```
<script type="text/javascript">
    function getInspiration() {
        if (morningDay == "depressed")
            start.Coding & get.Awesome;
    }
</script>
```

JavaScript

Mais sobre Funções

JS

JavaScript

parte 8

Alan Alencar, 2023

aprendendo Javascript

agenda

Já vimos como é possível criar funções em Javascript, declarar valores padrões para seus argumentos. Agora vamos aprender a declarar uma função como uma variável. É ... em Javascript é possível ter uma variável que aponta para uma função.

mas antes, funções dentro de função

função dentro do escopo da **função principal**

```js
// função dentro de função. Só é acessível dentro da função.
function soma(x = 1, y = 0) {
    x = typeof x !== 'number' ? 0 : x;
    y = typeof y !== 'number' ? 0 : y;

    exibeParametros();

    return x + y;

    function exibeParametros() {
        console.log(`Param X: ${x}`);
        console.log(`Param Y: ${y}`);
        return true;
    }
}

console.log( soma(10, 6) ); // retornará 16
console.log( soma('10', 6) ); // retornará 6
console.log( soma(10) ); // retornará 10
console.log( soma(10, null) ); // retornará 10
console.log( soma(undefined,5) ); // retornará 6
console.log( soma(5, true) ); // retornará 5
console.log( soma(false, true) ); // retornará 0
console.log( soma() ); // retornará 1
```

```
// Resultado das chamadas.

"Param X: 10"
"Param Y: 6"
16
"Param X: 0"
"Param Y: 6"
6
"Param X: 10"
"Param Y: 0"
10
"Param X: 10"
"Param Y: 0"
10
"Param X: 1"
"Param Y: 5"
6
"Param X: 5"
"Param Y: 0"
5
"Param X: 0"
"Param Y: 0"
0
"Param X: 1"
"Param Y: 0"
1
```

quando você cria uma função dentro do escopo de outra função é importante você saber que ela não poderá ser invocada fora deste escopo.

isto vai gerar um erro se você tentar invocá-la.

funções chamando função

construções avançadas com *functions*

```
// Função que retorna um valor aleatório.
function aleatorioNumber() {
    return Math.floor(Math.random() * 10)
}

// Usa a função de aleatório para gerar um valor.
function cubo(x = aleatorioNumber()) {
    return x * x * x;
}

console.log( cubo() );
console.log( cubo() );
```

