FIESC
CIESC
SESI
SENAI
IEL

SENAI

Iniciativa da FIESC - Federação das Indústrias do Estado de Santa Catarina

Serviço Nacional de Aprendizagem Industrial de Santa Catarina

Programação de Aplicativos

Prototypes e Herança

com JavaScript

Bruno Bandeira Fernandes
bruno.fernandes@edu.sc.senai.br

FIESC CIESC SESI SENAI IEL SENAI

Prototypes

O que são Prototypes?

- JavaScript é uma linguagem considerada baseada em **prototypes**;
- Todos os objetos do JS herdam propriedades e métodos do seu Prototype;
- Como vimos nos casos dos built in objects;
- **A ideia central é que:** todo objeto tenha um pai (ou seja, um Prototype);

A propriedade prototype

- As funções além de suas propriedades que já vimos, também vem com a propriedade **prototype** criada;
- Recebemos um objeto vazio, que pode ter propriedades e métodos adicionados;

```
function test() {return true};

console.log(test.prototype);
console.log(typeof test.prototype);
```

Adicionando props e métodos com prototype

- Vejamos agora como podemos adicionar propriedades e métodos;
- Perceba que não há diferença em acessá-las;

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

Pessoa.prototype.profissao = 'Estudante';
Pessoa.prototype.falar = function() {
  console.log("Olá mundo!");
}

let joao = new Pessoa("João", 15);

joao.falar();
console.log(joao.profissao);
```

Adicionando múltiplas props e métodos

- Não precisamos adicionar uma a uma as propriedades ou métodos;

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

Pessoa.prototype = {
  profissao: 'Estudante',
  falar() {
    console.log("Olá mundo!");
  }
}

let joao = new Pessoa("João", 15);

joao.falar();
console.log(joao.profissao);
```

Modificação do prototype

- Ao alterar o prototype, todas as instâncias ganham seus novos métodos ou propriedades;

```
let joao = new Pessoa("João", 15);

Pessoa.prototype.gritar = function() {
  console.log("AHHHHHHHH");
}

joao.gritar();
```

Props do obj x props do Prototype

- A ordem de acesso é: primeiro o objeto e depois o prototype;
- As propriedades podem coexistir;

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

Pessoa.prototype.idade = 10;
Pessoa.prototype.cabelo = 'castanho';

let pedro = new Pessoa("Pedro", 15);

console.log(pedro.idade);
console.log(pedro.cabelo);

pedro.cabelo = 'louro';

console.log(pedro.cabelo);
```

Maneira de utilizar o prototype se já tem prop

- Podemos deletar uma propriedade, e voltar a utilizar o prototype;
- Pois mesmo sendo sobrescrito, ainda estará disponível;

```
function Pessoa(name) {
    this.name = name
}

Pessoa.prototype.name = 'estava sobrescrito';

let pessoa = new Pessoa('teste');

console.log(pessoa.name)

delete pessoa.name;

console.log(pessoa.name);
```

Loop para objetos

- O loop mais indicado para percorrer objetos é o for … in;
- Isso por que o for normal serve mais para arrays;

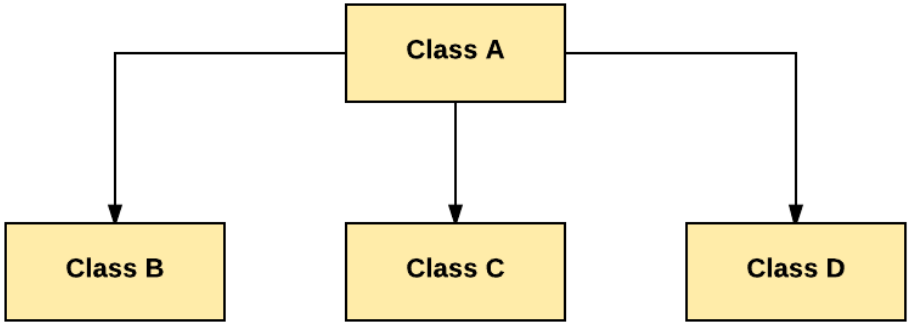
```
function Pessoa(name, lastname, age) {
    this.name = name;
    this.lastname = lastname;
    this.age = age;
}

pessoa = new Pessoa('Matheus', 'Battisti', 29);

for(prop in pessoa) {
    console.log(prop + " -> " + pessoa[prop]);
}
```

Herança

Como os Incas e Maias programavam...

A herança e o JavaScript

- Reutilzação de código;
- As **propriedades e métodos** são passadas para outros **objetos filhos**;
- A herança do JS pode ser aplicada via prototype chain;
  - Porém há outras maneiras de atingir este objetivo



Prototype chain

- É maneira default da linguagem de fazer herança;
- Podemos instanciar objetos no prototype de outros, criando a herança;

