PROGRAMAÇÃO I

JavaScript:

Programação Orientada a Objetos

(Conceitos avançados)

Mário Viana

FUNÇÕES

JavaScript

Mário Viana

FUNÇÕES ANÓNIMAS

- São funções que não possuem nome e que podem ser utilizadas nos seguintes casos:
  - Atribuindo a função anónima a uma variável;

```
var f = function(a){
   return a;
};
```

  - Passando a função anónima como parâmetro de outra função (*callback function*);
  - Definindo a função anónima e executá-la imediatamente (*immediate function*);

CALLBACK FUNCTIONS

- Como as funções podem ser atribuída as variáveis, também podem ser passadas como argumentos de outras funções.
- Exemplo:

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

function um(){
  return 1;
}

function dois(){
  return 2;
}

> invocaSoma(um, dois);   // devolve 3
```

CALLBACK FUNCTIONS

- Exemplo passando funções anónima:

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

> invocaSoma(
      function () {
         return 5;
      },
      function () {
         return 3;
      }
);    // devolve 8
```

CALLBACK FUNCTIONS

- Exemplo típico:

```
function multiplicaPorDois(a, b, c, callback){
  var i, ar = [];
  for(i = 0; i < 3, i++)}
     ar[i] = callback(arguments[i] * 2);
  }
  return ar;
}

function somaUm(a){
  return a + 1;

> myarr = multiplicaPorDois(1,2,3,somaUm);
  [3,5,7]
> multiplicaPorDois(1,2,3,function (a){
       return a + 2;});
  [4,6,8]
```

IMMEDIATE FUNCTIONS

- Consiste na chamada imediata de uma função anónima:

```
(function (nome) {
    alert("Olá " + nome + "!");
  }
)("Rui");
```

ou

```
(function (nome) {
    alert("Olá " + nome + "!");
 }("Rui"));
```

INNER (PRIVATE) FUNCTIONS

- Consiste em colocar funções dentro de funções, tornando as funções internas privadas.
- Exemplo:

```
var exterior = function(param){
  var interior = function (a){
    return a * 2;
  };
  return "O resultado é " + interior(param);
};

> exterior(2);
  "O Resultado é 4"
> exterior(8);
  "O Resultado é 16"
> interior(2);
  ReferenceError: interior is not defined
```

FUNÇÕES QUE RETORNAM FUNÇÕES

- Podendo as funções serem atribuídas a variáveis, é possível também devolver uma função.
- Exemplo:

```
function a(){
    alert("A!");
    return function (){
        alert("B!");
    };
}

> var newFunc = a();  // mostra A!
> newFunc(); // mostra B!
> a()();     // mostra A! e B!
```

FUNÇÕES QUE SE AUTOREESCREVEM

- Consiste em substituir uma função antiga por uma nova definida internamente.

- Exemplo:

```
function a(){
   alert("A!");
   return function (){
      alert("B!");
   };
}

> var a = a();// mostra A!
> a();    // mostra B!
```

- Exemplo:

```
function a(){
   alert("A!");
   a = function (){
      alert("B!");
   };
}

> a(); // mostra A!
> a(); // mostra B!
```

CLOSURES

- Esta estratégia tem a ver com o facto de uma função externa poder ter um tempo de vida menor que a função interna;
- Exemplo:

```
var incrementador = function (){
    var interna = 0;
    return function(){
      interna++;
      return interna;
    }
}();
var a = incrementador(); // 1
var b = incrementador(); // 2
```

CLOSURES

- O que se passou efetivamente?
- Em JavaScript todas as funções internas que sobrevivem à função onde estão definidas mantêm a referência ao contexto da função externa;
- Este comportamento leva a que variáveis locais da função "pai" não sejam destruídas no final da execução, permitindo que as funções internas "sobreviventes" acedam a essa informação;
- A este comportamento designa-se ***closure***.

CLOSURES – IMPLEMENTAÇÕES DIVERSAS

```
var f = function a(){
  var b = 0;
  var n = function(){
    b++;
    return b;
  };
  return n;
}

> var interna = f();
> interna(); // 1
> interna(); // 2
```

```
var interna;
var f = function(){
  var b = 0;
  var n = function(){
    b++;
    return b;
  }
  interna = n;
};
> f();
> interna(); // 1
> interna(); // 2
```

CLOSURES – IMPLEMENTAÇÕES DIVERSAS

```
function f (param){
  var n = function(){
    param++;
    return param;
  };

  return n;
}

> var interna = f(123);
> interna(); // 124
> interna(); // 125
```

CLOSURES NAS ESTRUTURAS ITERATIVAS

- Consideremos as seguinte implementação de *closure*:

```
function f (){
  var arr = [], i;
  for(i = 0; i < 3; i++){
    arr[i] = function(){
       return i;
    };
  }
  return arr;
}
> var arr = f();
> arr[0]();  // 3
> arr[1]();  // 3
> arr[2]();  // 3
```

- O expectável era devolver 1, 2 e 3. O que se passou?

CLOSURES NAS ESTRUTURAS ITERATIVAS

- NB: As funções não “lembram” valores, mas apenas deixam a referência ao ambiente onde foram criadas.
- No caso apresentado, a execução do *for* deixa a variável i no valor 3. Assim, todas as funções passam a apontar para o contexto da variável i que é 3.

CLOSURES NAS ESTRUTURAS ITERATIVAS

- Implementação correta para da closure apresentada:

- Solução 1:

```
function f (){
  var arr = [], i;
  for(i = 0; i < 3; i++){
    arr[i] = (function(x){
       return function(){
          return(x);
       };
    }(i));
  }
  return arr;
}

> var arr = f();
> arr[0]();  // 0
> arr[1]();  // 1
> arr[2]();  // 2
```

- Solução 2:

```
function f (){
  function n(x){
    return function(){
      return x;
    };
  }
  var arr = [], i;
  for(i=0; i < 3; i++){
    arr[i] = n(i);
  }
  return arr;
}
> var arr = f();
> arr[0]();  // 0
> arr[1]();  // 1
> arr[2]();  // 2
```

CLOSURES – GETTER/SETTER

- Este tipo de *closure* pretende proteger variáveis locais controlando a sua alteração.

- Exemplo:

```
var getValor, setValor;
(function (){
   var valor = 0;
   getValor = function (){
      return valor;
   };
   setValor = function (v){
      if(typeof v === “number”){
         valor = v;
      }
   };
}());
```

- Teste do código:

```
> getValor();
  0
> setValor(123);
> getValor();
  123
> setValor(true);
> getValor();
  123
```

CLOSURE - ITERADOR

- Permite implementar métodos com comportamento iterativo, podendo apresentar valores consecutivos de estruturas de dados.

