JavaScript

CJR

Aula 2

Condicionais

Operador Condicional Ternário (? :)

- Sintaxe:

```
condition ? expr1 : expr2
```

- Se ***condition*** é um valor truthy , o operador retornará o valor de ***expr1***; se não, ele retorna o valor de ***exp2***.
- Ele é frequentemente usado como um atalho para a instrução if.
- É muito utilizado em React para renderizações condicionais.
- Exemplos:

```
console.log("A taxa é " + (isMember ? "R$2.00" : "R$10.00"))

const period = age > 12 ? "Teenager" : "Kid"
```

referência ↗

Ternários aninhados

- É possível aninhar múltiplos ternários (equivale a if...else aninhados):

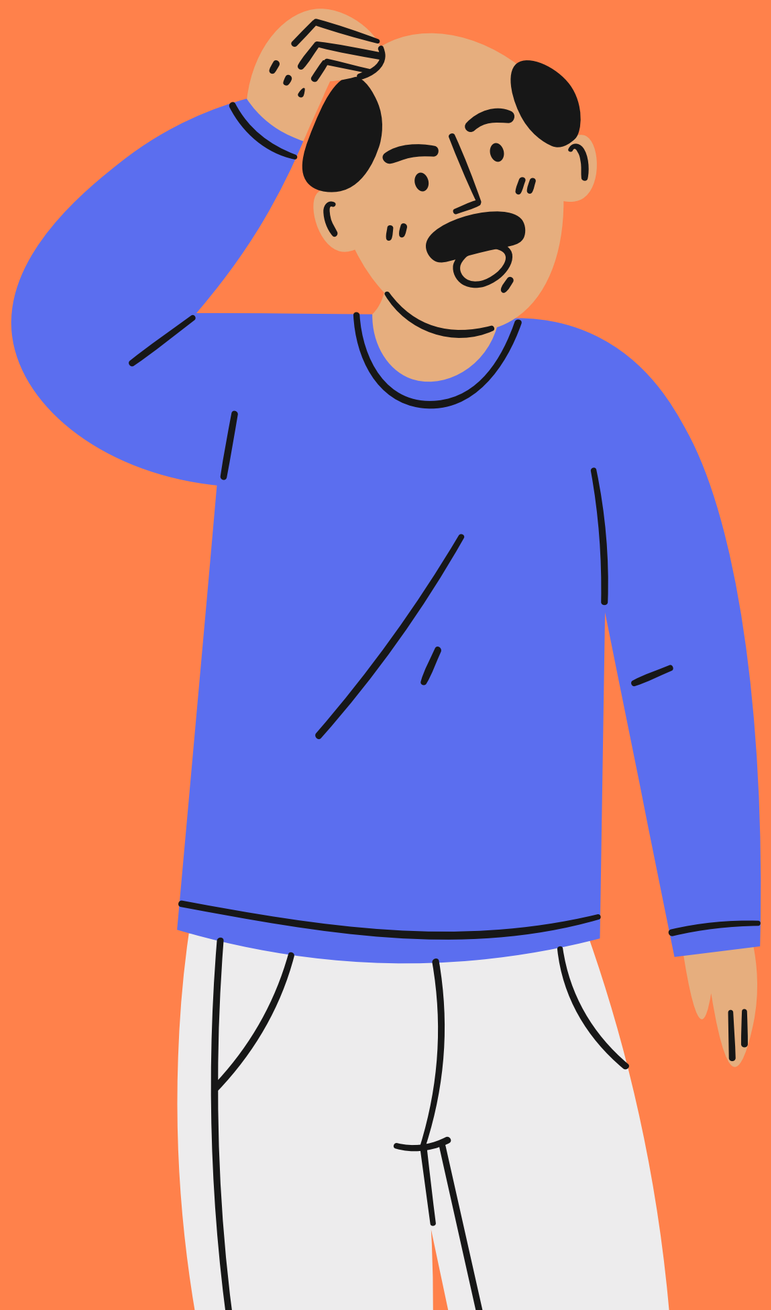
```
const period = age  < 12 ? "Kid" : age < 18 ? "Teenager" : "Adult"
```

- Não é uma boa prática, pois deixa o código menos legível, prefira usar if...else.

