Innominatum CONSULTORIA

Programação em JavaScript

ServiceNow - Deloitte

Rodrigo Costa

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL

Innominatum CONSULTORIA

JavaScript

Módulo 2

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL

Scope & Closures

Sessão 11

Objetivo geral

Capacitar os formandos na compreensão profunda da mecânica de execução do JavaScript, dominando os conceitos de gestão de memória e ciclo de vida de variáveis para a construção de aplicações robustas. A sessão foca na transição de uma compreensão superficial do código para o domínio do Contexto de Execução, ensinando a prever o comportamento da Call Stack e do Memory Heap como base para a escrita de algoritmos eficientes.

Objetivos específicos

Ao final da sessão, os formandos deverão ser capazes de:

- Explicar como o JavaScript executa código através do contexto de execução, **call stack** e **memory heap**;
- Explicar o conceito de **hoisting** e distinguir o comportamento de funções e variáveis;
- Compreender o conceito de **closure** como mecanismo de preservação de escopo;
- Explicar como o valor de **this** é determinado pelo contexto de chamada da função.

Innominatum CONSULTORIA

Contexto de Execução, Call Stack e Memory Heap

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL

JavaScript é uma linguagem single-thread

Uma thread (fio de execução) é o menor conjunto de instruções que um computador consegue executar de forma independente dentro de um processo. Quanto mais threads nosso processador possuir, mais operações em simultâneo ele é capaz de executar.

JavaScript é uma linguagem single-thread

Isso implica em dizer que o JavaScript (JS) executa o código **passo a passo**, seguindo uma ordem bem definida linha a linha, sem execução paralela, e para isso acontecer, o **motor (V8 do Chrome)** cria os chamados **contextos de execução**.

O que é um contexto de execução?

É um ambiente onde o JS:

- Sabe quais variáveis existem;
- Sabe quais funções estão disponíveis;
- Sabe onde está o this.

Tipos de contexto de execução

1. Contexto de execução global:
    - Criado quando o script começa.
2. Contexto de execução funcional:
    - Criado a cada chamada de função.

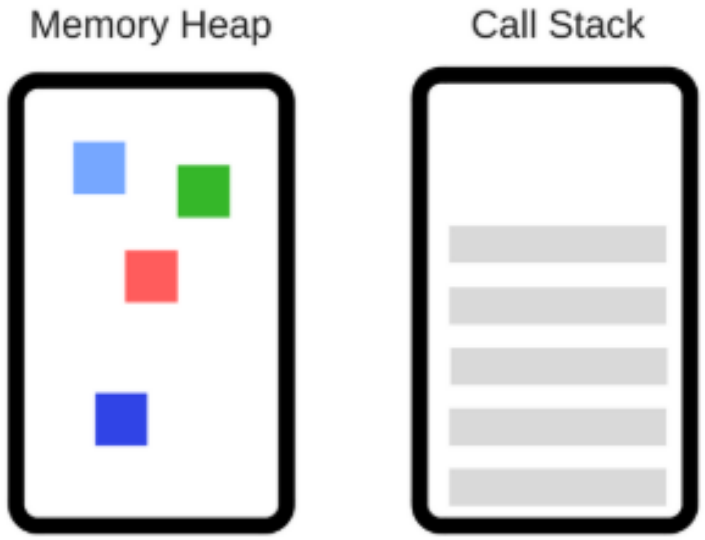
Somente um contexto poderá estar ativo por vez.

