TAWS_

Tecnologias para a Web Social

Licenciatura em Design e Multimédia
2º semestre 2011
Pedro Miguel Cruz

Javascript para programar

JAVASCRIPT

As implementações mais populares do Javascript são feitas por navegadores web. O Javascript aparece frequentemente misturado com fracas implementações do DOM e APIs confusas do navegador.

O Javascript é uma linguagem de programação que pode ser usada fora dos navegadores web (KDE, Rhino, SpiderMonkey, V8, WebKit).

É fácil implementar trabalho sem se conhecer muito da linguagem ou de programação – mas o melhor, é quando sabemos o que fazermos.

BOAS IDEIAS

Funções
As funções Javascript são objetos com o seu próprio espaço de nomes.

Tipificação fraca
Em Javascript os tipos de dados não são declarados, evitando-se a descrição de hierarquias complexas ou conversões de tipo. Por outro lado os compiladores de Javascript não são capazes de detetar erros de tipificação.

Objetos literais
Os objetos podem ser criados descrevendo os seus componentes através de uma notação simples e poderosa (JSON).

Herança baseada em protótipos
Não há classes! Apenas existem objetos que reutilizam componentes de outros objetos. Os paradigmas de programação clássicos são de aplicação pouco natural.

BOAS IDEIAS

Java	Javascript
Tipos fortes	Tipos fracos
Estática	Dinâmica
Clássica	Protótipos
Classes	Funções
Construtores	Funções
Métodos	Funções

MÁS IDEIAS

Variáveis globais

O Javascript não tem editores de ligações (responsável por definir os escopos dos vários objetos gerados pelo compilador). Desta forma a programação em Javascript é inerentemente dependente de variáveis globais que podem ser alteradas por qualquer função em qualquer altura da execução. Apesar de tal ser defensável para pequenos programas, frequentemente torna-se difícil gerir a fiabilidade de grandes programas.

Se os mesmos subprogramas possuírem variáveis globais com os mesmos nomes, os seus comportamentos poderão interferir um com o outro.

MÁS IDEIAS

Variáveis globais
Todas as variáveis globais são guardados num único objeto global. Nos navegadores o objecto global é a janela `window`.

Várias formas de declarar variáveis globais (use a primeira)
Declarar variáveis fora de funções, adicionar uma propriedade ao objeto global, usar uma variável sem a declarar (global implícita)

```
var foo = value;
window.foo = value;
foo = value;
```

MÁS IDEIAS

Escopos de bloco
A sintaxe do Javascript é adaptado do C. Em linguagens como o C ou Java é possível definir escopos de bloco, para além dos implícitos em métodos e funções. Contudo, tal não é possível em Javascript.

```
function azores(){
    { var cow = 1;}
    writeln(cow);
}
```

MÁS IDEIAS

Pontos e vírgulas
A inserção de pontos e vírgulas após cada declaração não é obrigatória. Contudo o Javascript possui um mecanismo de formatação de código que pode inserir pontos e vírgulas em locais erróneos e adulterar o comportamento do programa.

Deve-se escrever o ponto e vírgula depois de cada declaração.

MÁS IDEIAS

Operador typeof
Este operador devolve o tipo do operando.

```
/* Testando operandos */
typeof 98.6 //devolve 'number'

/* Testando se um operando é nulo */
value === null //pois typeof null devolve 'object’

/* Testando se um valor é um objeto */
if (value && typeof value === 'object') {
    //o valor é um objeto!
}
```

MÁS IDEIAS

Operador +
Pode ser uma adição ou uma concatenação, dependendo do tipo dos parâmetros.

```
9 + a; //devolve '9a'
9 + 9; //devolve 18
```

MÁS IDEIAS

Ponto flutuante
A norma adotada para operações em ponto flutuante pelo Javascript trás frequentemente imprecisões.

```
var res = 0.1 + 0.2; // res = 0.30000000000000004
```

```
/* solução 1: multiplicar por 10 ou 100 */
var res = 0.1 * 10 + 0.2 * 10; // res = 3;
res/=10;                       // res = 0.3;

/* solução 2: quando não é possível identificar
   a operação que deu origem ao erro    */

var res = 0.1 + 0.2;           // res = 0.30000000000000004
res = res.toFixed(1);          // res = '0.3'
var med = 1 + res;             // res = '10.3' oops
var med = parseFloat(res) + 1; // res = 1.3
```

MÁS IDEIAS

NaN
Valor ‘Not a Number’.

```
typeof NaN === 'number'     // true
parseInt('zo')              // NaN
NaN === NaN                 // false
NaN !== NaN                 // true

isNaN(NaN)                  // true
isNaN(0)                    // false
isNaN('oops')               // true
isNaN('0')                  // false
```

MÁS IDEIAS

Infinity
Valor para ‘infinito’.

```
1/0                     // Infinity
Infinity - Infinity     // NaN
Infinity - 1            // Infinity

isFinite('123');        // true
isFinite(NaN);          // false

function isNumber(value) {
   return typeof value === 'number' && isFinite(value);
}
```

VOLTANDO ÀS BOAS IDEIAS

Objetos
Funções
Herança
Arrays

OBJETOS

Javascript possui tipos simples como ‘number’, ‘string’, ‘boolean’, ‘null’ e ‘undefined’.