Switch

referência ↗

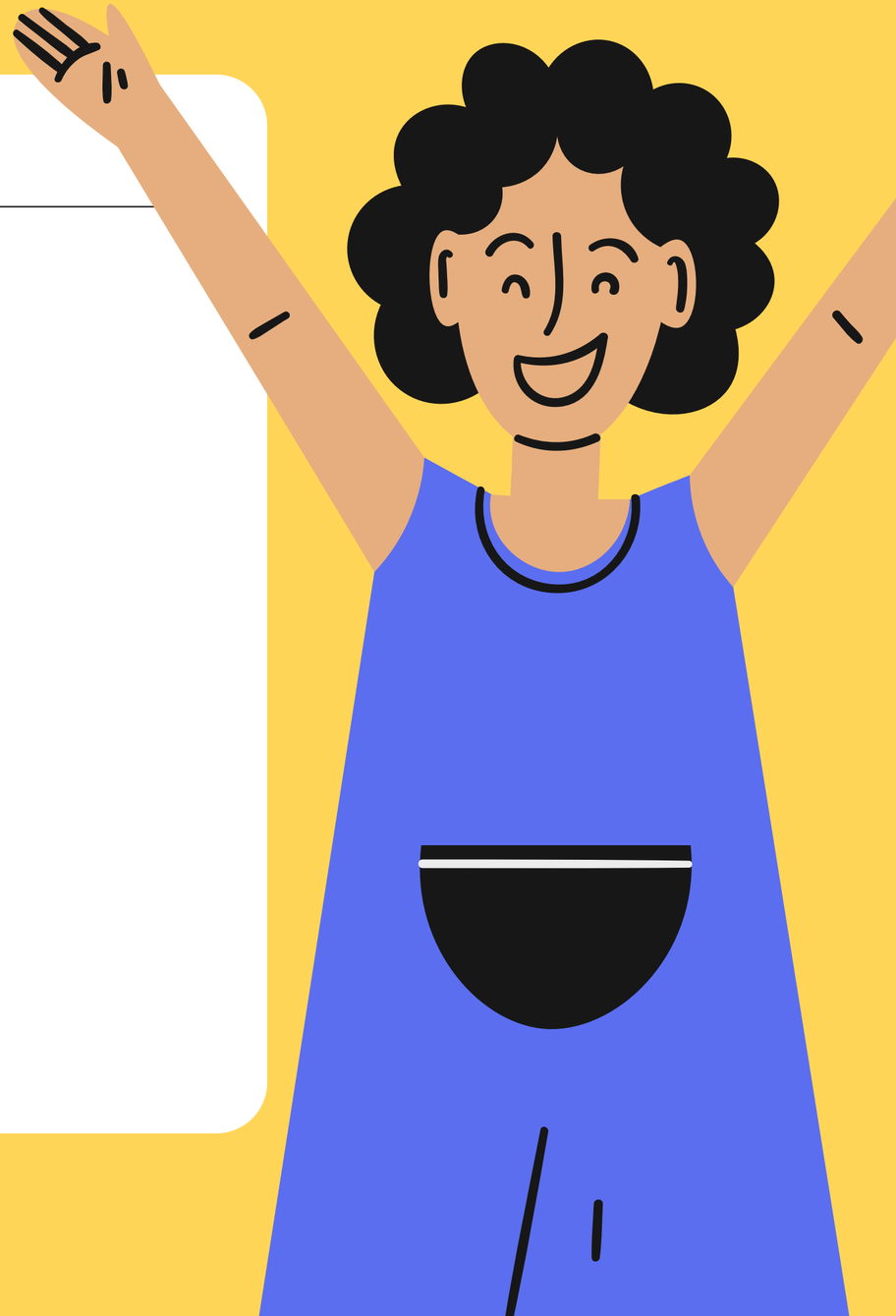
Substituindo a instrução switch por Object Literal ↗



Dúvidas?

Funções

referência ↗



- De modo geral, função é um "subprograma" que pode ser chamado por código externo à função.
- Assim como o programa em si, uma função é composta por uma sequência de instruções chamada corpo da função. Valores podem ser passados para uma função e ela vai retornar um valor.
- No JavaScript, há 3 formas de criar funções: ***function declaration***, ***function expression***, ***arrow function expression.***
- As funções podem ser **nomeadas** ou **anônimas**

Function declarations

- Para declarar uma função, utilizamos a palavra reservada ***function*** seguida do **nome** da função.
- Para executar a função, utilizamos o nome da funções seguida de dois parênteses.

```
function helloWorld() {
    console.log("Hello World!")
}

helloWorld() // "Hello World!"
helloWorld() //"Hello World!"
```

