Protótipos

Desenvolvimento Front-end III

DFE III

Gléderson L. dos Santos

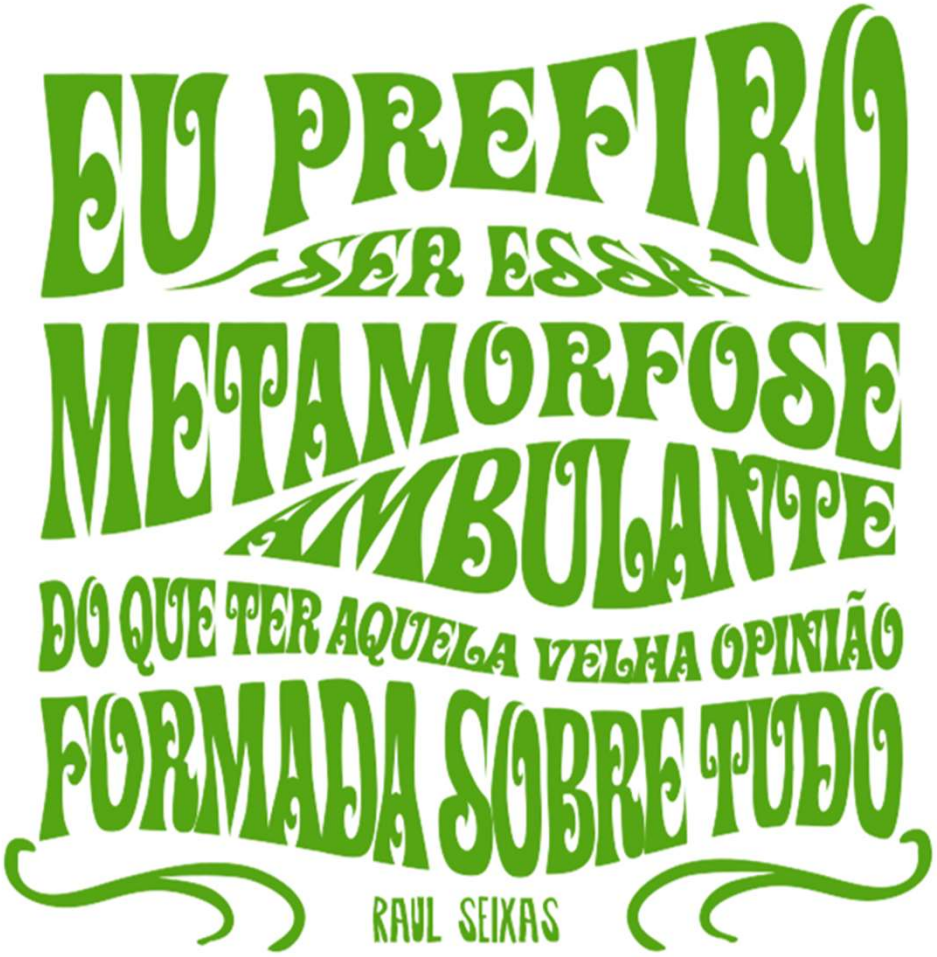
gledersonsantos@ifsul.edu.br

Campus Charqueadas
Tecnologia em Sistemas para Internet

Introdução

- Na aula anterior, acompanhamos a fundamentação básica acerca da manipulação de objetos no JS
  - Primeiramente, verificamos como criar objetos literais com o uso de JavaScript
  - Adicionalmente, aprendemos como definir métodos a esses objetos

Mutabilidade dos objetos



Objetos JavaScript são dito mutáveis (podem ter suas propriedades alteradas em tempo de execução)

Mutabilidade dos objetos

- **Quais as alternativas vistas até aqui para criação de objetos?**
  - Através de new Object e através do operador {}

- **Como adicionar as propriedades em objetos criados a partir de new Object()?**
  - Basta fazer uma atribuição para o novo atributo e este será criado
  - Isso vale também para objetos criados com o operador {}