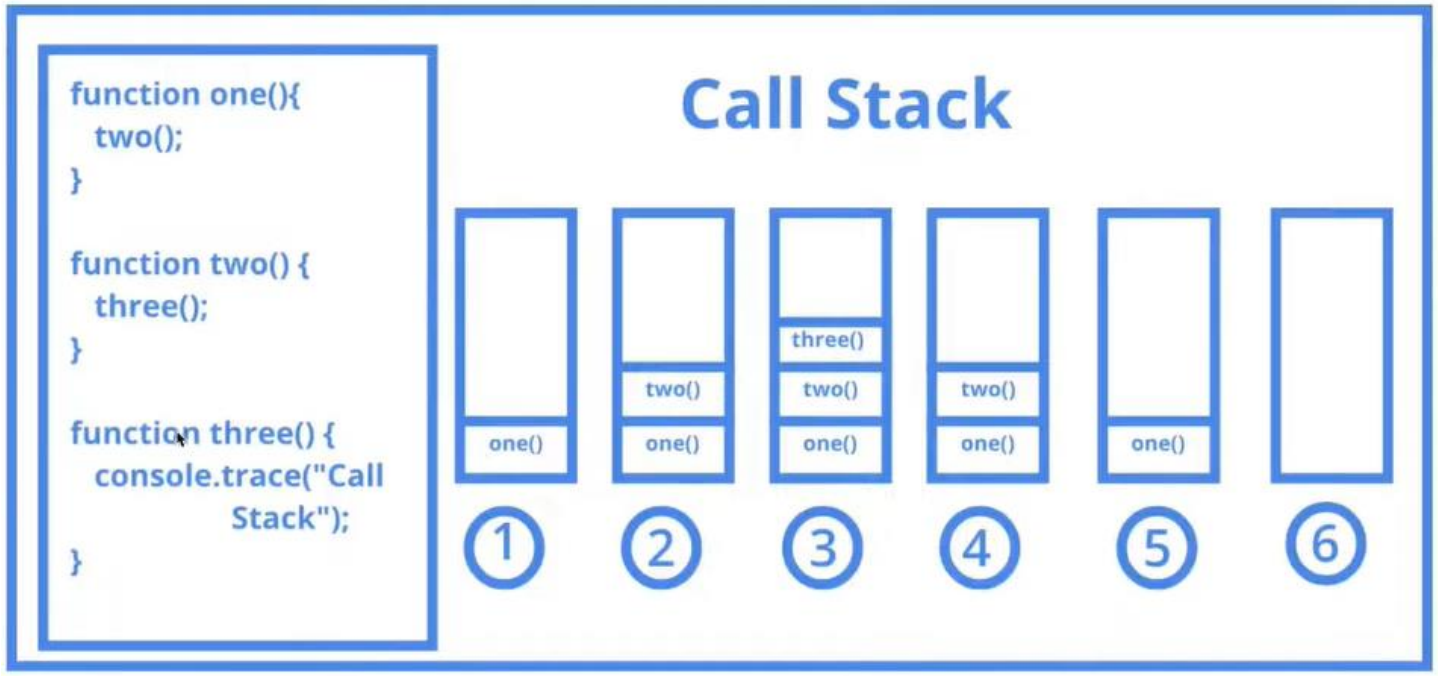
Call Stack x Memory Heap

Quando criado o contexto de execução o **runtime** do JavaScript é dividido em duas áreas principais:

- Memory Heap: É onde iremos armazenar as variáveis os objetos, e funções, contudo os dados complexos são apontados para o endereço de memória enquanto as variáveis primitivas tem o seu próprio valor;
- Call Stack: É uma pilha que controla a **ordem de execução das funções**, o JS executa uma **coisa por vez**, sempre no topo da fila.

Memory Heap x call Stack





Call Stack (Pilha de Execução)

A pilha de execução é:

- É do tipo LIFO (last in, first out);
- Guarda os contextos de execução;
- Controla qual função está a ser executada;

Exemplo simples de Call Stack

```
function primeiro() {
  console.log("Primeiro");
}

function segundo() {
  console.log("Segundo");
  primeiro();
}

function terceiro() {
  console.log("Terceiro");
  segundo();
}

terceiro();
```

```
$ node ex01.js
Terceiro
Segundo
Primeiro
```

A pilha vai ser iniciada com a **função terceiro**, que imprime o resultado no console, chama a **função segundo** e sai da fila de execução, a **função segundo** , entra na fila de execução, imprime o resultado no console sai da fila de execução e chama a **função primeiro**, que entra na fila de execução, imprime o resultado no console e sai da fila de execução, deixando a pilha agora limpa.

Hoisting

O que é hoisting?

É o comportamento do JavaScript (JS) onde:

- **Declarações** são processadas antes da execução;
- O código **não é movido**;
- O **Motor** apenas **regista** certas informações primeiro.

