USCS
UNIVERSIDADE DE SÃO CAETANO DO SUL

Linguagem de Programação

Prof. Renato Carioca Duarte

USCS
UNIVERSIDADE DE SÃO CAETANO DO SUL

Unidade 06

Programação Orientada a Objetos

Orientação a Objetos

- Em JavaScript, os objetos são uma estrutura de dados fundamental que nos permite armazenar e organizar informações relacionadas.
- Eles são usados para representar entidades do mundo real, conceitos abstratos ou qualquer coisa que possua propriedades e comportamentos associados.
- Um objeto em JavaScript é uma coleção de pares chave-valor, onde cada chave é uma string (também chamada de propriedade) que identifica um valor associado a ela.
- O valor pode ser qualquer tipo de dado, incluindo números, strings, booleanos, arrays, funções ou até mesmo outros objetos.

Criação de Objetos

- Os objetos podem ser criados de várias maneiras em JavaScript, mas uma das formas mais comuns é usando a notação de chaves `{}`.
- Veja um exemplo simples de como criar um objeto representando uma pessoa:

```
// Criando um objeto usando a notação de chaves {}
const pessoa = {
  nome: "João",
  idade: 30,
  profissao: "Engenheiro",
};
```

Criação de Objetos

- Nesse exemplo, criamos um objeto chamado `pessoa` com três propriedades: `nome`, `idade` e `profissao`, cada uma com seu respectivo valor.
- Para acessar os valores das propriedades de um objeto, podemos usar a notação de ponto `.` ou a notação de colchetes `[]`.
- Veja como podemos acessar os valores do objeto `pessoa`:

```
console.log(pessoa.nome);       // Output: 'João'
console.log(pessoa['idade']);   // Output: 30
console.log(pessoa.profissao); // Output: 'Engenheiro'
```

Criação de Objetos

- Além disso, é possível adicionar novas propriedades a um objeto ou modificar as existentes:

```
pessoa.cidade = 'São Paulo'; // Adicionando uma nova propriedade
pessoa.idade = 31;           // Modificando o valor de uma propriedade existente
```

- Também podemos excluir propriedades de um objeto usando o operador `delete`:

```
delete pessoa.profissao; // Removendo a propriedade 'profissao' do objeto
```

Outras Formas de Criação de Objetos

- Usando função Construtora de Objeto:

```
function Pessoa(nome, idade, profissao) {
    this.nome = nome;
    this.idade = idade;
    this.profissao = profissao;
}

const pessoa = new Pessoa('João', 30, 'Engenheiro');
```

Outras Formas de Criação de Objetos

- Usando método estático Object.create():

```
const pessoaProto = {
    saudacao() {
      console.log(`Olá, meu nome é ${this.nome} e tenho ${this.idade} anos.`);
    }
};

const pessoa = Object.create(pessoaProto);
pessoa.nome = 'João';
pessoa.idade = 30;
```

Outras Formas de Criação de Objetos

- Usando Função Factory:

```
function criarPessoa(nome, idade, profissao) {
    return {
        nome,
        idade,
        profissao
    };
}

const pessoa = criarPessoa('João', 30, 'Engenheiro');
```

Outras Formas de Criação de Objetos

- Usando Classes (a partir do ECMAScript 2015):

```
class Pessoa {
    constructor(nome, idade, profissao) {
        this.nome = nome;
        this.idade = idade;
        this.profissao = profissao;
    }
}

const pessoa = new Pessoa('João', 30, 'Engenheiro');
```

Propriedades (Atributos)

- As propriedades de um objeto são suas características ou atributos, que armazenam valores de qualquer tipo de dado, como números, strings, booleanos, arrays, outros objetos e até mesmo funções.
- Cada propriedade é identificada por uma chave única (também chamada de chave de propriedade), que é uma string.
- Para acessar o valor de uma propriedade em um objeto, podemos usar a notação de ponto `.` ou a notação de colchetes `[]`.

Propriedades (Atributos)

```
const pessoa = {
    nome: 'João',
    idade: 30,
    profissao: 'Engenheiro'
};

console.log(pessoa.nome);       // Output: 'João'
console.log(pessoa['idade']);   // Output: 30
console.log(pessoa.profissao); // Output: 'Engenheiro'
```

Métodos (Funções)

- Os métodos de um objeto são funções que estão associadas a esse objeto e permitem executar ações ou operações relacionadas a ele.
- Assim como as propriedades, os métodos também são definidos como pares chave-valor, mas a diferença é que o valor é uma função.
- Para chamar um método de um objeto, usamos a mesma notação de ponto `.` que usamos para acessar propriedades.