- Exemplo:

```
function serie(x){
    var i = 0;
    return function (){
      return x[i++];
    };
}
```

- Teste código:

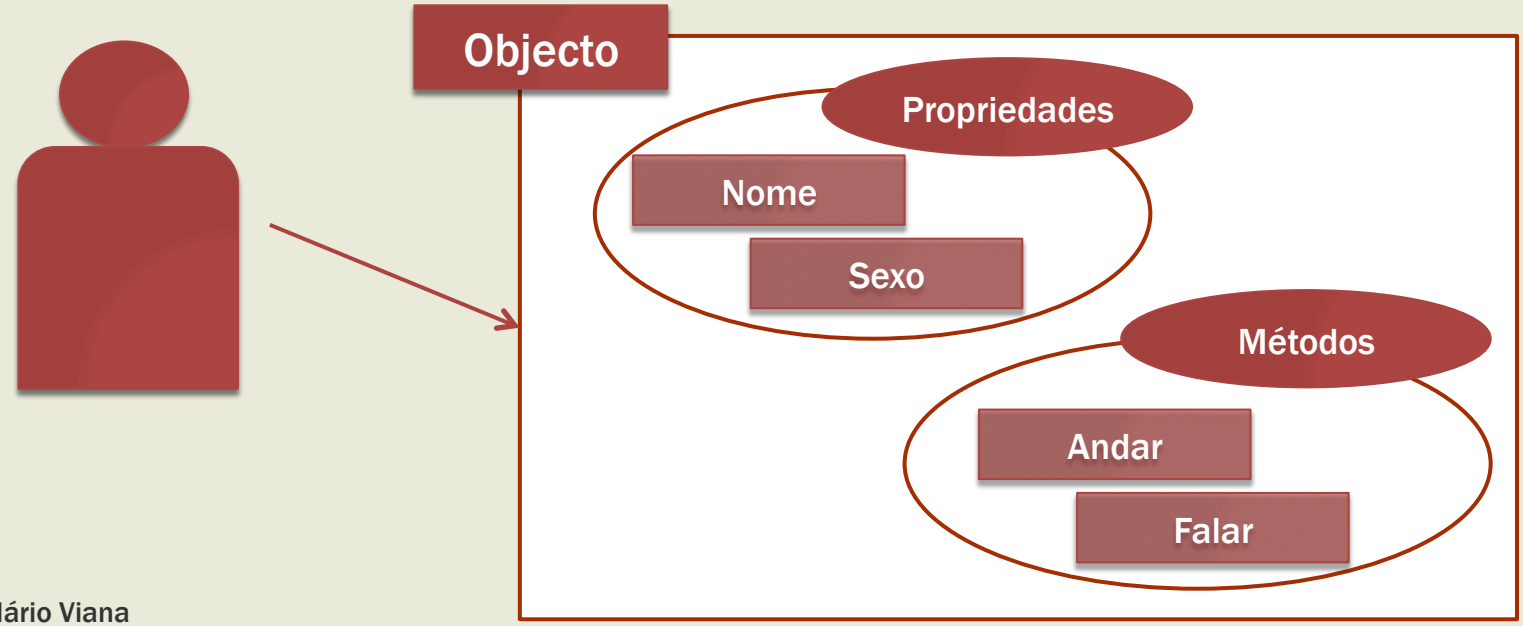
```
> var l=['a','b','c'];
> var next = serie(l);

> next();
  "a"
> next();
  "b"
> next();
  "c"
```

OBJETOS

JavaScript

FUNDAMENTOS DE OBJETOS

- Um objeto é uma entidade de software constituída por dados (propriedades) e funções (métodos);
- Permite, na programação, a modelização de situações do mundo real, através da manipulação das suas caraterísticas (propriedades) e funcionalidades (métodos).



MANIPULAÇÃO DE MÉTODOS

- Definir métodos:

```
var pessoa = new Object();
pessoa.nome = “Nicolau”;
pessoa.idade = 29;
pessoa.emprego = “Engenheiro de software”;

pessoa.apresentaPessoa = function(){
   alert(pessoa.nome + “, “ + pessoa.idade +
                  “ anos ,” + pessoa.emprego);
};
```

  - Notação literal

```
var pessoa = {
   nome: “Nicolau”,
   idade: 29,
   emprego: “Engenheiro de software”,

   apresentaPessoa: function(){
      alert(pessoa.nome + “, “ + pessoa.idade +
                  “ anos ,” + pessoa.emprego);
   }
};
```

MANIPULAÇÃO DE MÉTODOS

- Chamar métodos de objetos:

```
pessoa.apresentaPessoa();

pessoa["apresentaPessoa"]();

var metodo = pessoa.apresentaPessoa;
pessoa[metodo]();
```

- Apagar métodos de objetos:

```
delete pessoa.apresentaPessoa;

> pessoa.apresentaPessoa();
  "undefined"
```

O OPERADOR *this*

- No exemplo anterior, no método *apresentaPessoa()*, usamos *pessoa.nome* para aceder à propriedade *nome* do objeto pessoa.
- Existe um operador *this* que permite referenciar as propriedades do objetos.
- Objeto reescrito:

```
var pessoa = {
    nome: "Nicolau",
    idade: 29,
    emprego: "Engenheiro de software",

    apresentaPessoa: function(){
        alert(this.nome + ", " + this.idade +
                    " anos ," + this.emprego);
    }
};
```

- O operador *this* define o contexto: "este objeto" ou "o objeto atual"

FUNÇÕES CONSTRUTORAS

- Constitui um outro método de criar objetos, permitindo receber parâmetros para inicializar as propriedades dos objetos. Consideremos o exemplo:

```
function pessoa(nome, idade, emprego) {
   this.nome = nome,
   this.idade = idade,
   this.emprego = emprego,

   this.apresentaPessoa = function(){
      alert(this.nome + ", " + this.idade +
                  " anos ," + this.emprego);
   }
};
```

- Criar objetos a partir da função construtora:

```
var p1 = new pessoa("Carlos", 34, "Médico");
var p2 = new pessoa("Ana", 25, "Relações públicas");
p1.apresentaPessoa();
p2.apresentaPessoa();

var p3 = pessoa("José",23,"Jardineiro"); //O que acontece?)
```

FUNÇÕES CONSTRUTORAS

- Consideremos o exemplo:

```
function pessoa(nome, idade, emprego) {
    this.nome = nome,
    this.idade = idade,
    this.emprego = emprego,

    this.apresentaPessoa = function(){
        alert(this.nome + ", " + this.idade +
                    " anos ," + this.emprego);
    }
};

> var p3 = pessoa("José",23,"Jardineiro");
```

- A instrução sem o operador *new* cria um variável *p3* com a referência à função *pessoa*. Na verdade, as variáveis *nome*, *idade* e *emprego* são criada no âmbito global, assim como a função interna *apresentaPessoa()*.

```
> p3.nome; // Erro! "undefined"
> nome;    // "José"
> window.nome;   // "José"
```

A PROPRIEDADE *constructor*

- Quando um objecto é criado através de uma função construtora, a propriedade *constructor* é modificada de forma a conter a referência da função construtora usada na criação do objeto.
- É possível por isso:
  - Obter a referência à função construtora;

```
> p1.constructor;
  function pessoa()
```

  - e construir novos objetos a partir da propriedade *constructor*;

```
> var p5 = new p1.constructor("Marta",67,"Juíza");
> p5.nome;
  "Marta"
```

O OPERADOR *instanceof*

- O operador *instanceof* permite testar se um um objecto foi criado por uma uma função construtora específica.

```
> function pessoa() { }
> var p = new pessoa();
> var o = { };

> p instanceof pessoa;
  true
> p instanceof Object;
  true
> o instanceof Object;
  true
> o instanceof pessoa;
  false
```

ATRIBUIÇÃO DE OBJETOS

- Quando um **objeto** é **atribuído** a outra variável o que é passado é apenas a **referência do objeto original** → alterações efetuadas na variável afeta o objeto original.
- Exemplo:

```
> var original = {quantidade: 1};
> var copia = original;

> copia.quantidade;
  1
> copia.quantidade = 100;
  100
> original.quantidade;
  100
```

PASSAR OBJETOS A FUNÇÕES

- O mesmo acontece quando se passam objetos como parâmetros das funções.
- Exemplo:

```
> var original = {quantidade: 100};
> var repor = function (o) {o.quantidade = 0; };

> original.quantidade;
  100

> repor(original);

> original.quantidade;
  0
```

COMPARAÇÃO DE OBJETOS

- A comparação de objetos só devolve *true* se comparar duas referências para o mesmo objeto.
- Exemplo:

```
> var kikas = {raca: "cão"};
> var lucky = {raca: "cão"};

> kikas === lucky;  // dois objetos diferentes
  false

> kikas == lucky;
  false

> var meucao = lucky; // referenciam o mesmo objeto
> meucao === lucky;
  true
```

