**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.
* O grande objetivo inicial do Java e depois do JavaScript foi adicionar interatividade a páginas WEB. O JavaScript foi concebido para a execução de tarefas simples, para as quais o Java era demasiado pesado. Contudo, por várias razões, o JavaScript é usado atualmente em projetos grandes e de formas que ultrapassam as intenções originais. As características da linguagem que a tornam fácil de usar acabam por se revelar problemáticas nessas situações. (Votaremos a este assunto numa próxima aula).
* 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 que foi padronizada pela organização ECMA em 1997 e esse padrão tem evoluído: a versão atual é designada por 5.1. Há diferentes dialetos desse standard ECMA que, para além do núcleo padronizado, suportam diversas extensões. Para além do JavaScript, existe o JScript, o ActionScript, etc.
* A especificação ECMA diz que a linguagem deve poder ser interpretada, por forma a que seja possível correr scripts gerados dinamicamente.
* O corpo das funções tem uma sintaxe semelhante à do Java e outros 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 programação funcional, e ao nível dos objetos baseia-se nas ideias invulgares da linguagem Self. 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. Só por si, a linguagem nem sequer possui operações de entrada (input) ou de saida (output).
* 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.

**Versões do JavaScript**

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

* [SpiderMonkey](http://en.wikipedia.org/wiki/SpiderMonkey) - Escrito em C.
* [Rhino](http://en.wikipedia.org/wiki/Rhino_%28JavaScript_engine%29) - Escrito em Java para permitir usar scripting dentro de programas Java. Serviu de base ao novo package javax.script, introduzido no Java 6.0.

Outra implementação que importa referir é a seguinte:

* [node.js](http://en.wikipedia.org/wiki/Node.js) - Escrito pela Google e usado no Google Chrome. Muito rápido.

O JavaScript foi padronizado em 1997. O padrão chama-se ECMA-262, sendo ECMAScript o nome oficial da linguagem. No entanto, a linguagem continua a ser conhecida informalmente como JavaScript.

* JavaScript 1.0 - 1995
* [JavaScript 1.1](http://developer.mozilla.org/en/docs/New_in_JavaScript_1.1) - 1996
* ECMAScript 1 - 1997
* ECMAScript 2 - 1998
* ECMAScript 3 - 1999 - Introduz expressões regulares e exceções
* ECMAScript 5 - 2009 - Introduz JSON, geradores, iteradores, *compreensões* usando arrays, expressões let, etc.
* ECMAScript 5.1 - 2010
* ECMAScript 6 - 2015 - Introduz classes e modulos.
* ECMAScript 7 - 2016 - Introduz Array.prototype.includes.

**Bibliotecas e frameworks**

Existem bibliotecas e frameworks que ajudam a escrever programas JavaScript sofisticados, com interfaces ambiciosas. Eis alguns das mais usados:

* [JQuery](http://en.wikipedia.org/wiki/JQuery)
* [Prototype Framework](http://en.wikipedia.org/wiki/Prototype_JavaScript_Framework)
* [script.aculo.us](http://en.wikipedia.org/wiki/Script.aculo.us)

**Porque vamos trabalhar com a linguagem JavaScript?**

Na nossa cadeira, vamos estudar e trabalhar com a linguagem JavaScript por diversas razões. Eis as principais:

* Para tomarmos contacto com uma linguagem de scripting, para percebermos as características destas linguagens, e para sabermos quais as situações em que é vantajoso usá-las.
* Para estudarmos uma linguagem com um sistema de tipos dinâmico.
* Para estudarmos uma linguagem em que o sistema de objetos é baseado em protótipos e não em classes.
* Para revisitarmos o paradigma orientado pelos objetos.
* Para fazermos uma pequena introdução à programação Web do lado do cliente.

**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 Rhino corre num ambiente que fornece [diversas primitivas](https://developer.mozilla.org/en-US/docs/Rhino/Shell), das quais destacamos a função print.

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]  h  js> "hello, world".replace("hello", "goodbye")  goodbye, world  js> x = 12.4 + "144"  12.4144  js> typeof(x)  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  $ |

Com apenas duas exceções, as expressões anteriores também funcionam no "JavaScript Shell" (disponível na coluna da esquerda da página da cadeira) e também na consola do "Firebug", uma extensão do Firefox que pode ser instalada a partir do site http://getfirebug.com.

**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](http://en.wikipedia.org/wiki/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 [formulários HTML](http://en.wikipedia.org/wiki/Html_form)(HTML forms).

Convém saber que há incompatibilidades entre diferentes browsers em diferentes plataformas. Conseguir fazer um programa em JavaScript que funcione em todos os browser existentes é uma tarefa complicada.

**Exemplo 1**

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

|  |
| --- |
| **Adder**  Parte superior do formulário   +  =   Parte inferior do formulário |

Agora examine o código HTML e JavaScript que implementa o formulário anterior. Repare que no botão "Add" o atributo ONCLICK tem associado código que será executado sempre que o botão gerar um **evento de click**.

|  |
| --- |
| **<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 o formulário.

|  |
| --- |
| **Send Messages**  Parte superior do formulário  First name:  Last name:  email:   Male  Female  **Your Message**    Parte inferior do formulário |

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.

|  |
| --- |
| **Goto WEB page**  Parte superior do formulário  Enter URL:     Parte inferior do formulário |

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](http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Global_Functions:eval) do JavaScript. Como já foi dito antes, todas as implementações de JavaScript devem ser capazes de interpretar código gerado dinamicamente. Isso pressupõe a existência duma função eval para correr código JavaScript arbitrário.

|  |  |  |
| --- | --- | --- |
| **Calculator**  Parte superior do formulário   |  | | --- | |  | |  |   Parte inferior do formulário |

Eis o código HTML e JavaScript que implementa a calculadora. A implementação é verdadeiramente simples. Constrói no mostrador da calculadora, a pouco e pouco, a expressão introduzida pelo utilizador sob a forma de string. No final, quando o utilizador carrega na tecla '=', invoca a função eval para avaliar a o conteúdo da 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="14">  </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>** |

**Exemplo 5**

O seguinte exemplo ilustra o uso dum [**canvas**](http://pt.wikipedia.org/wiki/Canvas_%28HTML5%29) em JavaScript. Um canvas é simplesmente um retângulo onde se podem desenhar gráficos.

|  |
| --- |
| **Example of Canvas** |

Examine o código HTML e JavaScript que implementa o retângulo anterior.

O preenchimento do canvas só pode ser feito depois do canvas ter sido criado. Uma forma de fazer isso é colocar o código JavaScript nas linhas imediatamente a seguir à definição do canvas. Mas é considerado pouco elegante misturar código JavaScript com código HTML no body da página. É preferível colocar o código JavaScript dentro do header da página, como temos feito sempre, e invocar esse código depois da página ter sido carregada. Existe um **evento de onload** que é ativado quando a página é carregada. Esse evento é capturado usando o atributo ONLOAD no elemento BODY.

|  |
| --- |
| **<HTML>**  **<HEAD>**  <TITLE>Canvas</TITLE>  <SCRIPT>  function fillCanvas() {  var c=document.getElementById("canvas1");  var ctx=c.getContext("2d");  ctx.fillStyle="#FF0000";  ctx.fillRect(0,0,150,75);  ctx.moveTo(0,0);  ctx.lineTo(200,100);  ctx.stroke();  ctx.beginPath();  ctx.arc(95,50,40,0,2\*Math.PI);  ctx.stroke();  ctx.fillStyle="#00FF00";  ctx.font="30px Arial";  ctx.fillText("Hello World",10,50);  }  </SCRIPT>  **</HEAD>**  <**BODY** ONLOAD="fillCanvas()">  <H1>Example of Canvas</H1>  <CANVAS id="canvas1" width="200" height="100" style="border:1px solid #000000;">  This text is displayed if your browser does not support HTML5 Canvas  </CANVAS>  <**/BODY**>  <**/HTML**> |

**Exemplo 6**

Este exemplo mostra como se pode mudar um pedaço de texto arbitrário dentro duma página Web.

Exemplo: blá blá  0 blá blá.

Requer um pouco de planeamento, porque precisamos primeiro de associar um ID ao texto a alterar.

Se o texto já estiver marcado de alguma forma, por exemplo com a marca de bold assim **<b>o meu texto</b>**, então basta adicionar o ID desta forma, **<b id='o meu id'>o meu texto</b>**.

Se o texto não estiver marcado, então temos de o marcar com a marca **span** e adicionar o ID desta forma, **<span id='o meu id'>o meu texto</span>**.

Agora que o texto já está marcado e tem um ID, para lhe aceder em JavaScript basta usar a expressão **document.getElementById('o meu id').innerHTML**.

Veja o código do exemplo:

|  |
| --- |
| Exemplo: blá blá  <**INPUT** **TYPE**="button"  **VALUE**="Click Me"  **ONCLICK**="var el = document.getElementById('txt0'); el.innerHTML = parseInt(el.innerHTML,10) + 1;"  >  <**SPAN** id='txt0'>0</**SPAN**>  blá blá. |

**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.

Por exemplo, quando se usa JavaScript para adicionar interatividade a uma página Web, as componentes em causa, são os objetos DOM que representam essa página Web.

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 e o sistema de ficheiros. 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-resultado 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 colocar o output duma aplicação numa variável e processar a seguir o conteúdo dessa variável.

Eis um exemplo de script em bash, retirado [daqui](http://www.felixgers.de/teaching/shells/bash_script.html). 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/bash  # 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}${FIRST\_LINE}$'\n'  done  echo ${RESULT} | cat > FILE\_HEADS  echo "${RESULT}"  echo "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.

**Discussão**

As linguagens de scripting são geralmente concebidas para a execução de tarefas simples. Contudo, por vezes acabam por ser usadas em projetos grandes e de formas que ultrapassam as intenções originais. As características da linguagem que a tornam fácil de usar acabam por se revelar problemáticas nessas situações.

Quando comunidade de utilizadores é grande, então justifica-se fazer o seguinte:

* Criar ferramentas que ajudam a detetar mais erros prematuramente e/ou a aumentar a performance. Exemplos: [JSLint](http://en.wikipedia.org/wiki/JSLint), [Google Closure Tools](http://en.wikipedia.org/wiki/Google_Closure_Tools).
* Começa a ser melhorada a eficiência da implementação usando técnicas avançadas: [V8 JavaScript Engine](http://en.wikipedia.org/wiki/V8_(JavaScript_engine)).

**Elementos da linguagem JavaScript**

Regressemos ao estudo da linguagem JavaScrip, desta vez para apresentar os elementos de base mais importantes da linguagem.

**Comparação do JavaScript com o Java**

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

|  |  |
| --- | --- |
| **JavaScript** | **Java** |
| Suporte para programação funcional | DEsde o Java 8 suporta programação funcional limitada |
| Tipificação dinâmica | Tipificação estática |
| Baseada em protótipos | Baseada em classes |
| Herança usando o mecanismo dos protótipos | Herança através da hierarquia de classes |
| Adição dinâmica de novos membros a objetos | 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**](http://www.w3schools.com/js/js_reserved.asp)**do JavaScript**

Esta é a lista das principais palavras reservadas usadas 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 |

Note que 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 valores e não às variáveis. Exemplo:

|  |
| --- |
| var x = 34  x = "Hello!" // x pode conter um valor de qq tipo |

**Definição de variáveis**

A palavra var permite definir variáveis. Factos diversos:

* Dentro duma função, var define uma **variável local**.
* O **argumento duma função**, também define implicitamente uma variável local.
* Fora duma função, var define uma **variável global**.
* Usa-se **escopo estático** na resolução de nomes.
* Uma variável definida dentro dum bloco têm como âmbito toda a função envolvente, ou seja, **não há âmbito de bloco**. [Na versão mais recente do JavaScript foi introduzida a palavra let que permite definir variáveis com escopo de bloco. Para usar, basta trocar var por let.]

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 em JavaScript:

|  |
| --- |
| function f() {  var x = 1;  {  var x = 2;  }  print(x);  }  f();  // Output: 2 |

**Variáveis indefinidas**

Variáveis definidas 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 definida  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. Para garantir que a variável fica local, é preciso declará-la usando var.

|  |
| --- |
| x = 5;  y = 7 + 5; |

**Constantes**

A palavra const permite definir constantes. Mas, cuidado, que isto não faz parte do JavaScript padrão. Contudo está disponível no Rhino e Node.js, por exemplo.

|  |
| --- |
| const x = 5; |

**Tipos primitivos**

Os tipos primitivos são os seguintes:

* [**number**](http://www.javascriptkit.com/jsref/number.shtml) - Mistura reais e inteiros. O maior valor em várias implementações é 1.7976931348623157e+308. 0377 é um valor em octal e 0xFF é um valor em hexadecimal.
* [**boolean**](http://www.javascriptkit.com/jsref/boolean.shtml) - Tem os valores false e true.
* [**string**](http://www.javascriptkit.com/jsref/string.shtml) - 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.

Um literal de tipo string é uma porção de texto rodeada, ou por aspas ou plicas. Em código JavaScript, é comum usar plicas entre aspas e aspas entre plicas, como nestes exemplos: "He is called 'Neko-chan'"; 'He is called "Neko-chan"'.

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 << >> >>> (o último faz shift sem sinal)  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](http://www.javascriptkit.com/jsref/regexp.shtml) 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](http://www.javascriptkit.com/jsref/arrays.shtml) 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](http://devedge-temp.mozilla.org/library/manuals/2000/javascript/1.5/guide/fcns.html#1008302) 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( i in a )  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( i in arguments )  result += arguments[i];  return result;  }  var res = allAll(1,2,3,4,5); // Vale 15 |

A passagem de argumentos de tipos primitivos é feita por valor. Os objetos 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.

**O argumento this**

Todas as funções possuem automaticamente um primeiro argumento que não se declara e se chama **this**.

No caso de funções que não fazem parte de objetos, quando elas são chamadas, o argumento **this** fica associado a um objeto que depende da implementação. Por exemplo, no caso do Rhino e do Node.js, **this** fica associado a um objeto especial que guarda as ligações de todos os nomes globais. No seguinte exemplo, se a função f abaixo for chamada assim - f() - ela imprime-se a si própria de duas maneiras diferentes:

function f() {

print(f);

print(this.f);

}

No caso duma função f ser um método dum objeto obj e for chamada assim obj.f(), o argumento **this** fica associado ao próprio objeto (aliás, como também acontece em Java).

**A operação primitiva call**

Ao nível mais básico, em JavaScript todas as chamadas de função são feitas através da operação primitiva **call** que está disponível em todas as funções. Numa chamada de **call**, tem de se explicitar o primeiro argumento, **this**.

