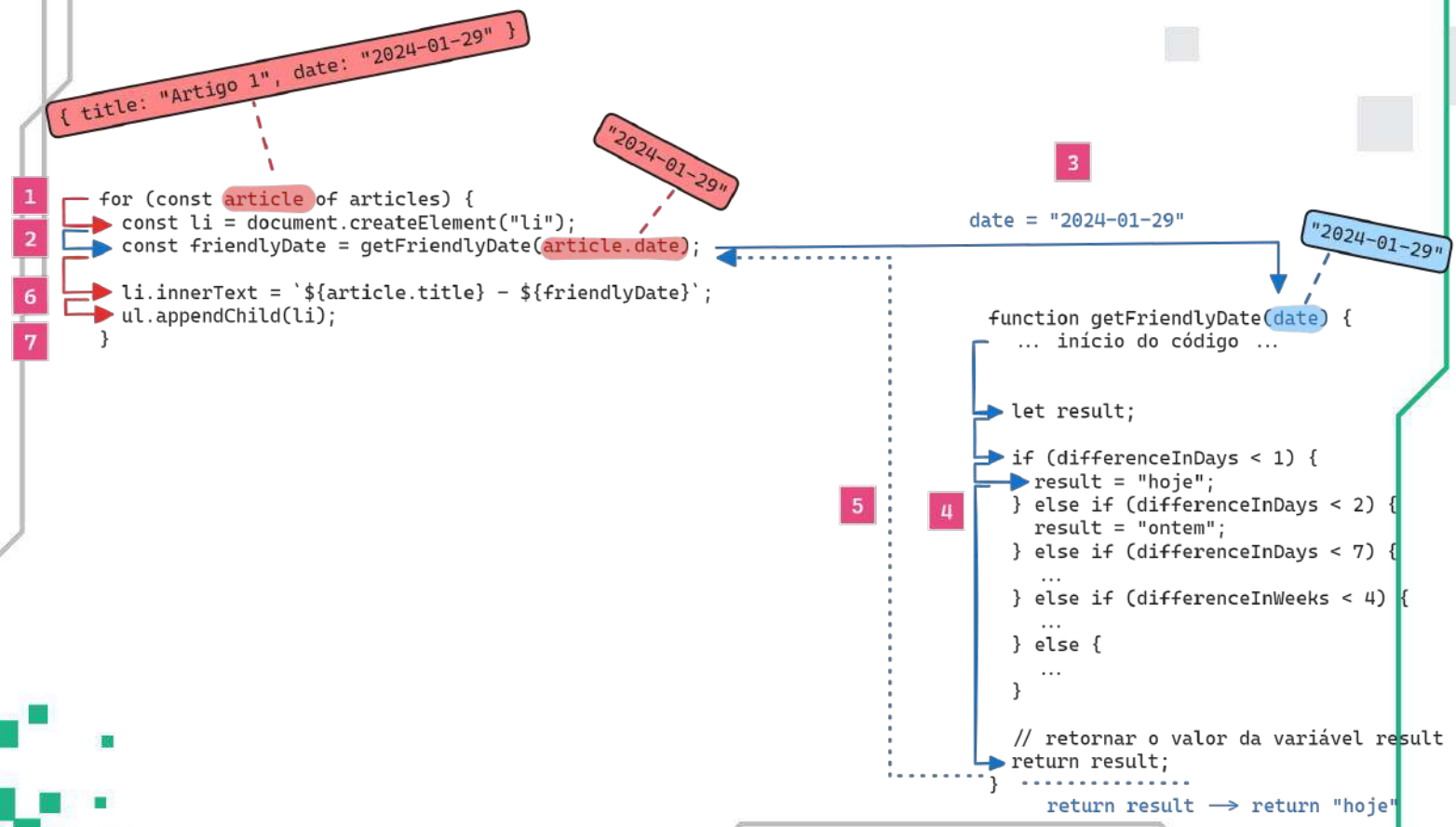
```
// a lógica do cálculo é simplificada, não
// se importe muito com a matemática envolvida,
// o importante é que ela monta uma string em
// formato "amigável".
function getFriendlyDate(date) {
  // calcular início do dia de hoje.
  // usar setHours, etc, para
  // voltar à hora 0 de hoje
  const todayDate = new Date();
  todayDate.setHours(0);
  todayDate.setMinutes(0);
  todayDate.setSeconds(0);
  todayDate.setMilliseconds(0);

  // objeto Date para a data de entrada
  const dateObj = new Date(date + "T00:00:00");

  // Diferença em milissegundos
  const differenceInTime = todayDate.getTime() - dateObj.getTime();

  const differenceInDays = differenceInTime / (1000 * 3600 * 24);
  const differenceInWeeks = differenceInDays / 7;
  // Aproximação para cálculo mensal
  const differenceInMonths = differenceInDays / 30;

  // usar if-else para decidir como exibir o nome
  // amigável, e guardar o resultado na variável
  // `result`
  let result;

  if (differenceInDays < 1) {
    result = "hoje";
  } else if (differenceInDays < 2) {
    result = "ontem";
  } else if (differenceInDays < 7) {
    const rounded = Math.round(differenceInDays);
    const plural = rounded !== 1 ? "dias" : "dia";
    result = `há ${rounded} ${plural}`;
  } else if (differenceInWeeks < 4) {
    const rounded = Math.round(differenceInWeeks);
    const plural = rounded !== 1 ? "semanas" : "semana";
    result = `há ${rounded} ${plural}`;
  } else {
    const rounded = Math.round(differenceInMonths);
    const plural = rounded !== 1 ? "meses" : "mês";
    result = `há ${rounded} ${plural}`;
  }

  // retornar o valor da variável result
  return result;
}
```