OBJETOS NATIVOS

JavaScript

OBJETOS "NATIVOS"

- Podem dividir-se em 3 grupos:
  - **Data wrapper objects**:
    - *Object*;
    - *Array*;
    - *Function*;
    - *Boolean*;
    - *Number*;
    - *String*
  - **Utility objects**:
    - *Math*;
    - *Date*;
    - *RegExp*
  - **Error objects**:
    - *Error*
    - outros mais específicos

OBJECTO NATIVO: *Object*

- ***Object*** **é o objecto pai de todos os objectos.**
- **Métodos e propriedades:**

```
> var o = new Object();

//devolve referência à função construtora
> o.constructor;

//devolve string descritiva do objeto
> o.toString();

//devolve um valor representativo do objeto
> o.valueOf();
```

OBJECTO NATIVO: *Array*

- *Array* é uma função nativa usada como construtora para criar arrays.
- Alguns métodos e propriedades:

```
> var a = [3,5,1,7,"test"];

> a.length;  // devolve nº de elementos
> a.push("new"); //acrescenta "new" ao final do array
> a.pop();  // retira e devolve o último elemento
> a.sort(); // devolve uma cópia ordenada do array
> a.join('||');  // devolve string com os elementos
separados pelo parâmetro indicado
> a.slice(1,3);  // devolve uma parcela do array
definida pelos parâmetros indicados (início e fim)
> a.splice(...); // remove, altera e insere elementos
```

OBJECTO NATIVO: *Array*

- Alguns métodos e propriedades:

```
> var a = [3,5,1,7,"test"];

> a.shift("first");  // acrescenta "first" ao ínicio
> a.unshift(); // retira e devolve o primeiro elemento
> a.reverse(); // inverte a ordem dos elemento
> a.concat(...); // acrescenta elementos indicados ao
array
> a.toString(); // apresenta o conteúdo do array em
string
```

OBJECTO NATIVO: *Function*

- ***Function*** **é um objeto que permite, de outra forma, criar funções.**
- **Métodos e propriedades:**

```
> var f = new Function('a','b','return a+b;');

//devolve referência à função construtora
> f.constructor;

//devolve o número de argumentos da função
> f.length;

//devolve a referência para o objeto ascendente
> f.prototype;
```

OBJECTO NATIVO: *Function*

- **Métodos e propriedades:**

```
var obj = {
    name: 'Ana',
    say: function (who){
      return "Hello " + who + ", I am a " + this.name;
    }
};

> var myobj = {name: "scripting guru"};
> obj.say.call(myobj, "Rui");
  "Hello Rui, I am a scripting guru"

> obj.say.apply(myobj, ["Rui"]);
  "Hello Rui, I am a scripting guru"
```

OBJECTO NATIVO: *Boolean*

- *Boolean* permite criar objetos que manipulam valores booleanos.
- Métodos e propriedades:

```
> var b = new Boolean();

//devolve o valor booleano representado pelo objeto
> b.valueOf();
```

- Quando utilizado o objeto *Boolean* sem *new* permite converter valores não booleanos em valores booleanos

```
> Boolean("test");
  true
> Boolean("");
  false
> Boolean({});
  true
```

OBJECTO NATIVO: *Number*

- ***Number*** **permite criar objetos que manipulam valores numéricos.**
- **Métodos e propriedades:**

```
> var n = new Number('123.456');

> n.toFixed(1);   // arredonda a n digitos: "123.5"
> n.toExponential(2);// devolve em notação científica:
"1.23e+2"
> n.toPrecision(5);//devolve valor com n dígitos de precisão
  "123.46"

// toString - devolve a string que representa o número
> var nr = new Number(255);
> nr.toString(); // "255"
> nr.toString(2); // "11111111"
> nr.toString(8);  // "377"
> nr.toString(16); // "ff"
```

OBJECTO NATIVO: *Number*

- Quando utilizado o objeto *Number* sem *new* permite devolver o valor primitivo e às suas propriedade.

```
> Number("101");
  101
> Number.MAX_VALUE;
  1.7976931348623157e+308
> Number.MIN_VALUE;
  5e-324
> Number.NaN;
  NaN
> Number.POSITIVE_INFINITY;
  Infinity
> Number.NEGATIVE_INFINITY;
  -Infinity
```

OBJECTO NATIVO: *String*

- ***String*** **permite criar objetos de tipo string.**
- **Métodos e propriedades:**

```
> var s = new String("javascript");

// número de carateres da string
> s.length;  // 10

//devolve o caratere da posição indicada
> s.charAt(0);

//devolve uma nova string concatenada (s + parâmetros)
> s.concat(" ECMA");  // "javascript ECMA"

//devolve posição onde começa a substring indicada
> s.indexOf("scr");  // 4; -1 se não encontrar
```

OBJECTO NATIVO: *String*

- **Métodos e propriedades:**

```
//o mesmo que indexOf mas começa a procura do final
> s.lastIndexOf("a");  // 3

//Compara duas strings; devolve: 0 se iguais; 1 se s > parâmetro; -1 se s < parâmetro
> s.localeCompare("crypt");  // 1
> s.localeCompare("sscript");  // -1
> s.localeCompare("javascript");  // 0

// aceita uma expressão regular e devolve um array de correspondências
> "R2-D2 and C-3PO".match(/[0-9]/g);// ["2","2","3"]

// devolve a posição da primeira correspondência
> "C-3PO".search(/[0-9]/); // 2
```

OBJECTO NATIVO: *String*

- **Métodos e propriedades:**

```
//substitui o conteúdo indicado nos locais onde
existe correspondência com a expressão regular
> "R2-D2".replace("/2/g", "-two"); // "R-two-D-two"

//devolve parte da string
> "R2-D2 and C-3PO".slice(4,13);// "2 and C-3"
> "R2-D2 and C-3PO".slice(4,-1);// "2 and C-3P"

// devolve uma String transformada num Array
> "1,2,3,4".split(/,/);// ["1","2","3","4"]
> "1,2,3,4".split(",", 2);// ["1","2"]

//devolve parte da String (igual a slice)
> "R2-D2 and C-3PO".slice(4,13);// "2 and C-3"
> "R2-D2 and C-3PO".slice(13,4);// "2 and C-3"
```

OBJECTO NATIVO: *String*

- **Métodos e propriedades:**

```
//devolve a string transformada em minúsculas
> "JavaScript".toLowerCase(); // "javascript"
> "JavaScript".toLocaleLowerCase(); // "javascript"

//devolve a string transformada em maiúsculas
> "JavaScript".toUpperCase(); // "JAVASCRIPT"
> "JavaScript".toLocaleUpperCase(); // "JAVASCRIPT"

// devolve string sem espaços, tabulações,
mudanças de linha,... (ECMAScript 5)
> " \t script \n".trim(); // "script"
> " \t script \n".replace(/\s/g); // "script" (ES3)
```

OBJECTO NATIVO: *Date*

- ***Date*** **permite criar objetos que manipulam datas.**
- **Criação de objetos:**

```
> var d = new Date();//data com o dia e hora
atuais

//argumento: nº de segundos desde 1/1/1970 00:00:01
> var d2 = new Date(1000);

//argumento: string data no formato RFC 1123
> var d3 = new Date("10/20/2001");

// argumentos: ano, mês, dia, hora, minuto, segundo
// milisegundo
> var d4 = new Date(2001,10,10,22,5,40,100);
```

OBJECTO NATIVO: *Date*

- **Métodos e propriedades:**

```
> var d = new Date("10/22/2001");

// devolve e altera dia do mês
> d.getDate();  // 22
> d.setDate(30);  // altera dia para 30

// devolve o número correspondente ao dia da semana da
// data [0..6] → 0=domingo, 1=2ª feira ... 6=Sábado
> d.getDay();

//devolve e altera o ano (4 dígitos)
> d.getFullYear();  // 2001
> d.setFullYear(2015);  // altera o ano da data

//devolve a data e/ou de forma descritiva
> d.toString();  // "Tue Nov 29 2016 00:00:00 GMT+0000 (WET)"
> d.toDateString();  // "Tue Nov 29 2016"
> d.toTimeString();  // "12:48:10 GMT+0000 (WET)"
> d.toLocaleDateString();  // "28/11/2015"
```

