Modern JavaScript

Thiago Alves Luiz

Senior FrontEnd Developer

@devthiago

contato@thiagoalv.es

Modern JavaScript

— Variables
— Template literals
— Arrow functions
— Array methods
— Destructuring objects and arrays
— Spread operator

Thiago Alves Luiz - @devthiago - contato@thiagoalv.es

Variables

var, *let* and *const*

Thiago Alves Luiz - @devthiago - contato@thiagoalv.es

var

— Forma comum de declaração de variáveis JS

— Function scoped

```
var name = 'Thiago';
```

var

```
var name = 'Thiago';

console.log(name); // "Thiago"

name = 'Alves';

console.log(name); // "Alves"

name = 21;

console.log(name); // 21

name = [ 1, 2, 3 ];

console.log(name); // [ 1, 2, 3 ]
```

Function scoped

```
function myFunction() {
  var name = 'Thiago';
  console.log(name); // "Thiago" - name é acessível dentro da função
}
console.log(name); // Throws a ReferenceError, name não é acessível fora da função
```

WARNING - var hoisting

isso:

```
console.log(myVar) // undefined -- no error raised

var myVar = 2;
```

WARNING - var hoisting

será interpretado como isso:

```
var myVar;

console.log(myVar) // undefined -- no error raised

myVar = 2;
```

let

— Funciona semelhante a `var`
— Não pode ser redeclarada no mesmo escopo

— Block scoped
— Reassignable

```
let name = 'Thiago';
```

let

```
let name = 'Thiago';

console.log(name); // "Thiago"

name = 'Alves';

console.log(name); // "Alves"

name = 21;

console.log(name); // 21

name = [ 1, 2, 3 ];

console.log(name); // [ 1, 2, 3 ]
```

Block scoped { }

```
if (condition) {
  let name = 'Thiago';
  console.log(name); // "Thiago" - name é acessível dentro do bloco
}
console.log(name); // Throws a ReferenceError, name não é acessível fora do bloco
```

Block scoped { }

```
let name = 'Alves';

if (condition) {

   let name = 'Thiago';

   console.log(name); // "Thiago"

}

console.log(name); // "Alves"
```

Temporal Dead Zone

```
console.log(myVar) // raises a ReferenceError !

let myVar = 2;
```

Temporal Dead Zone

Variáveis `var` recebem valor inicial `undefined`.

Variáveis `let` e `const` não são inicializadas até suas definições estarem disponíveis.

Podemos dizer que, enquanto não forem definidas, essas variáveis estão em uma `Zona Morta Temporal`

const

— Funciona semelhante a `let`
— Não pode ser redeclarada no mesmo escopo

— Block scoped
— Not reassignable

```
const name = 'Thiago';
```

const

```
const name = 'Thiago';

console.log(name); // "Thiago"

name = 'Alves'; // raises an error, reassignment is not allowed
```

Block scoped { }

```
if (condition) {
  const name = 'Thiago';
  console.log(name); // "Thiago" - name é acessível dentro do bloco
}
console.log(name); // Throws a ReferenceError, name não é acessível fora do bloco
```

Block scoped { }

```
const name = 'Alves';

if (condition) {
  const name = 'Thiago';
  console.log(name); // "Thiago"
}

console.log(name); // "Alves"
```

Mutable

Objetos:

```
const person = {
  name: 'Thiago'
};

person.name = 'Alves'; // this will work!
// person variable is not completely reassigned, but mutated

console.log(person.name) // "Alves"

person = 'Luiz'; // raises an error
// because reassignment is not allowed with const declared variables
```

Mutable

Arrays:

```
const person = [];

person.push('Thiago'); // this will work!
// person variable is not completely reassigned, but mutated

console.log(person[0]); // "Thiago"

person = ['Alves']; // raises an error
// because reassignment is not allowed with const declared variables
```

Variables

	Scope	Reassignable	Mutable	Temporal Dead Zone
var	Function	✅	✅	❌
let	Block	✅	✅	✅
const	Block	❌	✅	✅

Template literals

`${var} text`

Thiago Alves Luiz - @devthiago - contato@thiagoalv.es

Template literals

É uma expressão para interpolação de strings de uma ou múltiplas linhas.

Thiago Alves Luiz - @devthiago - contato@thiagoalv.es

Template literals

```
const name = "Thiago";

`Olá ${name}, você sabia que 5 + 5 = ${5+5}?`;

// Olá Thiago, você sabia que 5 + 5 = 10?
```

Template literals

```
function tmpl (name, age) {
  return `
    <section>
      <strong>${name}</strong>
      <span>${age}</span>
    </section>
  `;
}
```

Arrow Functions

```
() => {}
```

Thiago Alves Luiz - @devthiago - contato@thiagoalv.es

Arrow Functions

ES6 trouxe para o JS essa nova forma de declarar e usar funções.

