JavaScript

Introdução e Primeiros Conceitos

Conteúdo Previsto

1. Introdução
2. Arrays
3. Funções Gerais
4. Classes e Objetos

INTRODUÇÃO

Introdução

- JavaScript é uma linguagem de scripts com os nomes baseados em Java, mas não é Java!
- JavaScript é um dos “dialetos” de ECMAScript
- JScript não é JavaScript, ele foi originalmente criado pela Microsoft, baseado no ECMAScript

Introdução

- JavaScript é uma linguagem com amarração dinâmica de tipos e de métodos, diferente tanto de C/C#/C++, como de Java

Introdução

- Imagine esse exemplo em Java:

```
int x;
x = 0; Compila: 0 é um int
x = “Texto”; Não compila: “Texto” não é um int
```

- Isso ocorre pois Java (assim como C/C#/C++) tem amarração estática de tipos, ou seja, o tipo de uma variável é conhecido durante a compilação

Introdução

- Já em JavaScript, o exemplo:

```
var x;
x = 0; Funciona
x = “Texto”; Também funciona!
```

- Isso ocorre pois JavaScript possui amarração dinâmica de tipos, ou seja, o tipo de uma variável é conhecido apenas durante a execução do programa

Introdução

- Imagine esse exemplo em Java/C#:

```
String s;
s.m(); Não compila: m() não é um método de String
```

- Isso ocorre pois Java (assim como C#/C++) tem amarração estática de métodos, ou seja, os métodos de uma classe devem ser conhecidos durante a compilação

Introdução

- Em JavaScript isso passa a ser válido:

```
var x;
x = 0;
x.blah();
```

- Só será verificado se o método `blah` existe para o objeto durante a execução (amarração dinâmica)
- Caso o método realmente não exista para o objeto, um erro ocorrerá na linha da chamada ao método

Introdução

- O JavaScript ainda é capaz de executar um código arbitrário partindo de uma string, através da função `eval`:

```
var a = “var x = 4 + 4 + 1;”;
eval(a);
var b = x + 7;
```

- Aqui `b` valerá `16` (`4 + 4 + 1 + 7`), onde `9` (`4 + 4 + 1`) é o valor proveniente de `x`

Introdução

- JavaScript é uma linguagem que trata funções como objetos de primeira classe, e suas funções são funções de alto nível, diferente tanto de C/C#/C++, como de Java
- Tratar funções como objetos de primeira classe significa que as funções são tratadas como um tipo de dados como um valor inteiro, uma string...

Introdução

- Funções de alto nível são funções capazes de trabalhar com outras funções, como por exemplo, receber ou criar funções
- O fato de que uma função pode "receber" outra função NÃO deve ser confundido com a passagem de ponteiros para funções do C/C++, ou delegates do C#! Aqui não se passa ponteiros, mas A própria função

Introdução

- Exemplo de função como objeto de primeira classe:

```
var x;
x = 0;
x = function(a, b) { return a + b; };
```

- No segundo caso, x NÃO está recebendo o resultado da função (pois ela não está sendo executada), ele está recebendo a própria função

Introdução

- Continuando:

```
var x, y;
x = function(a, b) { return a + b; };
y = x(4, 5);
```

- A função só é executada quando ela for chamada, como é o caso da 3ª linha, onde `y` receberá `9`

Introdução

- Exemplo de função de alto nível:

```
function blah(a, b) {
    return a(b + 1);
}

var y = blah(function(x) { return 2 * x; }, 4);
```

- Aqui, `y` receberá `10`, que é proveniente do `2 * x`, onde `x = b + 1` e `b = 4`

Introdução

- Consequências de se poder criar funções em qualquer lugar: Quanto irão valer `a` e `b`?

```
function blah(x) {
    return function () {
                            return x;
                        }
}

var y = blah(4);
var a = y();
var b = y();
```

Introdução

- No exemplo anterior, `a` e `b` valem `4`, pois esse é o valor retornado pela função criada dentro de `blah`.
- Outro exemplo:

```
function blah(x) {
    return function () {
                          x = x + 1;
                          return x;
                        }
}

var y = blah(4);
var a = y();
var b = y();
```

Introdução

- Nesse último exemplo, `a` irá valer `5` e `b` irá valer `6`.
- Mais um exemplo:

```
function blah(x) {
    return function () {
                          x = x + 1;
                          return x;
                        }
}

var y = blah(4);
var z = blah(4);
var a = y();
var b = z();
```

Introdução

- Agora, tanto `a` como `b` irão valer `5`.
- Isso ocorre devido ao que é conhecido como fechamento de funções (function closure)
- No momento da criação de uma função, ela copia o conteúdo das variáveis locais e essa cópia passa a “fazer parte” da função, que a utiliza sempre que precisar

Introdução

- Formalmente, essas funções que foram criadas sem nome algum, como as vistas anteriormente, são formalmente conhecidas como funções lambda (lambda function)
- A linguagem C# simula esse comportamento de funções lambda através de funções anônimas nos delegates (fica para outro dia...)