Funções

A novidade aqui é o comando `return result`.
Ele significa que o valor da variável `result` deve ser considerado como **Resultado** da função.
Se quiser, experimente invocar essa função no Console para ver seus resultados:



Com a função pronta, a página HTML funciona da seguinte forma (somente a primeira iteração do loop por simplicidade, explicação em seguida):



Funções

- No passo 1, a iteração inicia criando o objeto `<li>`.
- No passo 2, é feita a invocação da função `getFriendlyDate`.
  O código dentro do loop fica congelado e inicia o código da função.
- No passo 3, o parâmetro `date` recebe implicitamente seu valor, e a função começa a executar.
- No passo 4, o código da função é executado e a variável local `result` adquire o valor `"hoje"`.
- No passo 5, é executado o comando `return result`.
  Como `result` vale `"hoje"`, esse comando fica `return "hoje"`.
  O comando `return` indica que a string `"hoje"` deve ser considerada como o **Resultado** da função, também chamado **Valor de Retorno**.
  A função termina de executar, e o resultado `"hoje"` sobe (linha pontilhada) até a linha de código que chamou a função: `const friendlyDate = getFriendlyDate(...)`.
  Como o resultado da função é a string `"hoje"`, essa linha de código se torna: `const friendlyDate = "hoje"`.
- Nos passos 6 e 7, o código dentro do loop define o texto do `<li>` e o coloca dentro da lista `<ul>`

Na próxima iteração do loop, esse diagrama vai se repetir, claramente com outros valores de data.

Mas o processo será o mesmo: a função `getFriendlyDate` será invocada, ela retornará o valor da variável `result`, o qual será recebido pela variável `friendlyDate`, etc.

Em resumo, se você quer que sua função tenha um resultado, deve usar o comando `return` 《`resultado`》.

O resultado pode ser de qualquer tipo de dados: número, string, booleano, objeto, ...

Se a função não tiver um `return`, o resultado dela é `undefined`.

Métodos de objeto

Introdução

Você se lembra do objeto `Math` ?

Ele é um objeto cujas propriedades armazenam funções matemáticas úteis.

Por exemplo `Math.floor` arredonda um número para baixo, `Math.random` sorteia um número aleatório, e várias outras.

Agora que você sabe como a subprogramação funciona (i.e. definir funções e invocá-las), pode aproveitar a capacidade do javascript de definir funções *dentro de propriedades de um objeto*, como acontece no `Math` (onde as propriedades `floor` e `random` são funções).

Por exemplo, ao longo desta aula definimos as funções `showNotification` e `getFriendlyDate`, poderíamos definí-las dentro de um objeto para ficar mais organizado (como o `Math` que reúne várias funções matemáticas).