OBJECTO NATIVO: *Date*

- **Métodos e propriedades:**

```
// devolve e altera o mês da data
> d.getMonth();  // 9 (0=Janeiro; 11=Dezembro)
> d.setMonth(0);  // altera o mês para janeiro

// devolve e altera hora, minuto, segundo e
// milissegundo
> d.getHours();
> d.setHours(23); // valores permitidos [0..23]
> d.getMinutes();
> d.setMinutes(45); // valores permitidos [0..59]
> d.getSeconds();
> d.setSeconds(30); // valores permitidos [0..59]
> d.getMilliseconds();
> d.setMilliseconds(230); // valores permitidos [0..999]

// devolve o nº que representa a data em milissegundos desde
1/1/1970 00:00:00 UTC
> d.getTime();
> d.setTime(14124123412); // 13/06/1970 12:22:03
```

OBJECTO NATIVO: *Math*

- É um objeto global que disponibiliza um conjunto de métodos e propriedades para operações matemáticas (Atenção: *Math* não é uma função → não pode ser usado *new*)

- Constantes:

```
> Math.PI; // valor de PI
> Math.SQRT2; // valor de raiz quadrada de 2
> Math.E;  // valor da constante de Euler
> Math.LN2; // valor do logaritmo natural de 2
> Math.LN10; // valor do logaritmo natural de 10
```

OBJECTO NATIVO: *Math*

- Alguns médodos:

```
// devolve um valor aleatório entre 0 e 1
> Math.random();

// devolve um inteiro arredondado para baixo
> Math.floor(1234.567); // 1234

// devolve um inteiro arredondado para cima
> Math.ceil(1234.123); // 1235

// devolve inteiro arredondado para valor próximo
> Math.round(1234.5678); // 1235
> Math.round(1234.234); // 1234

// devolve o cálculo da potência (base, expoente)
> Math.pow(2, 8); // 256

// função exponencial (Math.E, x)
> Math.exp(x); // e^x

// devolve o cálculo da raiz quadrada
> Math.sqrt(9); // 3
```

OBJECTO NATIVO: *Math*

- **Alguns médodos:**

```
// devolve o maior valor dos argumentos indicados
> Math.max(4.5, 101, Math.PI); // 101

// devolve o menor valor dos argumentos indicados
> Math.min(4.5, 101, Math.PI); // 3.141592653589793

// devolve o valor absoluto
> Math.abs(-101); // 101

// Funções trignométricas
> Math.cos(x);
> Math.sin(x);
> Math.tan(x);
> Math.acos(x);
> Math.asin(x);
> Math.atang(x);
```

OBJECTO NATIVO: *RegExp*

- É um objeto global que disponibiliza uma maneira poderosa de procurar e manipular texto. O JavaScript usa a sintaxe do Perl 5;
- Uma expressão regular define um padrão utilizado por algumas funções para encontrar correspondências;
- Um padrão é delimitado por barras / /;
- Dentro das barras utilizam-se metacarateres, com significado especial, para definir um padrão;

OBJECTO NATIVO: *RegExp*

- **Metacarateres:**
  `\ | ( ) [ ] { } ^ $ * + ? .`

- **. (ponto): substitui um carater :**

```
/snow./; // correspondência válida: snowy,snowe
Para representar o ponto: \.
/3\.4/ // correspondência válida: 3.4
```

- **[ ]: definir conjunto de carateres :**

```
/[abc]/; // correspondência válida: a, ou b ou c
/[a-h]/; // correspondência válida: a até h
```

- **^ : inverte padrão :**

```
/[^aeiou]/; // válida: qualquer letra exceto vogais
```

OBJECTO NATIVO: *RegExp*

- { } : define o número de vezes da repetição :

```
/xy{4}z/; // correspondência válida: xyyyyz
```

- Classes de carateres:

```
\d; // um dígito ⇔ [0-9]
\D; // não um dígito ⇔ [^0-9]
\w; // alfanumérico ⇔ [A-Za-z0-9]
\W; // não alfanumérico ⇔ [^A-Za-z0-9]
\s; // espaço em branco ⇔ [ \r\t\n\f]
\S; // não espaço em branco ⇔ [^ \r\t\n\f]
```

- Quantificadores: + (1 ou mais); * (0 ou mais); ? (um ou nenhum):

```
/x*y+z?/; // 0 ou mais x, seguido de 1 ou mais y e de um
ou nenhum z
/\d+\.\d*/; // válido: 45. ou 67.890
/[A-Za-z]\w*/; // uma letra seguida de 0 ou mais alfanuméricos
```

OBJECTO NATIVO: *RegExp*

- \b : limite de palavras (boundary)

```
/\bis\b/; // válido: "A tulip is a flower"
          // inválido: "A frog isn't"
```

- Âncoras (^ $)

```
^ → valida a sequência  só no início
/^pearl/  // válido: "pearls are pretty"
          // inválido: "My pearls are pretty"

$ → valida a sequência só no fim
/gold$/   // válido: "I like gold"
          // inválido: "golden"
```

- Modificadores de padrão:

```
g → correspondência global
i → ignora distinção entre maiúsculas/minúsculas
m → valida sequências multilinha
```

OBJECTO NATIVO: *RegExp*

- **Criar expressões regulares**

```
var re = new RegExp("j.*t");

ou

var re = /j.*t/;   // notação literal
```

- **Métodos do objecto RegExp**

test() – devolve true ou false conforme encontrou ou não a correspondência

```
> /j.*t/.test("Javascript")
  false

> /j.*t/i.test("Javascript")
  true
```

exec() – devolve um array de strings com as correspondências

```
> /j.*t/.exec("Javascript")
  []

> /(j.*a)(s.*t)/i.exec("Javascript")
  ["Javascript","Java","script"]
```

OBJECTO NATIVO: *RegExp*

- Métodos de String que aceitam expressões regulares como argumentos:

```
> var s = "HelloJavaScriptWorld";

match() – devolve um array de correspondências
> s.match(/a/);  // devolve a primeira correspondência
  ["a"];
> s.match(/a/g);  // devolve todas as correspondências
  ["a","a"];

search() – devolve a posição da primeira correspondência
> s.search(/j.*a/i);
  5

replace() – devolve uma string com as correspondências substituídas
por outra string
> s.replace(/[A-Z]/g, '');
  "elloavacriptorld";

> s.replace(/[A-Z]/g, '_$&');  // acrescenta _ antes do match
  "_Hello_Java_Script_World";

> var mail = "joaquim@sapo.pt";
> mail.replace(/(.*)@.*/, "$1"); // define o grupo $1 a devolver
  "joaquim";
```

OBJECTO NATIVO: *RegExp*

- **Substituição por callbacks:**

```
var s = "HelloJavaScriptWorld";

function replaceCallback(match){
    return “_” + match.toLowerCase();
}


> s.replace(/[A-Z]/g, replaceCallback);
  “_hello_java_script_world”;
```

A PROPRIEDADE *PROTOTYPE*

JavaScript
POO

A PROPRIEDADE *prototype*

- As funções em JavaScript são objetos que contêm propriedades e métodos:
  - Metodos: apply(), call()
  - Propriedades: length, constructor e prototype
- A propriedade ***prototype*** é um objeto vazio associado automaticamente a função construtora;
- Sendo um objeto podemos adicionar novos métodos e propriedades;
- A propriedade ***prototype*** não tem qualquer efeito na função mas nos objetos criado a partir da função construtora.

