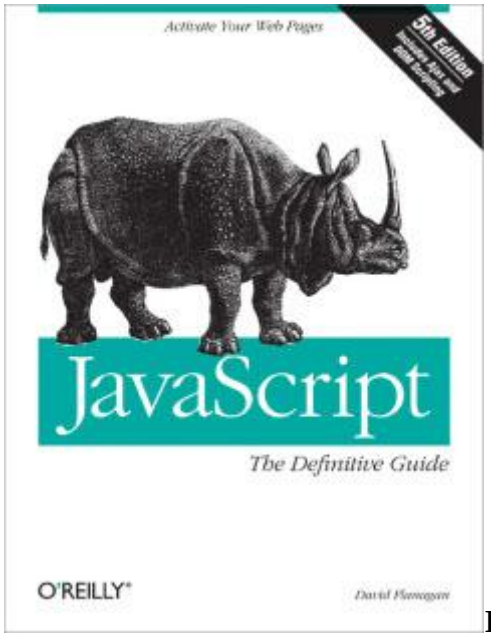
Introdução à linguagem JavaScript



Brendan Eich

Algumas características

- O principal responsável pelo desenho e implementação foi Brendan Eich. Esse trabalho começou a ser realizado no final de 1995 na empresa Netscape Communications. Inicialmente ele foi contratado para tornar a linguagem Java, já integrada no browser Netscape, mais fácil de usar por não-programadores. Mas rapidamente ele decidiu que que era necessário criar uma linguagem de scripting nova para facilitar a gestão dos elementos das páginas WEB. Acima de tudo era importante tornar a linguagem acessível a WEB designers com poucos conhecimentos de programação. (Correntemente, Brendan Eich é o CTO da Mozilla Corporation.)
- O grande objetivo inicial do Java e depois do JavaScript foi adicionar interatividade a páginas WEB.
- Hoje em dia, a linguagem JavaScript está integrada em diversos tipos de aplicações, não só em browsers. Só alguns exemplos: a aplicação Acrobat usa-a para manipular ficheiros PDF; os controlos remotos programáveis topo-da-gama da Philips usam-na; potencialmente, qualquer aplicação escrita em Java 6.0, ou versão mais recente, pode integrar JavaScript usando o package `javax.script`.
- É uma linguagem padronizada com standard ECMA.
- A especificação diz que a linguagem deve poder ser interpretada, por forma a que seja possível correr scripts gerados dinamicamente.
- Tem uma sintaxe semelhante à do Java e alguns dos seus aspetos superficiais imitam o Java (e.g. classe Math). Contudo todo o resto da linguagem é diferente do Java: trata-se duma linguagem com tipificação dinâmica, que suporta funções que retornam funções (ou seja programação funcional), e ao nível dos objetos ela baseia-se nas ideia de "protótipo" da linguagem Self e não usa classes como a linguagem Java. A escolha do nome JavaScript foi apenas uma decisão de marketing tomada em 1995.
- A maioria das implementações assume que os programas correm dentro dum ambiente, e.g. um browser, que disponibiliza os objetos e os métodos com os quais a linguagem vai interagir.
- Suporta o uso de expressões regulares na manipulação de texto (inspirado no Awk e Perl).
- Tem gestão automática de memória, tal como o Java, mas opcionalmente pode ser usada uma primitiva `delete` para apagar explicitamente objetos, tal como em C++.
- O JavaScript é uma linguagem híbrida que suporta os paradigmas imperativo (procedimental), funcional e orientado pelos objetos.

Padronizações do JavaScript

O esforço de padronização teve início em 1996 e o padrão é conhecido pelo nome ECMA 262. Atualmente, o padrão vai na versão 5.

Versões do JavaScript

A implementação original de Brendan Eich evoluiu com o tempo, e deu origem a duas implementações que atualmente são *open source* e são mantidas pela Mozilla Foundation:

- SpiderMonkey - Escrito em C.
- Rhino - Escrito em Java para permitir usar scripting dentro de programas Java. Serviu de base ao novo package `javax.script`, introduzido no Java 6.0.

O JavaScript tem vindo a evoluir a par com a evolução do padrão ECMA. A versão mais recente do JavaScript implementa a versão 3 do padrão, mas adiciona algumas extensões.

- JavaScript 1.0 - 1995
- JavaScript 1.1 - 1996
- JavaScript 1.2 - 1997
- JavaScript 1.3 - 1998
- JavaScript 1.4 - 1999
- JavaScript 1.5 - 2000
- JavaScript 1.6 - 2006
- JavaScript 1.7 - 2007 - Introduz geradores, iteradores, *compreensões* usando arrays, expressões `let`, etc.
- JavaScript 1.8 - 2008
- JavaScript 1.8.1 - 2008
- JavaScript 1.8.5 - 2010
- JavaScript 2 - Futuro

Algumas implementações disponíveis

A página da bibliografia de LAP disponibiliza implementações do SpiderMonkey JavaScrip para várias plataformas. Tratam-se de interpretadores que funcionam em ambiente de consola e que são ótimos para desenvolver e testar programas. Na mesma página encontra-se disponível um extensão para o Firefox que inclui uma consola de JavaScript e que se destina a fazer debugging de páginas WEB.

