Aula 11 - Introdução à Programação Orientada a Objetos

aTip Learn - Lógica de Programação com JavaScript

Descrição da Aula

Nesta aula, exploraremos programação orientada a objetos em JavaScript. Criaremos classes como modelos para representar objetos do mundo real, definindo propriedades para características e métodos para ações. Em seguida, abordaremos herança, permitindo criar novas classes a partir de existentes para reutilizar código. Aprenderemos a criar classes filhas que herdam propriedades e métodos, e como sobrescrever e estender comportamentos para personalizar objetos.

Objetivos

- Entender o conceito de programação orientada a objetos
- Criar classes e objetos em JavaScript
- Definir propriedades e métodos em classes
- Compreender o modificador de visibilidade privado.
- Utilizar herança para reutilizar código
- Sobrescrever e estender comportamentos de classes

Conteúdo

Introdução à Programação Orientada a Objetos

Até agora a maior parte do código que desenvolvemos em JavaScript baseia-se no uso de funções, em uma abordagem bastante próxima do que chamamos de programação procedural. Na programação procedural nós dividimos nosso código em procedimentos (funções) que recebem dados, processam e retornam resultados.

A programação procedural é bastante eficiente e simples, porém, em projetos de maior complexidade ela pode se tornar difícil de manter e evoluir. Isso porque nem sempre é possível modelar o problema que estamos tentando resolver em termos de procedimentos e funções.

Para reduzir esse problema, a programação orientada a objetos (POO) propõe uma nova abordagem, onde o foco principal é a criação de objetos que representam entidades do nosso problema. Cada objeto possui propriedades que representam suas características e métodos que representam suas ações. Objetos podem interagir uns com os outros por meio da chamada de métodos, permitindo a criação de sistemas complexos e bem organizados.

Na *POO* um objeto é uma instância de uma classe. Uma classe é um modelo que define as propriedades e métodos que um objeto pode ter. Por exemplo, podemos ter uma classe `Carro` que define propriedades como `modelo`, `ano`, `cor` e métodos como `ligar`, `desligar`, `acelerar` e `frear`. A partir dessa classe podemos criar vários objetos `Carro` com diferentes propriedades e métodos.

```
class Carro {
  constructor(modelo, ano, cor) {
    this.modelo = modelo;
    this.ano = ano;
    this.cor = cor;
    this.ligado = false;
  }

  ligar() {
    this.ligado = true;
  }

  desligar() {
```

```
    this.ligado = false;
  }

  acelerar() {
    if (this.ligado) {
      console.log('Acelerando...');
    } else {
      console.log('Ligue o carro antes de acelerar');
    }
  }

  frear() {
    console.log('Freando...');
  }
}

const carro1 = new Carro('Fusca', 1970, 'azul');
const carro2 = new Carro('Gol', 2000, 'prata');

carro1.ligar();
carro1.acelerar();
carro1.frear();
carro1.desligar();

carro2.acelerar();
```

Construtor

Uma classe é uma espécie de molde que define as propriedades e métodos que um objeto pode ter. Para criar uma classe em JavaScript utilizamos a palavra-chave `class` seguida do nome da classe e do corpo da classe, onde definiremos as propriedades e métodos.

```
class Carro { ... }
```

Toda classe possui um método especial chamado de construtor (`constructor`), que é responsável por inicializar as propriedades do objeto quando ele é criado. Para criar um objeto a partir de uma classe utilizamos a palavra-chave `new` seguida do nome da classe e dos argumentos do construtor.

```
class Carro {
  constructor(modelo, ano, cor) {
    ...
  }

  ...
}

const carro1 = new Carro('Fusca', 1970, 'azul');
```

