Aula 7

Escopo & Closures – O Coração da Linguagem

Agenda da Aula

- Revisão Rápida da Parte 1
- O que é Escopo? Onde as variáveis moram.
- Escopo Léxico: Onde o código foi escrito importa.
- Hoisting Revisitado: A "Zona da Morte Temporal".
- Closures: A memória secreta das funções.
- O Padrão de Módulo: O poder do encapsulamento.

O que é Escopo?

- Analogia: Escopo como as "regras de acesso" de um prédio.
- Define onde o motor do JavaScript procura por variáveis.
- Buscas LHS vs. RHS:
- LHS (Alvo): minhaVariavel = "valor"; (Onde vou colocar o valor?)
- RHS (Fonte): console.log(minhaVariavel); (De onde vou obter o valor?)

Escopo Léxico

- **Definição**:
  - O escopo é definido pela localização do código no momento em que é escrito, não quando é executado.
- **Analogia**:
  - "Bolhas" de escopo aninhadas. A bolha interna pode ver para fora, mas a externa não pode ver para dentro.

```
function exterior() {
    const variavelExterna = "Sou de fora!";

    function interior() {
        const variavelInterna = "Sou de dentro!";
        console.log(variavelExterna); // Consegue acessar?
        console.log(variavelInterna); // E esta?
    }

    interior();
    // console.log(variavelInterna); // E aqui?
}

exterior();
```

Escopo de Função vs. Bloco

- Escopo de Função:
  - `var` fica "preso" à função inteira.
- Escopo de Bloco:
  - `let` e `const` ficam "presos" a qualquer par de chaves { }.
- Princípio do Mínimo Privilégio:
  - Declare variáveis no menor escopo possível para evitar "poluição" e bugs.

Hoisting Revisitado

- Recapitulação:
  - Declarações são processadas antes da execução.
- Aprofundamento:
  - A "Temporal Dead Zone" (TDZ) para `let` e `const`.

- `var a` é içado e inicializado com `undefined`.
- `let b` é içado mas não inicializado. Acessá-lo antes da declaração causa um `ReferenceError`.

```
// console.log(idade); // Isto causaria um ReferenceError. 'idade' está na TDZ.
// console.log(nome); // Isto também causaria um ReferenceError. 'nome' está na TDZ.

const nome = "Ana"; // Fim da TDZ para 'nome'. Agora pode ser acedida.
let idade = 25; // Fim da TDZ para 'idade'. Agora pode ser acedida.

console.log(nome); // Funciona: imprime "Ana"
console.log(idade); // Funciona: imprime 25
```

Closures: O Conceito

- **Definição**:
  - Uma closure ocorre quando uma função "se lembra" do seu escopo léxico, mesmo quando é executada fora daquele escopo.
- **Analogia**:
  - Uma função que sai de casa com uma "mochila" contendo todas as variáveis que ela podia acessar em casa.

```
function criarSaudacao(saudacao) {
    const mensagemBase = saudacao + ", ";

    // A função retornada "fecha" (closes over)
    // a variável mensagemBase
    return function(nome) {
        console.log(mensagemBase + nome);
    };
}

// Criamos uma nova função 'dizerOla'
const dizerOla = criarSaudacao("Olá");

// A função 'criarSaudacao' já terminou a sua execução!
// O seu escopo deveria ter desaparecido.

// ... mas não desapareceu.
dizerOla("Ana"); // Imprime: "Olá, Ana"
dizerOla("João"); // Imprime: "Olá, João"
```

Closures na Prática

- Exemplo 1: Contadores privados (encapsulamento de estado).

```
function criarContador() {
     let contador = 0; // Esta variável é privada

     // Retornamos um objeto com os métodos públicos
     return {
          incrementar: function() {
                    contador++;
          },
          decrementar: function() {
                    contador--;
          },
          obterValor: function() {
                    return contador;
          }
     };
}

const meuContador = criarContador();

meuContador.incrementar();
meuContador.incrementar();
console.log(meuContador.obterValor()); // Imprime: 2

// Não há como acessar 'contador' diretamente!
// console.log(meuContador.contador); // Imprime: undefined
```

Closures na Prática

- Exemplo 2: Funções de fábrica (funções que criam e retornam outras funções).

```
function criarMultiplicador(fator) {
    // A função retornada "lembra-se" do
    'fator'
    return function(numero) {
        return numero * fator;
    };
}

const duplicar = criarMultiplicador(2);
const triplicar = criarMultiplicador(3);

console.log(duplicar(5)); // Imprime: 10
console.log(triplicar(5)); // Imprime: 15
```


O Padrão de Módulo

- A evolução natural das closures para organizar o código.
- Usa uma função externa que é executada imediatamente (IIFE - Immediately Invoked Function Expression).

```
(function() {
// Este código executa-se apenas uma vez, imediatamente.
})();
```

- Essa função retorna um objeto com a "API pública" (os métodos que queremos expor).
- Variáveis e funções internas permanecem privadas, acessíveis apenas pela API pública via closure.

Exemplo

```
const MeuModuloContador = (function() {
    // --- PARTE PRIVADA ---
    // Esta variável e função são invisíveis para o exterior.
    let contadorPrivado = 0;

    function alterarContador(valor) {
        contadorPrivado += valor;
    }

    // --- PARTE PÚBLICA (API) ---
    // Retornamos um objeto. Apenas o que está neste objeto
    // será acessível de fora.
    return {
        incrementar: function() {
            // Este método tem uma closure sobre 'alterarContador'
            // e 'contadorPrivado'.
            alterarContador(1);
        },

        obterValor: function() {
                return contadorPrivado;
        }
    };
})();

// Usando o módulo
MeuModuloContador.incrementar();
MeuModuloContador.incrementar();
console.log(MeuModuloContador.obterValor()); // Imprime: 2

// Não conseguimos aceder às partes privadas
console.log(MeuModuloContador.contadorPrivado); // undefined
// MeuModuloContador.alterarContador(10); // TypeError: não é uma função
```

Resumo da Parte 2 e Conclusão

- Escopo léxico, closures e o padrão de módulo são os pilares do encapsulamento em JS.
- Entender estes conceitos permite escrever código robusto, limpo e com menos bugs.
- Esta é a base para entender como frameworks e bibliotecas modernas funcionam.