



# Conceitos elementares de JavaScript

## Objetivos:

- Sintaxe JavaScript
- Interacção com o DOM
- Temporizadores
- Eventos

## 10.1 JavaScript

JavaScript (JS)[1] é uma linguagem interpretada, muito utilizada em páginas Web, mas não só. O facto de ser interpretada significa que não é necessário um passo explícito de compilação para produção de um objeto executável, como acontece com as linguagens *Java* ou *C*. A vantagem da interpretação é que permite um desenvolvimento mais rápido por executar diretamente o código escrito pelo programador. A desvantagem é que muitos erros só são detectados quando o fluxo de execução atinge a linha onde o erro está presente. Além disso, código interpretado é bastante mais lento do que código compilado. Na verdade, os browsers atuais fazem compilação Just-in-Time (JIT), que consiste em processar o código JS progressivamente e compilá-lo à medida que é necessário. Assim não é necessário repetir o passo de tradução em execuções posteriores do código, o que melhora o desempenho.

A linguagem JavaScript pode ser incluída em documentos HyperText Markup Language (HTML)[2] e tem a capacidade de aceder a elementos da página. Isto é, através de JS é possível interagir com a página, alterando o seu conteúdo e apresentação de forma dinâmica. Neste guião serão explorados aspetos genéricos de JS, como a sua sintaxe, e aspetos específicos de interação com elementos HTML.

## 10.2 Inclusão numa página

A inclusão de JS numa página *Web* é análoga à inclusão de estilos Cascading Style Sheets (CSS)[3]. Ou seja, através de elementos específicos na página, é possível incluir o código JS diretamente ou obter o código de uma localização externa.

O exemplo que se segue demonstra o processo de inclusão direta na página, através da marca `<script>`.

---

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      /* Código JavaScript aqui */
    </script>
  </head>

  <body>
    ...
  </body>
</html>
```

---

A alternativa é obter o código de um recurso externo, que por motivos de modularidade, reutilização e desempenho é o método recomendado.

---

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="labi.js"> </script>
  </head>

  <body>
    ...
  </body>
</html>
```

---

O conteúdo do ficheiro **labi.js** será código JS, seguindo a sintaxe definida na Secção 10.3. Tal como no caso dos estilos, é comum colocar os recursos de JS num diretório diferente da página. Nomes comuns para esse diretório são **scripts** ou **js**.

### Exercício 10.1

Construa uma pequena página, utilizando o exemplo apresentado anteriormente. Verifique que o navegador tenta obter o ficheiro **labi.js**.

A linguagem JS é bastante poderosa e o facto de executar em qualquer navegador permite desenvolver aplicações que podem ser distribuídas de forma muito eficaz. No entanto, note que qualquer código JS é sempre enviado ao cliente na sua forma textual, podendo ser facilmente copiado.

## 10.3 Sintaxe

A sintaxe da linguagem JS é inspirada na linguagem **C** e algo semelhante à linguagem *Java*. Este guião não irá explorar com detalhe todos os aspetos de sintaxe, ou todas as propriedades da linguagem, mas irá possibilitar uma utilização básica da mesma.

A sintaxe básica da linguagem JS é baseada em instruções, que são organizadas por linhas. Cada linha corresponde a uma instrução, podendo estas instruções ser terminadas com o carácter `;`. A utilização deste carácter é facultativo mas muito recomendado. JS é *case-sensitive*, o que significa que se deve ter cuidado na escrita.

O exemplo seguinte declara uma variável **x**, atribui-lhe um valor e apresenta o resultado na consola do navegador. Utilizando o exemplo anterior, este código estaria dentro do ficheiro **lab1.js**, mas também poderia ser declarado numa marca **<script>**.

---

```
/* Comentário */  
var s = "3";  
var x;  
x = 3;  
console.log(x);
```

---

Como deve identificar, existe uma chamada a uma função **console.log**, com o argumento **x**. Esta sintaxe é em tudo semelhante ao *Java*. A declaração de variáveis por sua vez é diferente. Isto deve-se ao facto de JS ser uma linguagem com tipos dinâmicos, não sendo necessário declarar explicitamente qual o tipo da variável. Portanto, todas as variáveis

são declaradas da mesma forma. É o conteúdo que determina como ela será utilizada.

