

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

JavaScript

Funções



JS

JavaScript

parte 8

aprendendo Javascript

agenda

comum em todas as linguagens de programação, estruturas conhecidas como **functions** também estão presentes no JavaScript.



definindo funções

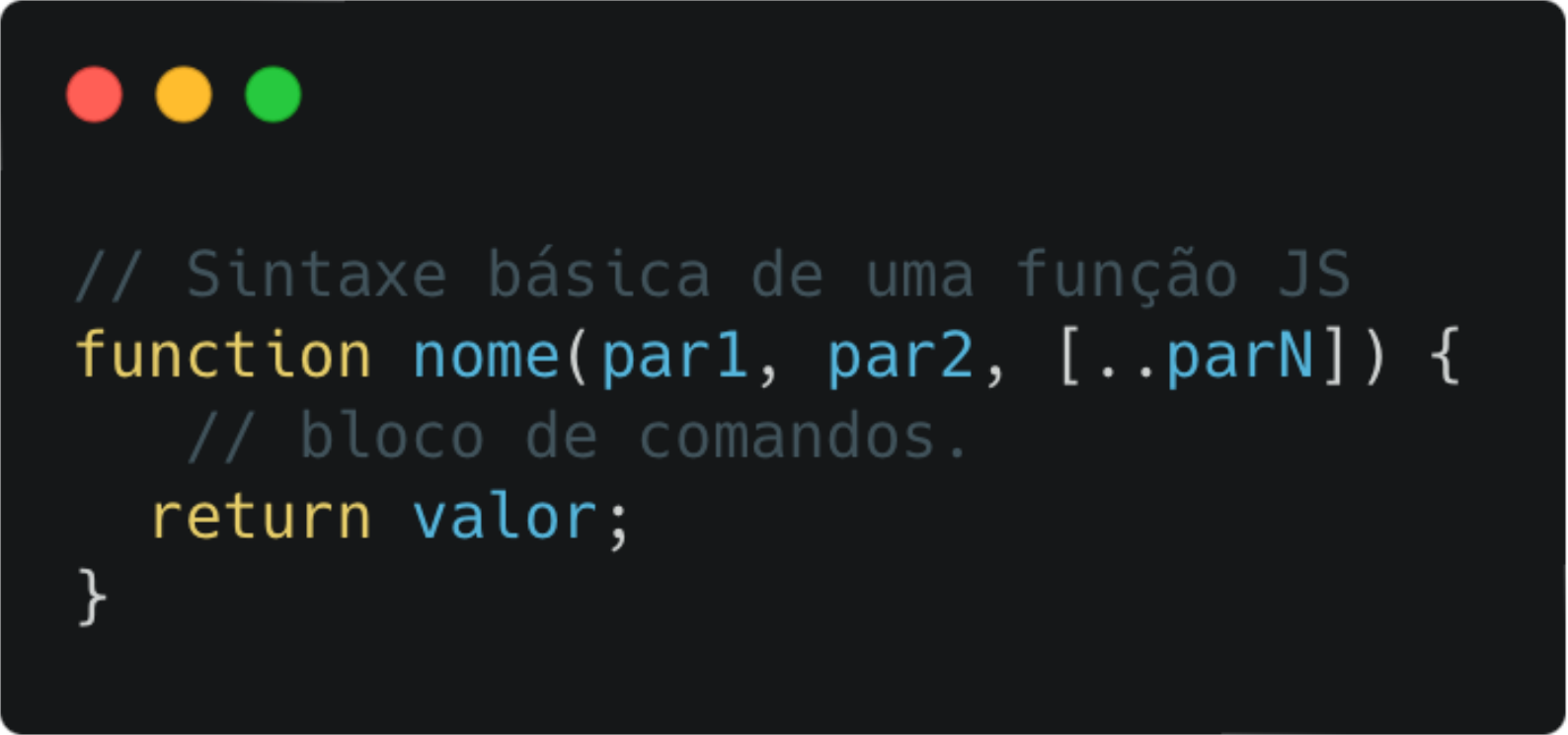
functions

imagine a situação em que um bloco de código seja utilizado frequentemente durante todo o seu programa. imagine como seria escrever este bloco de código várias vezes para atender a esta necessidade.

agora imagina como seria dar manutenção em todo este código repetitivo na estrutura do seu programa ?

inviável, contraproducente, demasiado complexo ou qualquer outro adjetivo desta natureza seria sua resposta. E é por isto que assim como em outras linguagens o JavaScript implementa o conceito de `functions` ou funções. e para manter a sina de ser uma linguagem moderna e diferente, JavaScript implementa **functions** de algumas **formas** bem peculiares como veremos a seguir.

a primeira construção de `function` é bem simples e muitíssimo parecida com de outras linguagens de programação.