Além disso, na coluna esquerda da página de LAP, no link "JavaScript Shell!" está disponível uma consola de JavaScript que funciona dentro do browser e é muitíssimo prática de usar. Escreva os seus programas JavaScript usando um editor de texto e depois vá fazendo copy&paste para dentro da consola para ir testando.

JavaScript embebido numa consola

O JavaScript só pode ser usado na prática, integrado num ambiente de execução que forneça objetos e métodos para interação com o exterior.

A versão de consola do SpiderMonkey corre num ambiente que fornece diversas primitivas, das quais destacamos as seguintes:

```
print     readline     load
build     help         quit
```

Examine bem esta pequena sessão de trabalho com a versão de consola do interpretador SpiderMonkey. Constitui um bom primeiro contacto com a linguagem.

```
// Correr js
$js

//Matemática simples
```

```
js> 1 + 1
2

js> d = Math.PI * 2 * 2
12.566370614359172

js> Math.random()
0.15057766821669438

js> typeof(1)
number

js> typeof(1.0)
number

// Strings
js> "hello, world"[0]
hd

js> "hello, world".replace("hello", "goodbye")
goodbye, world

js> x = 12.4 + "144"
12.4144

js> typeof(x)
string

js> typeof(readline())
> 123
string

js> typeof(parseInt("123", 10))
number

js> parseInt("aaa", 10)    // conversão base 10
NaN

js> parseInt("aaa", 16)    // conversão base 16
2730

// Ciclos
js> for (i = 0; i < 5; i++)
        print(i) ;
0
1
2
3
4

// Arrays
js> a = ["dog", "cat", "hen"]
dog,cat,hen

js> a.length
3

js> a[0] = 123.45
123.45

js> a
123.45,cat,hen

js> typeof(a[0])
number

js> typeof(a)
object
```

```
// Funções
js> function f(x) { return x + 1 ; }

js> typeof(f)
function

js> f(7)
8

js> function curriedAdd(x)
     { return function(y) { return x + y ; } }

js> var g = curriedAdd(5) ;

js> g(1) ;
6

// Fim
js> quit()    // ou CTRL-D

$
```

JavaScript embebido num WEB browser

Virtualmente todos os browsers atuais têm um interpretador de JavaScript embebido. Cada browser proporciona ao JavaScript um ambiente de execução com objetos e métodos que permitem usar essa linguagem para criar páginas WEB interativas.

Chama-se DOM - Document Object Model ao ambiente que qualquer browser é obrigado a disponibilizar ao JavaScript, caso queira suportar a linguagem. O DOM implementa a visão que o JavaScript tem das páginas escritas em HTML e do estado interno do browser. Usando o DOM, o JavaScript consegue examinar e modificar dinamicamente qualquer página WEB e ainda examinar e modificar o estado do browser.

DOM é um padrão controlado pela organização W3C - World Wide Web Consortium, a mesma organização que controla o padrão HTML, XML e muitos outros padrões da WEB.

Para usar JavaScript integrado numa página WEB é necessário saber um pouco de HTML. Nas nossas aulas vamos restinguir-nos à programação de HTML Forms.

Exemplo 1

O seguinte exemplo é uma form simples que permite somar números. A form é constituída por três caixas de texto, a terceira das quais é *read-only*, e por um botão. Por favor, brinque um pouco com a form para perceber o seu comportamento.