Propriedades

O construtor da classe é geralmente utilizado para definir suas propriedades. As propriedades são variáveis que armazenam informações sobre o objeto. Dentro do construtor, ou dos métodos da classe, podemos acessar e modificar as propriedades do objeto utilizando a palavra-chave `this` seguida do caractere `.` e do nome da propriedade.

```
class Carro {
  constructor(modelo, ano, cor) {
```

```
    this.modelo = modelo;
    this.ano = ano;
    this.cor = cor;
    this.ligado = false;
  }

  ...
}
```

Métodos

Além das propriedades, uma classe também pode definir métodos, que são funções que representam ações que o objeto pode realizar. Para definir um método em uma classe adicionamos o nome do método seguido da lista de parâmetros e do corpo do método.

```
class Carro {
  ...

  ligar() {
    this.ligado = true;
  }

  desligar() {
    this.ligado = false;
  }

  acelerar() { ...}
  frear() { ... }
}
```

Propriedades e Métodos Privados

Em JavaScript, todas as propriedades e métodos de uma classe são públicos por padrão, o que significa que eles podem ser acessados e modificados de fora da classe. Para tornar uma propriedade ou método privado, ou seja, acessível apenas dentro da classe, podemos utilizar o prefixo `#` antes do nome da propriedade ou método.

```
class Carro {
  #ligado = false;

  ligar() {
    this.#ligado = true;
  }

  desligar() {
    this.#ligado = false;
  }
}

const carro = new Carro();
console.log(carro.#ligado); // Erro: propriedade privada
carro.#ligado = true; // Erro: propriedade privada
```

Nós utilizamos o prefixo `#` para tornar a propriedade `ligado` privada, impedindo que ela seja acessada ou modificada de fora da classe. Isso é útil para proteger propriedades sensíveis ou métodos internos que não devem ser acessados diretamente. Imagine por exemplo que estamos criando uma classe que representa uma

conta bancária, o saldo da conta bancária é uma informação sensível que não deve ser acessada diretamente. Para isso podemos ter os métodos depositar e sacar que alteram o saldo da conta, mas o saldo em si é privado.

```
class ContaBancaria {
  #saldo = 0;

  depositar(valor) {
    this.#saldo += valor;
  }

  sacar(valor) {
    this.#saldo -= valor;
  }
}
```

Nesse exemplo temos um problema, afinal o saldo da conta pode se tornar negativo, o que não faz sentido caso essa conta não possua um limite de crédito. Para resolver esse problema podemos utilizar um método privado para verificar se o saldo é suficiente antes de realizar o saque.

```
class ContaBancaria {
  #saldo = 0;

  depositar(valor) {
    this.#saldo += valor;
  }

  sacar(valor) {
    if (this.#saldo >= valor) {
      this.#saldo -= valor;
    } else {
      console.log('Saldo insuficiente');
    }
  }

  #saldoSuficiente(valor) {
    return this.#saldo >= valor;
  }
}
```

Dessa forma limitamos o acesso as funcionalidades da classe ContaBancaria, garantindo que o saldo da conta seja sempre positivo. O uso de métodos e propriedades privadas é uma prática recomendada na *POO* pois protege o estado e as funcionalidades internas da classe de acessos indevidos.

Herança

A herança é um conceito fundamental na programação orientada a objetos que permite criar novas classes a partir de classes existentes. A classe que é estendida é chamada de classe pai ou superclasse, e a classe que estende é chamada de classe filha ou subclasse. A classe filha herda todas as propriedades e métodos da classe pai, permitindo reutilizar código e estender o comportamento da classe pai.