### Exercício 10.2

Seguindo o exemplo anterior, replique os dois exemplos anteriores no seu computador. Aceda à consola do navegador<sup>a</sup> e verifique o valor impresso. Experimente com outros valores.

Para voltar a executar o código JS basta atualizar a página do navegador, o que tipicamente se consegue através da tecla **F5**, ou da combinação **CMD + R** no caso do sistema OS X.

<sup>a</sup>No firefox, pode ativar a consola no menu Tools/Web Developer/Web Console.

### Exercício 10.3

Experimente substituir a chamada `console.log` por `document.write` e `alert`, de forma a verificar como o JS pode apresentar mensagens aos utilizadores.

Podem ser aplicados operadores aritméticos às variáveis, tais como a adição (+), ou a subtração (-). No entanto o significado desta operação irá variar com o tipo de variável (que depende do seu conteúdo atual). Um bom exemplo é o operador +, que no caso de números irá calcular a soma, mas no caso de sequências de caracteres irá concatená-las.

O exemplo seguinte demonstra a aplicação do operador +:

```
var s = "3";  
var x = 3;  
console.log(s+1);  
console.log(x+1);
```

Em que o resultado deverá ser:

31  
4

### Exercício 10.4

Replique o exemplo anterior no seu computador. Acesse à consola do navegador e verifique o valor impresso. Experimente com outros valores, números reais e sequências de caracteres.

### Exercício 10.5

Verifique o que acontece quando troca o tipo de uma variável. Isto é, considerando que `s` é uma *String*, defina que passa a ser um inteiro (`s=3;`), realize operações aritméticas e mostre o seu conteúdo.

Quando uma operação aritmética não é válida, a linguagem JS faz uso do termo **NaN** que significa *Not a Number*. Isto pode ser facilmente obtido se se subtrair um inteiro a uma *String*.

### Exercício 10.6

Verifique o resultado do seguinte pedaço de código. Poderá ser útil relembrar uma certa banda sonora.

```
for(var i=0;i<16;i++){
    document.write("uma-string" - 2);
    document.write("<br />");
}
document.write("Batman");
```

## 10.3.1 Funções

De forma a melhor organizar o código, e evitar a replicação desnecessária, é possível organizar um programa em funções. Estes elementos são constituídos por um nome, uma lista de argumentos e um corpo. Tal como a declaração das variáveis é indicada pela palavra reservada **var**, a declaração de funções faz uso da palavra reservada **function**, tal como descrito no exemplo seguinte:

```
function nome_da_funcao(arg1, arg2, arg3){
    /* ...Conteúdo... */
}
```

Comparando com a linguagem *Java*, verifica-se que não é necessário declarar qual o tipo

de retorno da função, nem os tipos dos parâmetros.

Um exemplo simples, de uma função que realiza a soma de dois números, pode ser declarada e invocada da seguinte forma:

```
function soma(x,y){  
    return x+y;  
}  
  
var resultado = soma(3,4);  
console.log(resultado);
```

### Exercício 10.7

Construa um programa em JS com quatro funções, uma para cada operação aritmética elementar. Invoque as funções criadas e apresente o resultado.

### 10.3.2 Condições

A execução condicional segue uma sintaxe semelhante ao *Java* e é implementada através das palavras reservadas **if**, **else**, no seguinte formato:

```
if ( /*condição*/ ) {  
    /* Instruções no caso verdadeiro */  
} else {  
    /* Instruções no caso falso */  
}
```

As chavetas podem ser omitidas caso apenas exista uma instrução a executar. Na condição poderá utilizar operadores de comparação tais como **<**, **>**, **>=**, **==**, **!=**, etc. Isto é em tudo semelhante ao *Java*. No entanto, existe uma diferença fundamental, que advém do facto de os tipos das variáveis serem dinâmicos. Considere o seguinte excerto:

```
var a = "3";  
var b = 3;  
  
if (a == b)  
    alert("Iguais");  
else  
    alert("Diferentes");
```

Se executar este excerto, irá verificar que em JS o operador igual (==) permite comparar tipos diferentes, convertendo os seus valores. No entanto, por vezes pretende-se efetuar uma comparação do valor e do tipo. Para isso, existe o operador === e a sua negação, o operador !==. Na linguagem JS diz-se que estes comparadores verificam se o valor é igual e o tipo idêntico. No caso anterior, **a** é igual a **b** mas as variáveis não são idênticas.

