# Javascript

Programa de atração de formação de talentos
Bradesco / Visionaire

**Aula 09 – Protótipos e classes**

Mark Joselli

mark.Joselli@pucpr.br

# *Javascript é orientado a objetos?*

Não no sentido clássico:

- Tipagem **fraca**
- **Não funciona** bem com o spread operator (...)
- Herança baseada em **protótipos**
- Classes são só um **syntax sugar** para os protótipos

- A linguagem não te estimula a **pensar em objetos**.

# *Métodos*

Um método nada mais é do que uma função no interior de um objeto:

```
const obj = {
    nome: ”Mark",
    falar: function(texto) {
        console.log(`Falou: ${texto}`);
    }
};
obj.falar("Olá!"); //Falou: Olá!
```

# *Palavra chave this*

Você pode usar a palavra chave `this`. Ela significa o objeto sobre o qual a função foi chamada:

```
const obj = {
    nome: ”Mark",
    falar: function(texto) {
        console.log(`${this.nome} disse: ${texto}`);
    }
};
obj.falar("Olá!"); //Mark disse: Olá
```

# *Palavra chave this*

Cuidado, **não é** o objeto onde a função está:

```
const obj = {
    nome: ”Mark",
    falar: function(texto) {
        console.log(`${this.nome} disse: ${texto}`);
    }
};

const obj2 = {
    nome: "Marcia",
    falar: obj.falar
};
obj2.falar("Olá!"); //Marcia disse: Olá!
```

# *Palavra chave this*

Você pode até definir quem vai ser o `this` com o método call.

```
const obj = {
    nome: ”Mark",
    falar: function(texto) {
        console.log(`${this.nome} disse: ${texto}`);
    }
};
const obj3 = { nome: "Roberta" };
obj.falar.call(obj3, "Olá!"); //Roberta disse: Olá!
```

# *Palavra chave this*

Isso pode ser especialmente confuso com funções chamadas internamente (ex: callbacks):

```
const obj = {
    nome: ”Mark",
    falar: function() {
        function talk() {
            return `${this.nome} disse: Olá!`;
        }
        console.log(talk());
    }
};
obj.falar(); //undefined disse: Olá!
```

# *Palavra chave this*

A solução é gravar o `this` em uma variável

```
const obj = {
    nome: ”Mark",
    falar: function() {
        const self = this;
        function talk() {
            return `${self.nome} disse: Olá!`;
        }
        console.log(talk());
    }
};
obj.falar(); //Mark disse: Olá!
```

# Prototypes

Um objeto pode ser indicado como protótipo de outro.

- Quando uma propriedade ou função for chamada e não for encontrada, o protótipo será consultado.
- Um protótipo também pode ter **outro objeto** como protótipo, formando uma **cadeia**.
- Alguns protótipos são definidos por padrão: objetos vazios possuem o ***Object***.prototype. Funções vazias o Function.prototype e arrays o ***Array***.prototype.

- Utiliza-se o comando ***Object***.create para definir o quem será o protótipo

# Prototypes

```
const Pessoa = {
    nome: "Fulano",
    falar: function(texto) {
        console.log(`${this.nome} diz: ${texto}`);
    }
};

const mark = Object.create(Pessoa);
mark.falar("Olá!"); //Fulano diz Olá!


mark.nome = ”Mark";
mark.falar("Olá!");  //Vinícius diz Olá!


Pessoa.falar("Olá!");    //Fulano diz Olá!
```

# Prototypes

A função `Object.getPrototypeOf(obj)` retorna o protótipo de um objeto.

Por exemplo, seguindo no programa anterior:

```
console.log(Object.getPrototypeOf(mark) === Pessoa); //true
```

# Construtores

As vezes, queremos garantir que um objeto é corretamente construído. Poderíamos criar uma função como:

```
function criarPessoa(nome) {
    const pessoa = Object.create(Pessoa);
    pessoa.nome = nome;
    return pessoa;
}
const mark = criarPessoa(”Mark");
```

# Construtores

- Todas as funções possuem uma propriedade chamada `prototype`, com um **objeto vazio**, que tem como protótipo `Object.prototype`
- Quando a palavra chave `new` é utilizada, ela cria um novo objeto, associando esse prototype automaticamente ao novo objeto criado.

**CUIDADO**

Não confunda! Esse objeto **não é** o protótipo da função em si. O da função ainda será `Function.prototype`

# Construtores

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

Pessoa.prototype.falar = function(texto) {
    console.log(`${this.nome} diz: ${texto}`);
};

const mark = new Pessoa(”Mark");
mark.falar("Olá!"); //Mark diz: Olá!
console.log(Object.getPrototypeOf(mark) === Pessoa.prototype); //true
```

```
console.log(Object.getPrototypeOf(Pessoa) === Function.prototype); //true
console.log(Object.getPrototypeOf(Pessoa) === Pessoa.prototype); //false
```


# Notação de classes

- Recentemente, o Java Script criou uma notação que lembra a de linguagens orientadas a objeto
- Ela emula com funções vários dos conceitos, mas de maneira adaptada aos protótipos
- É importante entender que, por baixo dos panos, o resultado será equivalente ao que fizemos até aqui
- Você notará que inclusive o protótipo da classe ainda será `Function.prototype`

# Notação de classes

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

    falar(texto) {
        console.log(`${this.nome} diz: ${texto}`);
    }
}

const mark = new Pessoa(”Mark");
mark.falar("Olá!"); //Mark diz: Olá!

console.log(Object.getPrototypeOf(mark) === Pessoa.prototype); //true
console.log(Object.getPrototypeOf(Pessoa) === Function.prototype); //true
console.log(Object.getPrototypeOf(Pessoa) === Pessoa.prototype); //false
```

