Padrões para Conteúdo Web I

ANA CAROLINA N R GRACIOSO

carol.nrg@gmail.com

Agenda

- POO (ES6)
- Escopo

Classe

```
class Carro {
    constructor(marca, modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }
    mostrarInfo() {
        console.log('Carro: ${this.marca} ${this.modelo}');
    }
}

const meuCarro = new Carro("Toyota", "Corolla");
meuCarro.mostrarInfo();
```

`constructor` é o método chamado ao criar uma nova instância.

`this` se refere à instância do objeto.

Encapsulamento

```
class ContaBancaria {
  constructor(saldo) {
    this._saldo = saldo;
  }

  get saldo() {
    return this._saldo;
  }

  set saldo(valor) {
    if (valor >= 0) {
      this._saldo = valor;
    }
  }
}

const minhaConta = new ContaBancaria(100); // trocar 100 por -10
console.log(minhaConta.saldo);
minhaConta.saldo = 200; // trocar 200 por -10
console.log(minhaConta.saldo);
```

O underline (_) é uma convenção para indicar um atributo privado.

`get` e `set` controlam como os valores são acessados e modificados.

Herança com Polimorfismo

```
class Animal {
  constructor(nome) {
    this.nome = nome;
  }

  fazerSom() {
    console.log(`${this.nome} está fazendo um som.`);
  }
}

class Cachorro extends Animal {
  fazerSom() {
    console.log(`${this.nome} está latindo.`);
  }
}

const meuAnimal = new Animal("Rex");
meuAnimal.fazerSom();

const meuCachorro = new Cachorro("Rex");
meuCachorro.fazerSom();
```

`extends` é usado para herdar de uma classe "pai".

A classe filha pode sobrescrever métodos da classe pai (Polimorfismo)

Classes Abstratas

```
class Pessoa {
  constructor(nome) {
    if (this.constructor === Pessoa) {
      throw new Error("Classe abstrata!");
    }
    this.nome = nome;
  }
}

class Aluno extends Pessoa {
  constructor(nome, matricula) {
    super(nome);
    this.matricula = matricula;
  }
}

try {
  const pessoa = new Pessoa("Carlos");
} catch (e) {
  console.error(e.message);
}

const aluno = new Aluno("Ana", "2023001");
console.log(aluno);
```

Embora o JavaScript não tenha suporte para classes abstratas, podemos simular o comportamento usando herança e lançando erros para códigos que precisam ser implementados nas subclasses.

Escopo

- O escopo refere-se à **visibilidade** e **acessibilidade** de variáveis dentro do código.

- Dois tipos principais: **Global** e **Local**.

Escopo Global

- Variáveis declaradas fora de qualquer função ou bloco de código têm escopo global e podem ser acessadas de qualquer parte do programa.

```
let nome = "Maria";

function saudar() {
  console.log("Olá, " + nome);
}

saudar();
```

A variável nome é global e pode ser acessada tanto dentro como fora da função.

Escopo Local

- Variáveis declaradas dentro de uma função só podem ser acessadas dentro dessa função.

```
function saudar() {
  let saudacao = "Olá, Mundo";
  console.log(saudacao);
}

saudar();
console.log(saudacao);
```

A variável `saudacao` é local à função e não pode ser acessada fora dela.

Escopo de Bloco

- Variáveis declaradas com **let** ou **const** dentro de blocos só existem dentro desses blocos.

```
if (true) {
  let mensagem = "Dentro do bloco";
  console.log(mensagem);
}

console.log(mensagem);
```

A variável `mensagem` só é acessível dentro do bloco `if`. Fora dele, ela não existe.

Escopo de Bloco

- Variáveis declaradas com **var** dentro de blocos existem fora desses blocos.

```
if (true) {
  var mensagem = "Dentro do bloco";
  console.log(mensagem);
}

console.log(mensagem);
```

A variável `mensagem` também é acessível fora do bloco `if`.

let, var e const

- var
  - escopo de função e permite reatribuição
- let e const
  - escopo de bloco
  - let: permite reatribuição
  - const: não permite reatribuição

Hoisting

- Variáveis declaradas com **var** são "elevadas" para o topo de seu escopo, mesmo antes da execução.

```
console.log(nome); // Saída: undefined
var nome = "Ana";
console.log(nome); // Saída: Ana
```

Quando o código é executado, a declaração `var nome;` é "elevada" ao topo do escopo.
A variável `nome` existe, mas seu valor é `undefined` até ser inicializada com `"Ana"`.

