Funções Construtoras no JavaScript

Desenvolvimento de Tecnologia RIA

DTR

Gléderson L. dos Santos

gledersonsantos@ifsul.edu.br

Campus Charqueadas
Tecnologia em Sistemas para Internet

Introdução

- Nas aulas passadas trabalhamos com os conceitos que envolvem tratamento de objetos no JavaScript
- Vimos que o JS é baseado em um paradigma que utiliza orientação a objetos baseada em protótipos
  - um objeto é uma coleção de propriedades
  - propriedades são associações entre um nome (ou chave) e um valor
  - Uma propriedade que possui como valor uma função é chamada de método
  - Um objeto possui também atributos, que são representações de características típicas desse objeto
  - Finalmente, objetos possuem um protótipo, que é o objeto antecessor do qual este herda propriedades.

Introdução

- Vimos também que um objeto possui atributos especificamente responsáveis por gerenciar o comportamento do referido objeto
  - Atributos de objeto: Responsáveis por gerenciar o comportamento do objeto e as origens do mesmo
    - Protótipo
    - Tipo (classe)
    - indicação de expansibilidade
  - Atributos de propriedade: Responsáveis por gerenciar o comportamento de uma propriedade específica do objeto
    - Editável
    - Enumerável
    - Configurável

Introdução - Criação direta de Objetos

- Identificamos 2 modos básicos de criar um objeto diretamente:
  - Objetos literais {}
  - new Object()

- Também vimos como criar um objeto como uma cópia de outro objeto, fazendo um melhor uso da característica prototípica da linguagem
  - Object.create()

Introdução - Varredura de propriedades de objetos

- Verificamos 3 métodos de realizar a varredura de propriedades de um objeto, a saber:
  - Através da iteração for...in
    - Itera sobre todas as propriedades enumeráveis de um objeto (bem como de sua cadeia de protótipos)
  - Object.keys(o)
    - Retorna um array com os nomes de todas as propriedades próprias e enumeráveis do objeto o
  - Object.getOwnPropertyNames(o)
    - Retorna um array contendo todas as propriedades próprias (enumeráveis ou não) do objeto o

Introdução

- A partir dessa estrutura básica, foi possível observar todo o potencial no tratamento dinâmico de propriedades de objetos
- Com o JavaScript, é possível adicionar e deletar facilmente propriedades de objetos (MUTABILIDADE)
- Associado com os mecanismos de varredura, objetos podem facilmente ser mixados, como foi exemplificado a partir da criação de funções que permitem executar mecanismos como por exemplo a extensão, o merge, a união e a intersecção de propriedades de objetos

Introdução

- Embora com grande potencial, tais mecanismos apresentados até aqui podem tornar o gerenciamento de objetos e o próprio código desenvolvido bastante confuso.

- Uma alternativa interessante tanto para melhor organizar o código quanto para aumentar seu reuso é adotar funções construtoras

Funções construtoras

- Como citado anteriormente, somente no ECMA 6 foi criado uma estrutura de orientação a objetos clássica no JavaScript, e até então não existia o conceito de classes na linguagem.

- Funções construtoras são mecanismos usados para permitir a criação de objetos a partir de uma função JavaScript.

Funções construtoras

- Com o uso de funções construtoras, o mecanismo de criação de objetos se baseia em dois passos
  - Definição do tipo de objeto a partir da criação de uma função construtora, que cria seus métodos e atributos
  - Criação de uma instância do objeto a partir do operador **new**
  - Essa técnica é chamada de **Constructor Paradigm** (Nicholas Zakas em "Professional JavaScript for Web Developers")

Exemplos de funções construtoras

```
function Carro(marca, modelo, ano){
        this.marca= marca;
        this.modelo = modelo;
        this.ano=ano;
}
function Pessoa(nome, sobrenome, email){
        this.nome= nome;
        this.sobrenome = sobrenome;
        this.email=email;
}
```

Instanciando objetos a partir de funções construtoras

```
var meuCarro = new Carro("Volkswagen", "Fusca",1973);

var pessoa1 = new Pessoa("Gléderson", "Lessa dos Santos", "glederson@gmail.com");

var pessoa2 = new Pessoa("João", "Santos da Silva", "xxxxxx@gmail.com");
```