Adder

1.2 + 3.4 = 4.6

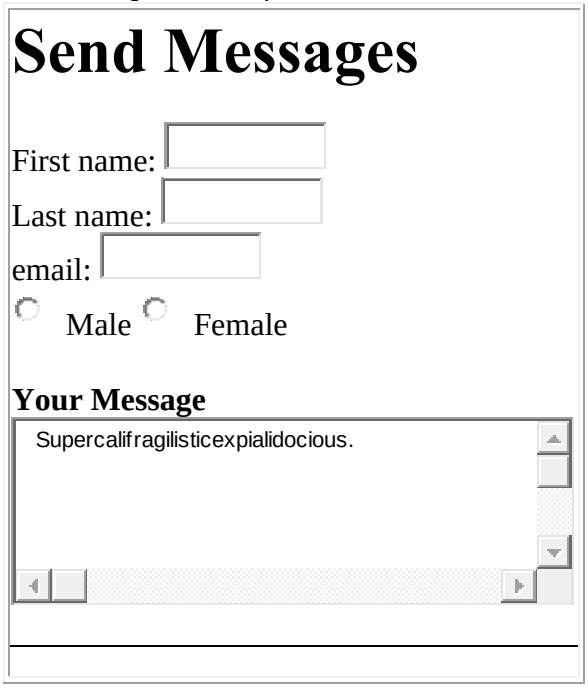
Agora examine o código HTML e JavaScript que implementa a form anterior:

```
<HTML>

<HEAD>
<TITLE>MyDocument</TITLE>

<SCRIPT TYPE="text/javascript">
function Compute(n1, n2) {
```

```
    return n1 + n2 ;
}
function RunForm1(form) {
    form.text3.value = Compute(parseFloat(form.text1.value),
parseFloat(form.text2.value)) ;
}
</SCRIPT>
</HEAD>

<BODY>
<H1>Adder</H1>
<FORM NAME="form1">
     <INPUT TYPE="text" NAME="text1" VALUE="1.2" SIZE=10 style="{ text-align: right }">
   + <INPUT TYPE="text" NAME="text2" VALUE="3.4" SIZE=10 style="{ text-align: right }">
   = <INPUT TYPE="text" NAME="text3" VALUE="4.6" SIZE=25 READONLY style="{ text-align:
right; font-weight : bold }">
 <p> <INPUT TYPE="button" NAME="button1" VALUE="Add" OnClick="RunForm1(form)">
</FORM>
</BODY>

</HTML>
```

Exemplo 2

O seguinte exemplo é uma *form* contendo botões de rádio e uma área de texto. Por favor, teste a form.



Eis o código HTML e JavaScript correspondentes:

```
<HTML>

<HEAD>
<TITLE>MyDocument</TITLE>

<SCRIPT>
function RunForm2(form) {
    var msg = "Sent by: " ;
    msg += form.firstname.value + " " + form.lastname.value ;
    msg += " (" + form.email.value + ") " ;
    if( form.sex1.checked )
        msg += " [male]" ;
    else if( form.sex2.checked )
        msg += " [female]" ;
    else
        msg += " []" ;
    msg += "\nMesg: " + form.textarea1.value ;
    alert(msg) ;
```

```
}
</SCRIPT>
</HEAD>

<BODY>
<H1>Send Messages</H1>
<FORM NAME="form2">
    First name: <INPUT TYPE="text" NAME="firstname"><BR>
    Last name: <INPUT TYPE="text" NAME="lastname"><BR>
    email: <INPUT TYPE="text" NAME="email"><BR>
    <INPUT TYPE="radio" NAME="sex1" VALUE="Male" OnClick="sex2.checked=false"> Male
    <INPUT TYPE="radio" NAME="sex2" VALUE="Female" OnClick="sex1.checked=false">
Female<BR>
    <P><B>Your Message</B><BR>
    <TEXTAREA NAME="textarea1" ROWS="5"
COLS="50">Supercalifragilisticexpialidocious.</TEXTAREA>
    <P><INPUT TYPE="button" NAME="button1" VALUE="Send" OnClick="RunForm2(form)">
    <INPUT type="reset">
</FORM>
</BODY>

</HTML>
```

Exemplo 3

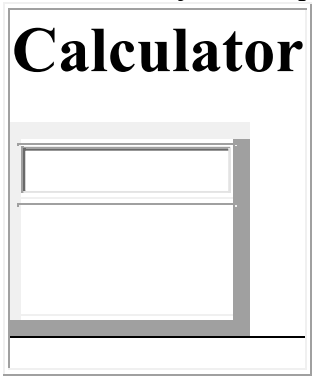
O seguinte exemplo mostra como se pode carregar um novo URL, mudando assim completamente o conteúdo do documento corrente. Também ilustra a criação duma caixa de alerta.

Eis o código HTML e JavaScript correspondentes:

```
<HTML>

<HEAD>
<TITLE>MyDocument</TITLE>

<SCRIPT>
function RunForm3(form) {
    var url = form.url.value ;
    if( url == "" )
        alert("Error: Empty URL") ;
    else
        document.location = form.url.value ;   // Change URL
}
</SCRIPT>
</HEAD>

<BODY>
<H1>Goto WEB page</H1>
<FORM NAME="form3">
    Enter URL: <INPUT TYPE="text" NAME="url" SIZE=40>
    <P><INPUT TYPE="button" NAME="button1" VALUE="Goto" OnClick="RunForm3(form)">
    <INPUT type="reset">
</FORM>
</BODY>

</HTML>
```

Exemplo 4

O seguinte exemplo serve para ilustrar a operação eval do JavaScript. Como já foi dito antes, todas as implementações de JavaScript devem ser capazes de interpretar código gerado dinamicamente. Isso pressupõe à existência duma função `eval` para correr código arbitrário.