Todas as chamadas normais de funções são automaticamente traduzidas para invocações de **call**. Concretamente, a chamada duma função independente é assim traduzida, em Rhino:

f(1,2,3) -> f.call(global(),1,2,3)

A chamada do método dum objeto é assim traduzida:

obj.f(1,2,3) -> obj.f.call(obj,1,2,3)

**A operação primitiva apply**

Existe ainda outra operação primitiva chamada **apply**, semelhante a **call**, com a única diferença que se passa para ela um array com todos os argumentos que aparecem a seguinte a **this**. Se estiver em causa um método, as seguintes três chamadas são equivalentes:

obj.f(1,2,3) <-> obj.f.call(obj,1,2,3) <-> obj.f.apply(obj,[1,2,3])

**Tratamento de exceções**

Semelhante ao Java. Exemplo:

|  |
| --- |
| try {  undefinedFunction();  alert('Afinal a função existe');  } catch(e) {  alert('Erro: ' + e.message);  } finally {  alert('Chega sempre aqui, independentemente do que aconteceu antes.');  } |

**Objetos**

Em JavaScript para além dos tipos primitivos, temos os tipos objeto. Os arrays também 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 existem os seguintes objetos predefinidos, que oferecem funcionalidades úteis: [Date](http://www.javascriptkit.com/jsref/date.shtml), [Array](http://www.javascriptkit.com/jsref/arrays.shtml), [Boolean](http://www.javascriptkit.com/jsref/boolean.shtml), [Function](http://www.javascriptkit.com/jsref/function.shtml), [Math](http://www.javascriptkit.com/jsref/math.shtml), [Number](http://www.javascriptkit.com/jsref/number.shtml), [RegExp](http://www.javascriptkit.com/jsref/regexp.shtml) e [String](http://www.javascriptkit.com/jsref/string.shtml) .

Mas no ambiente de execução envolvente, estão geralmente disponíveis 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 são praticamente simples dicionários, que mapeiam etiquetas em valores. Existe uma sintaxe própria para definir objetos diretamente.

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; // Notação 1  var n = p["name"]; // Notação 2  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"  fullname: function() { return this.first + " " + this.last}  },  age: 33,  hobbies: [ "chess", "jogging" ]  } |

Agora uma definição mais rigorosa e completa de objeto: um objeto é um dicionário enriquecido por uma propriedade privada especial que se chama **prototype**.

# Programação baseada em protótipos

Para criar múltiplos objetos semelhantes e para reutilizar código, o JavaScript não usa classes mas sim **protótipos**. Esta técnica foi inventada em meados dos anos 1980 no contexto da linguagem Self.

Comecemos por falar um pouco da linguagem Self.

Em Self, a criação de novos objetos é efetuada a partir de objetos existentes. Sempre que um objeto P é usado como base para a criação de novos objetos, diz-se que P é um **protótipo**.

A criação dum novo objeto a partir dum protótipo P (designemos a operação por copy(P)) é muito simples: cria-se um objeto vazio (sem propriedades) e faz-se a propriedade especial prototype do novo objeto referir o protótipo P. Todos os objetos criados a partir dum protótipo P começam vazios e ficam a referir esse mesmo P.

Cada objeto **herda dinamicamente** do respetivo protótipo. A herança funciona assim: quando se tenta aceder a um membro dum objeto, se esse membro não estiver diretamente disponível no objeto, então a procura continua no respetivo protótipo. Se também não estiver diretamente disponível no protótipo, então procura-se no protótipo do protótipo. E assim sucessivamente, ao longo duma cadeia de protótipos.

Note que qualquer objeto pode ser usado como protótipo. Qualquer objeto passa a ser considerado um protótipo a partir do momento em que é usado para criar novos objetos.

Agora regressemos à linguagem JavaScript.

Em JavaScript, os objeto são idênticos aos do Self na medida em que contêm uma propriedade especial que identifica um protótipo, e cada objeto herda dinamicamente do seu protótipo. No entanto, o mecanismo disponível para a criação de objetos é mais complicado do que o do Self (veremos esse mecanismo na secção seguinte).

Em JavaScript, o protótipo dum objeto é guardado na seguinte propriedade privada:

|  |
| --- |
| [[Prototype]] |

Algumas implementações de JavaScript expõem essa propriedade através da propriedade pública \_\_proto\_\_. Nas implementações de JavaScript usadas na nossa cadeira (Rhino e NodeJS), esta propriedade pública está disponível e é usada como se exemplifica:

|  |
| --- |
| obj.\_\_proto\_\_ |

De qualquer maneira, a maneira padronizada de testar se proto é protótipo de obj, é a seguinte:

|  |
| --- |
| proto.isPrototypeOf(obj) |

Todos os objetos definidos através dum literal partilham automaticamente um protótipo predefinido que se escreve:

|  |
| --- |
| Object.prototype |

Veja esta pequena sessão interativa que prova de duas maneiras diferentes que os objetos literais herdam realmente de Object.prototype:

|  |
| --- |
| js> var a = {name: "Pedro", address: "Lisboa", age: 42}  js> Object.prototype.isPrototypeOf(a)  true  js> a.\_\_proto\_\_ === Object.prototype // "===" verifica se dois objetos são o mesmo.  true |

# Programação orientada pelos objetos usando construtores

Em JavaScript, qual o mecanismo previsto para criar novos objetos e lhes associar o protótipo de onde herdam?

Convém começar por dizer que o mecanismo usado no JavaScript é um pouco complicado e foge ao que é tradicional nas linguagem baseadas em protótipos (que costumam imitar o Self). Foi provavelmente a pensar nos programadores de Java que se decidiu introduzir um mecanismo com alguma aparência de familiaridade.

O mecanismo usado em JavaScript para criar objetos é o mecanismo dos **construtores**. Um **construtor** serve para inicializar diversos objetos do mesmo tipo, que herdam do mesmo protótipo.

Um construtor **C** é um função com as seguintes particularidades:

* Destina-se a ser chamada no contexto do operador new, com esta sintaxe **new C()**. O operador new cria um objeto vazio que depois é inicializado pelo construtor. Dentro do construtor, o objeto novo é conhecido por this e a sua inicialização envolve tipicamente apenas campos de dados; não campos funcionais.
* O construtor **C** é dono dum objeto vazio, que se tornará o protótipo de todos os objetos criados a partir desse construtor. Esse objeto escreve-se **C.prototype** e diz-se que é o *protótipo do construtor*.

Regra geral, os campos funcionais são adicionados no protótipo do construtor, para ficarem acessíveis por herança. Quando se adicionam novas funções ao protótipo, ele deixa de ser vazio. Repare bem: geralmente, os campos de dados pertencem a cada objeto e não são partilhados; os campos funcionais são metidos no protótipo e são partilhados por herança (pois não vale a pena repeti-los em cada objeto).

Abaixo define-se um construtor chamado Car para representar e inicializar automóveis. Neste exemplo, cada objeto fica com três campos de dados próprios. Os campos funcionais são seguidamente instalados no protótipo de Car e ficam disponíveis através de herança. Veja tudo com atenção:

|  |
| --- |
| function Car(make, model, year) {  this.make = make;  this.model = model;  this.year = year;  }  Car.prototype.toString = function() {  return "A Beautiful " + this.year + " " + this.make + " " + this.model;  }  Car.prototype.changeMake = function(make) {  this.make = make;  }  var car1 = new Car("Toyota", "Corolla", 2002); |

Para aceder ao construtor dum objeto obj escreve-se:

|  |
| --- |
| obj.constructor |

Para aceder ao protótipo dum construtor C escreve-se:

|  |
| --- |
| C.prototype |

Para testar se o construtor dum objeto obj é C, escreve-se:

|  |
| --- |
| obj instanceof C |

#### Criação de hierarquias através dos construtores

Manipulando diretamente o membro prototype dos construtores é possível criar uma hierarquia de protótipos.

Eis um exemplo simples, que introduz um subtipo de Car. Repare que se muda o protótipo de FlyingCar para ser um objeto de tipo Car (em vez do habitual objeto vazio inicial).

|  |
| --- |
| function FlyingCar(make, model, year, maxFlightLevel) {  this.make = make;  this.model = model;  this.year = year;  this.maxFlightLevel = maxFlightLevel;  this.flightLevel = 0;  }  FlyingCar.prototype = new Car("", "", 0); // muda-se aqui o protótipo!!!  FlyingCar.prototype.fly = function() {  return "Flying level = " + this.flightLevel;  }  var flyingCar1 = new FlyingCar("Toyota", "CFly", 2099, 100); |

Para tirar possível dúvidas sobre o mecanismo de herança, estudemos a seguinte chamada:

|  |
| --- |
| flyingCar1.fly(); |

Primeiro procura-se um campo fly no objeto flyingCar1. Não se encontra. Depois procura-se no protótipo desse objeto. Encontra-se!

Estudemos agora a chamada:

|  |
| --- |
| flyingCar1.toString(); |

Primeiro procura-se um campo toString no objeto no objeto flyingCar1. Não se encontra. Depois procura-se no protótipo desse objeto. Não se encontra. Depois procura-se no protótipo do protótipo. Encontra-se!

# Programar em JavaScript, imitando as classes do Java

Programar em JavaScript usando construtores não é tão simples e agradável como devia ser. Cada vez há mais programadores de JavaScript que procuram outros estilos.

No ECMAScript 6, em 2015, foram introduzidas **classes** para trabalhar com objetos e herança. Essas [classes](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes) fazem lembrar as classes do Java. Oferecem sintaxe mais simples e familiar para lidar com os objetos do JavaScript. Infelizmente, por ser muito recente, o novo mecanismo ainda só está a começar a ser implantado.

Nesta cadeira, por diversas razões, também seguimos pelo mesmo caminho e usamos classes. Mas definimos a nossa própria implementação. Introduzimos **classes** e criamos os nossos objetos através duma primitiva **NEW**. Introduzimos também a primitiva **EXTENDS** para definir herança entre classes e uma operação **SUPER** para aceder à versão original dos métodos redefinidos.

Para programar neste estilo, precisamos de copiar a classe JSRoot (com a sua operação SUPER), mais as funções NEW e EXTENDS, para o início dos nossos programas. Não é preciso perceber os detalhes de implementação, embora seja instrutivo estudá-los: envolvem a introdução dum construtor vazio e a alteração direta do seu protótipo.

Usamos as palavras NEW, EXTENDS e SUPER em maiúsculas porque as correspondentes palavras em minúsculas estão reservadas no JavaScript.

|  |
| --- |
| **// Copiar este código para o início de cada programa.**  var JSRoot = {  **SUPER**: function(method) {  return method.apply(this,  Array.prototype.slice.apply(arguments).slice(1));  },  **INIT**: function() {  throw "\*\*\* MISSING INITIALIZER \*\*\*";  }  };  function **NEW**(clazz) { // Create an object and applies INIT(...) to it  function F() {}  F.prototype = clazz;  var obj = new F();  obj.INIT.apply(obj, Array.prototype.slice.apply(arguments).slice(1));  return obj;  };  function **EXTENDS**(clazz, added) { // Creates a subclass of a given class  function F() {}  F.prototype = clazz;  var subclazz = new F();  for(var prop in added)  subclazz[prop] = added[prop];  return subclazz;  };  // "print" é opcional, mas introduz compatibilidade com o Node.js  var **print** = typeof(console) !== 'undefined' ? console.log : print; |

A função **NEW** aplica-se a uma classe. Cria um objeto e inicializa-o usando o método de inicialização INIT que está definido na classe.

Precisamos de distinguir entre dois tipos de objetos:

* Objetos normais, criados e inicializados através da função **NEW**. Manipulamos estes objetos através de variáveis cujo nome começa por letra minúscula.
* Classes definidas através da função **EXTENDS** (exceto no caso da classe predefinida JSRoot). Guardamos as classes em variáveis cujo nome começa por letra maiúscula.

Ao programar neste estilo, consideramos apenas membros públicos.

Os dois exemplos que se seguem ilustram esta forma de programar em JavaScript. Veja como o código fica parecido com Java, em termos de conceitos e de estrutura, embora não em termos de sintaxe.

# Exemplo 1

Hierarquia de classes para representar pontos a uma dimensão, duas dimensões e três dimensões. Fatoriza-se o código ao máximo, inclusivamente usando a construção SUPER sempre que for aplicável.

|  |
| --- |
| var Point1 = **EXTENDS**(JSRoot, {  x: 0,  **INIT**: function(x) {  this.x = x;  },  equals: function(that) {  return this.x == that.x;  },  shift: function(deltax) {  this.x += deltax;  },  show: function() {  print("(" + this.x + ")");  }  });  var Point2 = **EXTENDS**(Point1, {  y: 0,  **INIT**: function(x, y) {  this.**SUPER**(Point1.INIT, x);  this.y = y;  },  equals: function(that) {  return this.**SUPER**(Point1.equals, that)  && this.y == that.y;  },  shift: function(deltax, deltay) {  this.**SUPER**(Point1.shift, deltax);  this.y += deltay;  },  show: function() {  print("(" + this.x + ", " + this.y + ")");  }  });  var Point3 = **EXTENDS**(Point2, {  z: 0,  **INIT**: function(x, y, z) {  this.**SUPER**(Point2.INIT, x, y);  this.z = z;  },  equals: function(that) {  return this.**SUPER**(Point2.equals, that)  && this.z == that.z;  },  shift: function(deltax, deltay, deltaz) {  this.**SUPER**(Point2.shift, deltax, deltay)  this.z += deltaz;  },  show: function() {  print("(" + this.x + ", " + this.y + ", " + this.z + ")");  }  });  var tests = {  testInheritance: function() {  var a = **NEW**(Point3,1,2,3);  var b = **NEW**(Point3,6,7,8);  a.show(); b.show();  a.shift(1, 1, 1); a.show(); b.show();  b.shift(1, 1, 1); a.show(); b.show();  },  testMixedTypes: function() { // a parte inválida das operações não tem efeito.  var a = **NEW**(Point1,1);  var b = **NEW**(Point2,6,7);  a.show(); b.show();  a.shift(1, 1, 1); a.show(); b.show();  b.shift(1, 1, 1); a.show(); b.show();  },  run: function() {  for(var fun in this)  if( fun != 'run' ) {  print(fun);  this[fun]();  }  }  };  // Run tests  js> tests.run();  testInheritance  (1, 2, 3)  (6, 7, 8)  (2, 3, 4)  (6, 7, 8)  (2, 3, 4)  (7, 8, 9)  testMixedTypes  (1)  (6, 7)  (2)  (6, 7)  (2)  (7, 8)  js> Point3.isPrototypeOf(NEW(Point3,1,2,3));;  true  js> Point1.isPrototypeOf(NEW(Point3,1,2,3));  true  js> Point2.isPrototypeOf(NEW(Point1,2));  false |

# Exemplo 2

Representação de expressões algébricas, onde pode ocorrer uma variável "x".

|  |
| --- |
| var ExpNode = **EXTENDS**(JSRoot, {  **INIT**: function() {},  big: function() { return size() > 1000; }  });  var BinNode = **EXTENDS**(ExpNode, {  l: null, r: null,  **INIT**: function(l, r) { this.**SUPER**(ExpNode.INIT); this.l = l; this.r = r; },  size: function() { return 1 + this.l.size() + this.r.size(); },  height: function() { return 1 + Math.max(this.l.height(), this.r.height()); }  });  var UnaryNode = **EXTENDS**(ExpNode, {  e: null,  **INIT**: function(e) { this.**SUPER**(ExpNode.INIT); this.e = e; },  size: function() { return 1 + this.e.size(); },  height: function() { return 1 + this.e.height(); }  });  var ZeroNode = **EXTENDS**(ExpNode, {  **INIT**: function() { this.**SUPER**(ExpNode.INIT); },  size: function() { return 1; },  height: function() { return 1; }  });  var AddNode = **EXTENDS**(BinNode, {  **INIT**: function(l, r) { this.**SUPER**(BinNode.INIT, l, r); },  eval: function(v) { return this.l.eval(v) + this.r.eval(v); },  deriv: function() { return **NEW**(AddNode, this.l.deriv(), this.r.deriv()); }  });  var MultNode = **EXTENDS**(BinNode, {  **INIT**: function(l, r) { this.**SUPER**(BinNode.INIT, l, r); },  eval: function(v) { return this.l.eval(v) \* this.r.eval(v); },  deriv: function() { return **NEW**(AddNode, **NEW**(MultNode, this.l, this.r.deriv()),  **NEW**(MultNode, this.l.deriv(), this.r)); }  });  var SimNode = **EXTENDS**(UnaryNode, {  **INIT**: function(e) { this.**SUPER**(UnaryNode.INIT, e); },  eval: function(v) { return -this.e.eval(v); },  deriv: function() { return **NEW**(SimNode, this.e.deriv()); }  });  var ConstNode = **EXTENDS**(ZeroNode, {  c: 0.0,  **INIT**: function(c) { this.**SUPER**(ZeroNode.INIT); this.c = c; },  eval: function(v) { return this.c; },  deriv: function() { return **NEW**(ConstNode, 0); }  });  var VarNode = **EXTENDS**(ZeroNode, {  **INIT**: function() { this.**SUPER**(ZeroNode.INIT); },  eval: function(v) { return v; },  deriv: function() { return **NEW**(ConstNode, 1); }  });  var tests = {  testSimple: function() {  var e = **NEW**(MultNode,  **NEW**(AddNode, **NEW**(ConstNode, 4), **NEW**(VarNode)),  **NEW**(ConstNode, 6.5));  print("((4 + x) \* 6.5)' (3) = " + e.deriv().eval(3));  },  run: function() {  for(var fun in this)  if( fun != 'run' ) {  print(fun);  this[fun]();  }  }  };  // Run tests  js> tests.run();  testSimple  ((4 + x) \* 6.5)' (3) = 6.5 |

## Método hasOwnProperty

Se estivermos na dúvida se uma dada propriedade está presente num objeto de forma direta ou indireta através de herança, podemos confirmar isso usando este método. Exemplo:

|  |
| --- |
| var P = { a:0 };  var x = NEW(P);  print(x.hasOwnProperty('a'));  x.a = 5;  print(x.hasOwnProperty('a'));  // Output  false  5  true |

## Propriedades privadas em objetos

Quando se escreve um objeto literal, esse objeto fica com todas as suas propriedades públicas. Nos objetos literais, não há suporte para propriedades privadas. No entanto, conseguimos obter um efeito equivalente usando funções anónimas com variáveis locais. Essas variáveis locais vão funcionar como campos privados dum objeto.