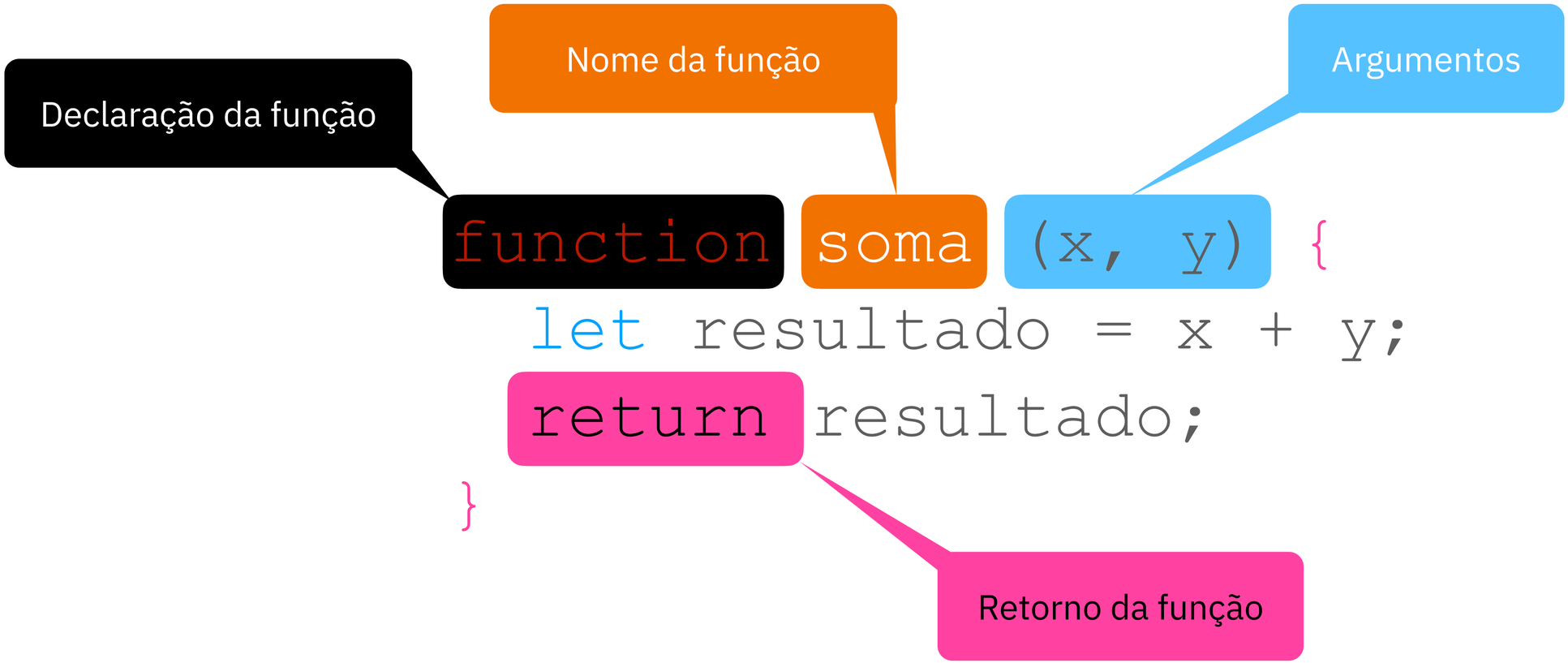
no exemplo a esquerda, mostra como um parâmetro pode ganhar um valor aleatório quando for do tipo `undefined`.

para isto é necessário não enviar o parâmetro e a chamada da função duas vezes mostra a aleatoriedade.

```
// Primeira chamada
216
729
// Segunda chamada
1
343
```