Métodos (Funções)


```
const pessoa = {
    nome: 'João',
    idade: 30,
    saudacao: function() {
      console.log(`Olá, meu nome é ${this.nome} e tenho ${this.idade} anos.`);
    }
};

pessoa.saudacao(); // Output: Olá, meu nome é João e tenho 30 anos.
```

Acessando e Utilizando Propriedades e Métodos

```
const pessoa = {
    nome: 'João',
    idade: 30,
    saudacao: function() {
      console.log(`Olá, meu nome é ${this.nome} e tenho ${this.idade} anos.`);
    }
};

console.log(pessoa.nome);       // Output: 'João'
console.log(pessoa['idade']);   // Output: 30
pessoa.saudacao();              // Output: Olá, meu nome é João e tenho 30 anos.
```

Acessando e Utilizando Propriedades e Métodos

- Os objetos em JavaScript são flexíveis e dinâmicos, o que significa que podemos adicionar, modificar ou remover propriedades e métodos em tempo de execução.
- Isso torna o JavaScript uma linguagem poderosa para trabalhar com objetos e manipular dados de maneira eficiente.

Classes

- Antes da introdução do ECMAScript 2015 (também conhecido como ES6), JavaScript não tinha uma sintaxe específica para a criação de classes como em outras linguagens de programação orientadas a objetos.
- No entanto, ainda era possível simular a criação de classes usando funções construtoras e o conceito de protótipos.
- A partir do ES6, foi introduzida a sintaxe de classes, que simplifica bastante a criação de classes em JavaScript.

- Vamos explorar ambas as abordagens para criar classes em JavaScript:

Classes (Pré-ES6)

Usando Funções Construtoras e Protótipos (Pré-ES6):

```
// Definindo uma função construtora para a classe Pessoa
function Pessoa(nome, idade) {
    this.nome = nome;
    this.idade = idade;
}

// Adicionando métodos à classe Pessoa através de seu protótipo
Pessoa.prototype.saudacao = function() {
    console.log(`Olá, meu nome é ${this.nome} e tenho ${this.idade} anos.`);
};

// Criando uma instância da classe Pessoa
const pessoa1 = new Pessoa('João', 30);
const pessoa2 = new Pessoa('Maria', 25);

// Chamando o método saudacao() em cada instância
pessoa1.saudacao(); // Output: Olá, meu nome é João e tenho 30 anos.
pessoa2.saudacao(); // Output: Olá, meu nome é Maria e tenho 25 anos.
```

Classes (Pré-ES6)

- Nesse exemplo, usamos uma função construtora `Pessoa` para criar um "modelo" para a classe Pessoa.
- Em seguida, adicionamos métodos à classe através do seu protótipo (`Pessoa.prototype`).
- Quando criamos uma instância da classe usando `new Pessoa()`, cada instância herda os métodos do protótipo, o que permite que todas as instâncias compartilhem os mesmos métodos.

Classes (A partir do ES6)

Usando Funções Construtoras e Protótipos (A partir do ES6):

```
class Pessoa {
    constructor(nome, idade) {
      this.nome = nome;
      this.idade = idade;
    }

    saudacao() {
      console.log(`Olá, meu nome é ${this.nome} e tenho ${this.idade} anos.`);
    }
}

// Criando uma instância da classe Pessoa
const pessoa1 = new Pessoa('João', 30);
const pessoa2 = new Pessoa('Maria', 25);

// Chamando o método saudacao() em cada instância
pessoa1.saudacao(); // Output: Olá, meu nome é João e tenho 30 anos.
pessoa2.saudacao(); // Output: Olá, meu nome é Maria e tenho 25 anos.
```

Classes (A partir do ES6)

- Com a sintaxe de classes introduzida no ES6, podemos definir uma classe usando a palavra-chave `class` e definir os métodos da classe dentro do bloco `class`.
- A palavra-chave `constructor` é usada para definir o construtor da classe, que é executado quando uma nova instância é criada.
- Os métodos definidos dentro da classe são automaticamente adicionados ao protótipo da classe, assim como fizemos manualmente no exemplo anterior.

Classes

- Ambas as abordagens resultam em classes JavaScript funcionais.
- No entanto, a sintaxe de classes do ES6 é mais simples, legível e se assemelha à sintaxe de classes em outras linguagens de programação orientadas a objetos, tornando-a mais amplamente utilizada em projetos modernos.

Construtores

- Em JavaScript, os construtores são funções especiais usadas para criar e inicializar objetos quando você está trabalhando com classes.
- Eles são chamados automaticamente quando você usa a palavra-chave `new` para criar uma nova instância da classe.
- O construtor é responsável por definir as propriedades iniciais do objeto e realizar qualquer inicialização necessária.
- Os construtores geralmente são usados para executar qualquer lógica de inicialização necessária para o objeto.

Construtores

- Isso pode incluir a configuração de propriedades iniciais, a chamada de outros métodos de inicialização ou a configuração de valores padrão.
- É importante lembrar que uma classe pode ter apenas um construtor.
- Se uma classe não tiver um construtor definido, o mecanismo JavaScript usará um construtor padrão implícito, que não faz nada além de criar um objeto vazio.
- Portanto, sempre que você precisar executar alguma lógica de inicialização ao criar uma instância de classe, defina um construtor personalizado para essa classe.