### Exercício 10.8

Repita o exemplo anterior e compare o resultado utilizando os operadores == e ===.

É possível ainda utilizar a operação **switch**, com uma sintaxe semelhante ao *Java*:

```
var a ="abc";
switch(a) {
  case "abc": alert("string abc"); break;
  case 3: alert("inteiro 3"); break;
  default: alert("outro");
}
```

### Exercício 10.9

Verifique como pode utilizar a operação **switch** misturando tipos diferentes.

#### 10.3.3 Ciclos

Para implementar ciclos, a linguagem JS suporta as mesmas instruções que *Java*: **while**, **do-while**, **for**.

```
do {
  /*instruções*/
} while (/*condição*/);

while (/*condição*/) {
  /*instruções*/
}

for (/*início*/ ; /*condição*/ ; /*incremento*/) {
  /*instruções*/
}
```

---

### Exercício 10.10

Pratique a utilização de ciclos em JS implementando um caso para cada um dos exemplos apresentados.

---

## 10.4 Interação com o DOM

O grande potencial da linguagem JS quando é executada no navegador é a possibilidade de aceder a qualquer elemento HTML, sendo possível manipular em tempo real qualquer aspeto. Isto é, através de JS é possível alterar o conteúdo da página, estilos e marcas após a página ter sido carregada no navegador. A característica que possibilita esta interação é chamada de Document Object Model. Tal como o nome indica, o Document Object Model (DOM)[4] cria um modelo de objetos da página HTML. Estes objetos podem depois ser manipulados na linguagem JS. Os objetos são entidades que possuem propriedades e métodos. É possível consultar ou alterar o valor das propriedades de um objeto (ex, o atributo `href` de uma marca `<a>`) e invocar métodos para produzir ações.

No DOM, os objetos estão organizados numa estrutura hierárquica com pais e filhos que reflete a hierarquia de elementos do documento HTML (ver Figura 10.1).