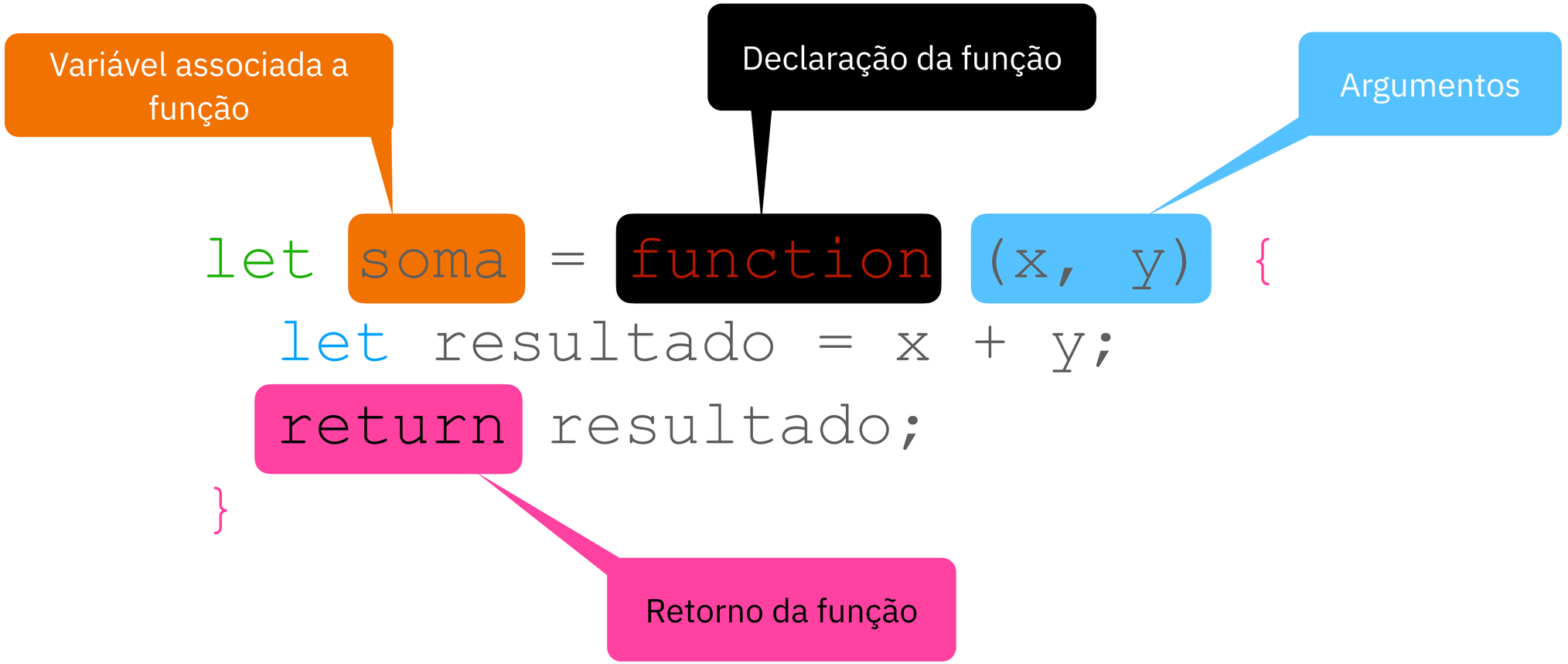
estrutura da função

function declaration



estrutura da função

function expression



não existe hoisting

agora precisa atenção total

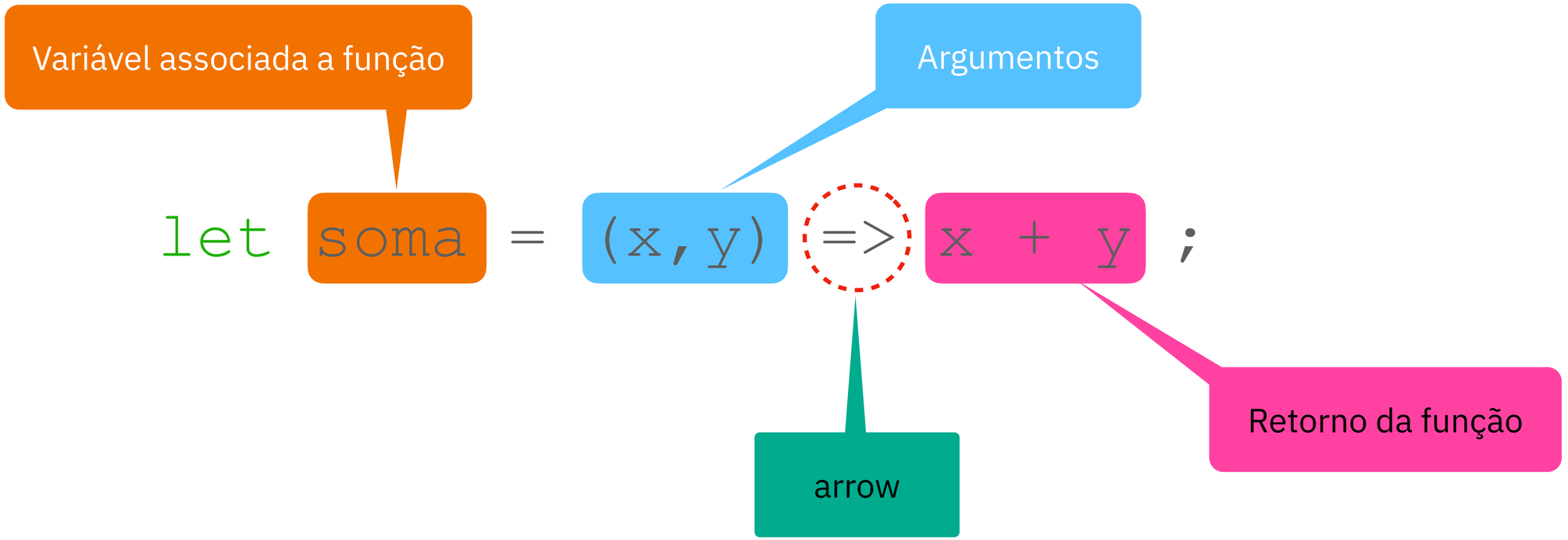
```
/*
 * function expression
 * Hoisting não é permitido com este tipo de declaração.
 */

let soma = function(x = 10, y = 100) {
    x = typeof x == 'number' ? x : 0;
    y = typeof y == 'number' ? y : 0;

    const resultado = x + y;
    return resultado;
}

console.log(soma(6, 1)); // 7
console.log(soma(5.87, 5.22)); // 11.09
console.log(soma(7.87)); // 107.87
console.log(soma()); // 110
```