Parâmetros

- Funções podem receber valores ou variáveis como parâmetros.
- Se forem passados menos argumentos do que a função espera, é atribuído **undefined** aos parâmetros restantes.
- No JavaScript, argumentos são sempre passadas por valor, ou seja, o valor da variável passada como argumento é copiado para o parâmetro.

```
function sayHello(name) {
    console.log(`Hello ${name}! 👋`)
}

sayHello("Rafael") // Hello Rafael! 👋
sayHello("Fernando") // Hello Fernando! 👋
```

```
function sum(a, b, c) {
    // a: 2, b: undefined, c: undefined
    console.log(`a: ${a}, b: ${b}, c: ${c}`)
}

sum(2)
```

```
let number = 2

function increment(number) {
    number = number + 1
    console.log(number)
}

increment(number) // 3
console.log(number) // 2
```

Parâmetros padrões

- É possível definir um valor padrão para um parâmetro o qual será atribuído caso o parâmetro não seja passado, ou caso seu valor seja **undefined**.

```
function sayHello(name = 'guy') {
    console.log(`Hello ${name}! 👋`)
}

sayHello('Rafael') // Hello Rafael! 👋
sayHello() // Hello guy! 👋
sayHello(undefined) // Hello guy! 👋
```

```
function sayHello(name) {
    name = name || 'guy'
    console.log(`Hello ${name}! 👋`)
}
```

(o equivalente sem default params)

Retorno

- Funções podem retornar valores, utilizando a palavra reservada return.
- Se a função não retornar nada, é retornado undefined por padrão.

```
function sum(a, b, c) {
    return a + b + c
}

const sum = sum(1, 2, 3)

console.log(sum) // 6
```

Higher Order Functions (HOF)

- "Uma **higher order function** (função de ordem superior) é uma função que recebe uma função como argumento ou retorna uma função."

- Em JavaScript, funções podem ser HOF, ou seja, além de receber/retornar variáveis ou valores, é possível receber/retornar funções.

- Isso possibilita a passagem de ***callbacks:***

  "Em programação de computadores, um método de callback é uma rotina que é passada como parâmetro para outro método. É esperado então que o método execute o código do argumento em algum momento. A invocação do trecho pode ser imediata, como em um (callback síncrono), ou em outro momento (callback assíncrono)"

Higher Order Functions (HOF)

```
function greeting(name) {
    alert('Olá ' + name)
}

function processUserInput(callback) {
    const name = prompt('Por favor insira seu nome.')
    callback(name)
}

processUserInput(greeting)
```

Função como parâmetro

Função como retorno

```
function randomFactory(min, max) {
    return function () {
        return Math.random() * (max - min) + min
    }
}

const getRandomBetween5and10 = randomFactory(5, 10)

console.log(typeof getRandomBetween5and10) // function
console.log(getRandomBetween5and10()) // 7.539457297
console.log(getRandomBetween5and10()) // 5.441404455
```

Closures

- Uma closure é uma função declarada dentro de outra função, e que tem acesso ao estado ao redor dela

```
function randomFactory(min, max) {
    function getRandomBetweenMinAndMax() {
        return Math.random() * (max - min) + min
    }

    return getRandomBetweenMinAndMax
}

const getRandomBetween5and10 = randomFactory(5, 10)
console.log(getRandomBetween5and10()) // 7.539457297
```

Function expressions

- Funções também podem ser criadas por uma **expressão de função.**
- As **expressões de função** podem ser anônima; não precisam ter um nome.
- Elas são convenientes ao passar uma função como um argumento para outra função.

```
const square = function (number) {
    return number * number
}

square(4) // 16
```

```
function processUserInput(callback) {
    const name = prompt('Por favor insira seu nome.')
    callback(name)
}

processUserInput(function (name) {
    alert('Olá ' + name)
})
```

Arrow functions

- É uma sintaxe mais curta para criar ***function expressions.***

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
// equivalente a: => { return expression; }

// Parênteses são opcionais quando só há um nome de parâmetro:
(singleParam) => { statements }
singleParam => { statements }