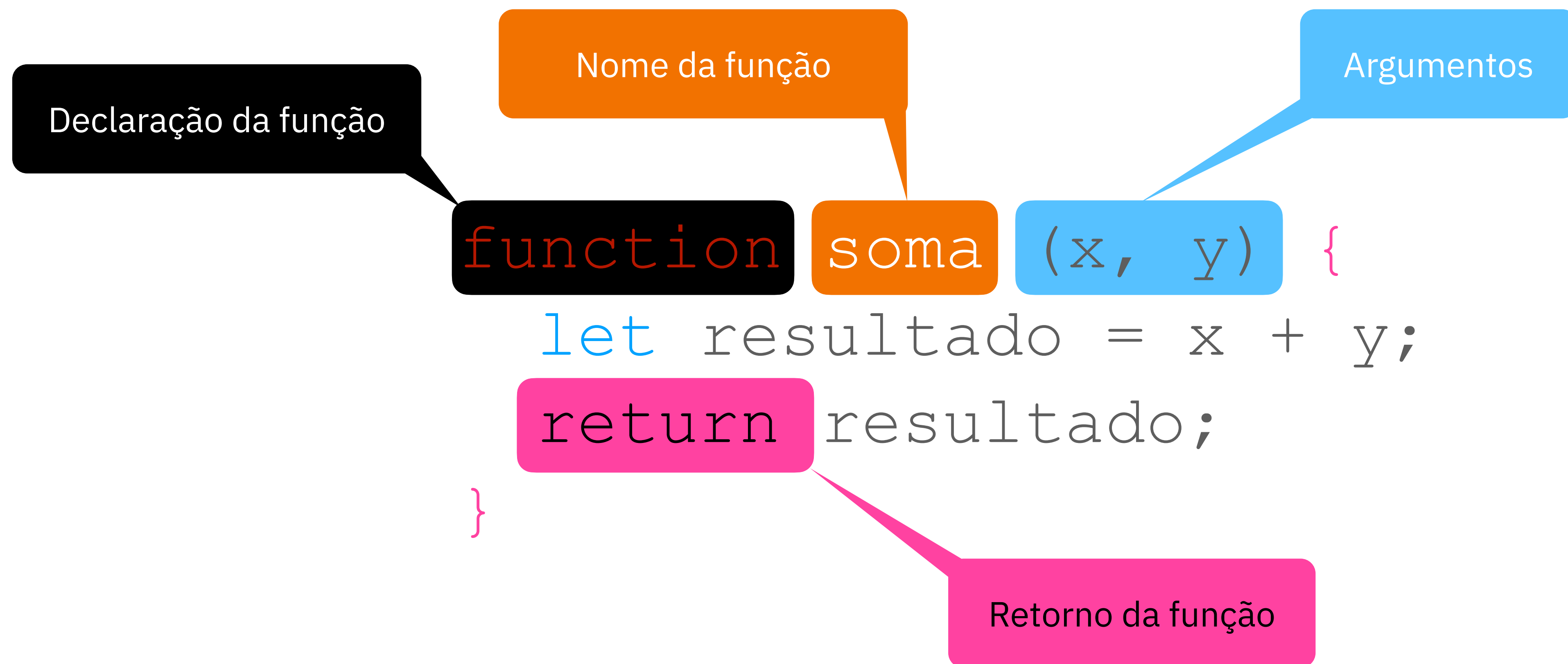


```
// Sintaxe básica de uma função JS
function nome(par1, par2, [..parN]) {
  // bloco de comandos.
  return valor;
}
```

funções são **blocos de construção fundamentais** em JavaScript. Uma função é um procedimento de JavaScript - um conjunto de instruções que executa uma tarefa ou calcula um valor. Para usar uma função, você deve defini-la em algum lugar no escopo do qual você quiser chamá-la.

estrutura da função

function declaration



um simples exemplo

quando usar functions



```
// Função que retorna a soma de dois números.  
function soma(x, y) {  
    const resultado = x + y;  
    return resultado;  
}  
  
// 1. como chamar a função atribuindo o resultado.  
const resultadoSoma = soma(6, 1);  
console.log(resultadoSoma);  
  
// 2. invocando a função e exibindo o resultado.  
console.log(soma(10, 5));
```

o exemplo a esquerda, mostra como se declara uma função que recebe dois parâmetros inteiros. (Veja que não há declaração de tipo)

ela soma os dois valores e atribui a uma constante e retorna o resultado para onde houve a invocação.

