Orientação a Objetos

Programação para Internet I

IFB - Tecnologia em Sistemas para Internet

Marx Gomes van der Linden

Criando objetos

- Como criar um objeto?

1) Criando diretamente: **`let obj = { … }`**
2) Usando **`Object.create(protótipo)`**
3) Encapsulando a criação em uma função
4) Criando um construtor

Função que cria objetos (Fábrica)

```
function criaCirculo(c, r) {
  let tmp = {
    cor: c,
    raio: r,

    mudaCor: function(c) {
        console.log("Mudando de cor...");
        this.cor = c;
    },

    toString: function() {
      return `Círculo ${this.cor} `
            + `de raio ${this.raio}`;
    }
  };
  return tmp;
}
```

Construtor

- Uma função comum que automatiza a criação de um objeto
- Convenção: inicial maiúscula
- Características:
  - » Usa a palavra-chave `this` para se referer ao objeto que está sendo criado
  - » Não precisa de `return`
  - » É chamada sempre com o operador `new`

Construtor

```
function Circulo(c, r) {
    this.cor = c;
    this.raio = r;

    this.mudaCor = function(c) {
        console.log("Mudando de cor...");
        this.cor = c;
    };

    this.toString = function() {
        return `Círculo ${this.cor} `
             + `de raio ${this.raio}`;
    }
}
```

Protótipos

- O reaproveitamento de propriedades e métodos em JS é baseado em **protótipos**

```
let prot = {
    nome: 'Novo usuário',
    pais: 'br',
    tipo: 'comum'
};

let user1 = Object.create(prot);
user1.nome = 'Cebolinha';

let user2 = Object.create(prot);
user2.nome = 'Donald Duck';
user2.pais = 'us';
```

Protótipo com construtor

- Toda função tem uma propriedade chamada **`prototype`**
- Compartilhada por todos os objetos criados pelo mesmo construtor
  - » Usada para adicionar métodos (não propriedades comuns)

Protótipo com construtor

```
function Circulo(c,r) {
    this.cor = c;
    this.raio = r;
}

Circulo.prototype.mudaCor = function(c) {
    console.log("Mudando de cor...");
    this.cor = c;
};

Circulo.prototype.toString = function() {
    return `Círculo ${this.cor} `
         + `de raio ${this.raio}`;
};
```

Protótipo com construtor

```
let circ1 = new Circulo('Azul', 200);
let circ2 = new Circulo('Vermelho', 1000);

circ2.mudaCor('Verde');

console.log(circ1.toString());
console.log(circ2.toString());
```

Sintaxe de classe (ES2015)

- A partir do EcmaScript 2015, é possível usar a sintaxe de classe semelhante a outras linguagens orientadas a objetos
- Apenas “açúcar sintático”
  - » Não muda a estrutura da linguagem
- Javascript continua sendo baseada em protótipos

Sintaxe de classe (ES2015)

```
class Circulo {
    constructor(c,r) {
        this.cor = c;
        this.raio = r;
    }

    mudaCor(c) {
        console.log("Mudando de cor...");
        this.cor = c;
    }

    toString() {
        return `Círculo ${this.cor} `
             + `de raio ${this.raio}`;
    }
}
```

Métodos estáticos

```
class Circulo {
    constructor(c,r) {
        this.cor = c;
        this.raio = r;
    }

    toString() {
        return `Círculo ${this.cor} `
             + `de raio ${this.raio}`;
    }

    static obterFormulaArea() {
        return "πr²";
    }
}

console.log(Circulo.obterFormulaArea());
```

Herança

- A sintaxe de classe permite a implementação mais fácil do conceito de herança.
- Palavra-chave **`extends`**

Classe Ave

Ave
+ voar() + vocalizar()

```
class Ave {
    voar(){
        console.log('Ave voando!');
    }

    vocalizar(){
         console.log('Ave vocalizando!');

    }
}
```

```
let ave = new Ave();
ave.voar();
ave.vocalizar();
```

Herança

Ave

+ voar()
+ vocalizar()

Superclasse
ou
classe “pai”

Arara

+ vocalizar()

Pato

+ vocalizar()
+ nadar()

Subclasses
ou classes “filhas”

Classe Pato

Pato
+ vocalizar() + nadar()

```
class Pato extends Ave {
    vocalizar(){
        console.log('Quack!');
    }

    nadar(){
        console.log('Pato nadando!');
    }
}
```

```
let pato = new Pato();
pato.nadar();
pato.vocalizar();
pato.voar();
```

Classe Arara

Arara
+ vocalizar()

```
class Arara extends Ave {
    vocalizar(){
        console.log('Arara!');
    }
}
```

```
let arara = new Arara();
arara.vocalizar();
arara.voar();
```

Construtor da superclasse

- Se a subclasse não definir construtor, o construtor da superclasse será chamado

```
class Usuario {
    constructor(){
        console.log("Construindo classe Usuario");
    }
}

class UsuarioPremium extends Usuario{
    toString() {
        return "Eu sou premium!";
    }
}

let prem = new UsuarioPremium();
```

`super()`

- Para se chamar o construtor da superclasse, deve-se usar **`super()`**
- Obrigatório quando a subclasse define um construtor

```
class Usuario {
    constructor(){
        console.log("Construindo classe Usuario");
    }
}

class UsuarioPremium extends Usuario{
    constructor(){
        super();
        console.log("Construindo classe UsuarioPremium");
    }

    toString() {
        return "Eu sou premium!";
    }
}

let prem = new UsuarioPremium();
```