Exercício

- Crie um objeto card, composto pelos atributos: título, subtítulo, mensagem, cor e container usando os mecanismos aprendidos na ultima aula.
- Após crie um método para esse objeto, denominado renderiza(), o qual deve criar um div no navegador contendo:
  - um h1 composto pelo conteúdo de título,
  - um elemento h2 com o subtítulo,
  - um p com a mensagem
  - cor de fundo é igual ao conteúdo presente em cor.
  - Associe o div resultante ao atributo container, como forma de criar um vínculo entre o elemento de tela e o objeto controlador aqui descrito

Observação

- Um objeto é, em muitos aspectos, similar a um array associativo.

- Entretanto, diferentemente de um simples mapeamento nome-valor, um objeto JavaScript também é capaz de herdar propriedades de outro objeto, conhecido como seu protótipo.

Criando protótipos de objetos

- Além dos mecanismos de criação de objetos até aqui apresentados podemos também criar objetos a partir da função Object.create()

- Veremos qual a motivação para criar objetos dessa forma e, de forma correlata, o conceito de protótipos

Protótipos

- Todos objetos JavaScript possuem um segundo objeto associado a ele. Esse objeto é chamado de protótipo e dele são herdadas propriedades

- Todos objetos literais possuem o mesmo protótipo, referenciado como Object.prototype

O que exatamente são protótipos?

Orientação a Objetos Clássica X Orientação a Objetos prototípica

- Linguagens como Java utilizam a orientação a objetos clássica
  - Tipos herdam de tipos (classes herdam de classes)
  - Uma classe é um molde para a criação de objetos

Orientação a Objetos Clássica X Orientação a Objetos prototípica

- Linguagens como JavaScript utilizam a orientação a objetos prototípica
  - Objetos herdam de Objetos (Objeto filho referencia um objeto pai ou protótipo)
  - Um filho inicialmente é um clone do seu pai (protótipo)

I Am Your Father!

Orientação a Objetos Clássica X Orientação a Objetos prototípica

- O protótipo é uma propriedade interna, e por isso ela não é armazenada com as propriedades do objeto instanciadas pelo programador

- O JavaScript busca uma propriedade em um objeto e, caso não encontre, buscará a propriedade no objeto pai

CUIDADO!

- Diferentemente do que poderíamos pensar em um primeiro instante, um objeto não copia as propriedades de um objeto protótipo.

- Na verdade as propriedades do protótipo são referenciadas (apontadas) pelo objeto filho.

Object.create()

- Função estática (não pode ser usada invocada em objetos individuais) que cria um novo objeto. Para isso, basta passar o protótipo de objeto desejado, que deve ser passado como parâmetro desta função.

- Criem um objeto ponto, composto pelos atributos x, y e pelo método distanciaOrigem que calcula a distancia desse ponto até a origem

Object.create()

```
let ponto={x:0,
      y:0,
      distanciaOrigem:function(){
         return Math.sqrt(this.x**2+this.y**2);
      }
}
let pt1=Object.create(ponto);
```

Object.create()

- Para criar um objeto vazio, similar ao objeto literal {} ou ao new Object() basta chamar repassando null como parâmetro

```
var objeto = Object.create(null);
```

Object.create()

- O método Object.create() serve para criar um objeto com uma referência ao objeto repassado como parâmetro.

- Nesse sentido, esse objeto se assemelharia (inicialmente) a um clone do objeto repassado como parâmetro

Object.create()

Lembre-se de que esses objetos são mutáveis!

Exemplo

```
let pai={ nome:"Glederson", sobrenome:"Santos"};
let filho = Object.create(pai);
pai;
filho;
filho.nome;
filho.sobrenome;
pai.sobrenome="Lessa";
pai;
filho;
filho.sobrenome;
filho.nome="Felipe";
pai;
filho;
pai.nome;
filho.nome;
```

Protótipos

- A função Object.getPrototypeOf(x) retorna o protótipo do objeto

- Verifique o protótipo da variável filho.
- Verifique também os protótipos de objetos Array e Date a partir do método Object.getPrototypeOf().