// A lista de parâmetros para uma função sem parâmetros deve ser escrita com um par de
parênteses.
() => { statements }
```

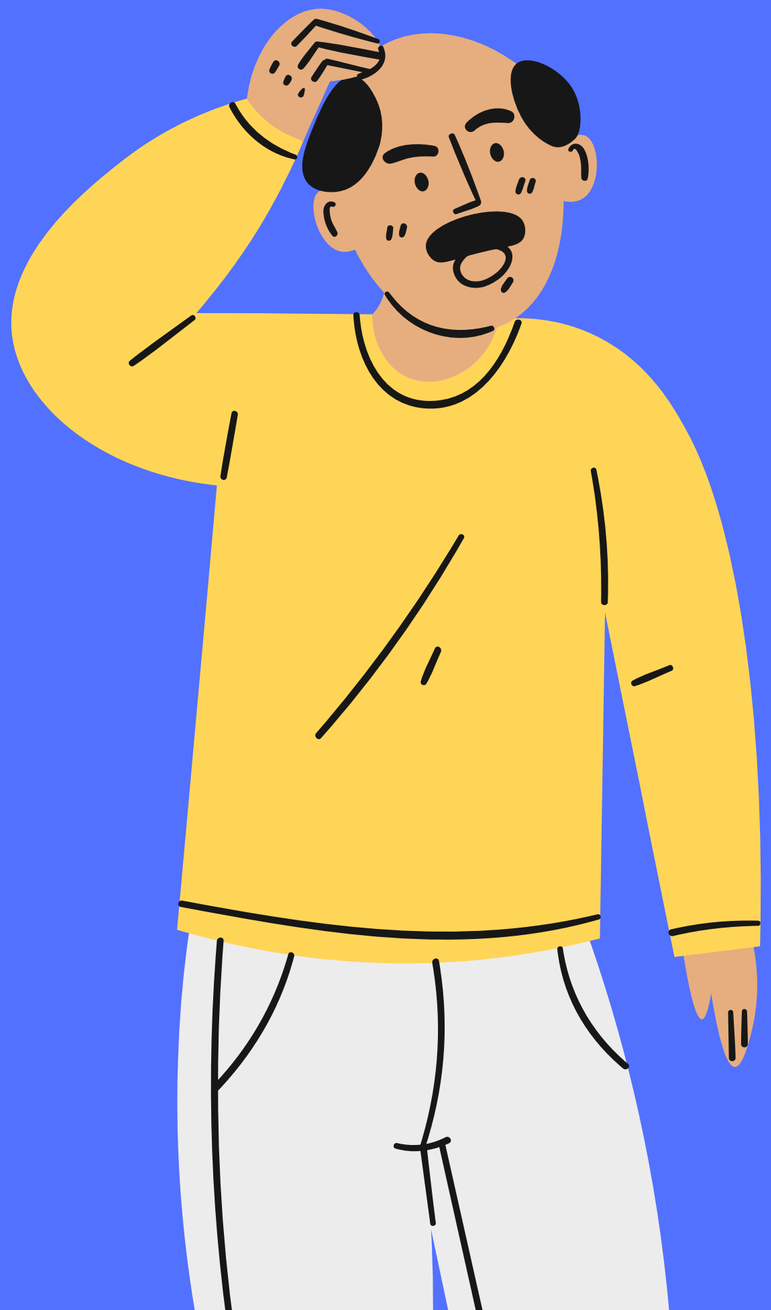
Arrow functions

```
const square = (number) => number * number
square(4) // 16
```

```
function randomFactory(min, max) {
    return () => Math.random() * (max - min) + min
}

const getRandomBetween5and10 = randomFactory(5, 10)
console.log(getRandomBetween5and10()) // 7.539457297
```

```
function processUserInput(callback) {
    const name = prompt('Por favor insira seu nome.')
    callback(name)
}

