**Intensivão JavaScript 🔥**

**Tópicos Avançados**

**Parte 1: Closures em JavaScript (❗importante) 🔒**

Um fechamento (closure) é a combinação de uma função agrupada (incluída) com referências ao seu estado circundante (o ambiente léxico). Em outras palavras, um closure fornece acesso ao escopo de uma função externa a partir de uma função interna. Em JavaScript, os closures são criados toda vez que uma função é criada, no momento da criação da função.

No exemplo abaixo, prefix é uma variável com fechamento.

const greet = () => {

const prefix = 'Mr'

return (name) => {

console.log(`${prefix} ${name}, welcome!`)

}

}

console.log(greet()('Codi'))

Esta seção terá muitas palavras bonitas, então tenha paciência comigo. Nós os cobriremos um por um.

Uma função agrupada com seu ambiente lexical forma um fechamento. Ok, o que é um ambiente lexical? É essencialmente o estado circundante – a memória local junto com o ambiente lexical de seu pai.

O quê? 🤯 Eu sei que é um pouco estranho. Vamos entender com um exemplo simples.

function x() {

var a = 7

function y() {

console.log(a)

}

return y

}

var z = x()

console.log(z) // [Function: y]

z()

Quando x é chamado, y é retornado. Agora, y está esperando para ser executado. Como uma arma carregada esperando para ser disparada! 🔫

Então, quando finalmente invocamos z, y é invocado. Agora, y tem que exibir o valor de a, então primeiro tenta encontrá-lo 🔍 na memória local, mas não está lá.

Ele vai para sua função pai. Encontra a lá. Voilá! Aí está, este é o Closure (fechamento).

Mesmo quando as funções são retornadas (no caso acima y), elas ainda se lembram de seu ambiente lexical (de onde veio).

Vantagens de Closures em JavaScript 😎

* **Currying (escovar)**

let add = function (x) {

return function (y) {

console.log(x + y)

}

}

let addByTwo = add(2)

addByTwo(3)

* **Ocultação/encapsulamento de dados**

Suponha que você queira criar um aplicativo contador. Toda vez que você chama, a contagem aumenta em 1. Mas você não quer expor a variável fora da função. Como fazer isso? Com fechamentos (Closures).

function Counter() {

var count = 0

this.incrementCount = function () {

count++

console.log(count)

}

}

console.log(count) // Error: count não está definida

var adder = new Counter()

adder.incrementCount() // 1

E já falamos sobre this na aula passada, se não se lembra muito bem do que se trata, volte e dê uma olhadinha lá.

Desvantagens de Closures em JavaScript 😅

Pode ocorrer consumo excessivo de memória ou vazamentos de memória.

Isso ocorre porque, mesmo que a função externa tenha sido executada, a função interna retornada ainda possui uma referência à variável fechada.

Nota: A coleta de lixo basicamente remove variáveis ​​não utilizadas da memória automaticamente.

**Parte 2: Objetos e Bind**

Assim como os arrays, os objetos são uma forma de armazenar dados. Fazemos isso com a ajuda de pares chave-valor.

const programador = {

nome: "Tulio",

idade: 23

}

nome é o key e Tulio é o value. Chaves são geralmente o nome das propriedades do objeto.

O this em JavaScript

Agora, trabalhar com objetos é diferente em JS do que em outras linguagens de programação populares como C++. E para entender isso corretamente, precisamos de uma boa compreensão da palavra chave this.

Vamos entender passo a passo.

Em um programa, às vezes, precisamos apontar para as coisas. Como dizer que esta função aqui pertence a este objeto. O this nos ajuda a obter esse contexto.

Você entenderá melhor o que estou dizendo quando olharmos alguns exemplos.

const programador = {

nome: "Tulio",

idade: 23,

aniversario: function(){

this.idade+=1;

}

}

programador.aniversario()

alert(programador.idade)

Agora, se em vez de usarmos o this dentro de um objeto, fizermos isso dentro de uma função?

function myFunc() {

console.log(this)

}

const obj = {

bool: true,

myFunc: myFunc,

}

obj.myFunc()

Agora, this vai apontar para o objeto. Então, o que está acontecendo aqui? o “this” está ligando ao objeto “Obj”, pois a função que o chamar é um método desse objeto.