O objetos em Javascript são coleções dinâmicas de chaves e valores.

Arrays são objetos, funções são objetos (apesar de serem objetos o seu tipo é ‘function’) e objetos são objetos.

Não existem classes para objetos. Os objetos são úteis para armazenar e organizar dados, podendo conter outros objetos e representar árvores ou grafos naturalmente.

OBJETOS

Objetos literais
São uma notação conveniente para criar novos objetos, denotada de grande expressividade e legibilidade.

```
var objeto_vazio = {};

var nome = {
    "primeiro-nome": "AqueleCujoNome",
    "sobrenome": "NãoPodeSerPronunciado"
};

var voo = {
    companhia: "TAP Air Portugal",
    numero: 'TP0115',
    partida: {
        codigo: "LIS",
        data: "2011-02-22 14:55",
        cidade: "Lisboa"
    },
    chegada: {
        codigo: "SAN",
        data: "2011-02-29 10:42",
        city: "San Diego"
    }
};
```

OBJETOS

Aceder a valores

```
nome["primeiro-nome"]              // "AqueleCujoNome"
voo.partida.codigo                 // "LIS"

nome["segundo-nome"]               // undefined
voo.estado                         // undefined
nome["PRIMEIRO-NOME"]              // undefined

var seg_nome = nome["segundo-nonme"] || "(vazio)";
var estado = voo.estado || "desconhecido";

voo.nave                           // undefined
voo.nave.modelo                    // throw "TypeError"
voo.nave && voo.nave.modelo        // undefined
```

OBJETOS

Modificar

```
nome['primeiro-nome'] = 'Tom';
```

Adicionar propriedades

```
nome['segundo-name'] = 'Marvolo';
nome.cognome = 'Voldemort';
voo.nave = {
    modelo: 'Boeing 777'
};
voo.estado = 'atrasado';
```

OBJETOS

Referenciar
Os objetos são passados por referência, nunca por cópia.

```
var x = nome;
x.cognome = 'Tom';
var alcunha = nome.cognome;
/* alcunha é 'Tom' pois x e nome
   são referências para o mesmo objeto */
```

```
var a = {}, b = {}, c = {};
/* a, b, e c referem-se cada um
   a um objeto vazio diferente */

a = b = c = {};
/* a, b, e c referem-se todos
   ao mesmo objeto vazio */
```

OBJETOS

Protótipos

Cada objeto está ligado a um objeto protótipo do qual pode herdar todas as suas propriedades. Todos os objetos literais estão ligados por omissão ao `Object.prototype` do Javascript.

```
if (typeof Object.partejar !== 'function') {
     Object.partejar = function (o) {
         var F = function () {};
         F.prototype = o;
         return new F();
     };
}

var outro_nome = Object.partejar(nome); // (new F()).prototype = nome;
```

Quando se fazem alterações a um objeto, o seu protótipo não é alterado.

```
outro_nome['primeiro-nome'] = 'Harry';
outro_nome['segundo-nome'] = 'Potter';
outro_nome.cognome = 'TheOneWhoLives';
```

OBJETOS

Protótipos e delegação
A beleza dos protótipos está na sua natureza dinâmica. Quando se adiciona uma propriedade a um protótipo, essa propriedade será imediatamente visível em todos os objetos baseados nesse protótipo.

A ligação com os protótipos apenas é usada quando se tenta aceder a uma propriedade de um objeto. Se essa propriedade não existe no objeto, é procurada no seu protótipo, e se não existe no protótipo, é procurada no protótipo do protótipo e assim sucessivamente. Se a propriedade não existe em toda a cadeia de protótipos, `undefined` é devolvido. A este processo chama-se *delegação*.

```
nome.profissao = "ator";
outro_nome.profissao // 'ator'
```

OBJETOS

Reflexão
A reflexão no contexto de ciência da computação consiste no processo de enumerar e modificar o comportamento e a estrutura de uma aplicação em tempo de execução.