Definindo funções como propriedades de um objeto

Seria feito assim:

```
// objeto `utils` (de "utilitários")
const utils = {
  showNotification(message, type) {
    const container = document.createElement("div");

    // ... resto do código ...

    if (type === "success") {
      container.style.backgroundColor = "seagreen";
    } else if (type === "error") {
      container.style.backgroundColor = "tomato";
    }

    // ... resto do código ...
  },

  getFriendlyDate(date) {
    const todayDate = new Date();
    todayDate.setHours(0);
    todayDate.setMinutes(0);
    todayDate.setSeconds(0);
    todayDate.setMilliseconds(0);

    let result;

    // ... resto do código ...

    return result;
  },
};
```

Métodos de objeto

Observe que não tem a palavra `function` quando uma função é definida como propriedade de um objeto.

Outra forma de definir seria assim:

```
const utils = {
  showNotification: function (message, type) {
    // código
  },

  getFriendlyDate: function (date) {
    // código
  },
};
```

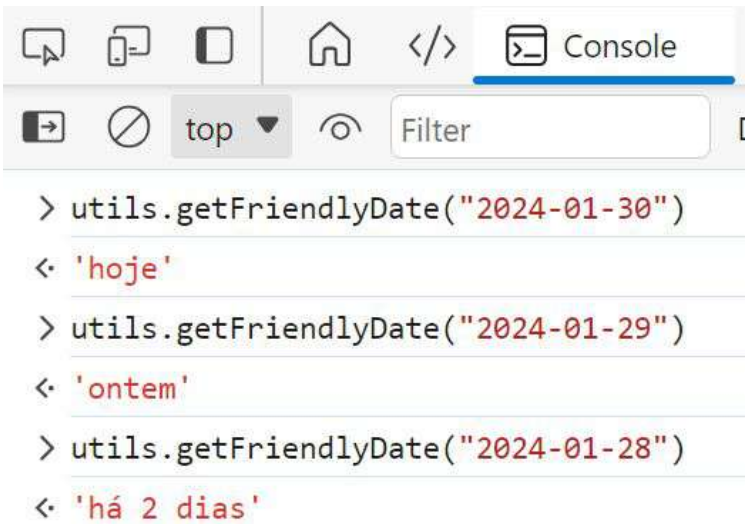
Onde usamos funções anônimas. Não há diferença prática entre as duas maneiras de definir as funções dentro do objeto `utils`.

Você pode até misturar: definir uma função da primeira maneira e a outra da segunda maneira.

Quando funções são definidas como propriedades, elas são geralmente chamadas de **Métodos**.

Invocando os métodos

Tendo construído o objeto `utils`, você pode testar que ele funciona pelo Console:



Note que `getFriendlyDate` é uma propriedade do objeto `utils`, então fazemos o acesso usando a notação de ponto que você já conhece.

Como você pode ver, os métodos são invocados da mesma forma que funções globais (funções definidas no escopo global como antes): você usa a notação de ponto para acessar o método dentro do objeto, digita os parênteses que significam a invocação, e coloca os valores dos parâmetros dentro dos parênteses.

Métodos de objeto

Palavra-chave this dentro de um método

A palavra-chave `this` também funciona dentro de um método, e tem um significado especial.

Para dar um exemplo, vamos lembrar de uma situação típica nas aplicações web feitas até agora, tomando como referência a aplicação de lista de tarefas que fizemos no final da aula 9.

Há um array global chamado `tasks`, contendo tarefas onde cada tarefa é um objeto da forma `{id: 1706589563604, title: "lavar a louça", priority: "high"}`.

Esse array é sincronizado com o Local Storage: quando a página inicia, ele é carregado do Local Storage, e toda modificação feita durante o uso da página é salva no Local Storage.

A página permite ao usuário adicionar uma nova tarefa, deletar uma tarefa existente, ou trocar a prioridade.

Nós podemos simplificar todas essas operações usando um objeto `tasksController`.

Para começar, não haverá mais um array global `tasks`, mas sim um objeto global `tasksController` onde o array de tarefas será uma propriedade:

```
const tasksController = {
  tasks: [], // colocaremos aqui as tarefas
};
```

Vamos começar simplificando o carregamento do Local Storage:

```
const tasksController = {
  tasks: [],

  load: function () {
    const tasksLocalStorage = localStorage.getItem("tasks");

    if (tasksLocalStorage !== null) {
      this.tasks = JSON.parse(tasksLocalStorage);
    }
  },
};
```

Note o uso da palavra-chave `this`.

Dentro de um método, o significado do `this` é: o próprio objeto.