Herança

- Para implementar a herança em JavaScript e compartilhar propriedades e métodos entre classes, é essencial entender o conceito de protótipos.
- A herança em JavaScript é baseada em protótipos, e cada objeto em JavaScript possui um protótipo associado a ele.
- A ideia é que uma classe (ou função construtora) possa herdar propriedades e métodos de outra classe através da cadeia de protótipos.

Passos para Implementar a Herança

1. Definir a Classe Pai (Superclasse):

- Comece definindo a classe pai, que é a classe da qual você deseja herdar propriedades e métodos.
- Normalmente, você define a classe pai como uma função construtora ou usando a sintaxe de classes do ES6.

```
// Classe Pai (Superclasse) usando função construtora
function Animal(nome) {
    this.nome = nome;
}

// Adicionar um método à classe Animal
Animal.prototype.som = function() {
    console.log('Fazendo barulho.');
};
```

Passos para Implementar a Herança

2. Definir a Classe Filha (Subclasse):

- Em seguida, defina a classe filha, que é a classe que deseja herdar as propriedades e métodos da classe pai.
- Para herdar a classe pai, você precisa chamar o construtor da classe pai na classe filha usando a função `call()` ou `apply()` dentro do construtor da classe filha.
- Isso garante que as propriedades da classe pai sejam inicializadas corretamente quando você cria uma nova instância da classe filha.

Passos para Implementar a Herança

2. Definir a Classe Filha (Subclasse):

```
// Classe Filha (Subclasse) usando função construtora
function Cachorro(nome, raca) {
    Animal.call(this, nome); // Chama o construtor da classe pai
    this.raca = raca;
}

// Estabelece a herança definindo o protótipo da classe filha
// como uma instância da classe pai
Cachorro.prototype = Object.create(Animal.prototype);

// Define o construtor correto da classe filha (Cachorro)
Cachorro.prototype.constructor = Cachorro;

// Adiciona um método específico à classe Cachorro
Cachorro.prototype.latir = function() {
    console.log('Au au!');
};
```

Passos para Implementar a Herança

3. Criar Instâncias da Classe Filha:

- Agora você pode criar instâncias da classe filha e usar seus métodos, incluindo os herdados da classe pai.

```
const rex = new Cachorro('Rex', 'Labrador');
console.log(rex.nome); // Output: 'Rex'
rex.som();             // Output: 'Fazendo barulho.'
rex.latir();           // Output: 'Au au!'
```

Herança (A partir do ES6)

- É importante notar que, com a introdução da sintaxe de classes do ES6, o processo de herança foi simplificado.
- Agora você pode usar a palavra-chave `extends` para definir a herança entre classes, o que é mais legível e mais semelhante a outras linguagens de programação orientadas a objetos.
- Usar a sintaxe de classes simplifica a implementação da herança em JavaScript e é a abordagem recomendada para projetos modernos.
- Aqui está o exemplo anterior reescrito usando a sintaxe de classes:

Herança (A partir do ES6)

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

    som() {
      console.log('Fazendo barulho.');
    }
}

class Cachorro extends Animal {
    constructor(nome, raca) {
      super(nome); // Chama o construtor da classe pai
      this.raca = raca;
    }

    latir() {
      console.log('Au au!');
    }
}

const rex = new Cachorro('Rex', 'Labrador');
console.log(rex.nome); // Output: 'Rex'
rex.som();             // Output: 'Fazendo barulho.'
rex.latir();           // Output: 'Au au!'
```

Encapsulamento

- O encapsulamento é um dos princípios fundamentais da programação orientada a objetos (POO) que visa esconder os detalhes internos de uma classe e fornecer acesso controlado aos seus membros (propriedades e métodos).
- Ele ajuda a proteger a integridade dos dados de uma classe e permite que as operações sobre esses dados sejam realizadas de maneira segura e consistente.

Encapsulamento

- O encapsulamento é alcançado através do uso de modificadores de acesso, como "public", "private" e "protected" (presentes em algumas linguagens de programação, mas não nativamente em JavaScript).
- Esses modificadores determinam se os membros de uma classe podem ser acessados diretamente por outras partes do código ou se o acesso é restrito somente à própria classe.
- Em JavaScript, você pode criar atributos privados em uma classe usando o símbolo # antes do nome do atributo. Isso indica que o atributo é privado e só pode ser acessado dentro da classe.

Encapsulamento

```
class Pessoa {
    #nome;
    idade;

    constructor(nome, idade) {
      this.#nome = nome;
      this.idade = idade;
    }

    getNome() {
      return this.#nome;
    }

    setNome(nome) {
      this.#nome = nome;
    }
}

const pessoa = new Pessoa('João', 30);
console.log(pessoa.getNome()); // 'João'
console.log(pessoa.idade); // 30
pessoa.setNome('Maria');
pessoa.idade = 35;
console.log(pessoa.getNome()); // 'Maria'
console.log(pessoa.idade); // 35
```

Dúvidas

??????