Eis o código HTML e JavaScript que implementa a calculadora. A implementação é verdadeiramente simples. Recria no mostrador da calculadora, a pouco e pouco, a expressão introduzida pelo utilizador. No final, quando o utilizador carrega na tecla '=', invoca a função `eval` para avaliar a expressão recebida, sob a forma de string. Este truque funciona bem porque as expressões aritméticas do JavaScript têm a sintaxe habitual da matemática.

```
<HTML>

<HEAD>
<TITLE>MyDocument</TITLE>
</HEAD>

<BODY>
<H1>Calculator</H1>
<FORM NAME="form4">
    <TABLE BORDER=10>
    <TR><TD>
        <INPUT TYPE="text" NAME="display" Size="18">
    </TD></TR>
    <TR><TD>
        <INPUT TYPE="button" NAME="1" VALUE="1" OnClick="form.display.value += '1'">
        <INPUT TYPE="button" NAME="2" VALUE="2" OnCLick="form.display.value += '2'">
        <INPUT TYPE="button" NAME="3" VALUE="3" OnClick="form.display.value += '3'">
        <INPUT TYPE="button" NAME="+" VALUE="+" OnClick="form.display.value += '+'">
    <br>
        <INPUT TYPE="button" NAME="4" VALUE="4" OnClick="form.display.value += '4'">
        <INPUT TYPE="button" NAME="5" VALUE="5" OnCLick="form.display.value += '5'">
        <INPUT TYPE="button" NAME="6" VALUE="6" OnClick="form.display.value += '6'">
        <INPUT TYPE="button" NAME="-" VALUE="-" OnClick="form.display.value += '-'">
    <br>
        <INPUT TYPE="button" NAME="7" VALUE="7" OnClick="form.display.value += '7'">
        <INPUT TYPE="button" NAME="8" VALUE="8" OnCLick="form.display.value += '8'">
        <INPUT TYPE="button" NAME="9" VALUE="9" OnClick="form.display.value += '9'">
        <INPUT TYPE="button" NAME="*" VALUE="*" OnClick="form.display.value += '*'">
    <br>
        <INPUT TYPE="button" NAME="C" VALUE="C" OnClick="form.display.value = ''">
        <INPUT TYPE="button" NAME="0" VALUE="0" OnClick="form.display.value += '0'">
        <INPUT TYPE="button" NAME="=" VALUE="=" OnClick="form.display.value =
eval(form.display.value)">
        <INPUT TYPE="button" NAME="/" VALUE="/" OnClick="form.display.value += '/'">
    <br>
    </TD></TR>
    </TABLE>
</FORM>
</BODY>

</HTML>
```

Teórica 26 (25/Mai/2011)

Linguagens de scripting. A linguagem Bash.

Tudo sobre a linguagem JavaScript.

Linguagens de scripting

As **linguagens de programação clássicas** são concebidas para criar estruturas de dados e algoritmos a partir do zero, usando os elementos primitivos da linguagem. Exemplos: ML, C, C++, Java.

Em constaste, nas **linguagens de scripting** assume-se que já existe uma coleção, pronta a ser usada, de componentes escritas noutras linguagens. As linguagens de scripting são concebidas para permitir ligar e organizar componentes existentes **de forma simples, expedita e flexível**. Exemplos: JavaScript, Bash, Tcl, Perl, PHP, Ruby, Python.

Chama-se **script** a um programa escrito numa linguagem de scripting.

Linguagens clássicas versus linguagens de scripting

As linguagens clássicas são geralmente estaticamente tipificadas para permitir detetar cedo os erros que podem ocorrer quando se utilizam elementos primitivos na construção de estruturas de dados complexas. Em constaste, as linguagens de scripting são sempre tipificadas dinamicamente pois um sistema de tipos estático seria uma complicação burocrática que só serviria para atrasar a escrita de scripts.

Outra diferença é o facto das linguagens de scripting serem geralmente interpretadas e não compiladas. Um dos objetivos disso é acelerar o desenvolvimento do código. Outro objetivo é facilitar a geração dinâmica de scripts que possam ser executados imediatamente pelo interpretador.

Nas linguagens de scripting dá-se pouca importância à questão eficiência. Em contrapartida dá-se a máxima importância à simplicidade, flexibilidade de utilização e a um grande poder expressivo que permita escrever scripts compactos. A questão da eficiência é pouco importante porque os scripts tendem a ser pequenos e porque a eficiência da linguagem é dominada pela eficiência das componentes, as quais são normalmente implementadas numa linguagens de programação clássica.

Algumas linguagem de scripting são também concebidas para serem usadas no interior duma aplicação de software, com o objetivo de fornecer ao utilizador um elevado grau de controlo do comportamento da aplicação, incluindo a adição de novas funcionalidades. Se é verdade que o utilizador não pode alterar o código de base da aplicação, ele pode escrever scripts para adaptar a aplicação às suas necessidades. Exemplos: Emacs Lisp é a linguagem de scripting do editor de texto emacs; JavaScript é linguagem de scripting mais usada nos browsers da WEB.