Se você fizer:

myFunc() // window

Nesse caso obtemos o objeto window. Então, podemos ver que o valor de this depende de como e onde estamos fazendo a chamada.

O que acabamos de fazer acima é chamado de **Implicit Binding** . O valor de this ficou vinculado ao objeto.

Existe outra forma de usar this. **A ligação explícita** é quando você força uma função a usar um determinado objeto como seu arquivo this.

Vamos entender por que precisamos de vinculação explícita por meio de um exemplo.

const student\_1 = {

name: 'Yuri',

displayName\_1: function displayName() {

console.log(this.name)

}

}

const student\_2 = {

name: 'Tulio',

displayName\_2: function displayName() {

console.log(this.name)

}

}

student\_1.displayName\_1()

student\_2.displayName\_2()

Estamos usando thiscorretamente, mas você pode ver o problema com o código acima?

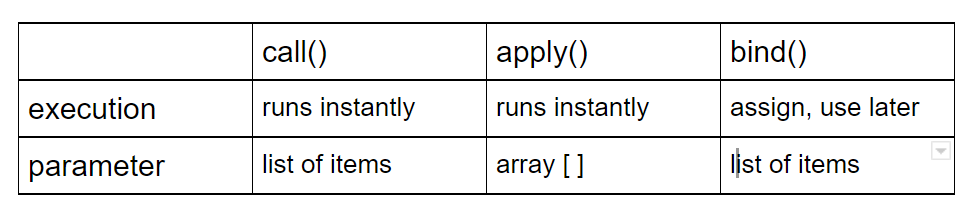
Estamos repetindo o código. E um dos princípios da boa programação é manter seu código LIMPO! (Sem repetições)

Então, vamos nos livrar do displayName\_2 e simplesmente fazer:

student\_1.displayName\_1.call(student\_2) // Tulio

O call forçou displayName\_1 a usar o segundo objeto como seu this.

Há muitas outras maneiras de fazer isso.



**Parte 3: Protótipos e herança prototípica em JavaScript 👪**

Sempre que criamos algo (como um objeto ou função) em JavaScript, o JS Engine automaticamente anexa essa coisa com algumas propriedades e métodos.

Tudo isso vem via **prototypes.**

\_\_proto\_\_ é o objeto onde JS está colocando tudo.

Vejamos alguns exemplos:

let arr = ['Codi', 'Academy']

console.log(arr.\_\_proto\_\_.forEach)

console.log(arr.\_\_proto\_\_) // mesmo que Array.prototype

console.log(arr.\_\_proto\_\_.\_\_proto\_\_) // mesmo que Object.prototype

console.log(arr.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_) // null

Tudo isso é chamado de prototype chain.

Podemos fazer o mesmo com objetos e funções também.

Sempre encontraremos Object.prototype por trás. É por isso que você deve ter ouvido que tudo em JS é um objeto. 🤯

O que é herança prototípica em JavaScript?

let object = {

name: 'Codi Academy',

city: 'Juiz de Fora',

getIntro: function () {

console.log(`${this.name}, ${this.city}`)

},

}

let object2 = {

name: 'Curso',

}

Se fizermos:

object2.\_\_proto\_\_ = object

O object2 obtém acesso às propriedades do objeto. Então, agora podemos fazer:

console.log(object2.city)

Esta é **a herança prototípica**.

**Parte 4: Temporizadores em JavaScript – setTimeout, setInterval, clearInterval ⏱️**

O setTimeout() chama uma função ou avalia uma expressão após um número especificado de milissegundos.

setInterval()faz o mesmo para intervalos especificados.

setTimeout(() => {

console.log('Se passaram 3 segundos!')

}, 3000);

const timer = setInterval(() => {

console.log('Eu vou continuar aparecendo até você me matar com clear!')