processUserInput((name) => {
    alert('Olá ' + name)
})
```

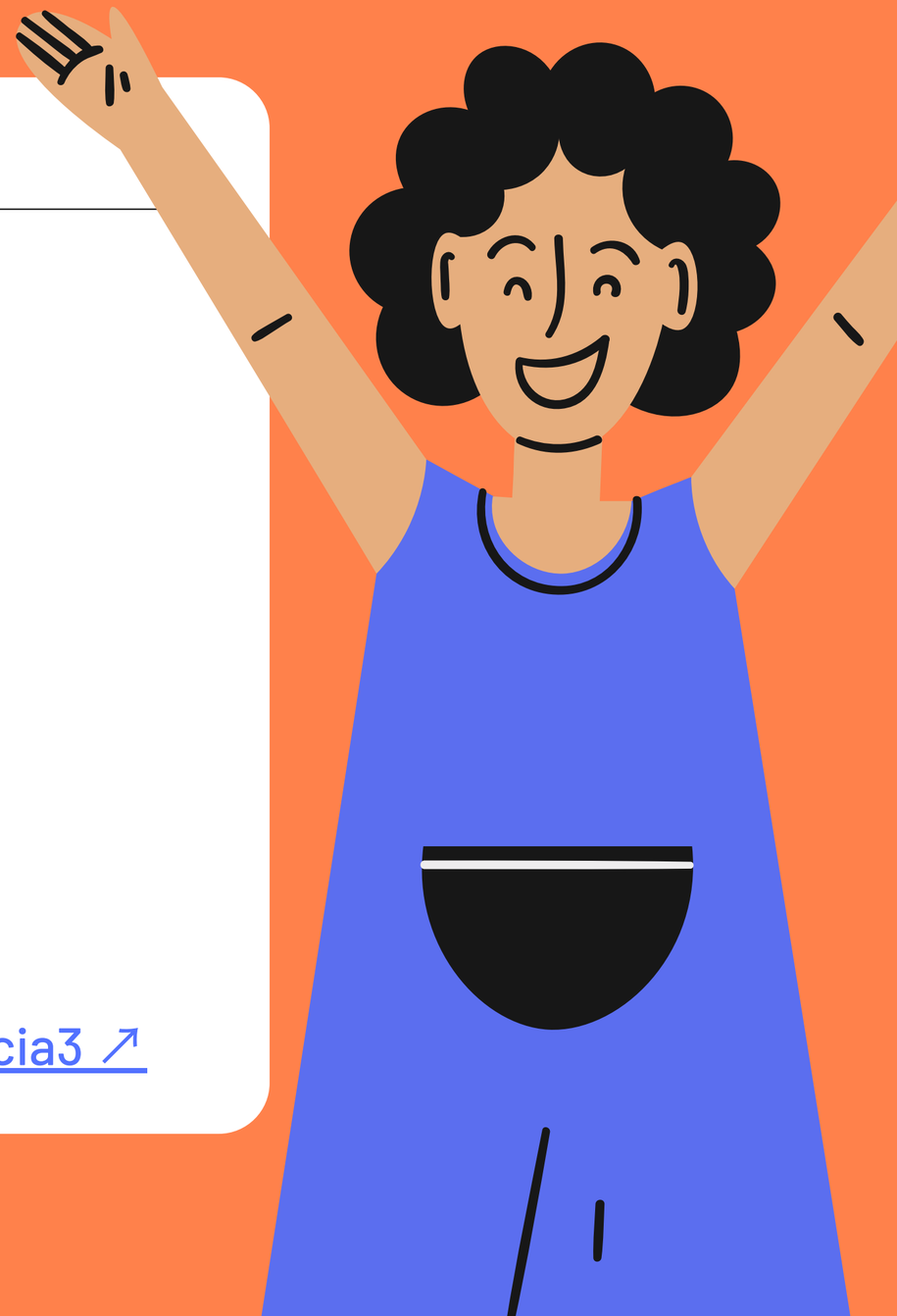
referência ↗



Dúvidas?

Objetos

referência1 ↗ referência2 ↗ referência3 ↗



- Um objeto é utilizado quando queremos armazenar mais de um valor em uma mesma estrutura.
- É uma coleção de propriedades nomeadas que possuem um valor (pares chave-valor) (é parecido com os dicionários do python ou as structs do C).
- Quando uma propriedade de um objeto é uma função, ela é chamada de método.
- Para criar um objeto, envolvemos pares de **chave: valor** separados por vírgula com um par de chaves ({})

```
const obj = {
    propriedade: 'valor',
    método: () => {
        console.log('função')
    }
}
```

- Também podemos utilizar **strings** ou **números** como chaves (ou symbols).