Como escolher entre linguagem clássicas e de scripting

Quando estão em causa aplicações que envolvem acima de tudo a coordenação de componentes já implementadas, podemos escolher programar essa aplicação numa linguagem clássica, e.g. Java, ou numa linguagem de scripting, e.g. JavaScript. Mas estudos mostram que se escolhermos uma linguagem de scripting, tanto o tempo de desenvolvimento como o tamanho da aplicação se reduzem num fator de 5 a 10, em média!

Quando estão em causa aplicações com algoritmos e estruturas de dados complexas, o melhor é usar uma linguagens de programação clássica. Usando uma linguagem de scripting, o script não ficaria mais pequeno, não haveria o benefício da tipificação estática para ajudar a apanhar antecipadamente erros subtis na utilização das estruturas de dados e, no final, o programa correria 10 vezes mais devagar.

Complementaridade dos dois tipos de linguagens

Se puderem ser usadas em conjunto, os dois tipos de linguagens permitem a criação de ambientes de desenvolvimento e execução de programas particularmente poderosos e flexíveis.

As linguagens de scripting sempre tiveram alguma popularidade, mas ultimamente a sua importância tem aumentado. A principal razão é a tendência atual para escrever aplicações baseadas em componentes já disponíveis. É o que se passa, por exemplo, quando está em causa o desenvolvimento de interfaces gráficas e de aplicações que correm sobre a WEB.

Exemplo: linguagem Bash

Bash (bourne-again shell) é uma linguagem de scripting muito usada no Linux na qual as "componentes" são as aplicações disponíveis. Em Bash, um script implementa uma nova funcionalidade usando as aplicações do sistema. Uma das construções mais importantes do Bash é o **pipe**, que permite ligar o output duma aplicação ao input de outra aplicação. Em Bash também é possível testar o código se saída duma aplicação e tomar decisões em conformidade (zero significa que a aplicação terminou sem erro; um valor diferente de zero representa um código de erro particular). Também é possível fazer é recolher o output duma aplicação numa variável e processar o conteúdo da variável a seguir.

Eis um exemplo de script em bash, retirado daqui. Este script lista na consola o nome de todos os ficheiros HTML que se encontram na diretoria corrente e, além disso, escreve a primeira linha de cada um desses ficheiros num ficheiro chamado File_Heads.

```
#!/bin/sh
# This is a comment
echo "List of files:"
ls -lA

FILE_LIST="`ls *.html`"
echo FILE_LIST: ${FILE_LIST}

RESULT=""
for file in ${FILE_LIST}
do
    FIRST_LINE=`head -1 ${file}`
    RESULT=${RESULT}${FIRST_LINE}
done

echo ${RESULT} | cat > FILE_HEADS

echo "'$RESULT' written Script done. "
```

No Windows, a linguagem de scripting chama-se PowerShell e foi introduzida em 2006 na versão Windows XP SP2. Antes do PowerShell usava-se a linguagem de scripting implementada pelo programa COMMAND.COM.

Comparação do JavaScript com o Java

A primeira é uma linguages de scripting. A segunda é uma linguagem clássica.

JavaScript	**Java**
Tipificação dinâmica	Tipificação estática
Baseada em protótipos	Baseada em classes
Herança usando o mecanismo dos protótipos	Herança através da hierarquia de classes

Podem ser adicionados novos membros a objetos individuais	Não é possível a adição dinâmica de novos membros
Estilo livre onde a maioria das declarações são opcionais	Estilo rígido a pensar na segurança

---

Palavras reservadas do JavaScript

Esta é a lista das principais palavras reservadas em JavaScript:

```
break     const      delete    for       import    new       this      void
case      continue   do        function  in        return    typeof    while
default   export     if        else      switch    var       with
```

Muitas das palavras anteriores são também palavras reservadas em Java.

As restantes palavras reservadas do Java também estão reservadas em JavaScript, apesar de não serem usadas de momento. Todos os interpretadores de JavaScript deveriam proibir a utilização destas palavras. Contudo alguns não o fazem.

Variáveis

O JavaScript é uma linguagem **dinamicamente tipificada**. Uma consequência disso é o facto das **variáveis não terem tipos associados**. Os tipos ficam associados aos os valores e não variáveis. Exemplo:

```
var x = 34 ;
x = "Hello!" ;   // x pode conter um valor de qq tipo
```

Declaração de variáveis

A palavra `var` permite declarar variáveis:

- Dentro duma função, `var` declara uma **variável local**.
- O **argumento duma função**, também declara implicitamente uma variável local.
- Fora duma função, `var` declara uma **variável global**.
- Usa-se **escopo estático** na resolução de nomes.
- Uma função declarada dentro dum bloco têm como âmbito toda a função envolvente, ou seja, **não há âmbito de bloco**.