Estude o seguinte exemplo:

O seguinte objeto literal tem todos os campos públicos:

|  |
| --- |
| var pub = {  x: 0,  y: 5,  show: function() { print("(" + this.x + " ," + this.y + ")") }  };  pub.show(); // Output: (0, 5) |

Para tornar privado o campo 'y', usa-se o esquema abaixo. Note que "this.y" foi reescrito "y".

|  |
| --- |
| var priv = (function () {  var y = 5;  return {  x: 0,  show: function() { print("(" + this.x + " ," + y + ")") }  }  })();  priv.show(); // Output: (0, 5) |

Explicação: Escreve-se uma função anónima com uma variável local chamada 'y'. No contexto dessa função define-se um objeto, o qual pode usar a variável 'y'. A função está preparada para retornar esse objeto quando for chamada. A função é chamada imediatamente e assim o objeto fica disponível no exterior; mas a variável 'y' permanece privada e só pode ser acedida a partir dos métodos do objeto.

**Extensibilidade e abstração**

Um **sistema extensível** é um sistema que se pode fazer crescer, sem que se tenha de se alterar o que já foi escrito.

A extensibilidade obtém-se pela via da **abstração**:

* Código que seja escrito com base em conceitos abstratos consegue lidar com as entidades iniciais descritas no enunciado do problema, mas também (e aqui é que surge a extensibilidade) com entidades a criar no futuro. Claro que isto só funciona se as entidades futuras se enquadrarem nas abstrações inicialmente consideradas.

Apesar de não ser obrigatório, muitas vezes queremos escrever software extensível. As principais razões são as seguintes:

* Os programas ficam organizados em torno de ideias gerais e claras.
* Os programas ficam mais fáceis de atualizar.
* O tempo de vida útil dos programas aumenta.
* Os programas ficam mais fiáveis.
* Poupa-se tempo e dinheiro.
* Os programas ficam mais elegantes e tornam-se fonte de prazer estético (pelo menos para quem os escreve).

**Extensibilidade através da fatorização**

A fatorização das classes é um **requisito** para obter extensibilidade. A fatorização faz parte do processo de desenho que ajuda a identificar as abstrações naturais dos programas.

Por vezes, basta fatorizar um sistema em torno de abstrações naturais, para ele ficar automaticamente extensível. Isso acontece quando não há iteração entre objetos, ou seja quando todas as operações disponíveis numa classe se aplicam ao objeto **this**, sem envolver outros objetos. Esta é a situação que apareceu nos exemplos finais da aula teórica 19 (em JavaScript). Também vai aparecer nos exercícios da aula prática 11.

Mas quando as operações envolvem o objeto **this** e outros objetos então a fatorização, só por si, **não é suficiente** para obter extensibilidade.

**A evitar: Testes explícitos de tipo**

Existe uma questão que, de forma dramática, dá origem a **código não extensível**. Trata-se do uso de **testes para determinar o tipo dum objeto**. Esse teste costuma ser feito de forma direta, usando instanceof ou isPrototypeOf, mas por vezes também é feito de forma indireta, testando algum atributo do objeto, por exemplo o nome ou a cor.

Tal código nunca pode ser extensível, pois a sua lógica está comprometida com os tipos concretos existentes: ou seja, esse código não conseguirá lidar com novos tipos a criar no futuro. Nessa situação, para estender a lógica a novos tipos, seria necessário reescrever o código para tratar mais casos concretos, o que significa que o código não seria extensível.

**A evitar: Quebra de modularidade**

Uma classe deve tratar dos seus objetos e não se preocupar com as tarefas de outros objetos. A tarefas dos outros objetos são para implementar nas respetivas classes.

Por outras palavras: cada objeto tratar de si e não se preocupar em fazer as tarefas dos outros objetos.

**Técnicas para evitar os testes explícitos de tipo**

Geralmente, nas funções que, para além do **this**, recebem outro objeto como argumento, surge a tentação de testar diretamente o tipo do argumento. Isso tem de ser evitado a todo o custo, se tivermos como objetivo a escrita de código extensível.

Vejamos algumas técnicas que permitem evitar os testes explícitos de tipo.

**Técnica do envio de mensagem**

Em vez de testar diretamente o tipo dum objeto, podemos enviar-lhe uma mensagem a perguntar alguma coisa. Tal código já é extensível pois funciona com quaisquer objetos que suportem uma dada função. Repare, funciona mesmo com objetos de tipos a criar futuramente.

Por exemplo, num jogo baseado numa matriz bidimensional, em que vários monstros perseguem um herói, o que é que um monstro deverá fazer quando se cruza com outra personagem? Imagine que se trata duma função meets(vizinho) da classe Monstro.

* **Versão errada** (visão dum mundo fechado) -- O monstro determina, usando a expressão Hero.isPrototypeOf(vizinho), se o vizinho é um herói. Se for um herói, então o monstro almoça-o. Se não for um herói, então não faz nada. Este código não é nada extensível pois a sua lógica está dependente das classes concretas iniciais.
* **Versão correta** (visão dum mundo aberto) -- O monstro envia ao vizinho a mensagem vizinho.comestível() e depois atua em conformidade com a resposta. Este código já é extensível, pois funciona com qualquer personagem, mesmo com uma personagem a criar futuramente, que saiba responder à pergunta comestível().

Em conclusão, conseguimos escrever código extensível **introduzindo o conceito abstrato** de *comestível*.

Relativamente a detalhes de implementação, tipicamente define-se o método comestível na raiz da hierarquia de classes retornando o valor false. Desta forma, por omissão, os personagens não são comestíveis. Os personagens que forem comestíveis redefinem o método para retornar true.

**Técnica dos níveis**

Suponha que no jogo existem muitas classes diferentes de personagens, e que entre essas classes de personagens se estabelecem complexas regras de alimentação, de tipo *cadeia alimentar*. Neste caso, uma simples função booleana *comestível* não chega para capturar tal riqueza de relações.

Neste caso, convém associar um *nível alimentar* a cada tipo de personagem e estabelecer a seguinte regra: uma personagem pode comer outra só no caso do nível alimentar da primeira ser superior ao da segunda. Concretamente, um objeto pode comer o seu vizinho se:

this.nivelAlimentar() > vizinho.nivelAlimentar()

**Métodos binários**

Chama-se método binário a um método com um argumento, em que se espera que o valor do argumento tenha exatamente o mesmo tipo concreto de this. [[Paper sobre métodos binários](http://lucacardelli.name/Papers/Binary.pdf)].

Por exemplo, para efeitos de reprodução, um objeto pode precisar de saber se um outro objeto, seu vizinho, é do mesmo tipo. Como fazer isso de forma geral, sem ter de referir o tipo concreto?

Faz-se assim em JavaScript:

this.\_\_proto\_\_ == vizinho.\_\_proto\_\_

Em Java:

getClass() == vizinho.getClass()

Em C++:

#include <typeinfo>

typeid(\*this) == typeid(\*vizinho)

**Exercício sobre métodos binários**

Observe o método equals das classes Point1, Point2, Point3 da aula teórica 20. Esse método não é binário pois permite comparar pontos de diferentes tipos: compara os campos em comum de this e that e ignora os campos que não sejam comuns.

Agora imagine que você pretende tornar binário o método equals, ou seja fazer com que ele só dê true no caso de this e that serem do mesmo tipo concreto e dos campos correspondentes serem iguais. Como fazer?

Tente resolver o problema modificando apenas o método equals na classe Point1.

**Fatorização sim, mas na medida certa**

Quando se fatoriza o código, é preciso cuidado para não exagerar a ponto de começar a trocar herança por generalidade. Ao trocar herança por generalidade perdemos extensibilidade, além de que o código começa a ficar confuso.

Repare que, no limite, a essência que qualquer hierarquia de classes pode ser capturada numa classe única, sem nunca se chegar a usar herança. Essa classe fica muito geral e muito complexo, cheio de variáveis que servem para representar as diferentes propriedades de diversos tipos de entidades ao mesmo tempo. As funções também ficam muito complexas e cheias de ifs porque têm de lidar com diferentes casos ao mesmo tempo. No limite até é possível criar uma única classe, muito geral, complexo e confuso, para representar ao mesmo tempo o Herói e os Monstros, sem nunca se chegar a usar herança.

Esse código não é extensível. Para incorporar um novo tipo de entidade, seria preciso alterar o código para passar a tratar mais um caso, o que prova que não é extensível.

# A Internet

A **Internet** é um sistema mundial de redes de computadores ligadas entre si. Teve origem num projeto militar dos anos 1960, mas a Internet moderna começou em meados dos anos 1980, quando as redes internas de diversas instituições académicas começaram a ser ligadas entre si. Nessa altura também apareceram as primeiras utilizações comerciais com serviços de email fornecidos por algumas companhias de comunicações.

## Desenho por camadas

A Internet foi desenhada por forma a ser escalável, robusta, e obedece a um conjunto de padrões oficiais bem definidos. Quem escreve programas sobre a Internet, usa os sockets disponibilizados pelo sistema operativo e por vezes tem um conhecimento relativamente superficial das tecnologias subjacente e isso basta. Mas, para apreciar devidamente a Internet e contextualizar o desenvolvimento de programas para ela, é importante conhecer um pouco mais dessas tecnologias (que só serão verdadeiramente estudadas em pormenor na disciplina do 3º ano "Redes de Computadores").

A Internet está organizada em sete camadas (de acordo com um padrão OSI), cada uma das quais se preocupa com um diferente aspeto das comunicações:

1. **Camada física** - modems, cabos telefónicos, fibra ótica, cabos Ethernet, etc.
2. **Camada de ligação de dados** - Protocolos que permitem aos aparelhos da camada física comunicarem uns com os outros: protocolo Ethernet, protocolos wireless, protocolo PPP, etc.
3. **Camada de rede** - Protocolo IP para transmissão de pacotes de dados entre dois pontos da rede, o que inclui questões de *routing* através nós intermédios. Esta camada implementa transmissão básica de dados, mas não é responsável pela fiabilidade da transmissão nem pelos aspetos semânticos da transmissão
4. **Camada de transporte** - Protocolos TCP, UDP e mais alguns, que adicionam diferentes graus de fiabilidade à rede e que definem diversos tipos de comunicação. O protocolo UDP suporta o envio de mensagem simples sem ser necessário estabelecer previamente uma conexão formal. O protocolo TCP suporta comunicação altamente fiáveis e requer o estabelecimento duma conexão formal temporária e dum diálogo entre os dois lados para a transmissão duma mensagem se poder efetuar com fiabilidade.
5. **Camada de sessão** - Permite o estabelecimento de conexões semipermanentes que são automaticamente restauradas em caso de desconexão.
6. **Camada de apresentação** - Trata de transformações básicas dos dados transmitidos e recebidos. Por exemplo: mudança de encoding dos carateres, uso de criptografia, serialização de documentos XML, etc.
7. **Camada da aplicação** - Implementa os protocolos usados pelas aplicações concretas. Exemplos de protocolos aplicacionais: Telnet (terminal virtual), SSH (terminal virtual com criptografia), FTP (acesso a ficheiros), HTTP (web), POP3/IMAP (email), BitTorrent (distribuição de ficheiros grandes), etc.

## Endereços IP

O protocolo IP (na camada de rede), identifica os nós da Internet usando um endereço único chamado **endereço IP**. Na variante IPv4 o endereço é um inteiro de 32 bits. Na variante IPv6 o endereço é um inteiro de 128 bits.

Do ponto de vista do computador, um endereço IP é um inteiro. Mas quando se escreve um endereço IP para ser lido por pessoas, usa-se a base 10 e os algarismos são agrupados de forma especial que exibe a estrutura lógica do endereço (estrutura essa que não vamos discutir aqui). Eis dois exemplos de endereços IP:

172.16.254.1 (IPv4)

2001:db8:0:1234:0:567:8:1 (IPv6).

Não é prático para os humanos memorizar endereços IP e por isso foi criado um serviço de rede chamado "DNS Domain Name System" que associa **nomes de domínio** a endereços IP. Por exemplo, o domínio **ctp.di.fct.unl.pt** está associado ao endereço **193.136.122.73**.

Por convenção, o nome **localhost** representa o computador local.

## Portas TCP

O protocolo TCP (na camada de transporte) permite que serviços diferentes partilhem o mesmo computador e a mesma ligação à Internet. Cada nó da rede disponibiliza um conjunto de **portas** identificadas por um inteiro de 16 bits sem sinal.

Cada serviço está associado a uma porta que pode ser escolhida livremente, embora existam muitas [associações padronizadas](https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers). Por exemplo: HTTP corre na porta 80, HTTPS na porta 443, FTP na porta 21, SSH na porta 22, Telnet na porta 23, BitTorrent na porta 30301.

# A World Wide Web

A **World Wide Web** (ou mais simplesmente **WWW** ou **Web**) (na camada de aplicação) resultou da seguinte ideia: Porque não criar na Internet um sistema de visualização de documentos, no qual interligamos os documentos entre si para facilitar a navegação neles?