diferente da declaração anterior, coloque o nome da função em uma variável e pronto. Agora basta chamar a variável passando os argumentos se existir. A atenção aqui é que neste tipo de construção não há **hoisting** para elevação da variável para o topo do escopo declarado.

então é obrigatório / mandatório criar antes de usar para que o resultado funcione.

estrutura da função

arrow *function*



sintaxe reduzida

conhecido também como *arrow function*

```
/*
 * function expression, com arrow function
 * Hoisting não é permitido com este tipo de declaração.
 */

let soma = (x = 10, y = 100) => {
    x = typeof x == 'number' ? x : 0;
    y = typeof y == 'number' ? y : 0;

    const resultado = x + y;
    return resultado;
}

console.log(soma(6, 1)); // 7
console.log(soma(5.87, 5.22)); // 11.09
console.log(soma(7.87)); // 107.87
console.log(soma()); // 110
```

arrow function permite uma sintaxe curta, sem necessidade de usar o operador `function` e usando `=>` para montar a sintaxe.

```
/*
 * function expression, com arrow function e
 * sintaxe curtíssima.
 */

let soma = (x = 10, y = 100) => x + y;

console.log(soma(6, 1)); // 7
console.log(soma(5.87, 5.22)); // 11.09
console.log(soma(7.87)); // 107.87
console.log(soma()); // 110
```

há ainda uma construção bem simples quando se quer apenas retornar algo. Não precisa do `{` e `}` e tampouco do `return` para devolver o resultado.

function expression vs arrow

apenas uma questão **sintática**

```js
/*
 * function expression
 * Hoisting não é permitido com este tipo de declaração.
 */

let soma = function(x = 10, y = 100) {
    x = typeof x == 'number' ? x : 0;
    y = typeof y == 'number' ? y : 0;

    const resultado = x + y;
    return resultado;
}

console.log(soma(6, 1)); // 7
console.log(soma(5.87, 5.22)); // 11.09
console.log(soma(7.87)); // 107.87
console.log(soma()); // 110
```

```js
/*
 * function expression, com arrow function
 * Hoisting não é permitido com este tipo de declaração.
 */

let soma = (x = 10, y = 100) => {
    x = typeof x == 'number' ? x : 0;
    y = typeof y == 'number' ? y : 0;

    const resultado = x + y;
    return resultado;
}

console.log(soma(6, 1)); // 7
console.log(soma(5.87, 5.22)); // 11.09
console.log(soma(7.87)); // 107.87
console.log(soma()); // 110
```


VAMOS PRATICAR
programming

revendo a prática

princípio básico

1. funções que consomem funções.
2. funções que consomem funções dentro do seu próprio escopo.
3. as **arrow functions** são bastante interessantes, mas requer uma especial atenção para construir um código que todos possam entender bem.