Protótipos

- Objetos criados com o atributo new possuem como protótipo o valor de protótipo de seu construtor

- Assim um objeto criado com new Object() possui como protótipo o Object.prototype e um objeto criado com new Date() possui como protótipo o Object.

Atributos de gerenciamento de objeto típicos do JavaScript

- Associado a cada objeto, existem três atributos de objeto:
  - O protótipo do objeto que é uma referência a outro objeto do qual as propriedades foram herdadas
  - O tipo do objeto
  - Uma flag indicando se mais propriedades podem ser adicionadas a um objeto

Tipos de propriedades

- Podemos classificar propriedades de objetos como sendo de dois tipos
  - Propriedade direta (own property): Propriedade definida diretamente em um objeto
  - Propriedade herdada: Propriedade definida a partir de um protótipo do objeto

Atributos de propriedades

- Além de um nome e um valor, cada propriedade possui uma série de valores associados, chamados atributos de propriedade:
  - editável (writable): especifica se o valor de uma propriedade pode ser setado
  - enumerável (enumerable): especifica se essa propriedade pode ser visualizada em laços de consulta.
  - configurável (configurable): especifica se uma propriedade pode ser deletada e se seus atributos de propriedade podem ser alterados

  - Até o ECMA 5 todas propriedades eram editáveis, enumeráveis e configuráveis. Entretanto, a partir do ECMA 5 essas propriedades podem ser modificadas.

Erros quando acessando propriedades

- Caso você cometa algum erro, e tente acessar uma propriedade inexistente (o objeto não possui a propriedade, ou seja, ela não é direta e nem herdada), o acesso a essa propriedade retornará undefined

```
var ponto={x: 1, y:1};
console.log(ponto.z);
```

Erros quando acessando propriedades

- Como a propriedade inexistente retorna undefined, poderemos chegar a erros de tipo caso busquemos uma cadeia de propriedades, como por exemplo:

```
var total = ponto.z.length;
```

- Esse erro interrompe o fluxo de execução do programa

Erros quando acessando propriedades

- Para evitar esse erro podemos usar um mecanismo que garanta o acesso a propriedades apenas se essas existem:

```
var total=undefined;
if(p){
        if(p.z) total=p.z.length;
}
```

- Tal mecanismo pode ser também sintetizado como:

```
var len=p&&p.z&&p.z.length;
```

Valores truthy e falsy

Curto-circuito (short-circuiting)

- O comportamento do operador && demonstrado anteriormente é chamado por vezes de curto-circuito e é usado por alguns desenvolvedores para explorar condicionalidades na execução do código.

- Por exemplo, esse comportamento faz com que os seguintes códigos tenham efeitos equivalentes:

```
var a=5;
var b=prompt("Digite 5");
if (a==b) alert("Entrou");
(a==b) && alert("Também entrou");
```

Deletando propriedades

- Conforme citado anteriormente, usamos o operador delete para remover propriedades de objetos.
- delete retorna true caso consiga remover o operador ou se esse operador já não existia.
- O delete não consegue remover propriedades que não são configuráveis (atributo configurable=false).

Deletando propriedades - Exemplo

```
delete Object.prototype;
var x=1;
delete x;
function f(){};
delete f;
var pai={nome:"Glederson",sobrenome:"Santos"};
pai;
delete pai.sobrenome
pai;
```

Testando propriedades

- Vimos que o operador **in** verifica se uma propriedade pertence a um objeto.