ADICIONAR MÉTODOS E PROPRIEDADES USANDO O prototype

- Adicionar métodos ou propriedades à propriedade *prototype* é outra forma de adicionar funcionalidades e caraterísticas aos novos objetos;
- Exemplo:

```
function Gadget(nome, cor){
  this.nome = nome;
  this.cor = cor;
  this.quemEs = function(){
    return "Eu sou um(a) " + this.nome +
                    " " + this.cor;
  };
}
```

ADICIONAR MÉTODOS E PROPRIEDADES USANDO O prototype

- Adicionando métodos e propriedades à *prototype*:

```
Gadget.prototype.preco = 100;
Gadget.prototype.qualidade = 3;
Gadget.prototype.informacao = function(){
  return “Qualidade: “ + this.qualidade + “; Preço: “ +
                              this.preco;
};

ou

Gadget.prototype = {
   preco: 100,
   qualidade: 3,
   informacao: function(){
       return “Qualidade: “ + this.qualidade +
                    “; Preço: “ + this.preco;
};
```

prototype
USAR DOS MÉTODOS E PROPRIEDADES

- A partir do momento que criamos um novo objeto através da função construtora temos acesso às propriedades e métodos definidos.

- Exemplos:

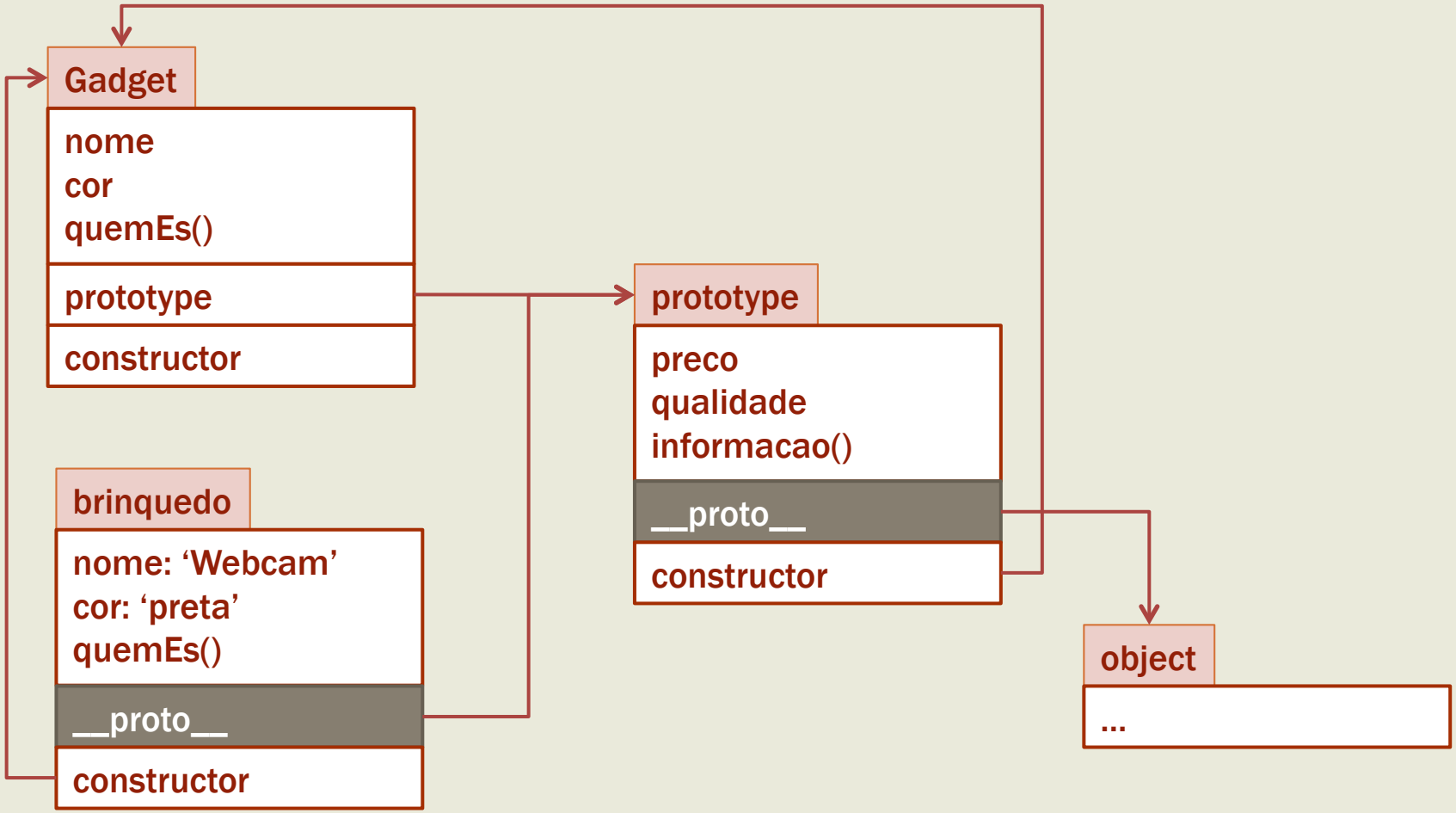
```
> var brinquedo= new Gadget("webcam", "preta");
> brinquedo.nome;
  "Webcam"
> brinquedo.cor;
  "preta"
> brinquedo.quemEs();
  "Eu sou um(a) Webcam preta"
> brinquedo.preco;
  100
> brinquedo.qualidade;
  3
> brinquedo.informacao();
  "Qualidade: 3; Preço: 100"
```

prototype
USAR DOS MÉTODOS E PROPRIEDADES

- Nota: o objeto ***propotype*** não é copiado para cada instância do objeto
- Logo, qualquer alteração no ***prototype*** da função *Gadget* é visível em todos os objetos criados através da função construtura
- Exemplos:

```
Gadget.prototype.get = function(prop){
    return this[prop];
}

> brinquedo.get('preco');
  100

> brinquedo.get('cor');
  "preta"
```

PROPRIEDADES PRÓPRIAS versus PROPRIEDADES DO PROTOTYPE



ACESSO ÀS PROPRIEDADE E MÉTODOS DA PROPRIEDADE *prototype*

- Sequência de acesso às propriedades e métodos:
- `> brinquedo.nome;`
  - 1º procura a propriedade na instância do objeto (neste caso encontra)
- `> brinquedo.preco;`
  - 1º procura a propriedade na instância do objeto (não encontra)
  - 2º procura no protótipo da função construtora → brinquedo.constructor.prototype (enconta);
  - Logo é possível aceder à propriedade *preco* através da instrução
    `> brinquedo.constructor.prototype.preco;`

REESCREVER DE UMA PROPRIEDADE DO PROTOTYPE NO OBJETO

- Se existir duas propriedades ou métodos com o mesmo nome no objeto e no prototype, qual tem precedência?
- O método/propriedade do objeto tem precedência.
- Exemplo:

```
function Gadget(nome){
    this.nome = nome;
}

> Gadget.prototype.nome = “espelho”;

> var brinquedo = new Gadget(‘pintura’);
> brinquedo.camera;
  “pintura”

> delete brinquedo.nome;

> brinquedo.nome;
  “espelho”
```

MÉTODO RELACIONADOS COM PROTOTYPE

- **hasOwnProperty()**: devolve *true* ou *false* indicando se um propriedade pertence a um objeto;

```
> brinquedo.hasOwnProperty('nome');
  true
```

- Podemos, por exemplo, descobrir a origem de determinada propriedade/método:

```
> brinquedo.toString();
"[object Object]"
> brinquedo.hasOwnProperty('toString');
  false
> brinquedo.constructor.hasOwnProperty('toString');
  false
> brinquedo.constructor.prototype.hasOwnProperty('toString');
  false
> Object.hasOwnProperty('toString');
  false
> brinquedo.prototype.hasOwnProperty('toString');
  true
```

MÉTODO RELACIONADOS COM PROTOTYPE

- **isPrototypeOf()**: este método diz-se se um determinado objeto é prototype de outro;

- Exemplo

```
var macaco = {
  pelo: true,
  comida: 'bananas',
  respira: 'ar'
};

function Humano(nome){
  this.nome = nome;
}

Humano.prototype = macaco;

> var ze = new Humano('Zé');

> macaco.isPrototypeOf(ze);
  true
```

ENUMERAR PROPRIEDADES

- É possível obter todas as propriedades através de um ciclo *for-in*.