Exemplo com variáveis globais e variáveis locais:

```
var x = 5 ;
var z = 7 ;

function f(x) {
    print(x) ;
    print(y) ;   // Não inicializada ainda
    x = 1 ;
    var y = 2 ;
    print(x) ;
    print(y) ;
    print(z) ;
    return x ;
}

f(6) ;
```

```
print(x) ;

// Output: 6  undefined  1  2  7  5
```

Exemplo com aninhamento de funções:

```
function f(x) {
    function g(y) {
        var x = 10 ;
        print(y) ;
        print(x) ;    // Imprime o x local
        return 0 ;
    }
    print(x) ;
    g(1) ;
    print(x) ;
}

f(6) ;

// Output: 6  1  10  6
```

Exemplo relativo ao facto de não existir âmbito de bloco:

```
function f() {
    var x = 1 ;
    {
       var x = 2 ;
    }
    print(x) ;     // Escreve 2
}
```

Variáveis indefinidas

Variáveis declaradas num determinado âmbito, ficam com o valor `undefined` enquanto não forem inicializadas.

Exemplos sobre variáveis indefinidas:

```
print(zzz) ;   // Lança a exceção ReferenceError se zzz nao estiver declarada
var zzz ;
if( zzz === undefined )   // Pode testar-se se uma variável está indefinida.
    print("zzz is undefined") ;
if( !zzz )     // undefined comporta-se como false num contexto booleano
    print("zzz is undefined") ;
```

Atribuição a variáveis

A atribuição efetua-se usando o operador =. É possível efetuar uma atribuição a um nome ainda não definido. Nesse caso é automaticamente criada uma variável global inicializada.

```
x = 5 ;
y = 7 + 5 ;
```

Constantes

A palavra `const` permite declarar constantes:

```
const x = 5 ;
```

Tipos primitivos

Os tipos primitivos são os seguintes:

- **number** - Mistura reais e inteiros. O maior valor é 1.7976931348623157e+308. 0377 é um valor em octal e 0xFF é um valor em hexadecimal.
- **boolean** - Tem os valores `false` e `true`.
- **string** - Por exemplo `""`, `''`, `"Hello"`, `'Hello'`, `"inner 'string' "`, `'inner "string" '`.
- **null** - Este tipo só tem o valor `null` e serve para atribuir a uma variável para indicar que esta não tem valor.
- **undefined** - Este tipo só tem o valor `undefined`, que é valor das variáveis não inicializadas.

Note que em JavaScript, uma string não é um objeto. No entanto também há objetos de tipo String que simplesmente encapsulam valores primitivos de tipo string. Em Javascript as strings são imutáveis, tal como em Java.

O operador `typeof` pode ser usado para saber o tipo de qualquer valor.

São efetuadas **conversões automáticas de tipo**, entre os tipos primitivos. Exemplos:

```
"The answer is " + 42        // produz "The answer is 42"
42 + " is the answer"        // produz "42 is the answer"
"37" - 7                     // produz 30
"37" + 7                     // produz "377"
true + 7                     // produz 8
```

Operadores

Eis a tabela de operadores do JavaScript ordenada por prioridade decrescente:

```
member                    .  []
call / create instance    ()  new
negation/increment        !  ~  -  +  ++  --  typeof  void  delete
multiply/divide           *  /  %
addition/subtraction      +  -
bitwise shift             <<  >>  >>>
relational                <  <=  >  >=  in  instanceof
equality                  ==  !=  ===  !==
bitwise-and               &
bitwise-xor               ^
bitwise-or                |
logical-and               &&
logical-or                ||
conditional               ?:
assignment                =  +=  -=  *=  /=  %=  <<=  >>=  >>>=  &=  ^=  |=
comma                     ,
```

Operadores de igualdade

```
==          Igualdade, produz true se os argumentos forem iguais (após possíveis
conversões automáticas de tipo).
!=          Desigualdade, produz true se os argumentos forem diferentes.
===         Igualdade estrita, produz true se os argumentos forem iguais e do mesmo tipo.
!==         Desigualdade estrita, produz true se os argumentos forem diferentes ou se
forem de tipos diferentes.
```

Expressões regulares

O JavaScript suporta expressões regulares semelhantes às da linguagem Perl.

A seguinte expressão regular representa dois "a"s seguidos de zero ou mais dígitos:

```
re = /aa\d*/ ;
re = new RegExp("aa\\d") ;          // Equivalente
```

A próxima expressão regular, mais abaixo na caixa, representa um "d" seguido de um ou mais "b"s seguido dum "d". As flags "i" e "g" indicam que o emparelhamento deve ignorar a caixa das letras e que deve ser global.

O método `test` determina se uma string emparelha com a expressão regular.