na primeira chamada, uma constante recebe o valor da soma e depois é exibida na console. na segunda forma, o comando `console.log` ecoa diretamente o resultado sem necessidade de armazenar primeiro para depois exibir. isto não é uma regra e sim uma escolha. se não é necessário guardar, pode-se exibir direto.

functions sofre hoisting

a engine eleva a function ao topo do escopo



```
// Funções sobre hoisting sempre.
```

```
console.log(soma(19, 35));  
console.log(soma(14.6, 18.87)); // soma fracionários  
console.log(soma(5, 2));
```

```
// Função que retorna a soma de dois números.
```

```
function soma(x, y) {  
  const resultado = x + y;  
  return resultado;  
}
```

em termos lógicos, parece não fazer sentido um código JavaScript começar chamando uma **function** que ainda não foi escrita.

a engine do JS vai aplicar **hoisting** ou elevação para que a `function` possa existir em memória antes de ser invocada pelo `console.log` das três primeiras instruções do nosso código.

todavia recomenda-se não usar deste protocolo já que em algumas situações não muito claras, pode acontecer do motor do Javascript apresentar erros.

coisas que você já sabe

tipagem fraca pode criar aberrações



```
// Função que retorna a soma de dois números?  
function soma(x, y) {  
  const resultado = x + y;  
  return resultado;  
}  
  
// Tipagem fraca pode ser polêmico o resultado.  
console.log(soma(19, 'X'));  
console.log(soma('14.6', 18.87));  
console.log(soma('5', '2A'));
```

a função recebeu um parâmetro `number` e um `string` e como já sabemos tomou a decisão de escolher o que fazer. Na teoria é bom, mas na prática se você não toma cuidado pode ser surpreendido. Não há erro.



```
// Resultado.
```

```
"19X"  
"14.618.87"  
"52A"
```


cuidado com tipos

previnindo do *undefined*

```
// Valor padrão para os parâmetros undefined
function soma(x = 1, y = 1) {
    // se for undefined por padrão receberá 1
    return x + y; // retornando direto.
}

console.log( soma(10, 6) ); // retornará 16
console.log( soma(10) ); // retornará 11
console.log( soma(undefined, 5) ); // retornará 6
console.log( soma() ); // retornará 2

// omitir o primeiro parâmetro retorna erro.
console.log( soma( , 6) ); // gera exceção.
```

a função agora verifica se o tipo da variável é `undefined` e caso seja, assinala com 1. Isto deve ser muito bem documentado no seu código para melhorar futuro entendimento do seu comportamento. Valores **defaults** em funções deve ficar bem claro para quem consome.

na prática não é possível omitir o primeiro parâmetro sem informar por exemplo um `undefined`. lembrando que `null` (nulo) não é `undefined`.

cuidado com tipos

abordagem ótima é testar



```
// Teste tipo e evite surpresas.  
function soma(x, y) {  
  x = typeof x !== 'number' ? 0 : x;  
  y = typeof y !== 'number' ? 0 : y;  
  
  return x + y;  
}  
  
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 5  
console.log( soma(5, true) ); // retornará 5  
console.log( soma(false, true) ); // retornará 0  
console.log( soma() ); // retornará 0
```

a função agora verifica se o tipo da variável é diferente do tipo `number` e assinala com 0. Isto deve ser muito bem documentado no seu código para melhorar futuro entendimento do seu comportamento.

esta é uma proteção muito importante quando se escreve funções em lógicas matemáticas, saldos, limites que necessariamente precisam de tipos `number`.

valores padrões

dupla proteção

```
// Teste tipo e evite surpresas.
```

```
function soma(x = 0, y = 0){  
  x = typeof x !== 'number' ? 0 : x;  
  y = typeof y !== 'number' ? 0 : y;  
  
  return x + y;  
}
```

```
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 5  
console.log( soma(5, true) ); // retornará 5  
console.log( soma(false, true) ); // retornará 0  
console.log( soma() ); // retornará 0
```

```
// Teste tipo e evite surpresas.
```

```
function soma(x = 1, y = 0){  
  x = typeof x !== 'number' ? 0 : x;  
  y = typeof y !== 'number' ? 0 : y;  
  
  return x + y;  
}
```

```
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 0
```




VAMOS PRATICAR

programming

revendo a prática

princípio básico

1. vimos que JavaScript é possível criar functions de várias maneiras e quando se usa **function declaration** ela sofre hoisting.
2. é possível dar valores padrões para os argumentos das variáveis quando se declara na assinatura da função.
3. deve-se tomar um cuidado especial com os tipos.

