ES6 para humanos

Sumário

Idiomas

- Original (em inglês)
- Versão em Chinês (Graças ao barretlee)
- Versão em Russo (Graças to etnolover)
- Versão em Coreano (Graças ao scarfunk)

1. let, const e escopo de bloco

`let` permite que você crie declarações limitadas a qualquer bloco, o que é chamado de escopo de bloco. Ao invés de usar `var`, que fornece escopo de função, é recomendado que se use `let` no ES6.

```
var a = 2;
{
    let a = 3;
    console.log(a); // 3
}
console.log(a); // 2
```

Outra forma de fazer declaração de escopo de block é usando `const`, que cria constantes. Em ES6, uma `const` representa uma referência constante para um valor. Em outras palavras, o valor não está congelado, apenas a atribuição dele. Segue um exemplo.

```
{
    const ARR = [5, 6];
    ARR.push(7);
    console.log(ARR); // [5,6,7]
    ARR = 10; // TypeError
    ARR[0] = 3; // valor não é imutável
    console.log(ARR); // [3,6,7]
}
```

Algumas coisas a se lembrar:

- "hoisting" de `let` e `const` variam da forma tradicional de "hoisting" de variáveis e funções. Ambos estão "hoistados", mas não podem ser acessados antes das suas declarações, por causa da Temporal Dead Zone
- `let` e `const` são escopadas aos fechamentos de bloco mais próximos.
- Quando usar `const`, use CAPITAL_CASING
- `const` deve ser definida na sua declaração

2. Arrow Functions

Arrow Functions são uma notação short-hand para escrever funções em ES6. A definição de arrow function consiste de uma lista de parâmetros `( ... )`, seguido de um `=>` e o corpo da função.

```
let adicao = function(a, b) {
    return a + b;
};

// Implementação com Arrow Function
let adicao = (a, b) => a + b;
```

Note que no exemplo acima, a arrow function `adicao` é implementada com um "corpo conciso" que não precisa de declaração de retorno explícita.

Segue um exemplo com o "corpo de bloco"

```
let arr = ['maçã', 'banana', 'laranja'];

let cafe_da_manha = arr.map(fruta => {
    return fruta + 's';
});
```

```
console.log(cafe_da_manha); // ['maçãs', 'bananas', 'laranjas']
```

Calma que ainda tem mais...

#

Arrow functions não só reduzem o tamanho do código. Também estão relacionados ao comportamento do binding do `this`

O comportamento das arrow functions com o `this` é diferente do das funções normais. Cada função no JavaScript define seu próprio contexto de `this`, mas as Arrow functions capturam o `this` do seu contexto delimitador. Dá uma olhada no código a seguir:

```
function Pessoa() {
    // O construtor de Pessoa() define `this` como uma instância dele mesmo
    this.idade = 0;

    setInterval(function envelhecer() {
        // Quando não se usa strict mode, a função envelhecer()
        // define `this` como o objeto global, que é diferente
        // do `this` definido pelo construtor de Pessoa()
        this.idade++;
    }, 1000);
}
var p = new Pessoa();
```

No ECMAScript 3/5, esse problema era corrigido ao atribuir o valor de `this` para uma variável.

```
function Pessoa() {
    var self = this;
    self.idade = 0;

    setInterval(function envelhecer() {
        // O callback se refere à variável  `self` cujo valor
        // é o objeto esperado
        self.idade++;
    }, 1000);
}
```

Como mencionado acima, Arrow functions capturam o valor de this do contexto delimitador, logo, o código a seguir funciona como esperado.

```
function Pessoa() {
    this.idade = 0;

    setInterval(() => {
        this.idade++; // `this` se refere corretamente ao objeto pessoa
    }, 1000);
}
```

```
var p = new Pessoa();
```

Leia mais sobre 'Lexical this' em arrows function aqui

3. Parâmetros Default de Funções

ES6 permite que você defina parâmetros default nas definições de funções. A seguir, uma pequena ilustração.

```
let getPrecoFinal = (preco, imposto = 0.7) => preco + preco * imposto;
getPrecoFinal(500); // 850
```

4. Operador Spread / Rest

O operador `...` é chamado de operador spread ou rest, dependendo de como e onde é usado.