```
const objeto = {
    0: 'valor',
    'minha prop': 2
}
```

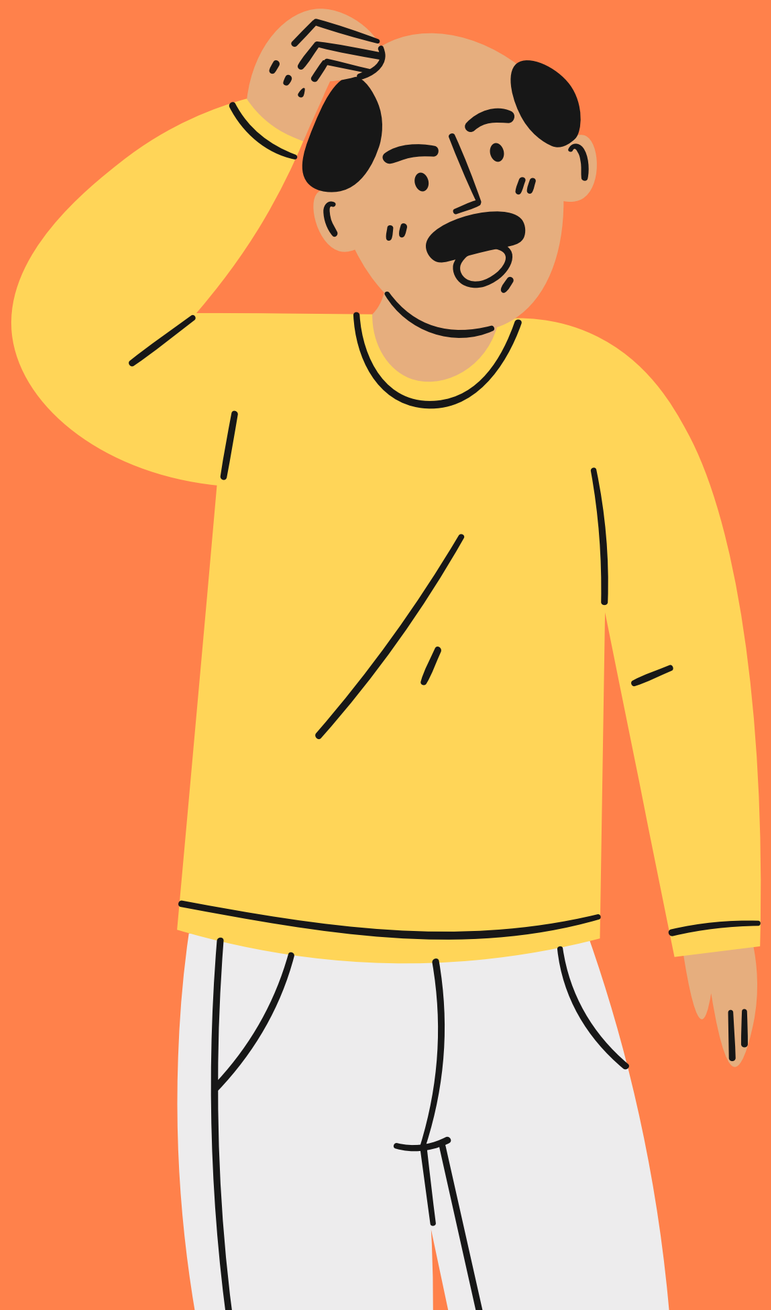
- Para acessar uma propriedade/método podemos utilizar o ponto (.) ou colchetes ([ ])
- Geralmente sempre utilizamos o ponto, usamos o colchetes quando o nome da propriedade não tem um nome válido (ex: contém espaços), quando queremos utilizar uma variável para acessar a propriedade ou quando a chave é um número.

```
const obj = {
    propriedade: 'valor',
    método: () => {
        console.log('função')
    },
    0: 'valor2',
    'minha prop': 2
}
```

```
obj.propriedade // 'valor'
obj['minha prop'] // 2
obj[0] // 'valor2'

obj.método()
obj['método']()


const propName = 'propriedade'
obj[propName] // 'valor'
```

- Se tentarmos acessar propriedades que não existem, é retornado **undefined**.

```
const objeto = {
    chave: 'valor'
}

objeto.naoExiste // undefined
```



Dúvidas?

Objetos aninhados

- Objetos são um tipo de dado como outro qualquer e podem ser utilizados como valores de um outro objeto.

```
const user = {
    name: 'Rafael',
    age: 19,
    address: {
        city: 'Brasília',
        UF: 'DF'
    }
}

console.log(user.address.city) // 'Brasília'
```

Definições de propriedade

- Também podemos definir ou alterar as propriedades de um objeto depois dele ser inicializado.

```
const user = {
    name: 'Não é o Rafael',
}

user.name = 'Rafael'
user.age = 19
user.sayHello = () => console.log("Hello")

user.name // Rafael
user.age // 19
user.sayHello() // Hello
```

this

- Nesse contexto, a palavra-chave this se refere ao objeto atual em que o código está sendo escrito — nesse caso o this se refere a user.

```
const user = {
    name: 'Rafael',
    age: 19,
}

user.happyBirthday = function () {
    this.age++
}

function describe() {
    console.log(`My name is ${this.name} and I am ${this.age}.`)
}

user.describe = describe

user.age // 19
user.happyBirthday()
user.age // 20
user.describe() // My name is Rafael and I am 20.
```

referência ↗

Objetos são referências

- Objetos armazenam a referência para seus dados (são ponteiros).
- Por isso, ao copiar um objeto para outra variável e depois modificá-la, o conteúdo do objeto original também é modificado. O que é copiado é a referência.
- Por isso também que que const "não funciona" para objetos, pois o que se mantém constante é a referência (que é o que realmente é o valor do objeto).