# Notação de classes

Também é possível usar com a sintaxe de declaração de variável

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

    falar(texto) {
        console.log(`${this.nome} diz: ${texto}`);
    }
}

const mark = new Pessoa(”Mark");
mark.falar("Olá!"); //Mark diz: Olá!

console.log(Object.getPrototypeOf(mark) === Pessoa.prototype); //true
console.log(Object.getPrototypeOf(Pessoa) === Function.prototype); //true
console.log(Object.getPrototypeOf(Pessoa) === Pessoa.prototype); //false
```

# Atributos

- Podemos declarar os atributos da nossa classe. O programa funcionará sem isso porém:
  - O javascript garante que o campo sempre existirá no objeto
  - IDEs poderão autocompletar melhor objetos da sua classe
  - É possível declarar campos privados, visíveis só no interior da classe. Basta iniciar seu nome com #

```
class Pessoa {
    nome; //publico
    constructor(nome) {
        this.nome = nome;
    }
}
```

```
class Pessoa {
    #nome; //privado
    constructor(nome) {
        this.#nome = nome;
    }
}
```

# Propriedades

- Propriedades permitem alterar ou ler o valor de atributos, ou criar valores calculados.
- São criadas através da palavra-chave get e set.
- São usadas como se fossem um atributo publico

# Propriedades

```
class Pessoa {
    #nome;
    constructor(nome) {
        this.nome = nome;
    }
    get nome() {
        return this.#nome;
    }
    set nome(valor) {
        this.#nome = valor || "?";
    }
    falar(texto) {
        console.log(`${this.nome} diz: ${texto}`);
    }
}
```

```
const mark = new Pessoa(”Mark");
mark.nome = undefined;
console.log(mark.nome); //Imprime ?
```

# Métodos estáticos

- É possível criar método e propriedades estáticas
- Eles devem ser acessados com base no nome da própria classe
- São úteis para métodos utilitários

```
class Ponto {
    static distancia(x1, y1, x2, y2) {
        return Math.sqrt((x1-x2)**2 + (y1-y2)**2);
    }
}
```

# Subclasses

- É possível criar subclasses
- No fundo, isso simplesmente associará a classe pai como protótipo da filha
- Note que no Javascript **não existem** classes abstratas ou interfaces. Dada a natureza dinâmica das variáveis, isso não é necessário.
  Usamos a palavra-chave `super` para chamar o construtor da superclasse. Ou `super.método` para chamar um método da superclasse.

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

    fazBarulho() {
        console.log(`${this.name} faz um barulho.`);
    }
}

class Cao extends Animal {
    constructor(name) {
        super(name);
    }

    fazBarulho() {
        console.log(`${this.name} late.`);
    }
}

let c = new Cao('Pretinha');
c.fazBarulho(); // Pretinha late
```

# Testando os tipos de dados

- O javascript possui 2 funções para testar o tipo de um dado:
  - `typeof`: Retorna uma string com o tipo do dado do parâmetro. Em objetos, o tipo retornado sempre será object. Em funções, o tipo será function. Pode ser usado também para tipos primitivos.
  - `instanceof`: Testa se a função construtora está em um dos protótipos na cadeia do objeto.
- Para o exemplo anterior:

```
console.log(c instanceof Cao); //true
console.log(c instanceof Animal); //true
console.log(typeof(c)); //object
```

# Métodos comuns a todos os objetos

- toString() e toLocaleString(): Usado automaticamente sempre que um objeto for representado na forma de texto
- valueOf: Retorna um valor que será usado se o objeto for convertido para um primitivo. Por padrão retorna o próprio objeto.
- hasOwnProperty: Testa se uma determinada propriedade está no objeto (não inclui herdadas)

# Objetos congelados

- A método estático Object.freeze impede que um objeto tenha novas propriedades adicionadas, removidas ou alteradas.
- Em resumo, torna o objeto imutável.
- O método estático Object.isFrozen pode testar se um objeto está congelado.
- É ideal para criação de enums:

```
const status = Object.freeze({
    ABERTO: 0, TRAMITANDO: 1, FECHADO: 2, CANCELADO: 3
});
```

# Atividade

1. Para praticar a sintaxe, experimente criar um objeto chamado Circulo com raio 3.
2. Em seguida, programe os métodos para calcular a área (PI*raio*raio) e o perímetro do círculo (2*PI*raio)

# Atividade

3. Altere seu objeto círculo para utilizar a sintaxe com uma função construtora chamada Circulo
4. Crie 2 círculos de raios diferentes. E imprima sua área e perímetro

# Atividade

5. Refatore o seu código para o círculo utilizar a sintaxe de classes
6. Crie o atributo do raio. Torne-o privado.
7. Adicione em sua classe um setter em que, caso o raio seja negativo, retire o sinal (um raio setado como -3 se tornaria 3)
8. Adicione getters para o método raio e também para a área e o perímetro.

# Atividade

9. Sobreponha o método `toString()` da sua classe Circulo para imprimi-la como Circulo de raio `${raio}`
10. Crie um método estático `criar` que aceita uma lista como parâmetro e gera um círculo para cada número positivo em seu interior, ignore os parâmetros que não puderem ser convertidos para números.

# Dúvidas?

- Materiais:
  - Livro Eloquent Javascript:
    - 3ª edição (inglês): https://eloquentjavascript.net/
    - 2ª edição (português): https://github.com/braziljs/eloquente-javascript
  - The Modern Javascript Tutorial: https://javascript.info/