Objetos compostos de outros objetos

- Como vimos anteriormente, nada impede que precisemos criar objetos compostos por outros objetos. Isso pode ser feito naturalmente a partir de funções construtoras

- Veja a seguinte alteração na função construtora Carro

```
function Carro(marca, modelo, ano, dono){
        this.marca= marca;
        this.modelo = modelo;
        this.ano=ano;
        this.dono=dono;
}
var meuCarro = new Carro("Volkswagen", "Fusca",1973, pessoa1);
```

Vantagens no uso de funções construtoras

- Além de uma maior organização do código, o uso de funções construtoras permite que tenhamos um controle maior sobre as instâncias criadas de nossa função construtora
  - Usando objetos literais, precisamos tomar o cuidado de não esquecer nenhuma propriedade do objeto
    - Object.create() resolve esse problema
  - Consistência nos dados
    - Objetos literais não conseguem estabelecer nenhum tipo de controle sobre a validação dos dados de inicialização das propriedades de um objeto, o que pode ser feito com funções construtoras

Exemplo

```
function Pessoa(nome, sobrenome, email){
        this.nome= nome;
        this.sobrenome = sobrenome;
        if (validaMail(email)) this.email=email;
        else this.email = "indefinido";
}
```

Adição e remoção de propriedades de objetos

- O uso de funções construtoras não afeta a liberdade de alterar dinamicamente instâncias de objetos

- Assim poderíamos criar uma propriedade nova, chamada cor, apenas para a instância armazenada por meuCarro

```
meuCarro.cor="branco"
```

- Outras instâncias de Carro, tanto já existentes quanto a serem criadas, não sofrem quaisquer tipo de alterações

Adição de métodos a uma função construtora

- A adição de métodos a uma função construtora segue a mesma estrutura vista até o momento. Suponha que queiramos adicionar o método apresentar() a Pessoa:

```
function Pessoa(nome, sobrenome, email){
        this.nome= nome;
        this.sobrenome = sobrenome;
        this.email=email;
        this.apresentar = function(){
        console.log("Olá, meu nome é"+this.nome+" "+
                this.sobrenome+" e meu e-mail é "+this.email+". ");
        };
}
```

O que diferencia uma função simples de uma função construtora?

- Sintaticamente, nada!

- Se chamarmos a função Pessoa como uma função simples ela será executada normalmente

```
var pessoa3=Pessoa("José", "Bastos", "xxxx@gmail.com");
```

- Pior ainda, verifique o conteúdo de pessoa3 e this
  - console.log(pessoa3);
  - console.log(this);

  - Essa ambiguidade das funções construtoras é alvo de críticas por diversos especialistas na área, como Douglas Crockford em “JavaScript The Good Parts”

E agora, o que fazer?

Padrão CamelCase

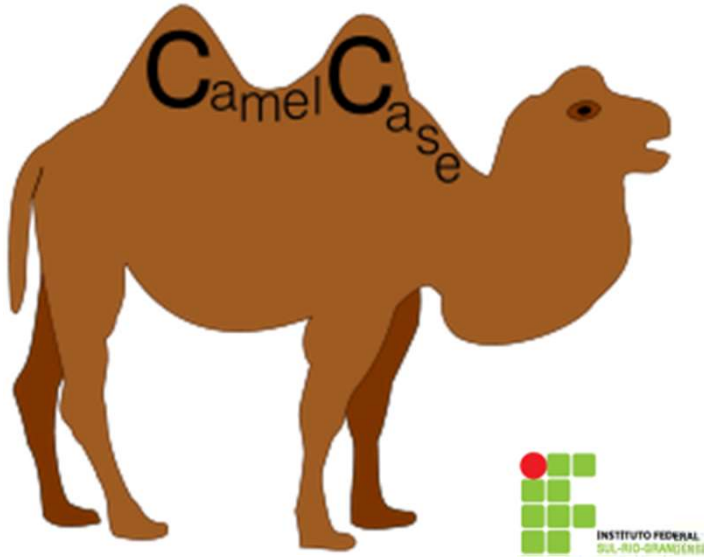
- A única forma de diferenciar o que é uma função construtora é a partir da nossa organização de código

- Adote a prática mais usual, identificando suas funções simples com letra minúscula e suas funções construtoras como se fossem classes (letra maiúscula).