O método `exec` produz um array com o resultado do emparelhamento na posição 0 do array, mais os resultados dos emparelhamentos das sub-expressões entre parêntesis. Se a expressão regular tiver a flag "g" ligada, então sucessivas chamadas de `exec` produzem sucessivos resultados de emparelhamentos até ser retornado `null`; sem a flag "g" apenas o resultado do primeiro emparelhamento é retornado.

```
var re = /d(b+)(d)/ig ;
re = new RegExp("d(b+)(d)", "ig") ;        // Equivalente
var b = re.test("cdbBdbsbz") ;             // resultado: true
var arr = re.exec("cdbBdbsdbdz") ;         // primeiro resultado: ["dbBd", "bB", "d"]
```

As expressões regulares suportam ainda os métodos `match`, `search`, `replace`, `split`.

Arrays

Em JavaScript os arrays podem ser inicializados, pelo menos de duas maneiras diferentes. Exemplo:

```
var colors = ["Red", "Green", "Blue"] ;
var colors = new Array("Red", "Green", "Blue") ;    // Equivalente
```

Tal como em Java os índices começam em zero e existe uma propriedade `length`.

```
var len = colors.length ;                           // Vale 3
```

Eis um exemplo dum array de comprimento 6 com apenas 4 elementos. Dois elementos estão indefinidos.

```
var colors = ["Red", , , "Green", "Blue","Yellow"] ;
```

Ao contrário do Java, os arrays crescem automaticamente. Basta atribuir a uma posição inexistente para o array crescer.

```
var colors = [] ;                                   // Array vazio
colors[2] = "Blue" ;
var len = colors.length ;                           // Vale 3
var t = typeof(colors[0]) ;                         // Vale undefined
```

Para fazer crescer um array na primeira posição livre, fazer assim:

```
colors[colors.length] = "Yellow" ;
colors.push("Yellow") ;                           // Equivalente
```

Para aceder e remover o último elemento dum array fazer:

```
var last = colors.pop() ;
```

É possível escrever diretamente na propriedade `length` dum array para fazer um array crescer, ou para truncar o array:

```
colors.length = 2 ;
```

Para percorrer os elementos dum array pode usar-se um `for`, mas também se pode fazer assim:

```
var colors = ["Red", "Green", "Blue"] ;
colors.forEach(function(c) { print(c) ; }) ;      // Iteração usando função anónima
```

... ou assim:

```
var colors = ["Red", "Green", "Blue"] ;
for( i in colors ) print(colors[i]) ;         // Iteração usando for..in
```

Eis um array a duas dimensões, 2x3:

```
var table = [[0, 1, 2],
             [3, 4, 5]] ;
var r = table[0][2] ;                                 // Vale 2
```

Outros métodos disponíveis para arrays: `join`, `reverse`, `shift`, `slice`, `splice`, `sort`, e muitos outros.

Funções

O JavaScript suporta o paradigma de programação funcional pois inclui funções anónimas, funções de ordem superior e funções que retornam outras funções.

```
function square(n) { return n * n ; }
var square = function(n) { return n * n ; } ;         // Equivalente
```

Eis um exemplo duma função de ordem superior, que depois é chamada usando uma função anónima como argumento:

```
function map(f, a) {
    var result = [] ;
    for( var i = 0 ; i < a.length ; i++ )
        result[i] = f(a[i]) ;
    return result;
}

var a = map(function(x) { return x * x ;}, [0, 1, 2, 3]) ;   // Vale [0, 1, 4, 9]
```

Nas chamadas das funções o número de argumentos não é validado: argumentos a mais na chamada são ignorados; argumentos a menos na chamada ficam indefinidos.

Dentro da cada função há um array predefinido chamado `arguments` que representa a sequência de argumentos realmente usados na chamada. Assim é fácil implementar funções com um número variável de argumentos, como no seguinte exemplo:

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

var i = allAll(1,2,3,4,5) ;                       // Vale 15
```

A passagem de argumentos de tipos primitivos é feita por valor. Os objeto-argumento são passados por referência.

Eis algumas funções predefinidas em JavaScript:

- **eval(string)** - Avalia uma string contendo código JavaScript.
- **isFinite(number)** - Testa se um número é finito.
- **isNaN(number)** - Teste se um número é a constante NaN.
- **parseInt(string, radix)** - Converte string em número inteiro.
- **parseFloat(string)** - Converte string em número real.

- **Number(obj)** - Converte um objeto num número.
- **String(obj)** - Converte um objeto numa string.

Objetos

Em JavaScript para além dos tipos primitivos, temos os tipos objeto. Os arrays são considerados objetos.

Como habitualmente, um objeto é um elemento de dados que possui identidade e que interage com outros objetos através da troca de mensagens.

Em JavaScript os objetos predefinidos principais são os seguintes: Date, Array, Boolean, Function, Math, Number, RegExp e String . Mas no ambiente de execução envolvente, estão geralmente disponíveis muitos mais objetos predefinidos. Por exemplo, no ambiente dum browser, todos os tipos de objetos previstos no DOM estão disponíveis: Document, Window, Form, Link, etc.

Objetos literais

Em JavaScript, os objetos comportam-se em grande medida como simples dicionários. Eis um exemplo de **objeto literal**, que define uma pessoa:

```
var p = {name: "Pedro", address: "Lisboa", age: 42} ;
```

Para aceder a uma componente dum objeto, há duas notações disponíveis:

```
var n = p.name ;
var n = p["name"] ;                    // Equivalente
p.name = "Pedrinho" ;                  // Muda nome
```

Se atribuirmos a um membro inexistente dum objeto, esse membro passa imediatamente a existir para esse objeto individual:

```
p.born = "Porto" ;
```

Para apagar um membro, usa-se a palavra `delete`:

```
delete p.born ;
```

Eis um objeto mais complexo:

```
var myStructure = {
  name: {
    first: "Mel",
    last: "Smith"
  },
  age: 33,
  hobbies: [ "chess", "jogging" ]
} ;
```

Construtores

Os objetos literais são muito úteis, mas não são suficientes quando é preciso definir diversos objetos do mesmo tipo: por exemplo, para partilha dos mesmos métodos, ou para efeitos de utilização do operador `instanceof`.

A definição de novos **tipos-objeto** faz-se através de **construtores**. Um construtor é uma função JavaScript que se preocupa em inicializar objetos dum dado tipo. A única particularidade determinante dum construtor é a seguinte:

- O construtor é uma função que se destina a ser chamada no contexto do operador `new`.

Dentro do construtor usa-se o nome `this` para inicializar os membros de cada objeto. O nome do construtor acaba por ser também o nome de tipo que pode ser usado do lado direito do operador `instanceof`.

Abaixo define-se um construtor chamado Car. Os métodos do objeto são membros funcionais atribuídos dentro do construtor. Note que os métodos usam a palavra `this` para referir o objeto a que são aplicados.

```
function carAsString() {
   return "A Beautiful " + this.year + " " + this.make + " " + this.model ;
}

function Car(make, model, year) {
   this.make = make ;
   this.model = model ;
   this.year = year ;
   this.asString = carAsString ;    // Também se podia ter usado uma função anónima
}

var car1 = new Car("Toyota", "Corolla", 2002) ;
```

Um objeto definido através dum objeto literal, considera-se que foi definido usando o construtor Object.

Modificação dinâmica dum tipo-objeto (Herança dinâmica)

Um tipo-objeto tem um membro chamado `prototype` através da qual é possível alterar a funcionalidade desse tipo, e consequentemente de todos os objetos desse tipo.

```
car1.changeMake("TTT") ;                                  // Gera um erro
Car.prototype.changeMake = function(make) { this.make = make ; } ;
car1.changeMake("TTT") ;                                 // Agora já não gera um erro
car1.asString() ;                                        // Permite confirmar que a marca
mudou
```

O código anterior funciona porque **todos os objeto herdam dinamicamente do respetivo protótipo**. Quando se tenta aceder a um membro dum objeto, se esse membro não estiver diretamente disponível no objeto, então a procura continua no protótipo.

O protótipo correspondente a um construtor é criado quando se chama o construtor pela primeira vez. Dessa primeira vez, além do protótipo é criado ainda um objeto normal; a partir daí só são criados objetos normais.

Usando esta técnica é possível alterar inclusivamente os membros dos tipos-objeto predefinidos, sendo por exemplo possível adicionar novos métodos ao tipo Array.

Criação de hierarquias

Manipulando diretamente o membro `prototype` de um ou mais tipos-objeto é possível criar uma hierarquia de tipos com herança dinâmica.

No seguinte exemplo definem-se dois tipos-objeto independentes, `Super` e `Sub`. Depois atribui-se uma instância de `Super` a `Sub.prototype`. Na prática isto tem o seguinte efeito: Quando se tenta aceder a um membro dum objeto de tipo Sub, se esse membro não estiver diretamente disponível no objeto, então a procura continua no respetivo protótipo, que agora é um objeto de tipo Super. Se também não estiver aí definido então a busca prossegue no protótipo desse objeto, ou seja em `Super.prototype`.

Usando esta ideia, vê-se que é fácil criar uma hierarquia de tipos, com tantos níveis quanto se deseje.

```
function Super() {
   this.p = function() { print("super") ; }
   this.q = function() { print("super") ; }
```

```
}

function Sub() {
   this.q = function() { print("sub") ; }
}

Sub.prototype = new Super() ;   // Alteração explícita do protótipo para efeitos de
herança

var a = new Super() ;
var b = new Sub() ;

a.p() ;                         // Escreve "super"
a.q() ;                         // Escreve "super"
b.p() ;                         // Escreve "super" - este método foi herdado
b.q() ;                         // Escreve "sub"
```

Dúvida que surgiu no final da aula

Será que os objetos normais têm um campo `prototype`?

O campo `prototype` está definido apenas nos tipo-objeto, estando disponível para ser consultado e alterado, como vimos.

Pode testar-se se um objeto herda dum determinado protótipo usando o operador `instanceof`.