- Exemplo:

```
var params= {
  produtoid: 555,
  seccao: 'produtos',
};

var url='http://exemplo.org/pagina.php?';
var i, query = [];

for(i in params){
  query.push(i + '=' + params[i]);
}

url += query.join('&');

> url;
  http://exemplo.org/pagina.php?produtoid=555&seccao=produtos
```

ENUMERAR PROPRIEDADES

- Nota: o ciclo extrai todas as propriedades enumeráveis do objecto, incluindo as propriedades do objeto *prototype*.

- Exemplo: extraindo todos as propriedades:

```
var listaprop = ‘’;
for(prop in brinquedo){
  listaprop += prop + “=“ + brinquedo[prop];
}
```

- Exemplo: extraindo apenas a propriedades do objecto:

```
var listaprop = ‘’;
for(prop in brinquedo){
  if(brinquedo.hasOwnProperty(prop)){
     listaprop += prop + “=“ +
                        brinquedo[prop];
  }
}
```

O LINK ESCONDIDO *__proto__*

- Considere-se o seguinte código:

```
var macaco = {
  comida: 'bananas',
  respira: 'ar'
};

function Humano(){}
Humano.prototype = macaco;

var programador = new Humano();

> programador.__proto__ === macaco;
  true
```

- Atenção que *__proto__* **não é o mesmo** que *prototype*. *__proto__* é uma propriedade de instancias de objetos; *prototype* é uma propriedade da função construtora usada para criar objetos,

```
> typeof programador.__proto__;
  "object"
> typeof programador.prototype;
  "undefined"
> typeof programador.constructor.prototype;
  "object"
```

AUMENTAR O PODER DOS OBJECTOS NATIVOS (BUILD-IN)

- É possível aumentar as funcionalidades dos objetos nativos (Array, String, Object, Function,...) mexendo na sua propriedade prototype.
- Exemplo: acrescentar um novo método ao objeto nativo *Array*

```
//Indica se um dado valor existe no array
Array.prototype.inArray = function(elemento){
  for(var i = 0, tam = this.length; i < tam; i++ ){
    if(this[i] === elemento){
      return true;
    }
  }
  return false;
};

> var cores = ['azul', 'verde', 'vermelho'];
> cores.inArray('azul');
  true
> cores.inArray('amarelo');
  false
```

ALTERAÇÃO DE prototype?...

- Na utilização do prototype é importante considerar dois aspetos do seu comportamento:
  - A cadeia do prototype está ativa, exceto se se substituir o objeto prototype;
  - O prototype.constructor não é fiável, sendo necessário reinicializar o construtor

ALTERAÇÃO DE prototype?...

- **Considere-se o seguinte exemplo:**

```
//Criar a função construtora
function Dog(){
  this.cauda = true;
}
//Criar 2 objetos
var benji = new Cao();
Var lucky = new Cao();
//Acrescentar método ao prototype
Dog.prototype.ladra = function(){
  return 'Wolf!';
}
> benji.ladra();
  Wolf!
> lucky.ladra();
  Wolf!
> lucky.constructor === Cao;
  true
> benji.constructor === Cao;
  true
```

ALTERAÇÃO DE prototype?...

- **Exemplo (continuação):**

```
// Reescrever o objecto prototype
Cao.protitype = {
   this.patas: 4,
   this.pelo: true
}
// Verificar a cadeia dos objetos
> typeof benji.patas;
   "undefined"
> benji.ladra();
   "Wolf!"
> Typeo benji.__proto__.say;
   "function"
> typeof benji.__proto__.patas;
   "undefined"
```

ALTERAÇÃO DE prototype?...

- **Exemplo (continuação):**

```
// criar novo objeto
var leao = new Cao();

// Verificar a cadeia dos objetos
> leao.ladra();
  TypeError: leao.ladra is not a function
> leao.patas;
  4
> leao.constructor;
  function Object()
> benji.constructor;
  function Dog()

// Reiniciar a propriedade constructor
> Cao.prototype.constructor = Cao;
> leao.constructor === Cao;
  true
```

EXERCÍCIOS

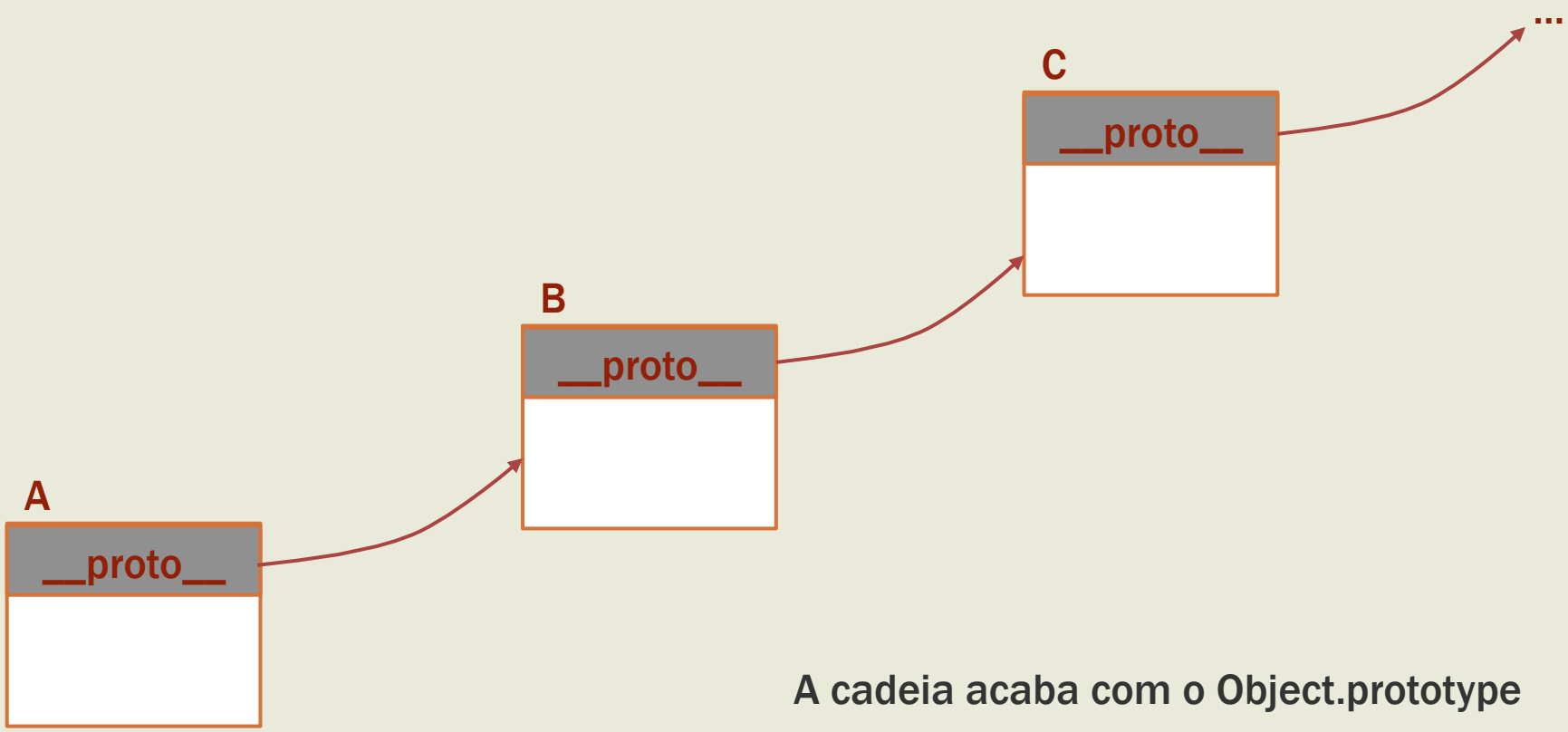
- Criar um objeto chamado Figura que tem a propriedade tipo e o método getTipo().
- Definir uma função construtora Triangulo() cujo protótipo é Figura. Os objetos criados a partir de Triangulo() devem possuir três propriedades a, b e c representando o tamanho dos lados do triângulo.
- Adicionar um novo método ao prototype chamado getPerimetro().
- Criar objetos e testar as suas propriedades e métodos.
- Criar um código que mostre as propriedade do objeto (não as do protótipo)
- Fazer o código que permita fazer com que seguinte instrução funcione:

```
>[1, 2, 3, 4, 5, 6, 7, 8, 9].baralha();
 [2,4,1,8,9,6,5,3,7]   //dados baralhados aleatoriamente
```