Quando usado com qualquer iterável, ele age como "spread" em elementos individuais.

```
function foo(x, y, z) {
    console.log(x, y, z);
}

let arr = [1, 2, 3];
foo(...arr); // 1 2 3
```

O outro uso comum de `...` é juntar uma série de valores em uma array. Nesse caso, o operador é chamado de "rest".

```
function foo(...args) {
    console.log(args);
}
foo(1, 2, 3, 4, 5); // [1, 2, 3, 4, 5]
```

5. Extensões de Objetos Literais

ES6 permite que se declare objetos literais com uma sintaxe abreviada para inicializar propriedades de variáveis e definir métodos das funções. Também permite ter índices de propriedades computadas em uma definição de objeto literal.

```
function getCarro(fabricante, modelo, valor) {
    return {
```

```
        // com a sintaxe abreviada de
        // valor da propriedade, você
        // pode omitir o valor da propriedade
        // se o índice casa com o nome
        // da variável



        fabricante,  // o mesmo que fabricante: fabricante
        modelo, // o mesmo que modelo: modelo
        valor, // o mesmo que valor: valor

        // valores computados agora funcionam
        // como objetos literais
        ['fabricante' + fabricante]: true,

        // sintaxe de abreviação de definição
        // de método omite a keyword `function`
        // e a vírgula

        depreciar() {
            this.valor -= 2500;
        }
    };
}

let carro = getCarro('Kia', 'Sorento', 40000);

// output: {
//     fabricante: 'Kia',
//     modelo:'Sorento',
//     valor: 40000,
//     fabricanteKia: true,
//     depreciar: function()
// }
```

6. Literais Octais e Binários

ES6 tem um novo suporte para literais octais e binários. Prefixar um número com `0o` ou `0O` vai convertê-lo em octal. Olha só:

```
let oValor = 0o10;
console.log(oValor); // 8

let bValor = 0b10; // 0b ou 0B para binário
console.log(bValor); // 2
```

7. Desestruturamento de Arrays e Objetos

Desestrututamento ajuda a evitar a necessidade de variáveis temporárias quando lida-se com

objetos e arrays.

```
function foo() {
    return [1, 2, 3];
}
let arr = foo(); // [1,2,3]

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

function bar() {
    return {
        x: 4,
        y: 5,
        z: 6
    };
}
let { x: a, y: b, z: c } = bar();
console.log(a, b, c); // 4 5 6
```

8. super em Objetos

ES6 permite que use o método `super` em objetos (sem classe) em protótipos. Um exemplo simples a seguir.

```
var pai = {
    foo() {
        console.log("Hello do Pai");
    }
}

var filho = {
    foo() {
        super.foo();
        console.log("Hello do Filho");
    }
}

Object.setPrototypeOf(filho, pai);
filho.foo(); // Hello do Pai
             // Hello do Filho
```

9. Templates Literais e Delimitadores

ES6 introduz uma forma mais fácil ainda de adicionar interpolação analisada automaticamente.

- `` `${ ... }` `` usado para renderizar as variáveis.
- `` ` `` Contra-aspa é usada como delimitador.

```
let user = 'Kevin';
console.log(`Olá ${user}!`); // Olá Kevin!
```

10. for...of vs for...in

- `for...of` itera em objetos iteráveis, tipo arrays.

```
let apelidos = ['zé', 'bobão', 'cabeçudo'];
apelidos.size = 3;
for (let apelido of apelidos) {
    console.log(apelido);
}
// zé
// bobão
// cabeçudo
```

- `for...in` itera sobre todas as propriedades enumeráveis do objeto.

```
let apelidos = ['zé', 'bobão', 'cabeçudo'];
apelidos.size = 3;
for (let apelido in apelidos) {
    console.log(apelido);
}
// 0
// 1
// 2
// size
```

11. Map e WeakMap

ES6 introduz uma nova série de estrutura de dados chamados `Map` e `WeakMap`. Mas na verdade, a gente usa maps em JavaScript toda hora. Inclusive todo objeto pode ser considerado um `Map`

Um objeto é feito de índices (sempre strings) e valores, enquanto num `Map`, qualquer valor (tanto objetos quanto valores primitivos) podem ser usados seja como índice ou como valor. Dá uma olhada nesse código:

```
var meuMap = new Map();

var indiceString = "uma string",
    indiceObj = {},
    indiceFuncao = function() {};

// atribuindo valores
meuMap.set(indiceString, "valor associado com 'uma string'");
meuMap.set(indiceObj, "valor associado com indiceObj");
```

```
meuMap.set(indiceFuncao, "valor associado com indiceFuncao");

meuMap.size; // 3

// recebendo os valores
meuMap.get(indiceString);    // "valor associado com 'uma string'"
meuMap.get(indiceObj);       // "valor associado com indiceObj"
meuMap.get(indiceFuncao);      // "valor associado com indiceFuncao"
```

WeakMap

Um `WeakMap` é um Map onde os índices são referenciados de forma fraca, o que não previne que seus índices sejam coletados pelo garbage collector, o que significa que você não precisa se preocupar com rombos de memória.