```
const user1 = {
    name: 'Rafael',
    email: 'rafaelrodrigues@cjr.ogr.br'
}

const user2 = user1
user2.name = 'Fernando'

user1.name // Fernando
```

Objetos são referências

- Outra consequência disso é que dois objetos distintos nunca são iguais, mesmo que tenham as mesmas propriedades. Apenas comparando o mesmo objeto de referência com ele mesmo produz verdadeiro.

```
// Duas variáveis, dois objetos distintos com as mesmas propriedades
var fruit = {name: "apple"};
var fruitbear = {name: "apple"};

fruit == fruitbear // return false
fruit === fruitbear // return false
```

```
// Duas variáveis, um único objeto
var fruit = {name: "apple"};
var fruitbear = fruit;  // assign fruit object reference to fruitbear

// Here fruit and fruitbear are pointing to same object
fruit == fruitbear // return true
fruit === fruitbear // return true
```

Qual será a saída no console para o seguinte código? Por quê?

```
const user = {
    name: 'Rafael',
    age: 19,
}

const addLastName = (user) => {
    user.name = `${user.name} Rodrigues`
}

addLastName(user)

console.log(user.name)
```

Objetos são referências

- Em funções, vimos que "No JavaScript, argumentos são sempre passadas por valor, ou seja, o valor da variável passada como argumento é copiado para o parâmetro."

- Como objetos são referências, o objeto recebido na função vai ter a mesma referência do que foi passado. Ou seja, funções podem modificar os dados de um objeto recebido como parâmetro.

```
const user1 = {
    name: 'Rafael',
    age: 19,
}

const addLastName = (user) => {
    user.name = `${user.name} Rodrigues`
    return user
}

const user2 = addLastName(user1)

user1.name // Rafael Rodrigues
user1 === user2 // true
```

Tudo é objeto

- No JavaScript, tudo é objeto, inclusive os **tipos primitivos** e **funções**, que são encapsulados em objetos e possuem algumas propriedades disponíveis por padrão.
- Temos disponíveis globalmente diversos objetos, como o console.

Objeto String↗

Objeto Number↗

Encadeamento opcional

- O operador de encadeamento opcional (?.) provê uma forma de simplificar o acesso a valores através de objetos conectados, quando é possível que uma referência ou função possa ser undefined ou null.