```
function Pessoa() {
    this.classe = 'Mamífero';
    this.falar = function() {
        console.log("Olá");
    }
}

function Advogado() {
    this.profissao = 'Advogado';
}

Advogado.prototype = new Pessoa();

let joao = new Advogado();

joao.falar();
```

Checando a herança

- Quando utilizamos a prototype chain, o objeto passa a virar instância de todos os 'pais';
- Podemos verificar isso pela instrução instanceof;

```
console.log(joao instanceof Advogado);
console.log(joao instanceof Pessoa);
console.log(joao instanceof Object);
```

Métodos e props no Prototype

- A ideia de utilizar o prototype é para que cada prop ou método adicionado **nele não se repita a cada objeto instanciado**;
- Então esta herança beneficia o código, **criando uma referência para os novos objetos**, deixando o programa mais performático;
- Não ocupando um novo espaço na memória a cada obj criado;

Exemplo

```
function Pessoa() {}

Pessoa.prototype.classe = 'Mamífero';
Pessoa.prototype.falar = function() {
    console.log("Olá");
}

function Advogado() {}

Advogado.prototype.profissao = 'Advogado';

Advogado.prototype = new Pessoa();

let joao = new Advogado();

joao.falar();
```

Aumentando ainda mais a eficiência

- Vimos que utilizar o prototype é uma boa prática;
- Então por que não clonar só o prototype em vez da instância do objeto?

```
function Pessoa() {}

Pessoa.prototype.classe = 'Mamífero';
Pessoa.prototype.falar = function() {
    console.log("Olá");
}

function Advogado() {}

Advogado.prototype.profissao = 'Advogado';

// clonando apenas o prototype de Pessoa
Advogado.prototype = Pessoa.prototype;

let joao = new Advogado();

joao.falar();
```

Precauções

- Utilizando a abordagem de clonar só o prototype tem um side effect;
- Se você muda o prototype, toda a cadeia que o utiliza, vai ser alterada também;
- **Então utilize desse jeito apenas quando não precisa mudar métodos e propriedades**;

```
let joao = new Advogado();

Advogado.prototype.falar = function() {
    console.log("Tchau");
}

let pedro = new Pessoa();

pedro.falar();
```

Construtor temporário

- Caso você tenha uma solução que não te deixaria optar por propriedades e métodos que não são alteráveis, você pode utilizar um construtor temporário e resolver o problema;

```
// clonando apenas o prototype de Pessoa, com construtor temporário
let F = function() {};
F.prototype = Pessoa.prototype;
Advogado.prototype = new F();
```

Isolando a herança em uma função

- Para facilitar as coisas e deixar a herança reutilizável também, podemos utilizar uma função;

```
function extend(Filho, Pai) {

    let F = function() {};
    F.prototype = Pai.prototype;
    Filho.prototype = new F();

}

function Pessoa() {}

Pessoa.prototype.classe = 'Mamífero';
Pessoa.prototype.falar = function() {
    console.log("Olá");
}

function Advogado() {}

Advogado.prototype.profissao = 'Advogado';

// herança
extend(Advogado, Pessoa)

let joao = new Advogado;

joao.falar();
```

Copiando propriedades

- Podemos em vez de utilizar o fake constructor,
  copiar as propriedades por um loop e
  realizar a herança;
- Precisamos utilizar a propriedade uber, que nos
  dará acesso ao obj Pai;
- A parte ruim desta abordagem é que ela recria
  as propriedades e métodos;

```
function extend(Filho, Pai) {
    let paiProto = Pai.prototype;
    let filhoProto = Filho.prototype;
    for(let i in paiProto) {
        filhoProto[i] = paiProto[i];
    }
    // filho tem acesso ao obj pai
    filhoProto.uber = paiProto;
}

function Veiculo() {}

Veiculo.prototype.motor = 1;
Veiculo.prototype.carenagem = 'aço'

function Carro(cor) {
    this.cor = cor;
}

Carro.prototype.portas = 4;

extend(Carro, Veiculo);

let bmw = new Carro('azul');

console.log(bmw.carenagem);
```

Outro problema ao copiar por loop

- Os arrays ficam alocados na memória e é criado apenas uma referência, fazendo com o que se o array do filho se altere o do pai também;