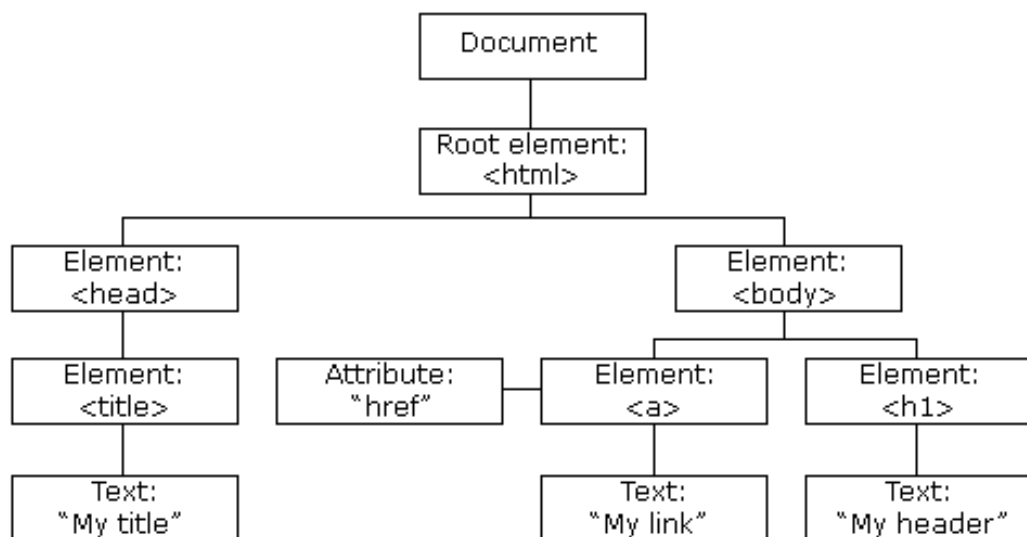


Figura 10.1: Estrutura hierárquica do DOM



Embora existam vários métodos para o fazer, esta secção irá focar-se na utilização do atributo **id** das marcas HTML. Considere o seguinte pedaço de HTML:

```
<body>
  <input id="op1" value="2"/>
  <input id="op2" value="3"/>
  <input id="res" value=""/>

  <script type="text/javascript" src="dom.js"></script>
</body>
```

Repare que a marca **<script>** é incluída depois de todos os outros elementos. Isto é necessário para que os elementos HTML já existam na DOM quando o código JS é executado. O conteúdo do ficheiro **dom.js** será o seguinte:

```
var x = document.getElementById( "op1" );
var y = document.getElementById( "op2" );
console.log( parseFloat(x.value) );
console.log( parseFloat(y.value) );
```

Note a utilização de dois métodos novos:

**document.getElementById:** Procura por um elemento descendente do objeto (**document**) que tenha o atributo **id** especificado no parâmetro (ex, **"op1"**).

**parseFloat:** Converte uma *String* (ex, **x.value**) num valor real (*float*).

Note ainda que se acede à propriedade **value** de cada um dos objetos devolvidos. No caso de **x**, o valor será 2, enquanto o que no caso de **y** o valor será 3. Esta propriedade é de escrita e leitura, o que significa que se pode facilmente alterar o texto apresentado num dado campo **<input>** apenas modificando a propriedade **value**.

### Exercício 10.11

Implemente o exemplo anterior, completando-o de forma a escrever no elemento **<input id="res"...>** o resultado da adição dos 2 valores.

Caso se procure um elemento inexistente, o valor devolvido pelo método **getElementById** será **null**, o que pode ser verificado usando uma condição:

---

```
var x = document.getElementById("nao-existe");
if(x == null)
    alert("Elemento não encontrado");
else
    alert(x.value);
```

---

### 10.4.1 Eventos

Até agora o código JS tem sido executado de forma automática quando a página é carregada. Na realidade só o código que se encontre fora de funções é que é automaticamente executado. Este pode depois invocar as diversas funções disponíveis. Ora, por vezes este não é o comportamento desejado, podendo o programador querer que nenhum código seja executado automaticamente, mas apenas após certos eventos. Repare que no caso anterior foi necessário mudar a inclusão do código JS para o final da página. A abordagem mais correta seria a de programar um evento, indicando que o código deve ser chamado após a página ser carregada completamente.

O exemplo seguinte melhora o código anterior, através da utilização do evento **window.onload**. Repare que é necessário colocar o código dentro de uma função. Neste caso a função tem o nome **calcular**.

---

```
function calcular(){
    var x = document.getElementById( "op1" );
    var y = document.getElementById( "op2" );
    console.log( parseFloat(x.value) );
    console.log( parseFloat(y.value) );
}

window.onload = calcular;
```

---

#### Exercício 10.12

Melhore o exercício 11 de forma a que o código da calculadora apenas seja executado após a construção completa da página.

Os eventos também se podem referir a ações do utilizador, nomeadamente mover o apontador, pressionar teclas ou simplesmente a modificação de algum elemento HTML. No caso de uma calculadora será útil incluir um botão que calcule o valor ou outro que indique a operação. Isto realiza-se através da inclusão de propriedades diretamente nas marcas HTML.<sup>1</sup>

---

<sup>1</sup>Para uma lista dos eventos, consulte [http://www.w3schools.com/tags/ref\\_eventattributes.asp](http://www.w3schools.com/tags/ref_eventattributes.asp).

Considere o seguinte excerto de HTML:

```
<html>
  <head>
    <script type="text/javascript" src="calculadora.js"></script>
  </head>
  <body>
    <input id="op1" value="2" />
    <span id="op-view">+</span>
    <input id="op2" value="3" />
    <input id="res" value="" /><br/>

    <button onclick="calcular()">Calcular</button>
  </body>
</html>
```

Repare como a marca **button** possui um atributo **onclick** que está definido para "**calcular()**". Isto significa que quando o utilizador clicar com o apontador em cima do botão, essa função será executada.