- Também chamado de

UpperCamelCase ou

PascalCase 😐



Desvantagem da técnica Constructor Paradigm

- Voltando a função construtora Pessoa, perceba o trecho de código apresentado na abaixo:

```
function Pessoa(nome, sobrenome, email){
    this.nome= nome;
    this.sobrenome = sobrenome;
    this.email=email;
    this.apresentar = function(){
        console.log("Olá, meu nome é"+this.nome+" "+
            this.sobrenome+" e meu e-mail é "+this.email+". ");
    }
}
```

Desvantagem da técnica Constructor Paradigm

- Embora tenhamos grandes ganhos em termos de organização, note que cada nova instância de objeto Pessoa gera um novo método apresentar
  - Verifique isso com

    console.log(pessoa1)

- Assim, tal método é carregado inúmeras vezes na memória, gerando um desperdício de recursos e, consequentemente uma perda de performance

Alterando o protótipo de uma função construtora

- Conforme visto na aula passada, todos objetos possuem um protótipo indicando qual é o seu objeto original que podia ser visualizado a partir do método Object.getPrototypeOf()
  - Em navegadores, normalmente o protótipo fica armazenado no atributo __proto__ do objeto (ISSO NÃO É UM PADRÃO!)

- Caso o método estivesse presente no protótipo do objeto, esse desperdício de recursos pelo armazenamento de múltiplas cópias de uma mesma função não ocorreria
  - Nesse caso só existe um método na memória, o do próprio protótipo do qual se baseia o objeto
  - Note que é por isso que na aula passada foi citado que propriedades herdadas não podem ser deletadas

prototype

- Para adicionar propriedades diretamente ao protótipo da função devemos associar essas propriedades diretamente ao protótipo da função construtora. Isso é feito a partir do **prototype**

- Vamos alterar o método apresentar para que seja um método do protótipo

Exemplo

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

Pessoa.prototype.apresentar = function(){
        console.log("Olá, meu nome é"+this.nome+" "+
         this.sobrenome+" e meu e-mail é "+this.email+". ");
}
```

prototype

- Esse método de se criar um objeto onde existe uma mistura da função construtora (com atributos) e o prototype desta foi apelidado por Douglas Crockford como **Pseudo-classical Pattern**, por buscar uma aproximação estrutural com o linguagens que seguem a orientação a objetos clássica, como o Java.

  http://www.crockford.com/

prototype

- O uso do prototype não se resume a um ganho de desempenho

- A partir do prototype podemos também definir atributos com valores comuns a todos as instâncias da função construtora
  - Tais valores podem ser tratados como valores default daquele atributo
  - Tais atributos podem simular ainda atributos de estáticos (ou de classe).

Exemplo

```
function Pessoa(nome, sobrenome, email){
   this.nome= nome;
   this.sobrenome = sobrenome;
   if (email) this.email=email;
}

Pessoa.prototype.apresentar = function(){
        console.log("Olá, meu nome é"+this.nome+" "+
         this.sobrenome+" e meu e-mail é "+this.email+". ");
}

Pessoa.prototype.email = "contato@ifsul.edu.br";
```

prototype em objetos nativos

- Com o uso do prototype podemos alterar objetos nativos do JavaScript adicionando novas funcionalidades aos mesmos.
- Tais funcionalidades estarão disponíveis a todos os objetos presentes nesta classe
- Exemplo

  strings.html
- O uso do prototype é bastante explorado por frameworks como o jQuery para aumentar as capacidades do JavaScript puro

Sobrecarga de protótipo

- Quando implementamos um sistema maior, e consequentemente classes mais complexas, podemos nos deparar com a situação de implementar uma grande quantidade de métodos diretamente no protótipo, o que alguns desenvolvedores argumentam que criaria um código mais poluído.
- Uma alternativa para essa situação que é adotada por alguns desenvolvedores é a sobrecarga completa do prototype

Sobrecarga de protótipo