Em 1989, foi Tim Berners-Lee que teve a ideia de aplicar à Internet a conhecida noção de **hipertexto** e começou a desenvolver o assunto. Criou a primeira versão do protocolo HTTP e da [linguagem HTML](https://en.wikipedia.org/wiki/HTML). Em 1990, também foi ele que escreveu o primeiro protótipo de servidor Web e o primeiro protótipo de browser Web. Eis o [resumo da história dos primeiros anos](http://info.cern.ch/hypertext/WWW/History.html). O primeiro site web foi [info.cern.ch](http://info.cern.ch/) e o primeiro endereço de página web foi <http://info.cern.ch/hypertext/WWW/TheProject.html>.

A Web é um dos vários serviços que correm na camada de aplicação da Internet. Usa os protocolos HTTP/HTTPS e um modelo de cliente/servidor. Os servidores Web estão ligados à Internet e correm um software específico (e.g. Apache ou IIS) que fica à espera de conexões. Os utilizadores ligam-se aos servidores usando um **Web browser**, por exemplo o Firefox, Chrome, Opera, Safari (apenas no Mac/Windows), ou Internet Explorer (apenas no Windows),

## URL

Para aceder a um site Web ou a recurso particular dum site Web usando um browser, o utilizador clica num link que tem associado o URL que identifica esse site ou recurso.

**URL** significa **Uniform Resource Locator**. Tem a seguinte estrutura geral:

protocolo://domínio:porta/caminho/recurso?query#fragmento

Alguns exemplos de URLs:

[http://en.wikipedia.org:80/wiki/Uniform\_resource\_locator#History](http://en.wikipedia.org/wiki/Uniform_resource_locator#History)

<http://www.google.com/search?q=UNL>

<http://whatismyipaddress.com/ip/193.136.122.73>

<http://translate.google.com/?langpair=en|pt&text=hello>

[https://clip.unl.pt/utente/eu/fun%E7%E3o/doc%EAncia/unidade/actividade/inscri%E7%F5es/pautas?tipo\_de\_per%EDodo\_lectivo=s&ano\_lectivo=2016&per%EDodo\_lectivo=2&institui%E7%E3o=97747&unidade\_curricular=8147&modo=pauta](https://clip.unl.pt/utente/eu/fun%E7%E3o/doc%EAncia/unidade/actividade/inscri%E7%F5es/pautas?tipo_de_per%EDodo_lectivo=s&ano_lectivo=2017&per%EDodo_lectivo=2&institui%E7%E3o=97747&unidade_curricular=8147&modo=pauta)

## O protocolo HTTP

O [protocolo HTTP](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) especifica um pequeno conjunto de comandos que o browser envia para o servidor Web a solicitar alguma ação sobre um recurso bem identificado. Eis os comandos mais importantes que o browser pode enviar para o servidor:

* **GET** - Obtém uma cópia do recurso especificado.
* **HEAD** - Pede ao servidor informação sobre o recurso especificado.
* **PUT** - Atualiza um recurso no servidor, no URL especificado.
* **POST** - Atualiza um recurso no servidor, aproximadamente no URL especificado, mas é o servidor que decide qual é o URL final no qual vai ficar guardado o recurso.
* **DELETE** - Elimina o recurso especificado.

O comando mais usado é o comando **GET**. É tipicamente usado para obter o conteúdo dum ficheiro guardado no servidor ou para obter uma página HTML para apresentar no browser.

Para ver o comando **GET** a funcionar, vamos enviá-lo para o servidor usando a aplicação **telnet** (também podiamos usar a aplicação **nc**) e observar a resposta do servidor. Neste exemplo, abrimos uma conexão através da porta 80 do servidor **ctp.di.fct.unl.pt** e depois damos um comando **GET** a pedir o conteúdo da primeira aula teórica de LAP.

|  |
| --- |
| $ **telnet ctp.di.fct.unl.pt 80**  Trying 193.136.122.73...  Connected to di73.di.fct.unl.pt.  Escape character is '^]'.  **GET /lei/lap/teoricas/01.html**  <body text="#00000" bgcolor="#E0F0E0" link="#0000EE" vlink="#551A8B" #FF0000">  <HR><HR><H1>Linguagens e Ambientes de Programação (2016/2017)</H1>  <HR><HR><H2>Teórica 01 (04/Mar/2017)</H2>  <p>Apresentação da disciplina.  <p>Discussão introdutória sobre alguns aspetos importantes da temática das Linguagens de Programação.  <HR>  ...  ... *muitas linhas omitidas* ...  ...  Connection closed by foreign host. |

Já agora, testemos também o comando **HEAD** sobre a mesma página:

|  |
| --- |
| $ **telnet ctp.di.fct.unl.pt 80**  Trying 193.136.122.73...  Connected to di73.di.fct.unl.pt.  Escape character is '^]'.  **HEAD /lei/lap/teoricas/01.html HTTP/1.0**  HTTP/1.1 200 OK  Date: Thu, 04 Jun 2017 21:38:10 GMT  Server: Apache/2.2.16 (Debian)  Last-Modified: Sat, 14 Mar 2017 12:54:29 GMT  ETag: "3ba5f-39bf-5113f1e172b40"  Accept-Ranges: bytes  Content-Length: 14783  Vary: Accept-Encoding  Connection: close  Content-Type: text/html  Connection closed by foreign host. |

Neste exemplo, comunicámos com o servidor usando a aplicação **telnet**. Mas em que circunstâncias é que o browser envia realmente comandos **GET** para o servidor? Eis dois exemplos:

#### Exemplo 1

O primeiro exemplo é trivial e consiste num simples link que refere um endereço da Google. Ao clicar nele, fazemos o browser enviar um comando **GET** ao servidor da Google. O browser recebe do servidor a página HTML correspondente ao URL e mostra-a.

<http://www.google.com/search?q=Internet>

Código:

|  |
| --- |
| <a href="http://www.google.com/search?q=Internet">http://www.google.com/search?q=Internet</a> |

#### Exemplo 2

No exemplo 1 interrogámos o servidor da Google com a query fixa "Internet". Suponhamos que agora queremos interrogar o servidor usando uma query variável, indicada pelo utilizador. A solução passa por usar um formulário HTML com um atributo "ACTION" e com o botão "submit", o que para nós é novidade pois não se falou disto na aula em que apresentaram os formulários HTML.

Parte superior do formulário

Google query:  

Parte inferior do formulário

Código:

|  |
| --- |
| <FORM NAME="form1" ACTION="http://www.google.com/search">  Google query:  <INPUT TYPE="text" NAME="q" VALUE="Internet">  <INPUT TYPE="submit" VALUE="Search">  </FORM> |

Neste formulário, quase tudo o que lá se encontra tem a sua razão de ser:

* No atributo "ACTION" especifica-se a parte inicial (prefixo) do URL.
* A caixa de input tem o nome "q", que vai fazer parte do URL gerado.
* Finalmente, existe um botão de tipo "submit". Premir esse botão faz o browser construir um URL completo e enviá-lo para o servidor num comando **GET**.

# Scripting do lado do servidor - Web dinâmica

Em 1991, a primeira versão do protocolo HTTP suportava apenas o comando **GET** e a resposta do servidor tinha de ser obrigatoriamente uma página HTML fixa. Não havia a possibilidade de gerar páginas dinamicamente do lado do servidor. Portanto, nessa altura, um site Web tinha associado uma simples coleção de páginas HTML fixas, geralmente organizadas numa árvore de ficheiros e diretorias.

A **Web dinâmica** começou em 1993, quando foram introduzidos os [formulários HTML](https://en.wikipedia.org/wiki/Form_(HTML)) (do lado do cliente) e o [Common Gateway Interface (CGI)](https://en.wikipedia.org/wiki/Common_Gateway_Interface) (do lado do servidor). Do lado do browser (cliente), usando formulários HTML, passou a ser possível efetuar pedidos com argumentos variáveis (como no último exemplo da secção anterior). Do lado do servidor, passou a ser possível correr scripts para gerar dinamicamente páginas HTML.

As linguagens Perl e C começaram por ser as mais usadas na escrita de scripts do lado do servidor. Mas logo foram desenvolvidas linguagens mais especializadas. Uma delas, o PHP, apareceu em 1995 e tornou-se rapidamente muito popular - atualmente continua a ser a linguagem mais usada do lado do servidor. Eis outras linguagens que se usam atualmente do lado do servidor: Java, ASP, Python, Ruby, Scala, TCL, JavaScript, etc.

Agora um detalhe importante dos servidores tradicionais: por cada pedido recebido, cria-se um processo independente para correr o script que vai gerar a página de resposta. Portanto, o servidor pode ficar congestionado se forem recebidos muitos pedidos num curto espaço de tempo. (Em todo o caso, demasiados acessos concorrentes à base de dados costuma ser o fator mais determinante de possíveis congestionamentos.)

Os exemplos 1 e 2, atrás, são representativos das funcionalidades existentes no início de 1996. Eram suportados formulários HTML com o atributo "ACTION", os servidores eram capazes de gerar páginas dinâmicas. Mas as páginas não eram interativas do lado do browser. Neste modelo, por cada pedido feito pelo browser, o servidor envia uma página completa que substitui a anterior (e o utilizador vê a página a ser completamente redesenhada).

Nessa altura ainda não tinha sido inventado o JavaScript, nem introduzido o atributo "ONCLICK" nos formulários HTML...

# Scripting do lado do cliente - JavaScript

Em 1996 foram testadas novas funcionalidades que ficaram padronizadas em 1997 sob o nome de **HTML 4**. Por exemplo, foram introduzidas frames e style sheets (CSS). Mas a adição mais revolucionária foi a introdução de scripting do lado do cliente. Com scripts a correr do lado do cliente, passou a ser possível ter páginas Web realmente interativas. A interatividade proporcionada pelos formulários era muito pobre.

A primeira linguagem de scripting usada do lado do cliente foi o JavaScript. Continua a ser a mais usada nesse contexto.

Dois atributos importantes que foram adicionados ao HTML, ligados ao scripting: "ONCLICK", que pode ser usado na maioria dos elementos HTML, mas que é mais usado nos botões, para invocar uma função JavaScript em resposta a um clique de rato; "ONLOAD", que é usado no elemento "BODY" para executar uma função JavaScript logo que o carregamento da página termine.

Com o JavaScript passou a ser possível, por exemplo:

* Validar o formato dos dados dos formulários do lado do cliente e dar feedback imediato em caso de erro.
* Inserir dinamicamente novo texto numa página HTML, sem passar pelo servidor.
* Reagir imediatamente a eventos gerados pelo utilizador através do teclado e rato.

Atualmente existe a tendência para implementar cada vez mais funcionalidades do lado do cliente e menos do lado do servidor. Do ponto de vista do utilizador a experiência de utilização torna-se mais rica e mais rápida (parecendo que se está a usar uma aplicação local). Contudo, há limites naturais para esta tendência. Por exemplo, as bases de dados, que contêm os dados consultados, são centralizadas e residem do lado do servidor.

## Modelo de eventos

Como é comum nas interfaces gráficas interativas, os scripts que correm no browser usam um modelo de programação baseado em eventos. Os eventos normalmente são gerados por ações do utilizador, através do rato ou do teclado. Os eventos são tratados de forma assíncrona.

O uso de lógica assíncrona tem consequências dramáticas no estilo de programação! Nos programas clássicos, o programa consegue controlar todo o fluxo de execução de forma estrita. Na programação assíncrona, escreve-se código para reagir a eventos e não há possibilidade de controlar o fluxo de execução geral.

Quando um programa JavaScript corre num browser, há uma thread principal que executa o código do programa e se chama a **thread dos eventos**. A thread dos eventos já está implementada (não pode ser alterada). Executa continuamente um ciclo que retira e processa os eventos guardados numa fila de espera. Por cada evento, a thread invoca a função de tratamento adequada, previamente associada a esse tipo de evento. Os eventos são tratados sequencialmente e enquanto não terminar o tratamento dum evento não se inicia o tratamento do seguinte. Por isso é importante que o tratamento de cada evento seja rápido - caso contrário, a página Web terá bloqueios desagradáveis. (Para além da thread dos eventos, há algumas threads auxiliares usadas na geração de eventos, mas elas trabalham discretamente nos bastidores, e o programador não pensa nelas.)

Eis a lista completa de [eventos do DOM](http://www.w3schools.com/jsref/dom_obj_event.asp) para os quais se podem instalar tratadores de eventos numa página HTML. Os tratadores de eventos podem ser instalados nos atributos [ONCLICK](http://www.w3schools.com/jsref/event_onclick.asp) e [ONLOAD](http://www.w3schools.com/jsref/event_onload.asp), ou usando a função [addEventListener](http://www.w3schools.com/jsref/met_element_addeventlistener.asp) em código JavaScript. Para criar um timer e especificar qual a função tratadora de eventos de tempo, usa-se a função [setInterval](http://www.w3schools.com/jsref/met_win_setinterval.asp).

# AJAX

Sem usar AJAX, um pedido do cliente (browser) ao servidor implica apresentar uma página completa nova. Foi o que vimos nos dois exemplos mais atrás.

[AJAX](https://en.wikipedia.org/wiki/Ajax_(programming)) é um mecanismo que permite ao cliente obter informação do servidor de forma completamente flexível. O cliente faz o pedido e depois pode usar a resposta (assíncrona) como entender, por exemplo para atualizar uma porção da página corrente, mostrar a resposta num alerta, ou outra coisa qualquer.

O mecanismo é fácil de usar e envolve uma API chamada [XMLHttpRequest](https://en.wikipedia.org/wiki/XMLHttpRequest), que está disponível para ser usada em JavaScript em todos os browsers modernos.

Os conceitos subjacentes ao AJAX foram inventados pela Microsoft por volta de 1998. A Google começou a usar AJAX de forma massiva nos seus sites em 2004. começando pelo GMail e Google Maps.

#### Pedido assíncrono

Eis um exemplo, onde se troca o texto "AJAX lava mais branco", pelo conteúdo do ficheiro "ajax.txt", que se encontra no servidor. Experimente clicar:

|  |
| --- |
| AJAX  AJAX lava mais branco. |

Eis o código do exemplo. A função update faz o seguinte: (1) cria um pedido XMLHttpRequest vazio, (2) indica como será tratada a futura resposta do servidor, (3) preenche o pedido, (4) envia o pedido. Sabemos que a resposta já está disponível quando **xhttp.readyState == 4** e sabemos que o operação teve sucesso se adicionalmente **xhttp.status == 200**.

|  |
| --- |
| <**SCRIPT** **TYPE**="text/javascript">  function update(elementname, filenameURL) {  var xhttp = new XMLHttpRequest(); // cria pedido vazio  xhttp.onreadystatechange = function() { // especifica tratamento da resposta  if( xhttp.readyState == 4 )  if( xhttp.status == 200 ) {  var el = document.getElementById(elementname);  el.innerHTML = xhttp.responseText;  }  else alert("Error");  };  xhttp.open("GET", filenameURL, true); // preenche o pedido  xhttp.send(); // envia o pedido para o servidor  }  </**SCRIPT**>  <**H1**>AJAX</**H1**>  <**INPUT** **TYPE**="button" **VALUE**="Click Me" **ONCLICK**="update('ajax', 'ajax.txt')">  <**SPAN** id="ajax">AJAX lava mais branco.</**SPAN**> |

Atenção, que por questões de segurança, a comunicação AJAX só funciona [se o cliente e o servidor estiverem no mesmo domínio](https://en.wikipedia.org/wiki/Same-origin_policy). O exemplo anterior funciona porque esta é uma página situada em **ctp.di.fct.unl.pt** a comunicar com um servidor também situado em **ctp.di.fct.unl.pt**.