- Uma alternativa ao operador **in** que vocês poderão encontrar é a expressão:

  ***propriedade* !== undefined;**

  - Exemplo:

    p.x!==undefined;

Você consegue visualizar uma possível diferença de resultados entre o in e o !== na identificação de propriedades? in permite diferenciar se uma propriedade não existe ou se ela existe e possui valor undefined

Testando propriedades

- O método hasOwnProperty() também faz essa verificação de existência de propriedade

- Diferentemente do operador **in**, o método hasOwnProperty só avalia propriedades diretas, sem avaliar aquelas herdadas do objeto protótipo.

```
var o = { x: 1 }
o.hasOwnProperty("x");
```

Testando propriedades

- Refinando ainda mais a função hasOwnProperty(), a função propertyIsEnumerable() retorna verdadeiro apenas se:
  - a propriedade é direta
  - seu atributo de propriedade enumerable é true (propriedade é enumerável)

Enumerando propriedades

- Todas as propriedades enumeráveis podem ser descritas a partir de um laço for .. in

```
var p = {x:1,y:1};
for (var prop in p){
        console.log(prop);
}
```

Refinando apresentação de propriedades

- Caso queiramos apresentar apenas as propriedades diretas de um objeto, podemos implementar o seguinte arranjo

Refinando apresentação de propriedades

- Verifique o que o seguinte exemplo faz:

```
var p = {x:1,y:1};
var q = Object.create(p);
q["z"]=1;
for(var prop in q){
        console.log(prop);
}
```

Coletando apenas propriedades diretas

```
var p = {x:1,y:1};
var q = Object.create(p);
q["z"]=1;
for(var prop in q){
        if(q.hasOwnProperty(prop)){
                console.log(prop);
        }
}
```

Como coletar apenas as propriedades (enumeráveis) herdadas?

Apresentando apenas métodos do objeto

```
for(p in o) {
        if (typeof o[p] === "function") {
                console.log (p);
        }
}
```

Como acessar apenas atributos do objeto?

Criando mecanismos auxiliares para trabalhar com objetos

- Os exemplos anteriores mostram como temos liberdade no gerenciamento de propriedades de objetos.
- A partir deles podemos criar algumas funções bem interessantes para esse tratamento de objetos, como a função extend, muito comum de ser encontrada em bibliotecas auxiliares:

https://api.jquery.com/jquery.extend/

função extend()

- A ideia é copiar as propriedades de um objeto em outro objeto de forma a estender o primeiro em um segundo objeto

```
function extend(o, p) {
        for(prop in p) {
                o[prop] = p[prop];
        }
        return o;
}
```

Exemplo: tenho um objeto que representa um calendário e outro objeto que representa um relógio e quero criar um objeto que representará um relógio com calendário.

função extend()

- Testem a função extend com um objeto Pessoa, composto de nome e sobrenome, e um objeto pFisica, composto inicialmente de cpf.

- O que acontece se pFisica tiver uma propriedade nome antes de ser usada a função extend()?

Observações

- O ECMA 6 prevê a palavra chave extends, que funciona de forma mais comum a outras linguagens orientadas a objetos.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/extends

O método extend implementado por nós não funciona no IE devido a um bug do navegador

merge de objetos

- Podemos alterar minimamente nossa função para copiar apenas propriedades não existentes no objeto original

```
function merge(o, p) {
        for(prop in p) {
                if (!(prop in o)){
                        o[prop] = p[prop];
                }
        }
        return o;
}
```

Funções auxiliares nativas do JavaScript

- Object.keys()
  - Retorna um array com o nome propriedades diretas e enumeráveis de um objeto

- Object.getOwnPropertyNames()
  - Funciona de forma similar ao Object.keys(), mas retorna o nome de todas as propriedades diretas de um objeto, incluindo aquelas não enumeráveis

Exercício extra: Outras funções auxiliares

- Tente criar as seguintes funções auxiliares:
  - restrict(o,p) – Remove propriedades de o que não estão presentes em p
  - substract(o,p) – Remove as propriedades de o que estão presentes em p
  - union(o,p) – Retorna um novo objeto com a união das propriedades de o e p
  - intersection(o,p) – Retorna um novo objeto com a intersecção das propriedades de o e p (aparecem em ambos objetos)
  - keys(o) – Retorna um array com os nomes das propriedades presentes em o