Ou seja, substitua mentalmente o `this` pela variável `tasksController`:

```
const tasksController = {
  tasks: [],

  load: function () {
    const tasksLocalStorage = localStorage.getItem("tasks");

    if (tasksLocalStorage !== null) {
      // o `this` é equivalente a `tasksController`
      tasksController.tasks = JSON.parse(tasksLocalStorage);
    }
  },
};
```

Métodos de objeto

O método `load` carrega os dados do Local Storage e, se eles não vierem `null`, insere esses dados na propriedade `tasksController.tasks`, que é o array de tarefas.

Assim, quando a página inicia, basta colocar o seguinte código no escopo global para fazer o carregamento do Local Storage:

```
tasksController.load();
```

Essa linha de código invoca o método `load`, que por sua vez insere na propriedade `tasksController.tasks` os dados carregados do Local Storage.

Agora podemos simplificar as operações de criar tarefa, editar prioridade e deletar:

```
const tasksController = {
  tasks: [],

  load: function () {
      // ...
  },

  // parâmetros title e priority deve
  // ser strings, como "lavar a louça" e "high"
  create: function (title, priority) {
    this.tasks.push({
      id: new Date().getTime(),
      title: title,
      priority: priority,
    });

    localStorage.setItem("tasks", JSON.stringify(this.tasks));
  },

  // id: o id numérico da tarefa, como 1706589563604.
  // newPriority: "normal" ou "high"
  editPriority: function (id, newPriority) {
    // achar o índice da tarefa que tem esse id
    const index = this.tasks.findIndex((task) => task.id === id);

    this.tasks[index].priority = newPriority;

    localStorage.setItem("tasks", JSON.stringify(this.tasks));
  },

  // id: o id numérico da tarefa, como 1706589563604.
  delete: function (id) {
    // achar o índice da tarefa que tem esse id
    const index = this.tasks.findIndex((task) => task.id === id);

    this.tasks.splice(index, 1);

    localStorage.setItem("tasks", JSON.stringify(this.tasks));
  },

  getTask: function (id) {
    const index = this.tasks.findIndex((task) => task.id === id);

    return this.tasks[index];
  },
};
```

Métodos de objeto

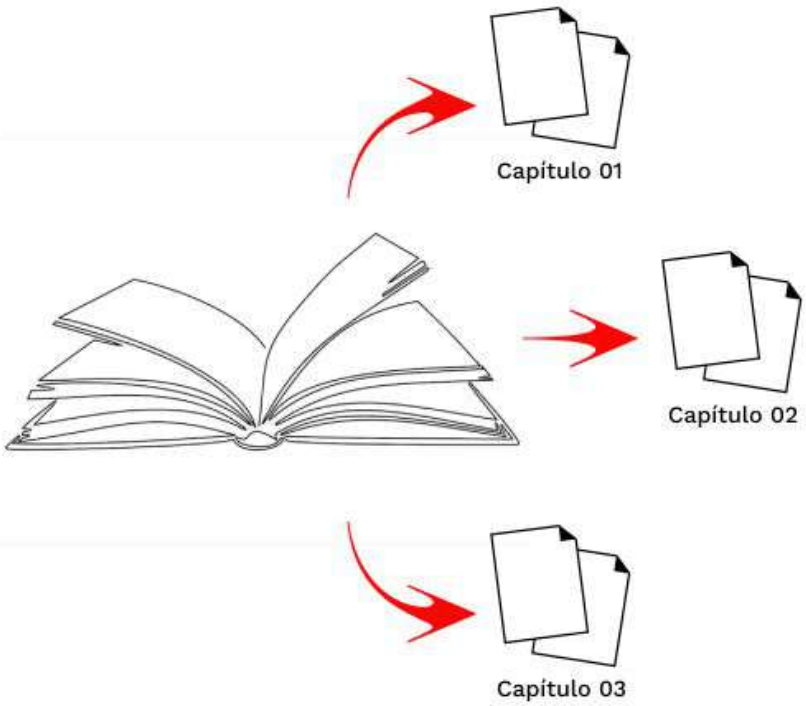
O método `create` é usado assim: `tasksController.create("lavar a louça", "normal")`.
Ele insere um novo objeto no array (com um ID calculado na hora), e já salva no Local Storage.
O método `editPriority` é usado assim: `tasksController.editPriority(1706589563604, "high")`.
Ele altera a prioridade do objeto tarefa no array, e já salva no Local Storage.
E o método `delete` é usado assim: `tasksController.delete(1706589563604)`.
Ele deleta o objeto tarefa do array, e salva no Local Storage.
Finalmente, o método `getTask` é usado assim: `tasksController.getTask(1706589563604)`.
Ele encontra no array o objeto com o ID solicitado, e retorna esse objeto.