#### Pedido síncrono

O pedido XMLHttpRequest anterior foi tratado de forma assíncrona que é o que geralmente se pretende. Mas se, numa situação excecional, não nos importarmos de fazer o pedido de forma síncrona, o que implica bloquear o programa por momentos à espera da resposta, o código da função anterior fica bastante simplificado. A seguinte função permite obter o conteúdo dum ficheiro situado no servidor. Neste código não é possível fazer tratamento de erros, pelo que o programa bloqueia simplesmente se o ficheiro não existir.

|  |
| --- |
| function getFile(filenameURL) {  var xhttp = new XMLHttpRequest(); // cria pedido vazio  xhttp.open("GET", filenameURL, false); // preenche o pedido  try {  xhttp.send(); // envia o pedido para o servidor  return xhttp.responseText;  }  catch(err) {  return null;  }  } |

#### Acesso local

Normalmente, cada pedido XMLHttpRequest é feito usando o protocolo HTTP. Mas muitos browser generalizam o mecanismo e também permitem pedidos XMLHttpRequest para obter recursos situados na mesma máquina do browser, ou seja recursos locais com URLs da forma "file://..." ou "localdir/localfile". Esta funcionalidade é útil para testar os programas com ficheiros locais.

O browser Firefox oferece esta funcionalidade diretamente. No caso do Chrome, esta funcionalidade precisa de ser ativada invocando o browser na linha de comando da seguinte forma:

chrome --allow-file-access-from-files

# XML

XML (eXtensible Markup Language) é uma linguagem que permite representar informação de forma estruturada. Nos sistemas de informação atuais, o XML é imensamente usado, tanto para representar informação armazenada nos servidores, como para representar informação transferidas entre servidores.

Olhando para um documento XML, o aspeto geral é semelhante ao duma página Web (documento HTML). No entanto há importantes diferenças entre os dois formatos:

* O HTML serve para representar o aspeto gráfico de páginas Web usando um conjunto predefinido de **marcas**, e.g. <p>, <h1>, <table>. Permite-se que um documento HTML tenha algumas pequenas omissões, como por exemplo a falta de fechamento de algumas marcas.
* O XML serve para representar informação em geral (independentemente de considerações gráficas) e permite aos utilizadores inventarem as suas próprias marcas. As marcas são escolhidas de forma a exprimir de forma intuitiva o significado atribuído aos dados. Um documento XML tem de cumprir rigidamente a sintaxe prevista.

Em todo o caso, o formato HTML é um caso particular do formato XML, se esquecermos o detalhe do formato HTML sem um pouco menos rígido sintaticamente.

Para exemplificar, eis um pequeno documento XML com alguma informação sobre uma senhora chamada Alice e os seus filhos:

<person gender='F'>

Alice

<children>

<person gender='M'> Andre <children></children> 5 years old </person>

<person gender='F'> Maria <children></children> 4 years old </person>

</children>

35 years old

</person>

Neste exemplo são usadas apenas duas marcas: <person> e <children>. Quando as marcas são bem escolhidas, o documento XML torna-se autoexplicativo. Você deverá conseguir responder a estas perguntas: Qual a idade de Alice? Quantos filhos tem? Qual o nome e idade dos filhos? Ela tem algum neto?

De forma geral, um documento XML tem a estrutura duma árvore n-ária. A parte do texto correspondente à árvore completa ou qualquer subárvore chama-se um **elemento XML**.

Sintaticamente, cada **elemento XML** começa com a abertura duma **marca**, por exemplo, <person>, e termina com o fecho desta mesma marca, </person>. A abertura duma marca pode, opcionalmente, incluir alguns **atributos** com os valores correspondentes, como neste exemplo <person gender='M'>. Finalmente, entre a abertura e o fecho duma marca, encontra-se a **conteúdo** do elemento, que consiste em texto simples com subelementos XML intercalados. A parte de texto simples é tradicionalmente chamada de **PCDATA**. No exemplo da Alice, o conteúdo do elemento principal consiste no seguinte: o segmento de PCDATA "Alice"; em seguida, um subelemento a descrever os filhos de Alice; finalmente, o segmento de PCDATA "35 anos de idade".

Um elemento sem conteúdo, como <children></children> pode ser abreviado <children/>.

Observemos agora um documento XML grande, contendo uma famosa comédia escrita por William Shakespeare: [A Midsummer Night's Dream](http://ctp.di.fct.unl.pt/miei/lap/teoricas/files/dream.xml.txt).

## Tecnologias associadas ao XML

Há um grande número de tecnologias associadas ao XML. Eis algumas:

* [**XML schema**](http://en.wikipedia.org/wiki/XML_schema): Servem para descrever um **tipo de documento XML**, com um conjunto específico de marcas e restrições sintáticas no conteúdo. Por exemplo, todas as peças de teatro de Shakespeare podem ser representadas usando o mesmo conjunto de marcas e a mesma estrutura geral - neste sentido pode falar-se no [tipo das peças de teatro de Shakespeare](http://www.kelschindexing.com/shakesDTD.html).
* [**XML Namespaces**](http://en.wikipedia.org/wiki/XML_namespaces): Permitem evitar conflitos de nomes, constituindo a única forma de reutilizar definições. Assim não há problema em tomar um documento existente e estendê-lo com nova informação e novas marcas.

**Style Sheets**: São mecanismos de especificação da forma como documentos XML devem ser visualizados. Estão disponíveis dois desses mecanismos: [XSL - XML Style Sheets](http://en.wikipedia.org/wiki/XSL_stylesheet) e [CSS - Cascading Style Sheet](http://en.wikipedia.org/wiki/Css). O primeiro mecanismo é o mais poderoso, e permite especificar transformações de XML em XML ou HTML (a sintaxe da linguagem de transformação é, ela própria, baseada em XML e usa o namespace predefinido "xsl"; os nós da árvore a serem tratados especificam-se usando uma linguagem de expressões chamada XPath). O segundo mecanismo permite associar a cada tipo de elemento informação de formatação, e.g. fonte, cor, margem, etc.

* **Protocolos de comunicação**: **SOAP** (Simple Object Access Protocol) consiste num protocolo de transferência de dados entre aplicações; as mensagem têm uma estrutura convencional, sendo codificadas usando o formato XML e recorrendo-se às marcas do namespace "soap". **WSDL** (Web Services Description Language) é protocolo de mais alto nível que estabelecer serviços a que os clientes acedem usando SOAP.
* **APIs para aceder a XML**: Facilitam a escrita de programas que manipulam XML. Estão disponíveis duas APIs padronizadas: **DOM** (Document Object Model) e **SAX** (Simple API for XML). A primeira é usada para construir explicitamente em memória a árvore que corresponde a um documento XML, fornecendo também primitivas para navegar e alterar essa árvore. A segunda permite processar um documento XML sem o ter de carregar em memória, funcionando com base em eventos que são ativados sempre que o parser reconhece algo: o início ou final dum elemento XML, o conteúdo de elemento XML, um erro de parsing, etc. As API DOM e SAX estão implementadas em diversas linguagens, incluindo Java (javax.xml.transform.dom, javax.xml.transform.sax).
* **Estabelecimento de relações**: **XLink** e **XPointer** são dois mecanismos para a criação de links entre documentos XML.

## Processamento de XML em JavaScript

Quando, a resposta a um pedido AJAX vem em XML temos de saber processar o documento recebido. A forma de processar XML é idêntica à do processamento de HTML. Está aqui o essencial do que é preciso saber: [The XML DOM](http://www.w3schools.com/xml/dom_intro.asp).

Eis um pequeno exemplo:

Código do exemplo:

|  |
| --- |
| <**SCRIPT** **TYPE**="text/javascript">  function xml() {  parser = new DOMParser();  serializer = new XMLSerializer();  var text =  "<person gender='F'>" +  "Alice" +  "<children>" +  "<person gender='M'> Andre <children></children> 5 years old </person>" +  "<person gender='F'> Maria <children></children> 4 years old </person>" +  "</children>" +  "35 years old" +  "";  xmlDoc = parser.parseFromString(text,"text/xml");  var nodes = xmlDoc.getElementsByTagName("person");  for(var i = 0 ; i < nodes.length ; i++ )  alert(serializer.serializeToString(nodes[i]));  }  <**/SCRIPT**>  <**INPUT** **TYPE**="button" **VALUE**="Click Me" **ONCLICK**="xml()"> |

# JSON e MongoDB

JSON (JavaScript Object Notation) é outro formato que permite representar informação de forma estruturada. Tem o mesmo poder expressivo do XML, mas é mais simples.

O formato JSON é independente de qualquer linguagem de programação, mas usa as mesmas convenções sintáticas dos objetos literais do JavaScript. Cada objeto JSON é constituido por uma coleção de pares etiqueta/valor e as sequências são representadas por arrays.

Eis um exemplo em XML

<person gender='F'><name>Alice</name><age>35</age><children><person gender='M'></person><person gender='F'></person></children></person>

e a respestiva tradução em JSON, feita [neste site](http://www.utilities-online.info/xmltojson/):

{

"person": {

"-gender": "F",

"name": "Alice",

"age": "35",

"children": {

"person": [

{ "-gender": "M" },

{ "-gender": "F" }

]

}

}

}

O sistema de base de dados [MongoDB](http://www.mongodb.com/) guarda documentos em formato JSON, permitindo indexar por quaisquer campos, mesmo que estejam muito aninhados no interior de subobjetos.

A naturalidade da modelação dos dados e o seu poder de indexação têm tornado o MongoDB [bastante popular](http://blog.mlab.com/2012/08/why-is-mongodb-wildly-popular/) (apesar de tudo está em [4º lugar](http://db-engines.com/en/ranking) atrás de concorrente bem mais poderosos).

O MongoDB é um "[document-oriented database system](http://db-engines.com/en/article/Document+Stores)" e a facilidade e informalidade da sua utilização é bem-vinda em muitas aplicações Web. Contudo, se a base de dados tiver uma grande quantidade de relações e de normalização, então deverá ser preferivel usar uma base de dados relacional clássica.

# Scripting do lado do servidor - Node.js

Como se disse na aula anterior, em 1993 foram introduzidos novos protocolos que passaram a permitir maior dinâmismo do lado do servidor, em particular, a possibilidade de gerar dinamicamente páginas HTML. Nessa altura a programação do lado do servidor começou a ser feita essencialmente em Perl e C, mas rapidamente começaram a aparecer e a ser usadas novas linguagens. Exemplos de linguagens usadas atualmente do lado do servidor: PHP, Java, ASP, Python, Ruby, Scala, TCL, JavaScript.

Um desenvolvimento relativamente recente, foi o aparecimento do Node.js. Trata-se dum interpretador de JavaScript que vem com uma biblioteca que o torna apto a ser usado do lado do servidor. Foi criado em 2009 por Ryan Dahl. Tem tido uma adoção rápida pela comunidade da programação para a Web.

Baseia-se na ideia de usar o modelo de eventos, já conhecido do lado do cliente, também do lado do servidor. Cada pedido que chega é considerado um evento e é tratado assincronamente. Tal como no caso do cliente, existe uma **thread dos eventos** que executa continuamente um ciclo que retira e processa os eventos guardados numa fila de espera. Os eventos são tratados sequencialmente e enquanto não terminar o tratamento dum evento não se inicia o tratamento do seguinte.

Apetece criticar o uso do modelo de eventos do lado do servidor argumentando que o tratamento dum pedido pode ser demorado - realmente pode envolver a consulta duma base de dados ou o carregamento dum ficheiro grande, por exemplo. Mas isso não é problema no caso do Node.js porque ele tem o cuidado de lançar em threads auxiliares as operações de biblioteca mais demoradas, gastando pouco tempo na thread dos eventos. Por outras palavras, muitas das operações da biblioteca do Node.js funcionam de forma assíncrona. As threads auxiliares, ao terminarem, geram elas mesmo eventos de finalização que permitem ao programador aceder ao resultado da operação e prosseguir com mais ações. No exemplo abaixo, o método de biblioteca **readFile** faz o carregamento integral do ficheiro cujo nome é passado no primeiro argumento. O últmo argumento de**readFile** é a função que fará o tratamento do reaultado da leitura quando o evento de finalização ocorrer.

Como já se disse na aula anterior, o uso de lógica assíncrona tem consequências dramáticas no estilo de programação! Nos programas clássicos, o programa consegue controlar todo o fluxo de execução de forma estrita. Na programação assíncrona, escreve-se código para reagir a eventos e não há possibilidade de controlar o fluxo de execução geral. Vamos ver um exemplo.

#### Exemplo - Servidor de ficheiros escrito em Node.js

Primeiro mostramos o código HTML do lado do browser. Vamos testar o servidor na mesma máquina do browser e por isso usamos o domínio "localhost". Escolhemos a porta 8080 por uma questão de tradição. Note que o formulário imediatamente abaixo só funcionará com o servidor estiver ativo, algo que só acontecerá durante a própria aula.

Parte superior do formulário

Get file:  

Parte inferior do formulário

Código:

|  |
| --- |
| <FORM NAME="form1" ACTION="http://localhost:8080">  Get file: <INPUT NAME="q" VALUE="aaa"> <INPUT TYPE="submit">  </FORM> |

Agora o script do lado do servidor. O código é suficientemente autoexplicativo, mas pode ser necessário consultar a [documentação de alguns objetos de biblioteca](https://nodejs.org/api/) (http, url, path, fs). Para correr o script, basta dar o comando **nodejs fileserver.js** do lado so servidor e usar o URL **http://localhost:8080/file?q=myfile.html** do lado do browser.

Agora o código do servidor. Note que o servidor HTTP é criado no último bloco de código e ativado usando o método listen. Repare também que há duas funções que funcionam assincronamente: fs.exists e fs.readFile.

|  |
| --- |
| *// fileserver.js*  var http = require("http");  var path = require("path");  var url = require("url");  var fs = require("fs");  function handleError(error, response) {  console.log("ERROR");  response.writeHeader(500, {"Content-Type": "text/plain"});  response.write(error + "\n");  response.end();  }  function handleNotFound(response) {  console.log("FILE NOT FOUND");  response.writeHeader(404, {"Content-Type": "text/plain"});  response.write("404 Not Found\n");  response.end();  }  function handleIgnore() {  console.log("IGNORE");  }  function handleReply(file, response) {  console.log("OK");  response.writeHeader(200);  response.write(file, "binary");  response.end();  }  var server = http.createServer(function(request, response) {  console.log("Request " + request.url);  var urlParsed = url.parse(request.url, true);  if(urlParsed.query.q == undefined) {  handleIgnore();  return;  }  var localPath = urlParsed.query.q;  var fullPath = path.join(process.cwd(),localPath);  console.log("FullPath " + fullPath);  fs.exists(fullPath, function(exists) {  if(!exists)  handleNotFound(response);  else  fs.readFile(fullPath, "binary",  function(error, file) {  if(error) handleError(error, response);  else handleReply(file, response);  });  });  });  server.listen(8080);  console.log("Fileserver is running on port 8080"); |

# Toolkits e framework para a Web

No mundo da programação Web, a nomenclatura usada para descrever coleções de código relacionado e reutilizável é muito fluida, mas vamos tentar introduzir aqui alguns termos, mais ou menos consensuais quando usados em determinados contextos:

* **API** - Genericamente descreve a interface externa dum conjunto de código, mas por vezes usa-se este termo para designar um módulo duma biblioteca, por exemplo quando se fala da API XMLHttpRequest.
* **Biblioteca** - Coleção de código que está disponível para ser chamado noutros programas. As operações disponíveis funcionam aproximadamente no mesmo nível de abstração. As operações podem ser muito diversas e muitas vezes não têm um objetivo específico comum, como por exemplo no caso da biblioteca do JavaScript. Mas também pode haver um objetivo bem definido, como no caso da biblioteca jQuery.
* **Toolkit** - Trata-se duma biblioteca especialmente rica, focada num domínio específico, e onde as operações trabalham a um nível de abstração elevado. Para uma biblioteca ser considerada um toolkit, é necessário que ofereça uma solução quase completa para o tipo de aplicação em causa. Exemplos: Bootstrap, Google Web Toolkit, etc.
* **Framework** - Trata-se dum sistema de programação focado num domínio específico e completamente genérico, cujas funcionalidades são adaptadas às necessidades usando código escrito pelo programador. O sistema de base já executa e prevê a maioria das necessidades do domínio específico (por vezes incluindo questões complicadas como segurança, escalabilidade e tempo de resposta), mas tem um comportamento por omissão que, muitas vezes, consiste em não fazer quase nada. Usando herança redefinem-se as partes desejadas, podendo ser chamadas funcionalidades do framework que estão disponíveis mas ainda não eram invocadas no código genérico. Por outras palavras, trata-se duma **biblioteca invertida** no sentido em que o fluxo de execução do programa é controlado pelo framework, sendo ele que chama o código do programador (exatamente o inverso do que acontece numa biblioteca normal). Exemplos: CakePHP, JavaServer Faces, Spring, Lift, Ruby on Rails, Express.js, etc.