```
let nestedProp = obj.first && obj.first.second;
```

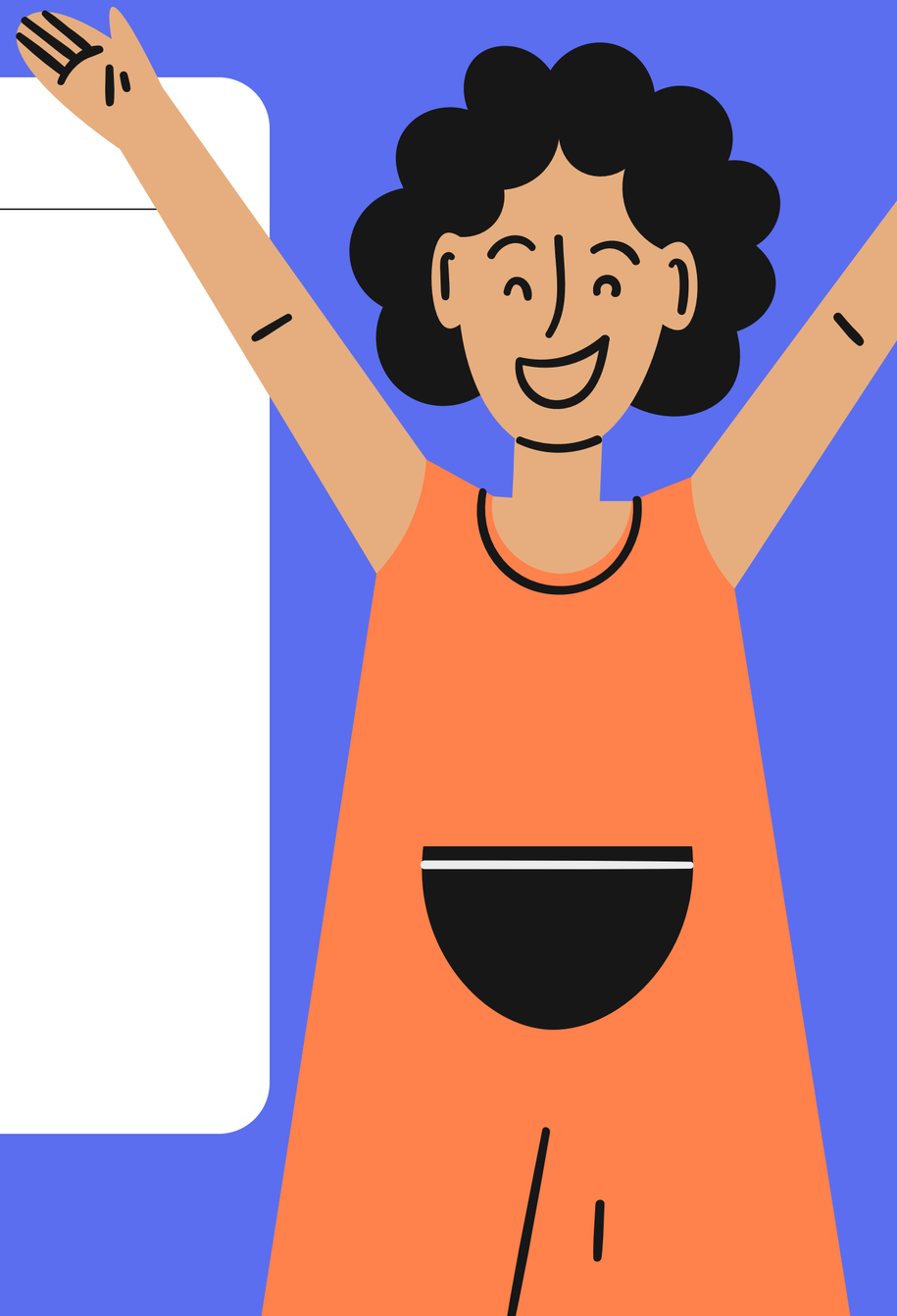
```
let nestedProp = obj.first?.second;
```

[referência ↗]

Arrays

referência1 ↗

referência2 ↗



- Array (a.k.a vetor, lista) é um meio de armazenar uma lista de itens em uma mesmo variável.
- É possível armazenar qualquer tipo de dado em um array e podemos acessar cada valor dentro dele individualmente.
- Os items de um array não precisam ser todos do mesmo tipo, eles podem ser misturados.
- Para definir um array, usamos colchetes ([ ]) e colocamos dentro dele cada item separado por vírgula .

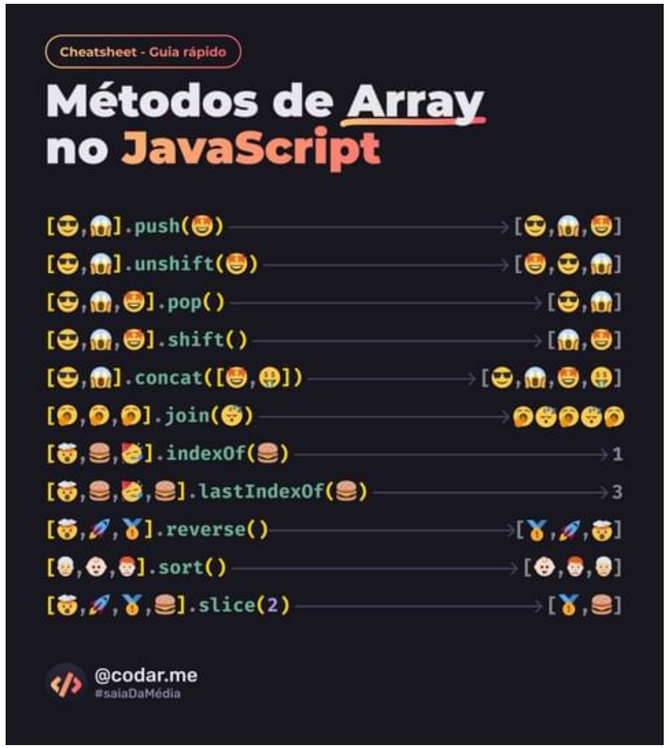
```
const fruits = ['apple', 'pear', 'banana', 'pineapple']

const randomArray = ['string', 1, true, { object: true }]
```

- Podemos acessar os itens de um array por meio da sintaxe de colchetes, colocando o index do elemento dentro dele.
- No JavaScript, o index de um array começa pelo 0.
- Podemos alterar o valor de um item do array depois dele ter sido inicializado.

```
const fruits = ['apple', 'pear', 'banana', 'pineapple']

fruits[0] // 'apple'
fruits[1] // 'pear'

fruits[2] = 'melon'
fruits[2] // 'melon'
```

Arrays são objetos

- Arrays também são objetos, ou seja, tem propriedades e métodos intrínsecos.



Fim!