Hoisting

- As variáveis declaradas com let e const também são "elevadas" ao topo, mas elas não são inicializadas até o ponto da declaração. Isso significa que, se você tentar acessá-las antes da declaração, um erro será lançado.

```
console.log(nome); // Erro: Cannot access 'nome' before initialization
let nome = "Ana";
console.log(nome); // Saída: Ana
```

Mesmo que a variável tenha sido "elevada", ela fica em um estado chamado de **zona morta temporal** (temporal dead zone) até a linha da declaração.
Isso significa que o JavaScript sabe que a variável existe, mas não permite o uso dela até que seja inicializada.

Hoisting de Funções

- As declarações de funções são completamente elevadas ao topo do escopo, o que significa que você pode chamar a função antes mesmo de ela ser declarada.

```
saudar(); // Saída: Olá, mundo!

function saudar() {
    console.log("Olá, mundo!");
}
```

Hoisting expressões de Funções

- Expressões de função (ou funções atribuídas a variáveis) não funcionam da mesma maneira:

```
saudar(); // Erro: Cannot access 'saudar' before initialization

let saudar = function() {
    console.log("Olá, mundo!");
}
```

No caso de expressões de função, o comportamento segue o de variáveis declaradas com `let` ou `const`, onde a função não é inicializada até que a linha de código correspondente seja executada.

Hoisting de Classes

- Classes em JavaScript não sofrem o mesmo tipo de hoisting que variáveis e funções. Se você tentar acessar uma classe antes de sua declaração, o JavaScript lançará um erro, porque as classes funcionam de forma semelhante ao let e const.

```
const obj = new MinhaClasse(); // Erro: Cannot access 'MinhaClasse'
before initialization

class MinhaClasse {
  constructor() {
    this.nome = "Exemplo";
  }
}
```

Referência

- https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

Você deverá desenvolver uma aplicação simples de gerenciamento de tarefas, onde o usuário pode adicionar novas tarefas, visualizar os detalhes de cada tarefa, marcá-las como concluídas e removê-las da lista. A aplicação será desenvolvida utilizando **Programação Orientada a Objetos**.

Requisitos:

1. **Classe Tarefa**:
    - A classe Tarefa deverá possuir:
        - Atributos nome, descricao e status (pendente ou concluída).
        - Um método concluir() para marcar a tarefa como concluída.
        - Um método detalhes() para exibir o nome, descrição e status da tarefa.
2. **Classe GerenciadorDeTarefas**:
    - A classe GerenciadorDeTarefas será responsável por gerenciar as tarefas e deverá:
        - Conter um array privado de tarefas.
        - Ter um método adicionarTarefa(tarefa) para adicionar uma nova tarefa.
        - Um método listarTarefas() para exibir todas as tarefas.
        - Um método marcarComoConcluida(index) para marcar uma tarefa como concluída.
        - Um método removerTarefa(index) para remover uma tarefa da lista.
        - Um método visualizarDetalhes(index) que permita visualizar os detalhes de uma tarefa específica.
3. **Encapsulamento**:
    - O array de tarefas deverá ser privado e acessível somente por métodos da classe.

Instruções:

- O formulário de entrada terá campos para o nome e a descrição da tarefa.
- Ao lado de cada tarefa haverá botões para:
    - **Visualizar Detalhes**: Exibir uma janela modal ou alerta com os detalhes da tarefa.
    - **Concluir Tarefa**: Marcar a tarefa como concluída.
    - **Remover Tarefa**: Remover a tarefa da lista.


Gerenciador de Tarefas
Nome da Tarefa
Descrição da Tarefa
Adicionar Tarefa
Criar API de Produtos
Detalhes
Concluir
Remover
Corrigir Bug na Tela de Checkout
Detalhes
Concluir
Remover
Melhorar Performance da Página Inicial
Detalhes
Concluir
Remover
Implementar Testes Unitários
Detalhes
Concluir
Remover