Módulos

Módulos no Javascript são uma forma padronizada de importar e exportar funcionalidades entre arquivos.

Mas o que de fato é um módulo?

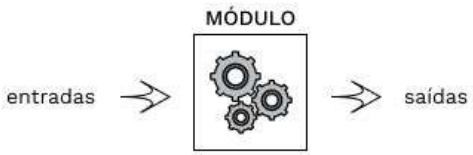
É possível fazer uma analogia entre módulos e os capítulos de um livro.



O código Javascript pode ser dividido em várias funções, cada uma com um objetivo específico, tais como os capítulos de um livro. O conjunto total de funções, trabalhando em conjunto, fazem a totalidade do sistema.

As principais características de um módulo são:

- **autosuficiente:** sua saída depende apenas das entradas que recebe



- **acoplamento:** os usuários do módulo não precisam saber como o módulo foi contruído, mas apenas os resultados retornados, de acordos com as entradas

- **reutilizável:** módulos são geralmente construídos para conter uma lógica única dentro do sistema como, por exemplo, um módulo responsável com conter a lógica necessária a construção de um botão sendo, portanto, reutilizáveis em vários locais do sistema

Há uma série de benefícios em usar módulos:

- **manutenibilidade:** por conterem lógica única e serem reutilizáveis, para realizar alterações nas funcionalidades desempenhadas pelo módulo, basta apenas alterar o seu código interno, por exemplo, para alterar o comportamento de todos os botões do sistema, basta alterar o módulo responsável pela construção do botão

- **evita o compartilhamento de variáveis globais:** como consequência do acoplamento, um módulo não precisa ter conhecimento de variáveis externas, o que se faz desnecessário o uso de variáveis globais no sistema

- **reusabilidade:** como já mencionado, módulos são reusáveis, diminuindo a quantidade de código duplicado na aplicação

Por dentro do assunto!

O uso de variáveis globais é absolutamente uma má prática em programação. Entre as razões para essa conclusão, estão:

- *variáveis globais podem ser utilizadas em várias locais do sistema, tornando o entendimento confuso, pois é difícil tem conhecimento de todos os locais que a utilizam*

- *variáveis globais podem ser lidas ou modificadas em qualquer parte do sistema, tornando difícil lembrar ou raciocinar sobre todos os usos possíveis e determinar qual valor está sendo armazenado em determinado momento da execução do sistema*

- *nomes globais estão disponíveis em todos os lugares do código, fazendo com que você possa, sem saber, acabar usando uma variável global quando pensa que está usando uma variável local*

Criando de módulos

Para utilizar módulos em uma página Web, com Javascript, devemos primeiramente definir o tipo de script do arquivo `.js` como `module`:

```
<!DOCTYPE html>
<html lang="pt-br">
    <head>
        ...
    </head>

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

Após isso, basta configurar os arquivos de cada módulo da página e relacioná-los com as declarações `export` e `import`.

Exportação de módulos

A declaração `export` é utilizada para exportar valores de um módulo Javascript. Os valores exportados poderão ser importados em outros módulos do sistema.

Há dois de exportações que podem ser utilizadas em um módulo Javascript:

Exportações nomeadas

A declaração `export` pode ser adicionada antes de cada valor a ser exportado:

```
// This module is intended to provide basic functions for arithmetic calculations.
// pi stores the value of the mathematical constant π, with 20 places of precision.
export const pi = 3.14159265358979323846;

// add adds two values.
export function add(v1, v2) {
    return v1 + v2;
}

// sub subtract v2 from v1.
export function sub(v1, v2) {
    return v1 - v2;
}

// mul multiplies two values.
export function mul(v1, v2) {
    return v1 * v2;
}

// div divide v1 by v2.
export function div(v1, v2) {
    return v1 / v2;
}
```

O resultado dessa importação é o seguinte objeto de valores:

```
{
  pi: 3.14159265358979323846,
  add: function(v1, v2) { ... },
  sub: function(v1, v2) { ... },
  mul: function(v1, v2) { ... },
  div: function(v1, v2) { ... }
}
```

Da mesma forma, a declaração `export` pode ser utilizada à parte, como mostrado abaixo:

```
// This module is intended to provide basic functions for arithmetic calculations.