Outra coisa a se notar aqui: Em um `Weakmap`, ao contrário do `Map`, *todo índice deve ser um objeto*.

Um `WeakMap` só tem 4 métodos: `delete(indice)`, `has(indice)`, `get(indice)` e `set(indice, valor)`

```
let w = new WeakMap();
w.set('a', 'b');
// Uncaught TypeError: Invalid value used as weak map key

var o1 = {},
    o2 = function(){},
    o3 = window;

w.set(o1, 37);
w.set(o2, "azerty");
w.set(o3, undefined);

w.get(o3); // undefined, pois é o valor setado

w.has(o1); // true
w.delete(o1);
w.has(o1); // false
```

12. Set e WeakSet

Objetos do tipo Set são coleções de valores únicos. Valores duplicados são ignorados, pois a coleção deve possuir apenas valores únicos. Os valores podem ser primitivos ou referências a objetos.

```
let mySet = new Set([1, 1, 2, 2, 3, 3]);
mySet.size; // 3
mySet.has(1); // true
mySet.add('strings');
mySet.add({ a: 1, b:2 });
```

Você pode iterar sobre um objeto do tipo set por ordem de inserção, usando ou `forEach`, ou `for...of`.

```
mySet.forEach((item) => {
    console.log(item);
    // 1
    // 2
    // 3
    // 'strings'
    // Object { a: 1, b: 2 }
});

for (let value of mySet) {
    console.log(value);
    // 1
    // 2
    // 3
    // 'strings'
    // Object { a: 1, b: 2 }
}
```

Objetos do tipo set também tem os métodos `delete()` e `clear()` .

WeakSet

Parecido com o `WeakMap`, o `WeakSet` permite que você armazene de forma fraca *objetos* em uma coleção. Um objeto no `WeakSet` só ocorre uma vez; ele é único na coleção do WeakSet.

```
var ws = new WeakSet();
var obj = {};
var foo = {};

ws.add(window);
ws.add(obj);

ws.has(window); // true
ws.has(foo);    // false, foo não foi adicionado ao set

ws.delete(window); // remove window do set
ws.has(window);    // false, window foi removido
```

13. Classes em ES6

ES6 introduz uma nova sintaxe para classes. Uma coisa a se notar é que no ES6 as classes não são um novo modelo de herança orientada a objetos. Elas apenas servem como sintaxe de açúcar sobre as heranças existentes de prototype do Javascript.

Dá para se olhar para as classes do ES6 como apenas uma nova sintaxe para se trabalhar com prototypes e funções construtoras que usaríamos no ES5

Funções definidas com a keyword `static` implementam funções estáticas/funções de classe na classe.

```
class Tarefa {
    constructor() {
        console.log("tarefa instanciada!");
    }

    mostrarId() {
        console.log(23);
    }

    static carregarTodas() {
        console.log("carregando todas as tarefas..");
    }
}

console.log(typeof Tarefa); // function
let tarefa = new Tarefa(); // "tarefa instanciada!"
tarefa.mostrarId(); // 23
Tarefa.carregarTodas(); // "carregando todas as tasks.."
```

extends e super em classes

Considere o código a seguir:

```
class Carro {
    constructor() {
        console.log("Criando um novo carro");
    }
}

class Porsche extends Carro {
    constructor() {
        super();
        console.log("Criando um Porsche");
    }
}

let c = new Porsche();
// Criando um novo carro
// Criando um Porsche
```

`extends` permite que classes filhas herdem da classe pai no ES6. É importante notar que o construtor derivado deve invocar super().

Você também pode chamar o método da classe pai na classe filha usando `super.nomeDoMetodoPai()`

Leia mais sobre classes aqui

Algumas coisas para se lembrar:

- Declaração de classes não são hoisted. Primeiro você precisa declarar suas classes e depois acessá-las. Caso contrário, vai dar throw num ReferenceError.
- Não há necessidade de usar `function` ao definir funções dentro de uma definição de classe.

14. Símbolo (Symbol)

Um símbolo é um tipo único e imutável de dado introduzido no ES6. A utilidade do símbolo é gerar um identificador único, que você nunca terá acesso a ele.