De todas estas variantes, os frameworks possuem a curva de aprendizagem mais longa. O investimento de tempo tem de ser razoavelmente grande. Não admira: é preciso conhecer todas as abstrações usadas pelo criador do framework e o nosso código tem de se submeter a essas abstrações, implementando especializações delas. Contudo, depois de aprendido o framework, há vantagens no nível de produtividade e na qualidade das aplicações escritas.

# Exemplo dum frameworks para a Web - Express.js

## Introdução

Para dar uma ideia de como o uso dum framework pode tornar mais expedita a escrita dum site Web, vamos usar o framework [Express.js](http://expressjs.com/) (de momento, talvez o mais popular para Node.js) e este [excelente tutorial](http://code.tutsplus.com/tutorials/build-a-complete-mvc-website-with-expressjs--net-34168) desenvolvido por Krasimir Tsonev em 2013. Para consulta, está também aqui a [documentação da API do Express.js](http://expressjs.com/en/4x/api.html).

Para começar, uma descrição rápida do site pretendido. O site deve permitir ao utilizador aceder a diversos tipos de informação, nas categorias "Blog", "Services", "Carrers", "Contacts". Deve também haver um painel de controlo protegido por password, para o administrador poder introduzir nova informação. É tudo.

Note que a forma essencial de usar diretamente o Node.js já foi apresentada atrás. No entanto, dentro dum framework o uso do Node.js é indireto: em muitas situações, trabalha-se a um nível de abstração mais elevado e parte da tarefa de programação consistem em configurar coisas, por exemplo especificando qual o código que deve correr em resposta a um pedido envolvendo um URL particular.

## Instalação

Você é aconselhado a fazer uma instalação efetiva para ganhar um pouco de experiência e poder testar a aplicação.

Assumimos que estamos a trabalhar num sistema Linux com suporte para o sistema de instalação de pacotes **apt**, por exemplo Ubuntu ou Debian.

**Passo 1** - Garantir que se tem a última versão do Node.js instalada e atualizar o gestor de pacotes **npm** do JavaScript.

# curl -sL https://deb.nodesource.com/setup\_0.12 | sudo -E bash -

...

# sudo apt-get install -y nodejs

...

# node -v

v0.12.14

# sudo npm install npm -g

...

# npm -v

3.9.0

**Passo 2** - Instalar o sistema de comunicações seguras Kerberos, o sistema de base de dados MongoDB, o framework Express.js e o o sistema de testes unitários jasmine-node. Note que o MongoDB é grande, ficando a ocupar cerca de 1GB no disco.

# sudo apt-get install libkrb5-dev mongodb

...

# sudo npm install express-generator -g

...

# sudo npm install jasmine-node -g

...

**Passo 3** - Descarregar e instalar a aplicação já escrita:

# wget https://github.com/tutsplus/build-complete-website-expressjs/archive/master.zip

...

# unzip master.zip

...

# cd build-complete-website-expressjs-master/app

...

# npm install # descarrega e instala todas as bibliotecas de que a aplicação depende

...

# jasmine-node ./tests # corre todos os testes unitários para validar instalação.

...

**Passo 4** - Lançar a aplicação. Numa consola fazer:

# cd build-complete-website-expressjs-master/app

...

# node app.js

Successfully connected to mongodb://127.0.0.1:27017

Express server listening on port 3000

GET / 200 1244ms - 2.54kb

GET /images/logo.png 304 158ms

GET /stylesheets/style.css 200 851ms - 3.25kb

GET /images/nav-bg.jpg 304 1ms

GET /images/home-title.jpg 304 1ms

GET /blog 200 29ms - 1.67kb

GET /images/logo.png 304 1ms

GET /stylesheets/style.css 304 3ms

GET /images/nav-bg.jpg 304 1ms

GET /services 200 7ms - 1.3kb

GET /images/logo.png 304 1ms

GET /stylesheets/style.css 304 4ms

GET /careers 200 5ms - 1.3kb

GET /stylesheets/style.css 304 1ms

GET /images/logo.png 304 1ms

GET /contacts 200 3ms - 1.3kb

GET /images/logo.png 304 0ms

GET /stylesheets/style.css 304 2ms

...

**Passo 5** - Para usar a aplicação de forma normal:

# google-chrome [http://localhost:3000](http://localhost:3000/)

...

Para usar a aplicação em modeo de administrador:

# google-chrome <http://localhost:3000/admin>

...

Para aceder ao painel de diagnósticos do MongoDB:

# google-chrome [http://localhost:28017](http://localhost:28017/)

...

## Padrão arquitectural MVC

Como é tipico de muitas aplicações Web, esta aplicação também usa o padrão arquitectural Modelo-Visualizador-Controlado ([Model-View-Controller](https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller)).

O padrão arquitectural MVC decompõe uma aplicação em três partes: (1) gestão dos dados, (2) lógica e (3) interface com o utilizador. Esta decomposição ajuda a tornar o programa mais compreensível e mais fácil de manter a longo prazo.

Eis alguma informação sobre os Modelos, Controladores e Visualizadores da nossa aplicação:

* **Modelos** - Cada modelo representa uma colecção de dados. Encarrega-se da gestão desses dados e da sua validação. Cada modelo está normalmente associado a uma tabela da base de dados, mas também pode estar associado a um ficheiro por exemplo.

Na nossa aplicação, para simplificar, todos os dados são do mesmo tipo e são guardados na mesma coleção. Cada registo tem os seguintes campos: title, text, picture, type. O campo type determina que é o controlador responsável pelo registo. Portanto temos apenas um modelo, a que foi dado o nome de ContentModel.

* **Controladores** - Cada controlador recebe os pedidos do exterior e determina como se deve responder a esses pedidos. A lógica da aplicação reside nos controladores. Costuma definir-se um controlador para cada modelo, mas por vezes há exceções, como no nosso caso em que definimos apenas um modelo.

Na nossa aplicação há vários controladores: Admin, Blog, Home, Page. São os controladores que decidem como é que a aplicação responde a pedidos HTTP e para isso usam-se as funcionalidades de [routing](http://expressjs.com/en/starter/basic-routing.html) do Express.js.

* **Visualizadores** - Os visualizadores definem a forma de apresentar os dados ao utilizadores, mas por vezes também a forma de formatar os dados que devem ser enviados para outros serviços da Web. Um visualizador é sempre comandado por um controlador, recebe do controlador os dados, formata esses dados e envia-os para o browser.

Na nossa aplicação só está definido um visualizador chamado Base. A parte de visualização ainda está pouco desenvolvida e será possível, no futuro, criar subclasses deste visualizador. O nosso visualizador produz HTML usando a primitiva de [render](http://expressjs.com/en/api.html#app.render) do Express.js.

## Geração automática

Como é habitual nestes fremeworks, o Express.js permite gerar o "esqueleto" duma aplicação, já com diversas coisas programadas e configuradas. No caso da nossa aplicação, o comando que Krasimir usou para criar o esqueleto da aplicação foi este:

# express --sessions --css less --hogan app

create : app

create : app/package.json

create : app/app.js

create : app/public

create : app/public/images

create : app/public/stylesheets

create : app/public/stylesheets/style.less

create : app/routes

create : app/routes/index.js

create : app/routes/users.js

create : app/views

create : app/views/index.hjs

create : app/views/error.hjs

create : app/bin

create : app/bin/www

install dependencies:

$ cd app && npm install

run the app:

$ DEBUG=app:\* npm start

create : app/public/javascripts

Depois Krasimir instalou as biblitecas de que a aplicação depende usando o comando:

# npm install

app@0.0.0 /media/big/amd2/xtra-ubuntu-14.04/home/amd/aa/app/app

+-- body-parser@1.13.3

| +-- bytes@2.1.0

| +-- content-type@1.0.2

| +-- depd@1.0.1

| +-- http-errors@1.3.1

| | +-- inherits@2.0.1

| | `-- statuses@1.3.0

| +-- iconv-lite@0.4.11

| +-- on-finished@2.3.0

| | `-- ee-first@1.1.1

| +-- qs@4.0.0

| +-- raw-body@2.1.6

| | +-- bytes@2.3.0

| | +-- iconv-lite@0.4.13

| | `-- unpipe@1.0.0

| `-- type-is@1.6.13

| +-- media-typer@0.3.0

| `-- mime-types@2.1.11

| `-- mime-db@1.23.0

+-- cookie-parser@1.3.5

| +-- cookie@0.1.3

| `-- cookie-signature@1.0.6

+-- debug@2.2.0

| `-- ms@0.7.1

+-- express@4.13.4

| +-- accepts@1.2.13

| | `-- negotiator@0.5.3

| +-- array-flatten@1.1.1

| +-- content-disposition@0.5.1

| +-- cookie@0.1.5

| +-- depd@1.1.0

| +-- escape-html@1.0.3

| +-- etag@1.7.0

| +-- finalhandler@0.4.1

| +-- fresh@0.3.0

| +-- merge-descriptors@1.0.1

| +-- methods@1.1.2

| +-- parseurl@1.3.1

| +-- path-to-regexp@0.1.7

| +-- proxy-addr@1.0.10

| | +-- forwarded@0.1.0

| | `-- ipaddr.js@1.0.5

| +-- range-parser@1.0.3

| +-- send@0.13.1

| | +-- depd@1.1.0

| | +-- destroy@1.0.4

| | +-- mime@1.3.4

| | `-- statuses@1.2.1

| +-- serve-static@1.10.2

| +-- utils-merge@1.0.0

| `-- vary@1.0.1

+-- hjs@0.0.6

| `-- hogan.js@3.0.2

| +-- mkdirp@0.3.0

| `-- nopt@1.0.10

| `-- abbrev@1.0.7

+-- less-middleware@1.0.4

| +-- less@1.7.5

| | +-- clean-css@2.2.23

| | | `-- commander@2.2.0

| | +-- graceful-fs@3.0.8

| | +-- mime@1.2.11

| | +-- mkdirp@0.5.1

| | | `-- minimist@0.0.8

| | +-- request@2.40.0

| | | +-- aws-sign2@0.5.0

| | | +-- forever-agent@0.5.2

| | | +-- form-data@0.1.4

| | | | +-- async@0.9.2

| | | | +-- combined-stream@0.0.7

| | | | | `-- delayed-stream@0.0.5

| | | | `-- mime@1.2.11

| | | +-- hawk@1.1.1

| | | | +-- boom@0.4.2

| | | | +-- cryptiles@0.2.2

| | | | +-- hoek@0.9.1

| | | | `-- sntp@0.2.4

| | | +-- http-signature@0.10.1

| | | | +-- asn1@0.1.11

| | | | +-- assert-plus@0.1.5

| | | | `-- ctype@0.5.3

| | | +-- json-stringify-safe@5.0.1

| | | +-- mime-types@1.0.2

| | | +-- node-uuid@1.4.7

| | | +-- oauth-sign@0.3.0

| | | +-- qs@1.0.2

| | | +-- stringstream@0.0.5

| | | +-- tough-cookie@2.2.2

| | | `-- tunnel-agent@0.4.3

| | `-- source-map@0.1.43

| | `-- amdefine@1.0.0

| +-- mkdirp@0.3.5

| `-- node.extend@1.0.10

| `-- is@0.3.0

+-- morgan@1.6.1

| +-- basic-auth@1.0.4

| `-- on-headers@1.0.1

`-- serve-favicon@2.3.0

O conteúdo do ficheiro /package.json ficou o seguinte:

{

"name": "app",

"version": "0.0.0",

"private": true,

"scripts": {

"start": "node ./bin/www"

},

"dependencies": {

"body-parser": "~1.13.2",

"cookie-parser": "~1.3.5",

"debug": "~2.2.0",

"express": "~4.13.1",

"hjs": "~0.0.6",

"less-middleware": "1.0.x",

"morgan": "~1.6.1",

"serve-favicon": "~2.3.0"

}

}

Para correr a aplicação vazia, fazer:

# npm start

e o seguinte link - [http://localhost:3000](http://localhost:3000/) - permite ver o que a aplicação vazia faz.

Para aprender mais sobre este exemplo, agora só mesmo estudando o [tutorial](http://code.tutsplus.com/tutorials/build-a-complete-mvc-website-with-expressjs--net-34168), consultando ocasionalmente a [documentação da API do Express.js](http://expressjs.com/en/4x/api.html).

[**Tipos**](http://en.wikipedia.org/wiki/Data_type)

Um **tipo** representa uma coleção de elementos de dados e tem associados um conjunto de *literais* mais um conjunto de *operações*.

Por exemplo, em OCaml o tipo int representa o conjunto do inteiros, tem associados os literais ..., -2, -1, 0, 1, 2, etc., e as operações +, -, \*, div, mod, succ, etc.

Numa linguagem sem tipos, e.g. Assembler, uma operação pode ser aplicada a quaisquer dados sem que qualquer validação seja feita. Os dados são vistos como simples sequências de bits e cada operação interpreta uma sequências de bits da maneira que lhe convém.

Numa linguagem com tipos, cada elemento de dados tem um tipo associado. Ao tentar aplicar uma operação a valores que não têm o tipo esperado obtém-se um **erro de tipo**.

Por outras palavras, ao associar um tipo a uma sequência de bits, estamos a atribuir um significado a essa sequência de bits. A linguagem de programação usa a informação de tipo para determinar que as utilizações da sequência de bits fazem sentido ou não. Quando não fizerem sentido, estamos perante um **erro de tipo**.

**Géneros de tipos**

Cada linguagem de programação oferece um conjunto de tipos simples e um conjunto de construtores de tipos. É assim possível distinguir dois géneros de tipos:

* **Tipos simples:** tipos predefinidos cujos valores são escalares, ou seja valores atómicos: inteiros, caracteres, reais, booleanos, enumerados.
* **Tipos estruturados:** tipos compostos definidos pelo programador aplicando determinadas construções a tipos existentes: arrays, registos, listas, tuplos, classes, funções, apontadores.

**Utilidade dos tipos**

Para que servem os tipos?