HERANÇA

Javascript

A CADEIA PROTOTYPE



A cadeia acaba com o Object.prototype

EXEMPLO DE HERANÇA

- Modo standard de implementação de herança

```
function Figura(){
   this.nome = 'Figura';
   this.toString = function(){
     return this.nome;
   };
}

function Figura2D(){
   this.nome = 'Figura 2D';
}

function Triangulo(lado, altura){
   this.nome = 'Triângulo';
   this.lado = lado;
   this.altura = altura;
   this.getArea = function(){
     return this.lado * this.altura / 2;
   };
}
```

EXEMPLO DE HERANÇA

- **Standard de implementação de herança (continuação)**

```
//Implementar a herança
Figura2D.prototype = new Shape();
Triangulo.prototype = new Shape2D();

// Como reescrevemos o prototype é
// importante redefinir o constructor
Figura2D.prototype.constructor = Figura2D;
Triangulo.prototype.constructor = Triangulo;

//Criar objetos
> var my = new Triangulo(5, 10);
> my.getArea();
  25
> my.toString();
  "Triângulo"
```

EXEMPLO DE HERANÇA

- **Standard de implementação de herança (continuação)**

```
// Como é natural é possível criar objetos
// a partir das outras funções construtoras
var f2d = new Shape2D();
var fig = new Shape();

> f2d.toString();
  "Figura 2D"
> fig.toString();
  "Figura"
```

HERANÇA – MOVER AS PARTILHAS PARA O PROTOTYPE

- As propriedades e métodos que não mudam nas instâncias devem ser movidas para o prototype.

- Exemplo anterior modificado:

```
function Figura(){}
// adicionar elementos partilhados
Figura.prototype.nome = 'Figura';
Figura.prototype.toString = function(){
  return this.nome;
};

function Figura2D(){}
// Cuidar primeiro do processo de herança
// antes de aumentar partilhas
Figura2D.prototype = new Figura();
Figura2D.prototype.constructor = Figura2D;
// Adicionar elementos partilhados
Figura2D.prototype.nome = 'Figura 2D';
```

HERANÇA – MOVER AS PARTILHAS PARA O PROTOTYPE

- Exemplo anterior modificado (continuação):

```
// No caso do Triangulo, deixamos as propriedades
// que são próprias de cada objecto (lado, altura)
function Triangulo(lado, altura){
   this.lado = lado;
   this.altura = altura;
}

// Cuidar primeiro do processo de herança
// antes de aumentar partilhas
Triangulo.prototype = new Figura2D();
Triangulo.prototype.constructor = Triangulo;
// Adicionar elementos partilhados
Triangulo.prototype.nome = 'Triângulo';
Triangulo.prototype.getArea = function(){
   return this.lado * this.altura / 2;
};
```

HERANÇA – MOVER AS PARTILHAS PARA O PROTOTYPE

- Testar este método de herança (continuação):

```
//Criar objetos
> var my = new Triangulo(5, 10);
> my.getArea();
  25
> my.toString();
  "Triângulo"
> my.hasOwnProperty('lado');
  true
> my.hasOwnProperty('nome');
  false
```

HERANÇA – HERDAR APENAS O PROTOTYPE

- Se por razões de eficiência se deve colocar propriedades e métodos para reutilizar no *prototype* - ➔ é boa ideia herdar apenas o *prototype*.
- Ou seja, é melhor, por exemplo, herdar *Figura.prototype* do que *new Figura()*.
- Ganha-se, assim, um pouco mais de eficiência porque:
  - Não se criam novos objetos só por causa da herança
  - Criam-se menos ligações em *runtime* na cadeia de herança

HERANÇA – HERDAR APENAS O PROTOTYPE

- Código modificado para herdar apenas o prototype:

```
function Figura(){}
// adicionar elementos partilhados
Figura.prototype.nome = 'Figura';
Figura.prototype.toString = function(){
  return this.nome;
};

function Figura2D(){}
// Cuidar primeiro do processo de herança
// antes de aumentar partilhas
Figura2D.prototype = Figura.prototype;
Figura2D.prototype.constructor = Figura2D;
// Adicionar elementos partilhados
Figura2D.prototype.nome = 'Figura 2D';
```

HERANÇA – HERDAR APENAS O PROTOTYPE

- Código para herdar apenas o *prototype* (continuação)

```
function Triangulo(lado, altura){
    this.lado = lado;
    this.altura = altura;
}

// Cuidar primeiro do processo de herança
// antes de aumentar partilhas
Triangulo.prototype = Figura2D.prototype;
Triangulo.prototype.constructor = Triangulo;
// Adicionar elementos partilhados
Triangulo.prototype.nome = 'Triângulo';
Triangulo.prototype.getArea = function(){
    return this.lado * this.altura / 2;
};
```

HERANÇA – HERDAR APENAS O PROTOTYPE

- Testar este método de herança (continuação):

```
//Criar objetos
> var my = new Triangulo(5, 10);
> my.getArea();
  25
> my.toString();
  "Triângulo"
> var f = new Figura();
> f.nome;
  "Triângulo2
```

- Este método embora mais eficiente, tem um efeito lateral: como os *prototype* de "descendentes" ou "ascendentes" aponta para o mesmo objeto ➔ as modificações feitas por um "descendente" no *prototype* são assumidas por "ascendentes" e "irmãos".

HERANÇA – CONSTRUTORES TEMPORÁRIOS

- Esta técnica permite resolver o problema anterior, recorrendo a uma função construtora intermediária para interromper cadeia de propagação.
- Consiste, na prática, em criar uma função construtora vazia, por exemplo *F()*, e através da chamada a *new F()* cria objetos que não têm propriedades, mas que herdam tudo do *prototype* ascendente.