Como criar um símbolo:

```
var sym = Symbol("descrição qualquer coisa");
console.log(typeof sym); // symbol
```

Perceba que não se usa `new` em `Symbol(…)`.

Se um símbolo é usado como propriedade/índice de um objeto, é armazenado de uma forma especial em que a propriedade não vai aparecer em enumerações normais das propriedades de um objeto.

```
var o = {
    val: 10,
    [Symbol("random")]: "Oi, sou um símbolo",
};

console.log(Object.getOwnPropertyNames(o)); // val
```

Para receber as propriedades de um símbolo, use `Object.getOwnPropertySymbols(o)`

15. Iteradores (Iterators)

Um iterador acessa os itens de uma coleção um por vez, enquanto mantém controle da sua posição atual na sequência. Ele tem um método `next()` que retorna o próximo item da sequência. Esse método retorna um objeto com duas propriedades: `done` e `value`

ES6 tem `Symbol.iterator` que especifica o iterador padrão de um objeto. Toda vez que um objeto precisar ser iterado (como no começo de um for...of), seu método @@iterator é chamado sem argumentos, e o iterador retornado é usado para obter os valores a serem iterados.

Vamos dar uma olhada numa array, que é um iterável, e o iterador que pode ser produzido para consumir seus valores.

```
var arr = [11,12,13];
var itr = arr[Symbol.iterator]();

itr.next(); // { value: 11, done: false }
itr.next(); // { value: 12, done: false }
itr.next(); // { value: 13, done: false }

itr.next(); // { value: undefined, done: true }
```

Perceba que você pode escrever iteradores customizados definindo `obj[Symbol.iterator]()` na definição do objeto.

16. Geradores (Generators)

Funções geradoras são um novo recurso que permite que uma função gere quantos valores forem necessários retornando um objeto que pode ser iterado para puxar mais valores da função, um por vez.

Uma função geradora retorna um **objeto iterável** quando é chamada. É escrita usando a nova sintaxe `*` além da nova keyword `yield` introduzida no ES6.

```
function *numerosInfinitos() {
    var n = 1;
    while (true) {
        yield n++;
    }
}

var numbers = numerosInfinitos(); // retorna um objeto iterável

numbers.next(); // { value: 1, done: false }
numbers.next(); // { value: 2, done: false }
numbers.next(); // { value: 3, done: false }
```

Toda vez que `yield` é chamada, o valor produzido se torna o próximo valor na sequência.

Perceba também que geradores computam seus valores produzidos sob demanda, o que permite que representem eficientemente sequências que são custosas de se produzir, ou até mesmo sequências infinitas.

17. Promises

ES6 tem suporte nativo para promises. Uma promise é um objeto que espera por uma operação assíncrona ser completa, e quando se completa, a promise se torna resolvida ou rejeitada.

A forma padrão de se criar uma Promise é usando o construtor `new Promise()` que aceita um manipulador que é dado 2 funções como parâmetros. O primeiro manipulador (geralmente nomeado `resolve`) é uma função pra chamar com o futuro valor quando ele estiver pronto; e o segundo manipulador (geralmente chamado de `reject`) é uma função pra chamar se a Promise não conseguir resolver o valor futuro, e for rejeitada.

```
var p = new Promise(function(resolve, reject) {
    if (/* condition */) {
        resolve(/* value */);  // resolvida com sucesso
    } else {
        reject(/* reason */);  // erro, rejeitada
    }
});
```

Toda promisse tem um método chamado `then`, que recebe um par de callbacks. O primeiro callback é chamado se a promise for resolvida, enquanto o segundo é chamado se for rejeitada.

```
p.then((val) => console.log("Promise Resolvida", val),
       (err) => console.log("Promise Rejeitada", err));
```

O valor de retorno de `then` vai ser passado como valor pro próximo `then`.

```
var hello = new Promise(function(resolve, reject) {
    resolve("Hello");
});

hello.then((str) => `${str} World`)
     .then((str) => `${str}!`)
     .then((str) => console.log(str)) // Hello World!
```

Quando retornar uma promise, o valor resolvido da promise vai ser passado para o próximo callback para encadeá-los efetivamente. Essa é uma forma simples de evitar o "callback hell"

```
var p = new Promise(function(resolve, reject) {
    resolve(1);
});

var EventualmenteAdiciona1 = (val) => {
    return new Promise(function(resolve, reject){
        resolve(val + 1);
    });
}

p.then(EventualmenteAdiciona1)
 .then(EventualmenteAdiciona1)
```

```
  .then((val) => console.log(val)) // 3
```