Imagine por exemplo que temos uma classe que representa um veículo, com propriedades como `modelo`, `ano`, `cor` e métodos como `ligar`, `desligar`, `acelerar` e `frear`. A partir dessa classe podemos criar classes filhas como `Carro` e `Moto` que herdam as propriedades e métodos da classe veículo.

```
class Veiculo {
  constructor(modelo, ano, cor) {
```

```
    this.modelo = modelo;
    this.ano = ano;
    this.cor = cor;
    this.ligado = false;
  }

  ligar() {
    this.ligado = true;
  }

  desligar() {
    this.ligado = false;
  }

  acelerar() {
    if (this.ligado) {
      console.log('Acelerando...');
    } else {
      console.log('Ligue o veículo antes de acelerar');
    }
  }

  frear() {
    console.log('Freando...');
  }
}

class Carro extends Veiculo {
  constructor(modelo, ano, cor, portas) {
    super(modelo, ano, cor);
    this.portas = portas;
  }

  abrirPortas() {
    console.log('Abrindo portas...');
  }

  fecharPortas() {
    console.log('Fechando portas...');
  }
}

class Moto extends Veiculo {
  constructor(modelo, ano, cor, cilindradas) {
    super(modelo, ano, cor);
    this.cilindradas = cilindradas;
  }

  empinar() {
    console.log('Empinando...');
  }
}

const carro = new Carro('Fusca', 1970, 'azul', 2);
```

```
const moto = new Moto('CG 125', 2000, 'preta', 125);

carro.ligar();
carro.acelerar();
carro.frear();
carro.desligar();
carro.abrirPortas();
carro.fecharPortas();

moto.ligar();
moto.acelerar();
moto.empinar();
moto.frear();
moto.desligar();
```

Você deve ter notado uma diferença nos construtores das classes `Carro` e `Moto`. Além de receber os argumentos da classe pai, eles também recebem argumentos específicos da classe filha. Entretanto, para inicializar as propriedades da classe pai, precisamos chamar o construtor da classe pai utilizando a palavra-chave `super` seguida dos argumentos do construtor da classe pai. **É importante chamar o construtor da classe pai antes de inicializar as propriedades da classe filha.**

```
class Carro extends Veiculo {
  constructor(modelo, ano, cor, portas) {
    super(modelo, ano, cor);
    this.portas = portas;
  }
}
```

Sobrescrita e Extensão

Quando uma classe filha herda um método da classe pai, ela pode sobrescrever o método para personalizar o comportamento. A sobrescrita é útil quando queremos alterar o comportamento de um método herdado para atender as necessidades da classe filha.

```
class Veiculo {
  acelerar() {
    console.log('Acelerando...');
  }
}

class Carro extends Veiculo {
  acelerar() {
    console.log('Acelerando o carro...');
  }
}

const carro = new Carro();
carro.acelerar(); // Acelerando o carro...
const moto = new Moto();
moto.acelerar(); // Acelerando...
```

Propriedades e Métodos Estáticos

Até agora, todas as propriedades e métodos que definimos em nossas classes são propriedades e métodos de instância, ou seja, são propriedades e métodos individuais de cada objeto. No entanto, em algumas situações

pose ser útil ter propriedades e métodos que pertencem à classe em si, e não a cada objeto individual. Para esses casos utilizamos as chamadas propriedades e métodos estáticos.

Essas propriedades e métodos não estão vinculados a um objeto específico, mas sim a classe em si. Para definir uma propriedade estática, utilizamos a palavra-chave `static` seguida do nome da propriedade. Para acessar uma propriedade estática, utilizamos o nome da classe seguido do caractere `.` e do nome da propriedade.

```
class Carro {
  static contadorCarros = 0;

  constructor(modelo, ano, cor) {
    this.modelo = modelo;
    this.ano = ano;
    this.cor = cor;
    Carro.contadorCarros++;
  }
}

const carro1 = new Carro('Fusca', 1970, 'azul');
const carro2 = new Carro('Gol', 2000, 'prata');
console.log(Carro.contadorCarros); // 2
```

Os métodos estáticos são definidos da mesma forma que as propriedades estáticas, utilizando a palavra-chave `static` seguida do nome do método. Para acessar um método estático, utilizamos o nome da classe seguido do caractere `.` e do nome do método.

```
class Calculadora {
  static somar(a, b) {
    return a + b;
  }
}

console.log(Calculadora.somar(2, 3)); // 5
```