— Sintaxe mais curta (concise body)

— `this` contexto vinculado

— retorno implícito

Arrow Functions

```
// block body
// explicit return
function func1 (number) {

  return number + 1;
}

func1(2); // 3
```

Arrow Functions

```
// concise body
// implicit return
const func1 = number => number + 1;

func1(2); // 3
```

Arrow Functions

```
const getFullName = (firstName, lastName) => `${firstName} ${lastName}`;

getFullName('Thiago', 'Luiz'); // "Thiago Luiz"
```

Arrow Functions

```
const sayMyName = () => `Thiago Luiz`;

sayMyName(); // "Thiago Luiz"
```

Arrow Functions

```
// block body
// explicit return
const getAge = (name, age) => {

  if (age > 18) return `${name} é maior de 18`;

  if (age === 18) return `${name} tem 18`;

  return `${name} é menor de 18`;

};

getAge('Thiago', 32); // "Thiago é maior de 18"
```

Arrow Functions

```
function myFunc() {
  this.myVar = 0;
  setTimeout(() => {
    this.myVar ++;
    console.log(this.myVar); // 1
  }, 0);
}
```

Array methods

map / filter / reduce

Thiago Alves Luiz - @devthiago - contato@thiagoalv.es

Array methods

- Paradigma funcional
- `.map()` - transforma todos os elementos de um array
- `.filter()` - decide quais elementos devem permanecer no array
- `.reduce()` - agrega os elementos em um valor único

Array.map()

```
const numbers = [0, 1, 2, 3, 4, 5, 6];

const doubledNumbers = numbers.map(n => n * 2);

console.log(doubleNumbers);  // [0, 2, 4, 6, 8, 10, 12]
```

Array.filter()

```
const numbers = [0, 1, 2, 3, 4, 5, 6];

const evenNumbers = numbers.filter(n => n % 2 === 0);

console.log(evenNumbers);  // [0, 2, 4, 6]
```

Array.reduce()

```
const numbers = [0, 1, 2, 3, 4, 5, 6];

const sum = numbers.reduce((acc, next) => acc + next, 0);

console.log(sum);  // 21
```

Array methods

```
const students = [
  { name: "Pedro", grade: 9 },
  { name: "Sara", grade: 10 },
  { name: "Julia", grade: 8 },
  { name: "João", grade: 5 }
];

const aboveSevenSum = students
  .map(student => student.grade) // [1]
  .filter(grade => grade >= 7) // [2]
  .reduce((acc, next) => acc + next, 0); // [3]

console.log(aboveTenSum) // 27

// [1] mapeamos o array de estudantes para um array de suas notas
// [2] filtramos o array resultante para valores maiores ou iguais a 7
// [3] reduzimos todos os valores em uma soma total
```

Destructuring objects and arrays

```
{ } = object
```

Thiago Alves Luiz - @devthiago - contato@thiagoalv.es

Destructuring objects and arrays

`Destruction` é uma abordagem de criação de novas variáveis extraindo valores que estão armazenados em `objects` ou `arrays`.

Object - sem destructuring

```
const person = {
   firstName: 'Thiago',
   lastName: 'Luiz',
   age: 32,
   sex: 'M'
};
```

```
const age = person.age;

const first = person.firstName;

const city = person.city || 'Santa Maria';
```

Object - com destructuring

```
const person = {
  firstName: 'Thiago',
  lastName: 'Luiz',
  age: 32,
  sex: 'M'
};

const { age, firstName: first, city = 'Santa Maria' } = person; // UMA LINHA

// A nova variável age é criada e é igual a person.age
// A nova variável first é criada e é igual a person.firstName
// Se tentarmos acessar firstName ocorrerá ReferenceError
// Quando fazemos `firstName: first`, estamos dando um novo nome de variável
// Podemos definir valores default caso este seja undefined
```

Object - destructuring

```
const person = {
  firstName: 'Thiago',
  lastName: 'Luiz',
  age: 32,
  sex: 'M'
};

const getFullName = ({ firstName, lastName }) => `${firstName} ${firstName}`;

getFullName(person); // 'Thiago Luiz'
```

Array - sem destructuring

```
const myArray = ['a', 'b', 'c'];

const x = myArray[0];
const y = myArray[1];
```

Array - com destructuring

```
const myArray = ['a', 'b', 'c'];

const [x, y] = myArray;
```

Spread operator

" ... "

Thiago Alves Luiz - @devthiago - contato@thiagoalv.es

Spread operator

Foi introduzido com o ES6 (ES2015) e é usado para expandir ou juntar objetos e/ou arrays.

Spread operator

```
const arr1 = ["a", "b", "c"];

const arr2 = [...arr1, "d", "e", "f"];

console.log(arr2); // ["a", "b", "c", "d", "e", "f"]

const [a, ...arr3] = arr1;

console.log(arr3); // ["b", "c"];
```

Spread operator

```
const { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };

console.log(x); // 1

console.log(y); // 2

console.log(z); // { a: 3, b: 4 }
```

```
const n = { y, ...z };

console.log(n); // { y: 2, a: 3, b: 4 };
```

Spread operator

```
function myFunc(x, y, ...params) {
  console.log(x);
  console.log(y);
  console.log(params);
}

myFunc("a", "b", "c", "d", "e", "f");
// "a"
// "b"
// ["c", "d", "e", "f"]
```

Posts

— JavaScript.info - A modern tutorial from the basics to advanced topics with simple, but detailed explanations.

— Robust Client-Side JavaScript - Guide focused on writing robust code by describing possible failures and explaining how to prevent them.

Thiago Alves Luiz - @devthiago - contato@thiagoalv.es

Posts

— Glossary of Modern JavaScript Concepts: Part 1 - Learn the fundamentals of functional programming, reactive programming, and functional reactive programming in JavaScript.

— Glossary of Modern JavaScript Concepts: Part 2 - Explains concepts like scope and closures, data flow, change detection, components, compilation, tree shaking.

Thiago Alves Luiz - @devthiago - contato@thiagoalv.es

Free eBooks

— Eloquent JavaScript – Covering the language and runtime specifics.

— You Don't Know JS (book series) – Series of books diving deep into language.

Free eBooks

— Speaking JavaScript - In-depth guide beginning with the basics.

— JavaScript Design Patterns - Classical and JavaScript specific design patterns.

Thiago Alves Luiz - @devthiago - contato@thiagoalv.es

Thiago Alves Luiz

Senior FrontEnd Developer

@devthiago

contato@thiagoalv.es