```
function Pessoa2(nome, sobrenome, email){
    this.nome= nome;
    this.sobrenome = sobrenome;
    if (email) this.email=email;
}

Pessoa2.prototype = {
    constructor: Pessoa2,
    apresentar: function(){
        console.log("Olá, meu nome é"+this.nome+" "+this.sobrenome+
          " e meu e-mail é "+this.email+".");
    },
    despedir:function(n){
        var frase="Até logo";
        if (n) console.log(frase+","+n+".");
        else console.log(frase+"!");
    },
    email:"contato@ifsul.edu.br"
};
var antonio= new Pessoa2("Antonio","Carlos");
console.log(antonio);
antonio.apresentar();
antonio.despedir(pessoa1.nome);
```

Como estamos sobrescrevendo todo o protótipo de Pessoa2, devemos adicionar também seu parâmetro **constructor** responsável por armazenar a função de criação do objeto (classe)

Desvantagem do Pseudo-classical Pattern

- Usando o Pseudo-classical Pattern, os métodos estão nos protótipos das funções construtoras, que só são carregados uma vez
  - Melhora de performance
  - Redução no consumo de memória

- Entretanto, não existe forma de criar propriedades privadas

Existem propriedades públicas e privadas no JavaScript

- Da mesma forma que na relação com classes, não possuímos parâmetros que indiquem se uma propriedade é publica ou privada

- Entretanto, podemos construir estruturas com comportamento análogo aos da visibilidade de propriedades de outras linguagens orientadas a objetos

http://javascript.crockford.com/private.html

Propriedades Públicas

- Membros de um objeto (propriedades) são todos considerados públicos. Logo, tais propriedades podem ser acessadas, modificadas ou até mesmo deletadas* conforme nosso interesse
  - *Desde que não sejam propriedades herdadas

- Assim, nos exemplos anteriores, sempre lidamos com propriedades que, na prática, eram públicas, tanto nas técnicas Constructor Paradigm quanto na Pseudo-classical Pattern.

Propriedades Privadas

- Propriedades privadas podem ser criadas apenas no construtor.

- Para isso só precisamos criar o atributo utilizando o parâmetro var

```
function Container(parametro){
        this.attrPublico=parametro;
        var attrPrivado=parametro;
        this.eu = function(){
                console.log(this.attrPublico);
                console.log(attrPrivado);
        };
}
var cont = new Container(5);
console.log(cont);
console.log(cont.attrPublico);
console.log(cont.attrPrivado);
cont.eu();
```

Pseudo-classical Pattern e propriedades privadas

- Funções criadas no prototype não são capazes de acessar atributos privados

- Adicione abaixo da função construtora de Container as seguintes linhas:

```
Container.prototype.eu2 = function(){
        console.log(this.attrPublico);
        console.log(this.attrPrivado);
}
var cont2 = new Container(5);
cont2.eu();
cont2.eu2();
```

Seria possível remover o operador **this** para corrigir esse comportamento?

Métodos Privileged (Privilegiados)

- No exemplo anterior, embora os métodos eu e eu2 sejam ambos públicos (podem ser acessados de qualquer ponto do código) somente o primeiro acessa propriedades privadas.

- Métodos que acessam propriedades privadas são chamados de **Privileged** (privilegiados)

- Métodos privilegiados podem ser criados apenas na função construtora

Métodos privados

- Da mesma forma que criamos atributos privados, podemos implementar métodos privados no construtor.

```
function Container(parametro){
  this.attrPublico=parametro;      //atributo público
  var attrPrivado=parametro-2;     //atributo privado
  this.eu = function(){            //função pública (privilegiada)
    console.log(this.attrPublico);
    console.log(attrPrivado);
  };
  function incrementa(){           //função privada
    attrPrivado +=1;
  }
}
Container.prototype.eu2 = function(){//função pública
  console.log(this.attrPublico);
  console.log(this.attrPrivado);
};
```

Tente acessar o método incrementa()

Resumo de visibilidade de propriedades

- Padrão de declaração
  - Público