// pi stores the value of the mathematical constant π, with 20 places of precision.
const pi = 3.14159265358979323846;

// add adds two values.
function add(v1, v2) {
    return v1 + v2;
}

// sub subtract v2 from v1.
function sub(v1, v2) {
    return v1 - v2;
}

// mul multiplies two values.
function mul(v1, v2) {
    return v1 * v2;
}

// div divide v1 by v2.
function div(v1, v2) {
    return v1 / v2;
}

export { pi, add, sub, mul, div };
```

As declarações `let`, `const`, `var`, `function` ou `class` podem ser utilizadas antes do `export`.

A declaração `default`

A declaração `default` é utilizada para exportar um único valor para o módulo:

```
// This module is intended to export the HTML structure for a button.

export default function(label, onClick) {
    const e = document.createElement("button");

    e.textContent = label;
    e.onclick = onClick;

    return e
}
```

Observe que não foi necessário nomear a função. Porém, também é possível exportar um valor nomeado como `default`.

```
// This module is intended to export the HTML structure for a button.

function buttom(label, onClick) {
    const e = document.createElement('button");

    e.textContent = label;
    e.onclick = onClick;

    return e
}

export default buttom;
```

A declaração `default` só pode ser usada uma única vez em um módulo.

Importação de módulos

Valores exportados em módulos Javascript podem ser importados onde forem necessários, com a utilização da declaração `import`:

```
import math from "./modules/math.js";
```

A declaração `from` indica o caminho do arquivo do módulo que está sendo importado. Essa importação indica que a estrutura de arquivos do projeto é como mostrada abaixo:

```
├── project
|   ├── modules
|   |   └── math.js
|   ├── index.html
|   └── index.js
```

A variável `math` será um objeto com todos os valores exportados pelo módulo `math`:

```
{
  pi: 3.14159265358979323846,
  add: function(v1, v2) { ... },
  sub: function(v1, v2) { ... },
  mul: function(v1, v2) { ... },
  div: function(v1, v2) { ... }
}
```

Os valores exportados poderão então ser utilizados no arquivo onde foram importados:

```
import math from "./modules/math.js";

console.log(math.pi);// 3.14159265358979323846
console.log(math.add(1, 2)); // 3
console.log(math.sub(2, 1)); // 1
console.log(math.mul(1, 2)); // 2

console.log(math.div(3, 2)); // 1.5
```

Outra forma de importar os valores exportados é atribuindo cada valor exportado a uma variável, função, etc., da seguinte forma:

```
import { pi, add, sub, mul, div } from "./modules/math.js";

console.log(pi); // 3.14159265358979323846
console.log(add(1, 2)); // 3
console.log(sub(2, 1)); // 1
console.log(mul(1, 2)); // 2
console.log(div(3, 2)); // 1.5
```

Obseve que cada valor importado passa a ser referenciado com o nome utilizado na importação.

O alias `as`

A declaração `as` pode ser utilizado para criar um alias diferente para um valor exportado:

```
import { pi as PI, add, sub, mul, div } from"./modules/math.js";

console.log(PI);// 3.14159265358979323846
console.log(add(1, 2)); // 3
console.log(sub(2, 1)); // 1
console.log(mul(1, 2)); // 2

console.log(div(3, 2)); // 1.5
```

No exemplo acima, a constante `pi` é agora referenciada como `PI`.

Um uso muito comum para alias é para importações `default`:

```
import * as math from "./modules/math.js";
```

O caractere `*` indica que todos os valores exportados são inseridos no escopo corrente e que podem ser referenciados com o alias `math`.

Componentes

Componentes de Software é o termo utilizado para descrever o elemento de software que encapsula uma série de funcionalidades. É uma unidade independente, que pode ser utilizada com outros componentes para formar um sistema mais complexo.

No contexto de uma página Web, cada elemento distinto da página (menu, cards, header, rodapé, etc.) pode ser construído como um componente.

Um bom exemplo é o componente de botão mencionado nesta aula:

```
// This module is intended to export the HTML structure for a button.

function buttom(label, onClick) {
    const e = document.createElement('button");

    e.textContent = label;
    e.onclick = onClick;

    return e
}

export default buttom;
```

Toda a lógica necessária a construção de um botão é encapsulada dentro deste componente, que pode ser utilizado onde for necessário na página Web.