```
Veiculo.prototype.opicionais = ['teto solar', 'aro de alumínio', 'diplay 8"'];

extend(Carro, Veiculo);

console.log(Veiculo.prototype);

Carro.prototype.opicionais.push('blindagem');

console.log(Veiculo.prototype.opicionais);
```

Resolvendo o problema

- Podemos utilizar uma estratégia de copiar um objeto, resolvendo este problema;
- Porém veja que o código fica complexo demais, talvez não seja o caso de utilizar herança para isso;
- Além de não utilizar prototypes nesta forma;

```
function objectClone(o, stuff) {
  var n;
  function F() {}
  F.prototype = o;
  n = new F();
  n.uber = o;
  for (var i in stuff) {
   n[i] = stuff[i];
  }
  return n;
}


function Veiculo() {
  this.carenagem = 'aço';
  this.opicionais = ['blindagem', 'lanterna LED'];
}

let v = new Veiculo;

let ferrari = objectClone(v, {
    rodas: 4,
  marca: 'Ferarri'
});

console.log(ferrari);
```

Herança múltipla

- Uma estrutura difícil de manter e o JS não nos dá esta funcionalidade de forma fácil. Precisamos criar a função;
- É difícil de manter;
- Melhor optar por prototype chain;

```
function multi() {
  var n = {}, stuff, j = 0, len = arguments.length;
  for (j = 0; j <len; j++) {
    stuff = arguments[j];
    for (var i in stuff) {
      if (stuff.hasOwnProperty(i)) {
        n[i] = stuff[i];
      }
    }
  }
  return n;
}

let pneu = {
    material: 'borracha'
}

let aro = {
    tamanho: 20
}

let pneuMontado = multi(pneu, aro);

console.log(pneuMontado);
```

Herança

Atualizando....

Definindo classes

Recordando....

- A declaração é bem parecida com constructor functions;
- As propriedades devem ficar num método especial chamado constructor;
  - Onde serão inicializadas;

```
class Tenis {
    constructor(modelo, cor) {
        this.modelo = modelo;
        this.cor = cor;
    }
}

console.log(typeof Tenis);

let allstar = new Tenis("All Star", "Branco");
```

Herança

- Utilizando a palavra extends, uma classe herda as propriedades e métodos de outra;
- **Bem mais fácil, não? :D**

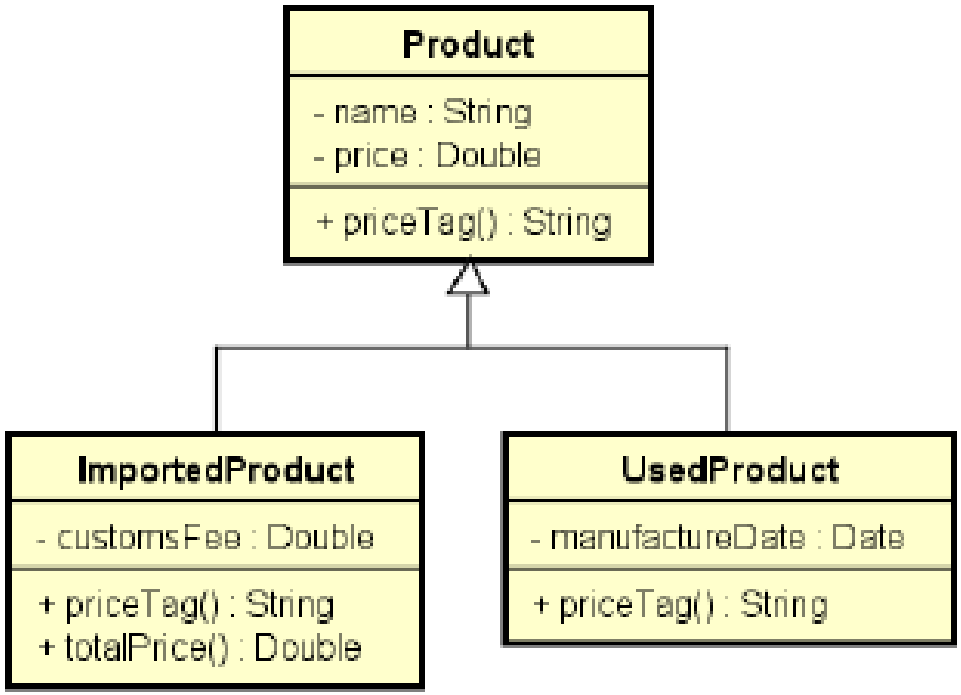
```
class Animal {
    constructor(nome) {
        this.nome = nome;
    }
}