}, 2000);

Você usa clearInterval()para parar o cronômetro.

clearInterval(timer)

Agora vejamos a algumas questões de entrevistas:

1. Qual a saída desse programa?

console.log('Olá')

setTimeout(() => {

console.log('Estagiário')

}, 0)

console.log('futuro')

// output

Olá

Futuro

Estagiário

1. Qual a saída desse programa?

for (var i = 1; i <= 5; i++) {

setTimeout(function () {

console.log(i)

}, i \* 1000)

}

// output

6

6

6

6

6

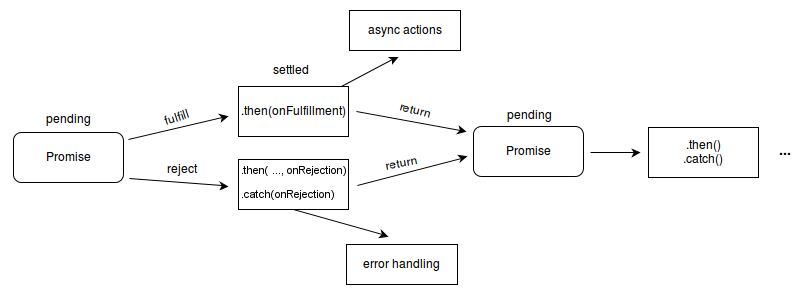
O que ocorre é que quando setTimeout volta à superfície, todo o loop foi executado e o valor de i tornou-se 6.

1. Como fazer com que a saída do programa anterior seja 12345?

Em vez de var➡️ use let.  
E por que isso vai funcionar? var tem escopo global, mas let tem escopo local. Portanto, com let um novo i é criado para cada iteração.

**Parte 5: Promessas em JavaScript (❗importante) 🤝**

O objeto Promise representa a eventual conclusão (ou falha) de uma operação assíncrona e seu valor resultante.



Uma promessa pode estar em um destes três estados:

* Pendente: estado inicial, nem cumprido nem rejeitado
* Cumprido: a operação foi concluída com sucesso
* Rejeitado: falha na operação

const promise = new Promise((resolve, reject) => {

let value = true

if (value) {

resolve('Hey, esse valor é verdadeiro')

} else {

reject('Erro, o valor é falso!')

}

})

promise

.then((x) => {

console.log(x)

})

.catch((err) => console.log(err))

**Parte 6: Debounce em JavaScript ⛹️‍♂️**Em JavaScript, uma função de debounce garante que seu código seja disparado apenas uma vez por cada entrada do usuário.

Debouncing é outro tópico favorito dos entrevistadores.

Vamos entendê-lo criando uma barra de pesquisa.

**Demonstração:** <https://codesandbox.io/s/debounce-input-field-o5gml>

Crie um campo de entrada simples index.html assim:

<input type='text' id='text' />

Agora, em index.js. Não se esqueça de adicioná-lo ao index.html primeiro:

const getData = (e) => {

console.log(e.target.value)

}

const inputField = document.getElementById('text')

const debounce = function (fn, delay) {

let timer

return function () {

let context = this

clearTimeout(timer)

timer = setTimeout(() => {

fn.apply(context, arguments)

}, delay)

}

}

inputField.addEventListener('keyup', debounce(getData, 300))

Primeiro, selecionamos a entrada e adicionamos um event listener a ela. Em seguida, criamos uma função debounce que recebe uma função de retorno de chamada e atraso.

Agora, dentro da função debounce, criamos um timer usando setTimeout. Agora, a função desse timer é garantir que a próxima chamada para getData só aconteça depois de 300 ms. Debouncing é isso.

Além disso, usamos clearTimeoutpara removê-lo. Não quero muitos deles por aí ocupando espaço de memória!

Ufa! Muita teoria. Vamos fazer um desafio divertido. Você deve ter visto a contagem regressiva antes de um jogo começar (é como 10, 9, 8, .... com algum atraso no meio). Tente escrever um programa para ele.