```
function Constructor(...){
        this.atributoPublico  = value;
}
Constructor.prototype.atributoPublico=value; //Use somente se quiser um valor default ou um
                                                //atributo comum a todos os objetos de Constructor
Constructor.prototype.metodoPublico=function(...){...};
```

  - Privado

```
function Constructor(...) {
        var atributoPrivado = value;
        function metodoPrivado(...) {...};
}
```

  - Privilegiado

```
function Constructor(...){
        this.metodoPrivilegiado = function(...){...};
}
```

Getters e Setters

- Mesmo sem termos classes e uma estrutura de visibilidade de propriedades clássicas, conseguimos criar funções construtoras e propriedades públicas e privadas.

- E métodos de get e set?

SIM! O JavaScript oferece suporte a métodos de get e set ☺

Getters e Setters

- Entretanto...
  - O uso de get e set é bastante discutido e, muitas vezes, é contraindicado por grande parte dos desenvolvedores
    - Exemplo: Google
  - Isso porque, dependendo da implementação, ele pode gerar diversos efeitos colaterais indesejáveis (altero uma propriedade com get/set e altero outra inesperadamente)

Getters e Setters

- Exemplo

```
var x = [1, 2, 3, 4];
x.length = 3;
console.log(x.length);
console.log(x);
```

Usando os bindings get e set

- O JavaScript pode definir funções especiais que se comportam como atributos, mas que na verdade são funções, usadas para atribuir ou receber valores.
  - Tais bindings são o get e o set

Get e Set em objetos literais

```
var pessoa = {
        nome: "Glederson",
        sobrenome: "Lessa dos Santos",
        get nomeCompleto() {
                return this.nome + " " + this.sobrenome;
        },
        set nomeCompleto(fullname){
                var names = fullname.split(" ");
                var len=names.length;
                this.nome = names[0];
                this.sobrenome=names[1];
                for(var i=2;i<len;i++){
                        this.sobrenome += " "+names[i];
                }
        }
};
```

Acessando Get e Set em objetos literais

```
console.log(pessoa.nomeCompleto);
pessoa.nomeCompleto = "Joaquim Costa e Silva";
console.log(pessoa.nomeCompleto);
console.log(pessoa);
```

Get e Set em protótipos

- Podemos definir funções getter e setter para um protótipo tanto quando usamos sobrecarga de protótipo quanto quando definimos novos métodos para o protótipo.
  - No segundo caso usaremos o método defineProperty()

Get e Set com sobrecarga de protótipo

```
function Pessoa(nome, sobrenome){
   this.nome= nome;
   this.sobrenome = sobrenome;
}
Pessoa.prototype = {
        constructor: Pessoa,
        get nomeCompleto() {
                return this.nome + " " + this.sobrenome;
        },
        set nomeCompleto(fullname){
                var names = fullname.split(" ");
                var len=names.length;
                this.nome = names[0];
                this.sobrenome=names[1];
                for(var i=2;i<len;i++){
                        this.sobrenome += " "+names[i];
                }
        }
}
```

Get e Set em com o defineProperty()

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

Object.defineProperty(Pessoa.prototype,”nomeCompleto”,{
        get: function() {
                return this.nome + " " + this.sobrenome;
        },
        set: function(fullname){
                var names = fullname.split(" ");
                var len=names.length;
                this.nome = names[0];
                this.sobrenome=names[1];
                for(var i=2;i<len;i++){
                        this.sobrenome += " "+names[i];
                }
        }

});
```

Get e Set com atributos privados

- Conforme vimos anteriormente, a prototype não consegue acessar propriedades privadas
- Nesse caso, temos que usar a função defineProperty() dentro da função construtora

Get e Set em com o defineProperty()

```
function Pessoa(n, s){
        var nome= n;
        var sobrenome = s;
        Object.defineProperty(this,"nomeCompleto",{
                get: function() {
                        return nome + " " + sobrenome;
                },
                set: function(fullname){
                        var names = fullname.split(" ");
                        var len=names.length;
                        nome = names[0];
                        sobrenome=names[1];
                        for(var i=2;i<len;i++){
                                sobrenome += " "+names[i];
                        }
                }
        });
}
```