class Cachorro extends Animal {
    latir() {
        console.log("Au au");
    }
}

let bob = new Cachorro("Bob");

bob.latir();

console.log(bob.nome);
```

Exercício de fixação (1)

- Fazer um programa para ler os dados de N produtos (N fornecido pelo usuário). Ao final, mostrar o preço de cada produto na mesma ordem em que foram digitados.

- Todo produto possui nome e preço. Produtos importados possuem uma taxa de alfândega, e produtos usados possuem data de fabricação. Estes dados específicos devem ser acrescentados no preço conforme exemplo. Para produtos importados, a taxa e alfândega deve ser acrescentada ao preço final do produto.

Exercício de fixação (1)



Exercício de fixação (1)

```
Entre com o número de produtos: 3
Dados do 1ª Produto:
Comum, usado ou importado (c/u/i)? i
Nome: Tablet
Preço: 260.00
Taxa aduaneira: 20.00

Dados do 2ª Produto:
Comum, usado ou importado (c/u/i)? c
Nome: Notebook
Preço: 1100.00

Dados do 3ª Produto:
Comum, usado ou importado (c/u/i)? u
Nome: Iphone
Preço: 400.00
Data de fabricação (dd/mm/aaaa): 01/01/2019

Preços:
Tablet R$ 280.00 (taxa aduaneira: R$ 20.00)
Notebook R$ 1100.00
Iphone (used) R$400.00 (data de fabricação: 01/01/2019)
```

Exercício de fixação (2)

- Um animal contém um **nome**, **comprimento**, **número de pata**s, uma **cor**, **ambiente** e uma **velocidade** (em m/s).
- Um peixe é um animal, tem 0 patas, o seu ambiente é o mar, cor cinzenta. Além disso, o peixe tem como **característica**: tem barbatanas.
- Um mamífero é um animal, o seu ambiente é a terra.
  - Um urso é um mamífero, cor castanho e o seu **alimento** preferido é o mel.

- Codifique as classes **Animal**, **Peixe** e **Mamífero**

Exercício de fixação (2)

A classe **Animal** possui os **atributos**:

- nome (texto), comprimento (int), patas (int), cor (texto), ambiente (texto) e velocidade (float)

- **Animal** (nome, cor, ambiente, comprimento, velocidade, patas)

Para a classe Animal, codifique os **métodos**:

- AlterarNome(nome)
- AlterarComprimento(comprimento)
- AlterarPatas(patas)
- AlterarCor(cor)
- AlterarAmbiente(ambiente)
- AlterarVelocidade(velocidade)
- Dados()//imprime os dados do animal

Exercício de fixação (2)

Para a classe **Peixe**, codifique:

- **Peixe**(nome, caracteristica, comprimento, velocidade)
- AlterarCaracteristica(caracteristica)
- Caracteristica()//retorna a característica de um determinado peixe
- Dados()//imprime todos os dados

Para a classe **Mamifero**, codifique:

- **Mamifero**(nome, cor, alimento, comprimento, velocidae, patas)
- AlterarAlimento(alimento)
- Alimento()//retorna o alimento de um determinaod uso
- Dados()// imprime todos os dados de um mamifero

Exercício de fixação (2)

- Por último, crie a rotina de teste de forma a ter um jardim zoológico com os seguintes animais: **camelo, tubarão e urso-do-canadá**.

Exemplo de execução:

```
Zoo:
-----------------------------
Animal: Camelo
Comprimento: 150 cm
Patas: 4
Cor: Amarelo
Ambiente: Terra
Velocidade: 2.0 m/s
-----------------------------
Animal: Tubarão
Comprimento: 300 cm
Patas: 0
Cor: Cinzento
Ambiente: Mar
Velocidade: 1.5 m/s
Caracteristica: Barbatanas
-----------------------------
Animal: Urso-do-canadá
Comprimento: 180 cm
Patas: 4
Cor: Vermelho
Ambiente: Terra
Velocidade: 0.5 m/s
Alimento: Mel
-----------------------------
```

FIESC
CIESC
SESI
SENAI
IEL

SENAI

Iniciativa da FIESC - Federação das Indústrias do Estado de Santa Catarina

Serviço Nacional de Aprendizagem Industrial de Santa Catarina