Reflexão – adicionar propriedades
Já vimos como se adicionam propriedades em exemplos anteriores. Contudo, pode fazer sentido descobrir se determinadas propriedades já existem. Note-se que quando se avalia uma propriedade também se avaliam as propriedades do protótipo do objeto, pelo que é preciso tratar todos os tipos possíveis de retorno.

```
typeof voo.numero      // 'number'
typeof voo.estado      // 'string'
typeof voo.chegada     // 'object'
typeof voo.descricao   // 'undefined'
typeof voo.toString    // 'function'
typeof voo.constructor // 'function'
```

Várias vezes faz sentido evitarmos inspecionar a cadeia de protótipos.

```
voo.hasOwnProperty('numero')            // true
voo.hasOwnProperty('constructor')       // false
nome.hasOwnProperty('profissao');       // true
outro_nome.hasOwnProperty('profissao'); // false;
```

OBJETOS

Reflexão – enumerar

O ciclo `for in` permite percorrer todas as propriedades de um objeto, incluindo as dos seus protótipos. Se apenas estivermos interessados em funções, podem-se considerar os seguintes métodos disponíveis para qualquer objeto.

```
Object.prototype.getAllMethods = function(){
   var methodsArray = new Array();
   for (var method in this) {
         if (typeof this[method] == 'function') {
             methodsArray.push(method);
         };
   };
   return methodsArray;
};
```

```
Object.prototype.getOwnMethods = function(){
   var methodsArray = new Array();
   for (var method in this) {
         if (typeof this[method] == 'function' && this.hasOwnProperty(method))
         {
             methodsArray.push(method);
         };
   };
   return methodsArray;
};
```

OBJETOS

Reflexão – enumerar

```
outro_nome.qualquerNome = function(){
   //operador ternário
   return Math.random() < 0.5 ? this['primeiro-nome'] : this['sobrenome'];
};

outro_nome.getAllMethods(); // ["qualquerNome", "getAllMethods",
                                "getOwnMethods"]
outro_nome.getOwnMethods()  // ["qualquerNome"]
```

Reflexão – eliminar

É possível eliminar uma propriedade de um objeto usando o operador `delete`. O operador afeta apenas o objeto referenciado e não os seus protótipos, podendo revelar propriedades dos mesmos que possam ter sido ocluídas no objeto atual.

```
outro_nome["primeiro-nome"] = "Harry";
delete outro_nome["primeiro-nome"];
outro_nome["primeiro-nome"]              // 'AqueleCujoNome'
```

OBJETOS

Moderar as variáveis globais
Evitar alterações ilegítimas por parte de outras aplicações usando uma variável para conter a aplicação.

```
var MYAPP = {};

MYAPP.nome = {
    "first-name": "Harry",
    "last-name": "Potter"
};

MYAPP.voo = {
    companhia: "TAP Air Portugal",
    numero: 'TP0115',
    partida: {
        codigo: "LIS",
        data: "2011-02-22 14:55",
        cidade: "Lisboa"
    },
    chegada: {
        codigo: "SAN",
        data: "2011-02-29 10:42",
        city: "San Diego"
    }
};
```

FUNÇÕES

Os objetos literais estão inerentemente ligados a `Object.prototype`. As funções são objetos inerentemente ligados a `Function.prototype` (que por sua vez está também ligado a `Object.prototype`).

Cada função tem ainda duas propriedades escondidas: o contexto da função e o código que implementa o seu comportamento.

Como as funções são objetos podem ser usadas como qualquer outro parâmetro, podendo ser guardadas em variáveis, objetos ou arrays. As funções podem ser passadas como parâmetros ou serem devolvidas a partir de uma outra função.

Finalmente, como as funções são objetos, elas próprias podem ter métodos.

FUNÇÕES

Funções literais

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

Uma função literal pode aparecer em qualquer lugar de uma expressão. Podem portanto ser definidas dentro de outras funções. Uma função interna/privada tem acesso aos parâmetros e variáveis da função em que está aninhada. Quando a função exterior é executada é criado o ***fecho*** da função interna que consiste no seu código e nas referências ao escopo externo.

FUNÇÕES

Invocação
Em adição às variáveis declaradas, cada função recebe mais dois parâmetros: os argumentos e o `this`. O valor do `this` é determinado pelo padrão de invocação.

Invocação por métodos

```
var meuObjeto = {
    valor: 0;
    incremento: function (inc) {
        this.valor += typeof inc === 'number' ? inc : 1;
    }
};

meuObjeto.incremento();   // 1
meuObjeto.incremento(2);  // 3
```

FUNÇÕES

Invocação por função

```
var soma = adicao(3, 4);  // soma é 7
```

Para estender objetos, note-se que aninhar várias funções requer guardar o `this`.

```
meuObjeto.duplicar = function(){
   this.valor = adicao(this.valor, this.valor);//6
};
meuObjeto.duplicar();

meuObjeto.duplicar = function(){
   var that = this;
   var dup = function(){
         that.valor = adicao(that.valor, that.valor);
   };
   return dup();
};

meuObjeto.duplicar();                              // 6
```