No caso das linguagens com tipificação estática:

* Permitem a deteção antecipada de erros, em tempo de compilação. **O compilador deve garantir que um programa validado já não vai ter erros de tipo em tempo de execução.**
* Ajudam a documentar os programas, tornando mais claras as intenções do programador (exceto se for usada inferência de tipos).
* Suportam a definição de módulos, que ocultam a representação interna dos valores.
* São úteis na otimização de código gerado.

No caso das linguagens com tipificação dinâmica:

* Suportam um modelo de execução no qual se lança uma exceção logo que seja detetado que um erro de tipo. **O código errado já não chega a ser executado.**

[**Sistemas de tipos**](http://en.wikipedia.org/wiki/Type_system)

Um **sistema de tipos** é um conjunto de regras que:

* Associa tipos aos literais e às expressões duma linguagem. Por exemplo o literal 2 e a expressão 2+5\*3 têm tipo inteiro.
* Determina quais são as manipulações permitidas dos valores de cada tipo. Por exemplo, em OCaml os inteiros podem ser manipulados usando apenas as operações associadas ao tipo inteiro.
* Define como os valores dos vários tipos interagem. Por exemplo, em C a expressão 2 + 5.2 e considerada válida, mas o valor 2 é automaticamente convertido em inteiro.

Para exemplificar, eis três regras que fazem parte do sistema de tipos da linguagem OCaml:

**exp : bool exp' : bool**

**-------------- ------------- ---------------------------**

**false : bool true : bool exp && exp' : bool**

Nestas regras, por cima da barra são colocadas pré-condições e por baixo da barra são colocadas conclusões. As duas primeiras regras são axiomas e dizem que false e true são valores válidos de tipo bool; a terceira regra diz que se exp e exp' forem expressões válidas de tipo bool, então exp && exp' é também uma expressão válida de tipo bool.

Não mostramos mais regras, pois a formalização de sistemas de tipo não é matéria de LAP. Mas, pelo menos, fica a ideia.

Estas regras são úteis por várias razões. Por exemplo, servem para o utilizador compreender melhor a linguagem, servem para provar que o sistema de tipos é consistente (ou seja, que a cada expressão é associado um tipo único), servem como base para a escrita duma parte importante do compilador.

**Verificação de tipos (**[**Type checking**](http://en.wikipedia.org/wiki/Type_checking#Type_checking)**)**

Chama-se **verificação de tipos** ao processo de validação dum programa face às regras dum sistema de tipos.

**Tipificação estática e suas limitações**

Diz-se que uma linguagem usa **tipificação estática** se a verificação de tipos for efetuada em tempo de compilação. Os erros de tipo são detetados antes do programa começar a correr. Nestas linguagens declaram-se os tipos das variáveis, dos argumentos e do resultado das funções para o compilador ter uma base para fazer as suas validações. Em algumas linguagens, como o OCaml, não se declaram tipos porque é feita inferência de tipos.

As seguintes linguagens usam tipificação estática: OCaml, C, C++ e Java.

Numa linguagem com tipificação estática, repare que os tipos são associados às expressões que aparecem no texto dos programas e é o próprio texto que é validado. Depois, durante a execução do programa já compilado, os tipos dos valores são ignorados porque o compilador já garantiu a ausência de erros de tipo.

Mas atenção: Num programa validado, ou seja num programa sem erros de tipo, podem mesmo assim ocorrer outro género de problemas durante a execução:

1. Certos **valores particulares** podem gerar exceções: **zero** usado como denominador duma divisão, ou **null** usado como recetor de mensagem. O sistema de tipos lida com classes de valores (os tipos) e não com valores individuais.
2. O programa pode entrar num **ciclo infinito**.
3. O programa pode estar **incorreto**, ou seja ter erros do lógica. Apesar de funcionar, não faz o que o programador pretendia.

**Tipificação estática e conservadorismo**

Uma característica importantíssima dos sistemas de tipos das linguagens com tipificação estática é serem **conservadores**. Repare que quando se usa tipificação estática, os tipos representam informação incerta, correspondendo a "aproximações" dos valores que realmente serão usados em tempo de execução (os valores exatos não se podem prever, em geral). Para garantir segurança, o compilador tem sempre de assumir o pior caso.

Por exemplo, no código Java que se segue, a variável a, de tipo Animal recebe um gato. Depois, mais adiante, tenta-se fazer miar o animal referido pela variável a. Mas o compilador tem considerar que, nessa altura, a variável a poderá já não referir um gato e, por isso, produz um erro de compilação; no entanto, em tempo de execução até podia não haver problema se estivesse um gato na variável...

Animal a = new Cat() ;

...

a.meou() ; // ERRO DE TIPO

**Tipificação dinâmica e suas limitações**

Diz-se que uma linguagem usa **tipificação dinâmica** se a verificação de tipos for efetuada em tempo de execução. Os erros de tipo são detetados durante a execução dos programas. Nestas linguagens não de declaram os tipos das variáveis, dos argumentos e do resultado das funções. Nestas linguagens uma mesma variável pode conter valores de tipos diferentes, em diferentes momentos da execução.

As seguintes linguagens usam tipificação dinâmica: JavaScript, Prolog, Lisp, Perl, Python, Ruby, APL e Smalltalk.

Numa linguagem com tipificação dinâmica, os tipos são associados aos valores que são usados em tempo de execução e não às variáveis. Portanto, em tempo de execução tem de existir informação de tipo associada aos valores. Sempre que se aplica uma operação a alguns valores, o tipo desses valores é testado pelo sistema de execução.

A limitação característica dos sistemas de tipos dinâmicos é o facto dos erros de tipo só serem detetados durante a execução dos programas. Mesmo um erro de tipo básico pode ficar por descobrir durante muito tempo, por se se situar em código que é executado muito raramente. O erro só será apanhado quando esse código for executado pela primeira vez.

Repare também que numa linguagem com tipificação dinâmica se gasta mais memória com a informação de tipo que é guardada nos valores e se gasta mais tempo a fazer validação de tipos em tempo de execução.

**Tipificação dinâmica e ausência de conservadorismo**

Os sistemas de tipos das linguagens com tipificação dinâmica **não precisam de ser conservadores** pois exercem a sua influência durante a execução dos programas, quando os argumentos exatos das operações são conhecidos e estão disponíveis para teste.

Por exemplo, no código JavaScript que se segue, a variável a recebe um gato. Depois, mais adiante, tenta-se fazer miar o animal referido por essa variável a. O compilador aceita o código, mas depois em tempo de execução, poderá ser detetado ou não um erro de tipo. Se a variável a referir mesmo um gato, tudo correrá bem. Se referir um valor de outro tipo, então ocorrerá um erro de tipo e será lançada uma exceção.

var a = NEW(Cat) ;

...

a.meou() ;

**Elementos de tipificação dinâmica dentro de linguagens com tipificação estática**

Há linguagens de programação com tipificação estática que incluem alguns elementos de tipificação dinâmica.

Um exemplo em Java: a operação **instanceof** permite ao programador testar dinamicamente o tipo de qualquer objeto; a operação de cast aplicada a um objeto também executa um teste de tipo implícito em tempo de execução. Para isto funcionar, os objetos em Java precisam de registar a classe a que pertencem.

**Duas escolas de programação**

Nos dias de hoje, entre as linguagens de programação mais usadas encontram-se tanto linguagens com tipificação estática como linguagens com tipificação dinâmica. Existem alguns adeptos enquadrados em cada uma das duas escolas de programação, os quais defendem renhidamente a sua "dama" (veja por exemplo esta [discussão](http://lambda-the-ultimate.org/node/983)).

Por exemplo, os adeptos da tipificação estática acreditam que os seus programas são mais seguros depois de verificados, mas os adeptos da tipificação dinâmica argumentam que conseguem programar melhor as suas ideias sem constrangimentos artificiais e que apesar de tudo os seus programas têm provado ser robustos e conter poucos erros.

Na verdade é perfeitamente possível ser bons resultados dentro de cada uma das escolas de programação desde que se usem boas técnicas de desenvolvimento de software. Uma das técnica mais importante é certamente a técnica de escrever muitos [testes unitários](http://en.wikipedia.org/wiki/Unit_test) para validar sistematicamente todos os aspetos do software em desenvolvimento.

Só uma curiosidade: O CLIP, o sistema da informação da FCT, é um exemplo de software da escola da tipificação dinâmica - está escrito em Prolog.

**Sistemas de tipos com e sem falhas de proteção**

Um sistema de tipos diz-se **sem falhas** (em Inglês, ***safe***) se conseguir garantir que todos os erros de tipo são detetados e que código com erros de tipo nunca chegará a ser executado.

Os sistemas de tipos do C e C++ têm falhas porque:

* Alguns casts permitem violar o sistema de tipos de forma muito básica. Por exemplo é possível converter um inteiro num apontador para função.
* Mesmo sem usar um cast, usando aritmética de apontadores é fácil pôr um apontador inteiro a apontar para uma variável de tipo double e depois escrever nela (ou em parte dela) um inteiro.
* A gestão explícita de memória também é um problema. Suponhamos que temos um apontador para uma variável dinâmica inteira e que libertamos essa variável. Suponhamos que logo a seguir criarmos uma variável dinâmica de tipo double. Suponhamos ainda que, por acaso, a variável é criada na zona de memória anteriormente libertada, mas ainda apontada pelo apontador inteiro...

Exemplos de linguagens com sistemas de tipos sem falhas:

* OCaml
* Java
* Smalltalk
* JavaScript

Repare que na lista de linguagens com sistemas de tipos sem falhas aparecem linguagens com tipificação estática e linguagens com tipificação dinâmica.

**Polimorfismo**

Os programadores gostam de escrever código geral que possa ser aplicado a vários tipos de dados. É penoso, e causador de erros, ter de reescrever um algoritmo com ligeiras variações só porque surgiu a necessidade de o aplicar a um tipo de dados diferente.

Uma **função polimórfica** é uma função que pode ser aplicada a argumentos de vários tipos. A nossa conhecida função len em OCaml é polimórfica pois aplica-se a listas de qualquer tipo:

len : 'a list -> int

Um **tipo polimórfico** é um tipo cujas operações se aplicam a valores de mais do que um tipo. Em OCaml o tipo da listas 'a list é polimórfico. Em Java o tipo Vector<E> também é.

Uma **variável polimórfica** é uma variável mutável que pode conter valores de tipos diferentes. Em Java, uma variável de tipo Animal pode referir qualquer objeto cujo tipo seja subtipo de Animal. Em C uma variável de tipo void \* pode guardar qualquer apontador.

Entidades que não sejam polimórficas dizem-se **monomórficas**.

Muitas das linguagens com tipificação estática modernas suportam polimorfismo. Todas as linguagens com tipificação dinâmica suportam polimorfismo de forma inerente.

**Variedades de polimorfismo**

O seguinte diagrama identifica as variedades e subvariedades de polimorfismo de funções em linguagens com tipificação estática, de acordo com [Cardelli](http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf):

Polimorfismo

* + Universal
    - Paramétrico
    - Inclusão (ou subtipo)
  + Ad hoc
    - Overloading
    - Coerção

**Polimorfismo universal** - A função trabalha de forma uniforme sobre uma diversidade infinita de tipos que partilham a mesma estrutura. A implementação é única e o mesmo código consegue lidar com todos os tipos considerados.

**Polimorfismo paramétrico** - É uma forma de polimorfismo universal onde a função polimórfica tem um parâmetro de tipo implícito ou explicito. Na chamada da função o parâmetro de tipo pode ser ou não inferido. A função len em OCaml é polimórfica paramétrica, sendo 'a o nome do parâmetro de tipo:

len : 'a list -> int

let rec len l =

match l with

[] -> 0

| x::xs -> 1 + len xs

;;

A seguinte função em Java é polimórfica paramétrica, sendo T o nome do parâmetro de tipo:

<T> void fromArrayToCollection(T[] a, Collection<T> c) {

for (T o : a) {

c.add(o) ;

}}

**Polimorfismo de inclusão** - É uma forma de polimorfismo universal que resulta da noção de subtipo. Uma função que declare aceita argumentos dum dado tipo, digamos Animal, também aceita argumentos de subtipos desse tipo, digamos Cat ou Lion. Qualquer linguagens com subtipos suporta polimorfismo de inclusão.

A seguinte função em Java é polimórfica de inclusão:

int weight(Animal a) { ... }

**Polimorfismo ad hoc** - A função trabalha de forma não uniforme sobre uma diversidade finita de tipos que não partilham a mesma estrutura. Existem múltiplas implementações, uma para cada tipo considerado.

**Overloading** - O mesmo nome de função é usado para denotar diferentes implementações monomórficas. No ponto da chamada usa-se o contexto para descobrir qual das implementações deve ser usada. Portanto esta forma de polimorfismo não é mais do que uma conveniência sintática. Um exemplo: o operador "+" em Java denota três operações monomórficas distintas, não relacionadas entre si: (1) soma de inteiros; (2) soma de reais; (3) concatenação de strings.

**Coerção** - Uma coerção é uma conversão automática de tipo. As coerções fazem com que funções essencialmente monomórficas se tornem polimórficas, pois passam a poder ser chamadas com argumentos de diferentes tipos. A seguinte função em C foi escrita para ser monomórfica

double inc(double d) { return d + 1 ; }

mas devido ao facto de em C existir coerção de inteiros para doubles, a função passa a poder ser aplicada tanto a reais como inteiros.

Lendo as descrição anteriores, percebe-se porque razão o "polimorfismo ad hoc" também se chama "polimorfismo aparente". É apenas "açúcar sintático" que está em jogo.

# Ambientes de programação

Um **ambiente de programação** é uma coleção de ferramentas usadas no desenvolvimento de software. Um bom sistema de desenvolvimento aumenta a produtividade dos programadores e também contribui um pouco para aumentar a qualidade do software produzido.

Um ambiente minimal é tipicamente constituído por:

* um sistema de ficheiros,
* um editor de texto,
* um interpretador (ou, em alternativa, um compilador e um ligador)

Um ambiente sofisticado é tipicamente constituído por:

* um sistema de ficheiros,
* numerosas ferramentas, incluindo editor, depurador (debugger), gerador de documentação, framework para testes unitários, etc.
* essas ferramentas estão bem integradas umas com as outras;
* são oferecidas ao utilizador duas interfaces uniformes: uma interfaces gráfica e uma interface baseada em linha de comando.

# O ambiente de desenvolvimento JDK para Java

O ambiente de desenvolvimento não-visual JDK (Java Developer's Kit) consiste num conjunto de ferramentas que ajudam a **desenvolver**, **testar**, **documentar** e **executar** programas em Java. Estas ferramentas podem ser usadas na linha de comando, mas também podem ser integradas em ambientes de desenvolvimento gráficos, como por exemplo o Eclipse. O JDK é gratuitamente disponibilizado pela empresa Sun e existem versões para Windows, MacOS, Linux, etc.

## Ferramentas do JDK

**javac** Compilador

**java** Executor de *programas independentes*

**javadoc** Gerador de documentação

**appletviewer** Executor de *applets*

**jdb** Depurador

**javap** Disassembler

**javah** Gerador de *header files* para métodos nativos escritos em C

#### Como se compilam programas em Java?

* Os programas em Java compilam-se por meio do comando **javac**. Por exemplo: javac Sets.java. O comando **javac** permite compilar tanto **programas independentes** como **applets**.
* Os ficheiros compilados pelo comando **javac** têm obrigatoriamente a extensão ".java".
* Quando cativado, o compilador cria na diretoria corrente diversos ficheiros com a extensão ".class", um por cada classe ou interface definidas nos ficheiros fonte. Os ficheiros ".class" contêm código executável, num formato padrão chamado **bytecode**.