O Hoisting acontece na fase de criação do contexto.

O comportamento estranho

```
dizerOla();

function dizerOla() {
  console.log("Olá!");
}
```

A função assim como o var foi varrida pelo JS, já existe na memory heap e portanto, já está pronta para ser utilizada. Isso nos permite criar fluxos onde uma função encadeada consegue ser executada, mesmo quando ela ainda não foi declarada.

Let e const também sofrem Hoisting?

As declrações **let** e **const** sofrem sim os hoisting, **mas de uma forma diferente**. O **let** e o **const** são registados na fase de criação, mas o motor do JS **proíbe o acesso a eles** até que a execução chegue à linha onde foram declarados esse conceito é conhecido como **Temporal Dead Zone (TDZ)**.

Function declaration x Function Expression

Uma function declararion function somar(), já sabemos que terá o comportamento do var e será içada pela memory heap, contudo, o que acontece com a fuction expression quando ela é utilizada com o token const e let, torna essa função protegida e, portanto, uma função declarada dessa forma: const somar = (a,b) => a + b, é vista pelo memory heap como valor, não como função.

Closures

O que já temos até aqui

Já aprendemos que:

- Funções criam **novo escopo**;
- Variáveis têm **tempo de vida**;
- O JS usa **scope chain**.

As Closures nascem exatamente aqui.

Definição de closure

Uma Closure é o **elo de ligação** (o "fecho") que mantém o escopo pai vivo na memória, mesmo que o contexto de execução desse pai já tenha saído da Call Stack.

Lexical scope – Escopo léxico

O Escopo Léxico, é a capacidade da variável saber onde ela nasceu, mesmo após ter seu contexto destruído, isso acontece, quando a nossa função contador, retornou uma outra função. Aqui o JS entende que a variável count provavelmente ainda poderá ser usada, e assim, mantém tanto a variável, quanto o seu valor vivo no memory heap, pronto para ser utilizado, caso seja invocada.

```
function contador() {
  let count = 0;
  return function () {
    count++;
    console.log(count);
  };
}
const incrementar = contador();
incrementar();
incrementar();
incrementar();
```

Para que servem as closures?

As closures nos permite recuperar uma variável e o seu valor criado em um contexto que não existe mais, permitindo que outras funções possam continuar utilizando-se dessa variável. Isso nos permite a não utilizar mais variáveis globais para servirem de auxiliares de contadores, podendo criar variáveis com escopo locais protegidas contra alterações indesejadas.

A Palavra this

O grande mito do this

Ideia errada comum: “O **this** refere-se à função onde estou”.

Na verdade o **this** depende **de quem chama a função**, não de onde ela foi escrita. A regra de ouro aqui é: “O valor de this é definido no momento da chamada da função”, nunca no momento da definição.

This numa função normal

```
function mostrarThis() {
  console.log(this);
}

mostrarThis();
```

No resultado o JS "preenche o vazio" com o objeto de nível mais alto disponível: o Global Object (que no navegador é o window). Isto significa que qualquer propriedade que tentes aceder dentro dessa função via this será procurada no objeto global da janela do navegador.

This num método de objeto

```
const pessoa = {
  nome: "Ana",
  falar() {
    console.log(this.nome);
  }
};

pessoa.falar();
```

this → pessoa Quem chama é o objeto.

Como a função é chamada através da referência pessoa., o this é automaticamente vinculado a esse objeto específico.

O Resultado: this.nome torna-se, na prática, pessoa.nome

Síntese

- Compreensão de como o JS executa código através de contextos;
- Entendimento de como o JavaScript regista declarações antes da execução;
- Compreensão de como funções podem preservar e aceder ao seu escopo léxico mesmo após a execução do contexto externo;
- Entendimento de que o valor de this é determinado pelo contexto de chamada da função.

Conclusão

Innominatum CONSULTORIA

Programação em JavaScript

ServiceNow - Deloitte

Rodrigo Costa

IPCA POLITÉCNICO DO CÁVADO E DO AVE
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU
IAPMEI Parcerias para o Crescimento
PORTUGAL DIGITAL