HERANÇA – CONSTRUTORES TEMPORÁRIOS

- Código modificado para utilização de construtores temporários:

```
function Figura(){}
// adicionar elementos partilhados
Figura.prototype.nome = 'Figura';
Figura.prototype.toString = function(){
  return this.nome;
};

function Figura2D(){}
// Cuidar primeiro da herança
var F = function(){};
F.prototype = Figura.prototype;
Figura2D.prototype = new F();
Figura2D.prototype.constructor = Figura2D;
// Adicionar elementos partilhados
Figura2D.prototype.nome = 'Figura 2D';
```

HERANÇA – CONSTRUTORES TEMPORÁRIOS

- Código modificado para utilização de construtores temporários (continuação):

```
function Triangulo(lado, altura){
    this.lado = lado;
    this.altura = altura;
}

// Cuidar primeiro do processo de herança
var F = function(){};
F.prototype = Figura2D.prototype;
Triangulo.prototype = new F();
Triangulo.prototype.constructor = Triangulo;
// Adicionar elementos partilhados
Triangulo.prototype.nome = 'Triângulo';
Triangulo.prototype.getArea = function(){
    return this.lado * this.altura / 2;
};
```

HERANÇA – CONSTRUTORES TEMPORÁRIOS

- Testar este método de herança (continuação):

```
//Criar objetos
> var my = new Triangulo(5, 10);
> my.getArea();
  25
> my.toString();
  "Triângulo"
> var f = new Figura();
> f.nome;
  "Figura"
```

HERANÇA – ACESSO DOS DESCENDENTES AOS ASCENDENTES

- É possível obter a referência do *prototype* do objeto ascendente de forma que o objeto descendente faça uso de um método ascendente, sendo possível acrescentar mais qualquer coisa.
- Modificação do código para contemplar esta funcionalidade:

```
function Figura(){}
// adicionar elementos partilhados
Figura.prototype.nome = 'Figura';
Figura.prototype.toString = function(){
   var superior = this.constructor;
   return superior.uber ?
            superior.uber.toString()
              + ', ' + this.nome : this.nome);
};
```

HERANÇA – ACESSO DOS DESCENDENTES AOS ASCENDENTES

- Modificação do código para contemplar esta funcionalidade (continuação):

```
function Figura2D(){};

// Cuidar primeiro da herança
var F = function(){};
F.prototype = Figura.prototype;
Figura2D.prototype = new F();
Figura2D.prototype.constructor = Figura2D;
Figura2D.uber = Figura.prototype;
// Adicionar elementos partilhados
Figura2D.prototype.nome = 'Figura 2D';
```

HERANÇA – ACESSO DOS DESCENDENTES AOS ASCENDENTES

- Modificação do código para contemplar esta funcionalidade (continuação):

```
function Triangulo(lado, altura){
   this.lado = lado;
   this.altura = altura;
}

// Cuidar primeiro do processo de herança
var F = function(){};
F.prototype = Figura2D.prototype;
Triangulo.prototype = new F();
Triangulo.prototype.constructor = Triangulo;
Triangulo.uber = Figura2D.prototype;
// Adicionar elementos partilhados
Triangulo.prototype.nome = 'Triângulo';
Triangulo.prototype.getArea = function(){
   return this.lado * this.altura / 2;
};
```

HERANÇA – ACESSO DOS DESCENDENTES AOS ASCENDENTES

- Testar este método de herança (continuação):

```
//Criar objetos
> var my = new Triangulo(5, 10);
> my.getArea();
  25
> my.toString();
  "Figura, Figura 2D, Triângulo”
```

HERANÇA – ISOLAR A HERANÇA ATRAVÉS DE UMA FUNÇÃO

- É possível isolar o processo de herança através de uma função, simplificando, assim, o código:

```
function extend(descendente, ascendente){
    var F = function(){};
    F.prototype = ascendente.prototype;
    descendente.prototype = new F();
    descendente.prototype.constructor = descendente;
    descendente.uber = ascendente.prototype;
}
```

HERANÇA – ISOLAR A HERANÇA ATRAVÉS DE UMA FUNÇÃO

- Continuação:

```
function Figura(){};
// adicionar elementos partilhados
Figura.prototype.nome = 'Figura';
Figura.prototype.toString = function(){
   return (this.constructor.uber ?
             this.constructor.uber.toString()
                   + ', ' + this.nome : this.nome);
};

function Figura2D(){};
// Cuidar primeiro da herança
extend(Figura2D, Figura);
// Adicionar elementos partilhados
Figura2D.prototype.nome = 'Figura 2D';
```

HERANÇA – ISOLAR A HERANÇA ATRAVÉS DE UMA FUNÇÃO

- Continuação:

```
function Triangulo(lado, altura){
   this.lado = lado;
   this.altura = altura;
}

// Cuidar primeiro do processo de herança
var F = function(){};
extend(Triangulo, Figura2D);
// Adicionar elementos partilhados
Triangulo.prototype.nome = 'Triângulo';
Triangulo.prototype.getArea = function(){
   return this.lado * this.altura / 2;
};
```

HERANÇA – OBJETOS DE OBJETOS

- Podemos também, criar novos objetos a partir de outros objetos, usando a notação literal de objetos.
- O processo começa por definir uma função que recebe um objecto e devolve um nova cópia desse objeto:

```
function extendCopy(p){
    var c = {};
    for(var i in p){
        c[i] = p[i];
    }
    c.uber = p;
    return c;
}
```

HERANÇA – OBJETOS DE OBJETOS

- Herança por cópia de objetos (continuação):

```
var Figura = {
    nome: 'Figura',
    toString: function(){
        return this.nome;
    }
};

var Figura2D = extendCopy(Figura);
Figura2D.nome = 'Figura 2D';
Figura2D.toString = function(){
    return this.uber.toString()
                + ', ' + this.nome;
};
```

HERANÇA – OBJETOS DE OBJETOS

- Herança por cópia de objetos (continuação):

```
var Triangulo = extendCopy(Figura2D);
Triangulo.nome = 'Triângulo';
Triangulo.getArea = function(){
   return this.lado * this.altura / 2;
};

> Triangulo.lado = 5;
> Triangulo.altura = 10;
> Triangulo.getArea();
  25
> Triangulo.toString();
  "Figura, Figura 2D, Triângulo"
```

HERANÇA MISTA (PROTOTYPE E CÓPIA)

- Esta técnica consiste em criar um novo objeto herdando um, adicionando então propriedades e métodos adicionais.
- Isto é possível através da chamada a uma função que combina as duas técnicas (*prototype* e cópia).

```
function objectPlus(o, adicional){
    var n;
    function F(){};
    F.prototype = o;
    n = new F();
    n.uber = o;

    for(var i in adicional){
        n[i] = adicional[i];
    }
    return n;
}
```

HERANÇA MISTA (PROTOTYPE E CÓPIA)

- Implementação de herança mista (continuação):

```
var Figura = {
    nome: 'Figura',
    toString: function(){
        return this.nome;
    }
};

var Figura2D = objectPlus(Figura, {
    nome: 'Figura 2D',
    toString: function(){
        return this.uber.toString()
                        +  ', ' + this.nome;
    }
});
```

HERANÇA MISTA (PROTOTYPE E CÓPIA)

- Implementação de herança mista (continuação):

```
var Triangulo = objectPlus(Figura2D, {
    nome: 'Triângulo',
    getArea: function(){
      return this.lado * this.altura / 2;
    },
    lado: 0,
    altura: 0
});

var my = objectPlus(Triangulo, {
    lado: 4,
    altura: 4
});

> my.getArea();
  8
```

MÚLTIPLA HERANÇA

- Como noutras linguagens é possível implementar um sistemas de múltipla herança, permitindo que um objeto herde propriedades e métodos de mais do que um objecto.
- Para tal, implementa-se uma função que aceita qualquer número de objetos:

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

MÚLTIPLA HERANÇA

- Implementação múltipla herança (continuação):

```
var Figura = {
  nome: 'Figura',
  toString: function(){
    return this.nome;
  }
};

var Figura2D = {
  nome: 'Figura 2D',
  dimensoes: 2
};
```

MÚLTIPLA HERANÇA

- Implementação múltipla herança (continuação):

```
var Triangulo = multi(Figura, Figura2D, {
  nome: 'Triângulo',
  getArea: function(){
    return this.lado * this.altura / 2;
  },
  lado: 5,
  altura: 10
});

> Triangulo.getArea();
  25
> Triangulo.dimensoes;
  2
> Triangulo.toString();
  "Triângulo"
```