#### Como se executam programas em Java?

* Se se tratarem de **programas independentes**, executam-se usando o comando **java**. Por exemplo: java TestSets. Neste exemplo, "TestSets" refere-se ao ficheiro "TestSets.class" na diretoria corrente: repare que o comando java não admite a escrita da extensão ".class".
* Se se tratarem de **applets**, executam-se usando o comando **appletviewer** ou então por intermédio dum browser WWW.
* A componente de software que permite executar ficheiros de de *bytecode* chama-se **Java Virtual Machine**. Está incorporada no comando **java**, no comando **appletviewer** e na maioria dos browsers WWW modernos.

#### Como se documentam programas em Java?

* O gerador de documentação, **javadoc** converte ficheiros ".java" em ficheiros HTML, visualizáveis usando qualquer browser WWW. Estes ficheiros HTML incluem documentação sobre as classes, interfaces, métodos, variáveis e exceções definidos nos ficheiros fonte e ainda o conteúdo de comentários especiais, da forma /\*\* ... \*/, escritos pelo programador. A vasta documentação sobre a plataforma Java que é fornecida pela Sun foi gerada pelo comando **javadoc**.

# As ferramentas do OCaml

A distribuição do OCaml inclui uma vasto conjunto de ferramentas disponíveis no Windows, Linux e MacOS. Estas ferramentas podem ser usadas na linha de comando, mas também podem ser integradas em ambientes de desenvolvimento gráficos.

Repare que a lista de ferramentas abaixo inclui um [profiler](http://en.wikipedia.org/wiki/Profiler_%28computer_science%29) que serve para o programador descobrir quais as partes do programa onde se gasta mais tempo. Essas são as partes do programa que mais interessa optimizar.

## Ferramentas do OCaml

**ocaml** Interpretador

**ocamlc** Compilador para a máquina virtual CAML

**ocamlopt** Compilador de código nativo

**ocamlrun** Executor da máquina abstrata CAML

**ocamlbrowser** Permite navegar e inspecionar o conteúdo dos módulos de biblioteca

**ocamldebug** Depurador de código fonte

**ocamldoc** Gerador de documentação

**ocamldep** Inspeciona um conjunto de ficheiros fonte e gera automaticamente informação de dependências para a ferramenta make

**ocamlcp** Profiler, insere no código fonte a contagem de quantas vezes cada função é chamada, etc.

**ocamlprof** Interpreta o output da execução de programas processados usando ocamlcp

**ocamllex** Gera reconhecedores de expressões regulares diretamente a partir dessas expressões

**ocamlyacc** Gera parsers diretamente a partir de gramáticas LALR(1)

**ocamlp4** Processador do código fonte de programas ocaml (para pretty-print, por exemplo)

**ocamldumpobj** Disassembler de ficheiros .cmo

**ocamlobjinfo** Inspeciona ficheiros .cmo, .cmi, .cma

# Unix e Gnu/Linux

O Unix é um sistema operativo mas também um sistema de desenvolvimento e manutenção de software. Atualmente a sua versão open-source, chamada de Gnu/Linux, é muito popular e tem sido usada no desenvolvimento de software "open-source" e de software comercial.

O Unix disponibiliza uma grande quantidade de ferramentas de desenvolvimento, mas deve ser dito que essas ferramentas foram nascendo de forma caótica e que não há uma interface consistente entre elas. Algumas ferramentas são também muito complexas, embora cumpram os seus objetivos de forma admirável, na maior parte dos casos.

## [gnu tools](http://en.wikipedia.org/wiki/GNU_build_system)

Os **gnu tools** são um conjunto de ferramentas que ajudam a produzir software portável, que funciona em diversas versões do Unix e noutros sistemas operativos. Estas ferramentas foram desenvolvidas e adquiriram maturidade especialmente durante os anos 1990.

**autoconf** Ferramenta para criar projetos de software portáveis e reconfiguráveis

**automake** Gerador automático de makefiles

**libtool** Ferramenta para criar bibliotecas de software

## [make](http://en.wikipedia.org/wiki/Makefile)

A utilitário **make** pode ser usado para gerir automaticamente pequenos projetos. A sua principal utilização é determinar automaticamente que partes do projeto precisam de ser recompiladas e produzir os comandos para concretizar a recompilação. O make considera as dependências entre os diversos ficheiros do projeto para saber o que é precisos fazer em cada momento.

Geralmente, cada projeto contém um ficheiro chamado **Makefile** onde se declaram as dependências entre os ficheiros, e onde se definem regras que dizem como os diversos ficheiros devem ser atualizados. Para além da recompilação, podem ser definidas outras ações, tais como apagar os ficheiros auxiliares do projeto, ou produzir o arquivo da distribuição do projeto.

O seguinte exemplo é o ficheiro Makefile que foi usado no primeiro projeto de LAP. O ficheiro declara as dependências entre ficheiros e tem regras que dizem como obter uns ficheiros a partir de outros (as regras começam obrigatoriamente por um tab).

|  |
| --- |
| APP = dt  VERSION = 0.1  COMP = ocamlc  INTERFS = DTree.mli  OBJS = DTree.cmo dt.cmo  GROUP = 123\_456  $(APP): $(OBJS)  $(COMP) -o $(APP) $(INTERFS) $(OBJS)  %.cmo: %.ml  $(COMP) $(INTERFS) -c $<  clean:  rm -f $(APP) \*.cmo \*.cmi  dist: clean  mkdir $(GROUP)  cp \*.ml \*.mli $(GROUP)  zip -r $(GROUP).zip $(GROUP)  rm -rf $(GROUP) |

Eis uma pequena sessão interativa, que pressupõe a existência do ficheiro Makefile na diretoria corrente:

|  |
| --- |
| $ **ls**  dt.ml DTree.ml DTree.mli Makefile  $ **make DTree.cmo**  ocamlc DTree.mli -c DTree.ml  $ **make**  ocamlc DTree.mli -c dt.ml  ocamlc -o dt DTree.mli DTree.cmo dt.cmo  $ **make clean**  rm -f dt \*.cmo \*.cmi  $ **make dist**  rm -f dt \*.cmo \*.cmi  mkdir 123\_456  cp \*.ml \*.mli 123\_456  zip -r 123\_456.zip 123\_456  adding: 123\_456/ (stored 0%)  adding: 123\_456/DTree.ml (deflated 72%)  adding: 123\_456/dt.ml (deflated 67%)  adding: 123\_456/DTree.mli (deflated 65%)  rm -rf 123\_456  $ **ls**  123\_456.zip dt.ml DTree.ml DTree.mli Makefile |

## [gdb](http://en.wikipedia.org/wiki/Gdb)

O Gnu Debugger trabalha com linguagens implementadas nativamente e tem um funcionalidade extensíssima. Mostramos só como usar o gdb para descobrir qual a função onde um programa está a *rebentar*.

O seguinte programa está errado:

|  |
| --- |
| char str[128] ;  void f(void)  {  char \*pt = str ;  int i ;  for( i = 0 ; i < 10000 ; i++ ) <-- Provoca estoiro  \*pt++ = 'a' ;  }  void g(void) {  f() ;  }  int main()  {  g() ;  return 0 ;  } |

O gdb permite mostrar o conteúdo da pilha de execução no momento do estoiro, o que geralmente é muito útil para descobrir as razões do erro.

|  |
| --- |
| $ **./a**  Segmentation fault (core dumped)  $ **gdb ./a**  GNU gdb 6.6-debian  Copyright (C) 2006 Free Software Foundation, Inc.  (gdb) **run**  Starting program: /media/EXTERN/amd1/z/a  Program received signal SIGSEGV, Segmentation fault.  0x0804835d in f () <-- Estoiro na função f  (gdb) **backtrace**  #0 0x0804835d in f ()  #1 0x0804837b in g ()  #2 0x08048390 in main () <-- Pilha de execução no momento do estoiro  (gdb) |

## [cvs](http://en.wikipedia.org/wiki/Concurrent_Versions_System)

O **cvs** (Concurrent Versions System) um sistema de apoio ao desenvolvimento de projetos de software. Tem duas funcionalidades essenciais:

* Suporta o **registo cronológico** das alterações introduzidas nos ficheiros fonte do projeto;
* Suporta a **colaboração** entre os diversos participantes de projeto, e sabe lidar com o problema da submissão de **alterações conflituantes**.

O CVS, divulgado em 1986, foi o primeiro sistema de controlo de versões a suportar um repositório partilhado por vários utilizadores.

Uma ferramenta como o CVS é essencial para a viabilidade do software livre, pois este tipo de software é tipicamente desenvolvido coletivamente por um grande número de voluntários geograficamente espalhados e não pagos. Nenhum desses voluntários teria tempo nem paciência para processar manualmente as numerosas submissões de código novo ou corrigido.

Uma ferramenta do tipo do CVS também faz falta no desenvolvimento de software proprietário, mas no caso do software livre a questão é mais evidente.

#### Exemplos de software originalmente desenvolvido com a ajuda do CVS

* Apache web server
* X window system
* GNOME desktop
* Mozilla web browser
* OpenOffice
* GCC gnu compiler collection
* Debian linux distribuition
* Anjuta IDE for rapid application development

Em 1999 foi lançado o [Sourceforge](http://www.sourceforge.net/), um repositório de código fonte CVS baseado na Web. Foi um dos primeiros sistemas Web a oferecer a possibilidade de gestão colaborativa de projetos de código aberto. Neste momento o sistema hospeda mais de 300 000 projetos, embora nem todos estejam ativos. O próprio sistema Sourceforge era inicialmente baseado em código aberto, embora tal não seja verdade atualmente.

#### Alternativas modernas ao CVS

O CVS tem algumas limitações aborrecidas, como por exemplo a falta de suporte para a noção de renomear uma diretoria. Atualmente há muitas alternativas ao CVS e, para iniciar projetos novos, convém evitar o CVS.

* Alternativas *open source*: há cerca de duas dezenas de alternativas, por exemplo, [Subversion](http://subversion.apache.org/), [Monotone](http://www.monotone.ca/), [Git](https://git-scm.com/), [Bazzar](http://bazaar.canonical.com/en/).
* Alternativas comerciais: há apenas meia dúzia, por exemplo, [BitKeeper](http://www.bitkeeper.com/) e [Plastic SCM](https://www.plasticscm.com/home.html).

**Subversion** (2004) - O Subversion foi criado para resolver as insuficiências mais evidentes do CVS. Foi a primeira alternativa popular ao CVS e continua a ser muito usado.

**Git** (2005) - Mas, atualmente, o sistema mais usado é provavelmente o Git, por causa de diversas funcionalidades específicas e duma boa velocidade de resposta, mesmo em projetos grandes. O plugin para o Eclipse [EGit](http://www.eclipse.org/egit/) constitui uma interface prática para usar o Git.

**GitHub** (2008) - Atualmente é o maior repositório de código fonte do mundo. Usa o Git e está acessível na Web. Tem mais de 21 milhões de projetos instalados, embora nem todos estejam ativos.

**Detalhes sobre o Git** - Todos os participantes dum projeto possuem a sua cópia local do projeto e de todo o código. Trabalham localmente sobre o código e periodicamente usam os comandos do Git para atualizar o estado do projeto local. Sempre que um participante considerar conveniente, dá o comando que incorpora no projeto central as alterações entretanto introduzidas no projeto local. Eis alguns exemplos de comandos:

git clone https://user@github.com/Project/project.git

git status

git add myfile

git commit

git push

# [Conhecer melhor o Git em 15 minutos!](https://try.github.io/levels/1/challenges/1)

# IDEs

Um [IDE](http://en.wikipedia.org/wiki/Integrated_development_environment), ou ambiente de desenvolvimento integrado, é um programa único dentro do qual se processa todo o desenvolvimento do software. Geralmente a interface é gráfica.

Exemplos: Eclipse, Visual Studio .Net, Anjuta, Code::Blocks.

Os IDEs mais sofisticados são estendíveis através de plugins. Isso permite que o mesmo sistema possa ser usado para trabalhar em diversas linguagens.

**Escolha duma linguagem de programação**

A escolha duma linguagem de programação para usar num projeto de programação depende de muitas circunstâncias. Em contextos diferentes deve dar-se maior ou menor importância a diferentes aspetos, e pode ser preferível usar uma determinada linguagem em vez de outra. Em algumas situações é conveniente usar mais do que uma linguagem.

* Experiência prévia com a linguagem.
* Facilidade em aprender a linguagem.
* Facilidade em usar a linguagem.
* Suporte de boas práticas de programação.
* Suporte de tecnologias de desenvolvimento de software recentes.
* Qualidade das ferramentas de desenvolvimento disponíveis.
* Disponibilidade de grande número de programadores.
* Suporte para deteção atempada de bugs.
* Padronização da linguagem.
* Expressividade e produtividade.
* Riqueza das bibliotecas disponíveis.
* Portabilidade.
* Segurança.
* Velocidade de execução.
* Custo.
* Facilidade em escrever scripts.
* Adequação a programação de sistemas.
* Suporte para concorrência.
* Suporte para introspeção.

As características das próprias linguagens são um dos fatores mais importantes a levar em conta. No entanto, há que saber que, com a passagem do tempo, as linguagens mais usadas tendem a evoluir e a libertar-se de algumas das suas restrições iniciais. Há uma tendência para as linguagens evoluirem no sentido de suportar programação multiparadigma e para as linguagens mais lentas terem a sua implementação otimizada e ficarem mais eficientes.

Por exemplo: a úmtima versão do Java já suportar programação funcional em alguma medida; a linguagem PHP também ganhou suporte para programação funcional; a linguagem CamlLight ganhou classes e objetos e passou a chamar-se OCaml; foi criada uma versão do C com classes e objetos chamada C++; foi criada uma versão do JavaScript chamada TypeScript com tipificação estática, mais adequada ao desenvolvimento de projetos grandes; a linguagem JavaScript era lenta mas agora existem implementações muito rápidas; etc.

**Qualidade duma Linguagem de Programação**

Revisitar a aula teórica 1.

**Discussão sobre as linguagens usadas na disciplina**

**OCaml**

* Linguagem simples, fácil de usar, suporta abstrações muito poderosas.
* A programação pode ser de muito alto-nível e os programas tendem a ser compactos.
* Tem um conjunto de bibliotecas rico e há muitas ferramentas desenvolvidas para serem usadas com o OCaml.
* Suporta programação funcional, imperativa e orientada pelos objetos. (Na nossa cadeira usámos só a primeira parte, por razões pedagógicas.)

**C**

* Grande liberdade para explorar os recursos da máquina -> programação de sistemas.
* Linguagem insegura mas muito eficiente.
* Nível excelente de interoperabilidade - quase todas as linguagens suportam mecanismos de ligação ao C.
* Depuração difícil devido ao uso de apontadores e à necessidade de gerir a memória explicitamente. Os programas podem ser difíceis de ler.

**JavaScript**

* Linguagem muito prática de usar, dá grande liberdade ao programador, mantendo um razoável nível de segurança quando o que está em causa é a escrita de pequenos scripts.
* Tem um sistema de tipos dinâmico e um sistema de objetos baseado em protótipos.
* Serve para programar páginas WEB interativas, mas também é usada como linguagens de scripting em muitas outras aplicações.