FUNÇÕES

Invocação por construtor
Quando uma função é invocada com o prefixo new, um novo objeto é criado em que o construtor é a função invocada (o protótipo do novo objeto possui uma ligação para este construtor).

```
// Um objeto com a propriedade status
var Quo = function (string) {
    this.status = string;
};

/* Atribuir a todas as estancias de Quo
   um método get_status */
Quo.prototype.get_status = function () {
    return this.status;
};

// Instanciar Quo.

var meuQuo = new Quo("confuso");
meuQuo.get_status();  // ‘confuso’
```

FUNÇÕES

Invocação por aplicação

O método `apply` permite-nos construir um array de argumentos para invocar uma função, assim como escolher o valor para `this`.

```
var array = [3, 4];
var sum = adicao.apply(null, array);    // soma é 7
```

FUNÇÕES

Argumentos
O parâmetro `arguments` disponibiliza um array com todos os argumentos.

```
var somatorio = function () {
    var i, somatorio = 0;
    for (i = 0; i < arguments.length; i += 1) {
        somatorio += arguments[i];
    }
    return somatorio;
};
```

FUNÇÕES

Cascata
Se os métodos alterarem um objeto e o devolverem, é possível programar em cascata.

```
getElement('myBoxDiv').
    move(350, 150).
    width(100).
    height(100).
    color('red').
    border('10px outset').
    padding('4px').
    appendText("Please stand by").
    on('mousedown', function (m) {
        this.startDrag(m, this.getNinth(m));
    }).
    on('mousemove', 'drag').
    on('mouseup', 'stopDrag').
    later(2000, function (  ) {
        this.
            color('yellow').
            setHTML("What hath God wraught?").
            slide(400, 40, 200, 200);
    }).
    tip('This box is resizeable');
```

FUNÇÕES

Notação abreviada no Javascript 1.8

```
function(d) Math.sqrt(d);
function(d) { return Math.sqrt(d); }
```

HERANÇA

Pseudo clássica

```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
```

```
Function.method('inherits', function (Parent) {
    this.prototype = new Parent();
    return this;
});
```

HERANÇA

Pseudo clássica

```
var Mammal = function () {
    this.say = "r-r-r-r";
};

var Cat = function (name) {
    this.name = name;
}.
    inherits(Mammal).
    method('says', function () {
        return this.say;
    }).
    method('get_name', function () {
        return this.name;
    });

var cat = new Cat("bixo");
cat.says(); // "r-r-r-r";
```

HERANÇA

Baseada em protótipos
Usando o `Object.partejar()`, especificando a *herança por diferenciação*.

```
var myMammal = {
    name : 'Herb the Mammal',
    get_name : function (  ) {
        return this.name;
    },
    says : function (  ) {
        return this.saying || '';
    }
};
```

```
var myCat = Object.partejar(myMammal);
myCat.name = 'Henrietta';
myCat.saying = 'meow';
myCat.purr = function (n) {
    return '-'+s;
};
myCat.get_name = function (  ) {
    return this.says + ' ' + this.name + ' ' + this.says;
};
```

ARRAYS
STRINGS
NUMBERS

```
    array.join(separator)
array.pop( )
array.push(item...)
array.reverse( )
array.shift( )
array.slice(start, end )
array.sort(comparefn )
array.splice(start, delCount, items)
array.unshift(item...)
function.apply(thisArg, argArray )
number.toExponential(fractionDigits )
number.toFixed(fractionDigits )
number.toPrecision(precision )
number.toString(radix )
object.hasOwnProperty(name )
regexp.exec(string )
regexp.test(string )
```

```
string.charAt(pos )
string.charCodeAt(pos )
string.concat(string...)
string.indexOf(searchString, position )
string.lastIndexOf(searchString,
position )
string.localeCompare(that )
string.match(regexp )
string.replace(searchValue,
replaceValue )
string.search(regexp )
string.slice(start, end )
string.split(separator, limit )
string.substring(start, end )
string.toLocaleLowerCase( )
string.toLocaleUpperCase( )
string.toLowerCase( )
string.toUpperCase( )
string.fromCharCode(char...)
```

JSON

JavaScript Object Notation (JSON)
Apesar de baseada na notação de objetos literais do Javascript, é independente da linguagem.

```
[
    {
        "first": "Jerome",
        "middle": "Lester",
        "last": "Howard",
        "nick-name": "Curly",
        "born": 1903,
        "died": 1952,
        "quote": "nyuk-nyuk-nyuk!"
    },
    {
        "first": "Louis",
        "last": "Feinberg",
        "nick-name": "Larry",
        "born": 1902,
        "died": 1975,
        "quote": "I'm sorry. Moe, it was an accident!"
    }
]
```

Usar o `$.parseJSON(json)` do jQuery.

REFERÊN_ CIAS

Douglas Crockford, Javascript: The Good Parts