### Exercício 10.13

Complete o excerto anterior implementando a função **calcular()**. Verifique o funcionamento da página quando pressiona o botão.

Podemos generalizar este exemplo de forma a que se possa especificar a operação a executar através de campos de seleção:

```
<html>
  <head>
    <script type="text/javascript" src="calculadora.js"></script>
  </head>
  <body>
    <input id="op1" value="2" />
    <span id="op-view">+</span>
    <input id="op2" value="3" />
    <input id="res" value="" /><br/>

    <select onchange="operacao()">
      <option value="+"> Soma </option>
      <option value="-"> Subtração </option>
    </select>

    <button onclick="calcular()">Calcular</button>
```

```
</body>
</html>
```

Neste caso, a marca **<select>** invocará a função **operacao()**.<sup>2</sup> A função simplesmente irá definir uma variável global com a operação a realizar:

```
var op = "+"; //Deverá estar no topo do ficheiro.

function operacao() {
  var elemento = event.target;
  var elementoSeleccionado = elemento.options[elemento.selectedIndex];
  op = elementoSeleccionado.value;
  console.log("Operação: "+op);
}
```

A utilização de **event.target** é útil pois permite indicar à função qual o elemento que invocou o evento. Neste caso permite aceder imediatamente ao elemento onde se clicou, para saber qual a operação a realizar, evitando usar **document.getElementById()**. No entanto, este elemento não está disponível em todos os navegadores!

#### Exercício 10.14

Integre o código anterior numa página funcional. E verifique o funcionamento da mesma. Terá de alterar a função **calcular()** de forma a aplicar uma operação diferente, de acordo com o valor da variável **op**.

#### Exercício 10.15

Adicione suporte para mais operações, tais como a multiplicação, divisão, ou resto da divisão inteira.

#### Exercício 10.16

Altere a propriedade **innerHTML** do elemento **id="op-view"** de forma a que a página apresente sempre a operação correta.

---

<sup>2</sup>Ver [http://www.w3schools.com/jsref/dom\\_obj\\_select.asp](http://www.w3schools.com/jsref/dom_obj_select.asp).

Como pode ver em [http://www.w3schools.com/tags/ref\\_eventattributes.asp](http://www.w3schools.com/tags/ref_eventattributes.asp), os eventos existentes são inúmeros, podendo inclusive reagir à posição do rato.

Considere que modifica o código anterior de forma a adicionar o evento **onmouseover** e a propriedade **id** à marca **<button>**:

```
...  
  <button id="btn" onclick="calcular()" onmouseover="mover('btn')">Calcular</button>  
...
```

Pode-se agora implementar a função **mover()**, que será ativada sempre que o apontador se encontre em cima do botão. Por exemplo, a função seguinte move um elemento especificado por parâmetro para uma posição aleatória dentro dos limites da janela:

```
function mover(elemento){  
  var e = document.getElementById(elemento);  
  
  e.style.position = "absolute";  
  e.style.top = (Math.random() * window.innerHeight)+"px";  
  e.style.left = (Math.random() * window.innerWidth)+"px";  
}
```

### Exercício 10.17

Componha o exemplo anterior e verifique o que acontece quando o apontador passa por cima do botão.

### Exercício 10.18

Generalize o último exemplo de forma a que todos os elementos se tornem móveis.

## 10.5 Temporizadores

A linguagem JS tem a possibilidade de atrasar a execução das funções. Isto é útil para implementar animações e controlar a sua duração. Por exemplo, considere o seguinte código que faz um elemento diminuir de altura até desaparecer:

```
function diminuirVertical(elemento) {  
  var altura = parseInt(elemento.style.height,10);
```

```
for( ;altura> 0; altura--){  
    elemento.style.height = altura+"px";  
}  
}
```

---

Neste exemplo não existe maneira de controlar o tempo de execução, ou seja, o tempo que o elemento demora a desaparecer. Na realidade o efeito irá executar rapidamente, não sendo sequer visível qualquer animação.