Sugestão:

let count = 10

for (let i = 0; i < 10; i++) {

function timer(i) {

setTimeout(() => {

console.log(count)

count--

}, i \* 500)

}

timer(i)

}

**Parte 7: Gerenciamento de Memória**

**Introdução**  
Linguagens de baixo nível, como C, tem primitivas de gerenciamento de memória de baixo nível como malloc() e free(). Em contrapartida, os valores do JavaScript são alocados quando coisas (objetos, strings, etc.) são criadas e "automaticamente" liberadas quando não são mais usadas. Este último processo se chama garbage collection. Facilmente se torna uma fonte de confusão e dá a impressão aos desenvolvedores JavaScript (e outras linguagens de alto nível) que eles não precisam se preocupar com o consumo de memória. Isto é um erro.

**Ciclo de vida da memória**

Independentemente da linguagem de programação, o ciclo de vida da memória é praticamente sempre o mesmo:

1. Alocar a memória que você precisa
2. Utilizar a memória alocada (ler, escrever)
3. Liberar a memória alocada quando não é mais necessária

A primeira e a segunda parte são explícitas em todas as linguagens. A última parte é explicita em linguagens de baixo nível, porém implícito em linguagens de alto nível como JavaScript.

**Alocação no JavaScript**

**Inicialização de valor**  
A fim de não incomodar o programador com alocações, o JavaScript faz isso com os valores conforme são declarados.  
var n = 123; // aloca memória para um número

var s = "azerty"; // aloca memória para uma string

var o = {

a: 1,

b: null

}; // aloca memória para um objeto e seus valores contidos

// (assim como o objeto) aloca memória para o vetor e

// seus valores contidos

var a = [1, null, "abra"];

function f(a) {

return a + 2;

} // aloca uma função (que é um objeto que pode ser chamado)

// expressões de funções também alocam um objeto

someElement.addEventListener('click', function(){

someElement.style.backgroundColor = 'blue';

}, false);

Alocação via chamada de uma função  
Algumas funções quando chamadas resultam na alocação de um objeto.

var d = new Date();

// aloca um elemento do DOM

var e = document.createElement('div');

Alguns métodos alocam novos valores ou objetos:

var s = "azerty";

var s2 = s.substr(0, 3); // s2 é uma nova string

// Como as strings são valores imutáveis,

// o JavaScript pode decidir não alocar memória,

// mas apenas armazenar o intervalo [0, 3].

var a = ["ouais ouais", "nan nan"];

var a2 = ["generation", "nan nan"];

var a3 = a.concat(a2);

// novo vetor com 4 elementos sendo

// a concatenação dos elementos a e a2

**Utilização de valores**A utilização de valores basicamente significa leitura e escrita em memória alocada. Isto pode ser feito ao ler ou escrever o valor de uma variável ou a propriedade de um objeto ou até mesmo ao passar um argumento para uma função.

Libere quando a memória não for mais necessária

A maioria dos problemas relacionados ao gerenciamento de memória aparecem nesta fase. A tarefa mais complicada aqui é descobrir quando "a memória alocada não é mais necessária". Geralmente exige que o desenvolvedor determine a onde no programa tal pedaço da memória não é mais necessária e liberá-la.

Linguagens de alto nível vêm com um pedaço de software chamado "garbage collector" (coletor de sujeira), cujo trabalho é monitorar a alocação de memória a fim de descobrir quando um pedaço de código não é mais necessário e neste caso, automaticamente liberá-lo. Este processo é algo aproximado já que, em geral, saber se um pedaço de memória é necessário é algo [indecidível](http://pt.wikipedia.org/wiki/Decidibilidade) (que não pode ser resolvido através de um algoritmo).

**Garbage collection**  
Como foi mencionado acima, em geral o problema de automaticamente descobrir se a memória "não é mais necessária" é indecidível. Como consequência, os garbage collections implementam uma limitação de uma solução ao problema em geral. Esta seção irá explicar os conceitos necessários para entender os principais algoritmos de garbage collection e suas limitações.