Introdução

- Mais um exemplo:

```
function blah(x) {
    return 2 * x;
}

var a = blah(4);
var b = blah(4, 5);
```

- Neste exemplo tudo funciona corretamente, sendo que `a` e `b` valerão `8`, pois em JavaScript é possível passar uma quantidade de parâmetros diferente da que foi explicitada

Introdução

- É possível passar nenhum parâmetro para uma função declarada com dois, ou ainda passar três parâmetros para uma função que foi declarada com nenhum!
- Para acessar os parâmetros reais utiliza-se uma variável especial, que é um array chamado de `arguments`

Introdução

- Por exemplo:

```
function blah() {
    return arguments[1] + arguments.length;
}

var a = blah(4, 8);
var b = blah(4, 8, 16);
```

- Aqui `a` valerá `10` (`8 + 2`), e `b`, `11` (`8 + 3`)

ARRAYS

Arrays

- A construção de arrays em JavaScript pode ocorrer de diversas formas:

```
var a = new Array();
a[0] = 10;
a[1] = 20;
a[2] = 30;
a[3] = 40;
var x = a[5];
```

- Apesar do tamanho não estar explícito, ele é ajustado conforme o necessário, assim a atribuição `x = a[5]` não dá erro, pois `a[5]` irá apenas retornar um valor não definido (`undefined`)

Arrays

- Um fato importante (não relacionado apenas a arrays): `undefined` é diferente de `null`
- Um valor `null` existe, porém não contém nada

Arrays

- Além disso, a construção abaixo também funciona:

```
var a = new Array();
a[0] = 10;
a[1] = 20;
a[5] = 60;
```

- O que ocorre é que para todos os índices intermediários (2, 3 e 4), será atribuído o valor `undefined`

Arrays

- É possível também criar um array sem utilizar o `new Array()`, mas já preenchendo seus valores:

```
var a = [10, 20, 30, 40, 50];
```

- Além disso, arrays em JavaScript são heterogêneos, diferentes dos arrays de C/C++ e Java, que são homogêneos, tornando isso possível:

```
var a = [10, “Texto”, new Pessoa()];
```

Arrays

- A classe Array possui alguns métodos e propriedades para auxiliar em seu uso
- Alguns dos mais importantes são:
- `length`
  - Retorna o tamanho do array
- `concat()`
  - Concatena dois ou mais arrays, ou seja é possível coisas como:

```
a.concat(b, c, d);
```

Arrays

- `join()`
  - Concatena todos os elementos, utilizando um separador passado:

```
var a = [1, 2, “rafael”];
a.join(); retorna “1,2,rafael”
a.join(“ e ”); retorna “1 e 2 e rafael”
a.join(“”); retorna “12rafael”
```

Arrays

- `sort(func)`
  - Ordena todos os elementos, utilizando uma função de ordenação passada:

```
var a = [“b”, “a”, “d”, “c”];
a.sort(); a passa a conter [“a”, “b”, “c”, “d”]

function ordenador(x, y) {
    compara x e y, retornando:
    0 se forem iguais
    valor negativo se x vem antes de y
    valor positivo se x vem depois de y
    *Esse é o mesmo comportamento dos sorters
    do C/C++ e do Java
}
a.sort(ordenador);
```

Arrays

- Para mais informações sobre arrays, ver: www.w3schools.com/jsref/jsref_obj_array.asp

FUNÇÕES GERAIS

Funções Gerais

- Diferente do Java, onde tudo deve ser acessado através de classes e objetos, o JavaScript possui algumas funções gerais, que podem ser chamadas a qualquer momento
- Algumas das mais importantes são:

Funções Gerais

- `encodeURI(str)`
  - Codifica a string passada, substituindo espaços e caracteres especiais (exceto , / ? : @ & = + $ #) na representação de bytes UTF-8:

    ```
    var a = encodeURI(“a/e i/õ/u”);
    a passa a valer “a/e%20i/%C3%B5/u”
    ```
- `decodeURI(str)`
  - Realiza o processo inverso

Funções Gerais

- `encodeURIComponent(str)`
  - Codifica a string passada, substituindo espaços e caracteres especiais (incluindo , / ? : @ & = + $ #) na representação de bytes UTF-8:

    ```
    var a = encodeURI(“a/e i/õ/u”);
    a passa a valer “a%2Fe%20i%2F%C3%B5%2Fu”
    ```

- `decodeURIComponent(str)`
  - Realiza o processo inverso

Funções Gerais

- `parseInt(str)`
  - Converte a string passada em um número inteiro
- `parseFloat(str)`
  - Converte a string passada em um número, considerando casas decimais (o separador decimal é o .)

Funções Gerais

- Para mais informações sobre funções em geral ver:
  www.w3schools.com/jsref/jsref_obj_global.asp
- Para funções matemáticas, ver:
  www.w3schools.com/jsref/jsref_obj_math.asp
- Para funções com números, ver:
  www.w3schools.com/jsref/jsref_obj_number.asp
- Para funções com datas, ver:
  www.w3schools.com/jsref/jsref_obj_date.asp
- Para funções com strings, ver:
  www.w3schools.com/jsref/jsref_obj_string.asp

CLASSES E OBJETOS

Classes e Objetos

- A criação de classes e objetos se dá de maneira bem diferente de Java e C++, visto que o construtor pode ser uma função qualquer, como:

```
function Pessoa(n, e) {
    this.nome = n;
    this.email = e;
    this.mostraNome = function() {
        alert(this.nome); };
}
var p = new Pessoa(“rafael”, “btec”);
p.mostraNome();
```

Classes e Objetos

- Assim como Java, todo objeto possui o método toString, e para sobrescrevê-lo (override) o processo é similar:

```
function Pessoa(n, e) {
    this.nome = n;
    this.email = e;
    this.mostraNome = function() {
        alert(this.nome); };
    this.toString = function() {
        return this.nome; };
}
```

Classes e Objetos

- É possível obter o construtor de um objeto através da propriedade `constructor`, a qual todos os objetos possuem (inclusive números e strings!):

```
function Pessoa(n, e) { ... }

var p = new Pessoa(“rafael”, “btec”);

var c = p.constructor;
var p2 = new c(“rafael 2”, “btec 2”);
```

Classes e Objetos

- Como as amarrações são dinâmicas, é possível tentar obter uma propriedade inexistente de um objeto, sem que isso seja um erro (retornando-se `undefined`):

```
function Pessoa(n, e) { ... }

var p = new Pessoa(“rafael”, “btec”);

var a = p.telefone; O valor de a é undefined
```

Classes e Objetos

- Contudo, ocorrerá um erro durante a execução, caso se tente executar um método inexistente de um objeto (equivaleria a tentar executar o valor `undefined`, como se ele fosse uma função):

```
function Pessoa(n, e) { ... }

var p = new Pessoa(“rafael”, “btec”);

p.voar(); Erro na execução!
```

Classes e Objetos

- Diferente das outras linguagens, um objeto pode "ganhar" métodos e propriedades em tempo de execução:

```
function Pessoa(n, e) { ... }

var p = new Pessoa("rafael", "btec");

var s = p.salario; s será undefined

p.salario = 450;

s = p.salario; s valerá 450
```

Classes e Objetos

- Ao se fazer isso, apenas aquele objeto "ganha" o método ou propriedade, outros objetos da classe não:

```
function Pessoa(n, e) { ... }

var p = new Pessoa("rafael", "btec");
var p2 = new Pessoa("rafael 2", "btec 2");

p.salario = 450;

var s = p.salario; s valerá 450
s = p2.salario; s será undefined
```

Classes e Objetos

- Para se acrescentar um método o processo é similar:

```
function Pessoa(n, e) { ... }

var p = new Pessoa(“rafael”, “btec”);

p.falar(); Ocorrerá um erro: falar não existe

p.falar = function() { ... };

p.falar(); Agora tudo funciona
```

Classes e Objetos

- É possível acessar uma propriedade ou método como em um array:

```
function Pessoa(n, e) { ... }

var p = new Pessoa(“rafael”, “btec”);

var a = p[“nome”]; a valerá “rafael”

p.falar = function() { ... };

p[“falar”](); Executa o método falar
```