### Exercício 10.19

Implemente o exercício anterior e verifique qual o resultado quando aplicado ao evento **onclick** de uma imagem.

A alternativa que a linguagem JS fornece é a utilização de temporizadores com uma resolução de 1 milissegundo. É assim possível ativar funções de forma periódica, sendo igualmente possível controlar o intervalo entre execuções. No caso de animações é possível controlar a duração da animação. As funções relevantes são:

**setInterval("função", intervalo):** Define que a função indicada no parâmetro deve ser invocada a cada intervalo de tempo. O intervalo de tempo é expresso em milissegundos. A função devolve um objeto para que seja possível cancelar o temporizador;

**clearInterval(variável):** Apaga o temporizador passado no argumento;

**setTimeout("função", atraso):** Define que a função indicada deve executar depois do atraso especificado, em milissegundos. Neste caso a função é executada apenas uma vez.

No exemplo seguinte, altera-se a altura de um elemento (tal como uma imagem), por 10px de cada vez e o processo é executado a cada 10ms. Neste caso a função além de reduzir a imagem, deteta através da variável global **temp** que é a primeira execução e programa o temporizador. No final cancela-o.

---

```
var temp = null;  
  
function diminuirVertical(elemento){  
    if(temp == null){  
        var elemento = event.target;  
        temp = setInterval("diminuirVertical("+elemento.id+")",10);  
    }  
}
```

```

}

var altura = parseInt(elemento.style.height) - 10 ;
elemento.style.height = altura+"px";

if(altura == 0){
    window.clearInterval(temporizador);
    temp = null;
}
}

```

---

Através da função **setTimeout** também é possível executar o mesmo processo, sendo que neste caso o programa fica mais compacto. O princípio de funcionamento é ligeiramente diferente. Neste caso, a cada execução, se a altura for superior a 0, a função programa uma nova execução de si própria para um tempo futuro.

```

function diminuirVertical(elemento){
    if(elemento == null) //Primeira execução
        elemento = event.target;

    var altura = parseInt(elemento.style.height) - 10 ;
    elemento.style.height = altura+"px";

    if(altura > 0){
        setTimeout("diminuirVertical('+elemento.id+')",10);
    }
}

```

---

### Exercício 10.20

Implemente diversas animações utilizando estes métodos. Pode recorrer aos seguintes atributos de estilo:

- **opacity**: Valor real entre 0 e 1 que traduz a opacidade do elemento;
- **width**: Valor inteiro (em px) que representa a largura do elemento;
- **top**: Valor inteiro (em px) que representa a deslocação a partir do topo;
- **left**: Valor inteiro (em px) que representa a deslocação a partir da esquerda;
- **display**: Define a visibilidade do elemento. Ver [http://www.w3schools.com/cssref/pr\\_class\\_display.asp](http://www.w3schools.com/cssref/pr_class_display.asp).

## 10.6 Para aprofundar

### Exercício 10.21

Controlando o atributo **display** do estilo de um elemento `<div>` implemente um *popup* ativado por um clique num botão. Este popup deverá ter outro botão que o faz desaparecer.

### Exercício 10.22

Implemente um relógio usando JS. Pode obter a data atual utilizando o seguinte conjunto de instruções:

```
var hoje = new Date();  
var horas = hoje.getHours();  
var minutos = hoje.getMinutes();  
var segundos = hoje.getSeconds();
```

### Exercício 10.23

Verifique a página <http://getbootstrap.com/javascript> que descreve as funcionalidades de animações, através de JS que o *Twitter Bootstrap* fornece. Experimente criar *popups* (classe **modal**) e outros elementos dinâmicos.

### Exercício 10.24

Construa uma pequena página com um botão que utilize *Twitter Bootstrap*. Adicione um evento ao botão de maneira que, quando o apontador clicar nele uma primeira vez, ele adquira a classe **active**. E quando clicar novamente, seja retirada a classe.

## Glossário

|             |                           |
|-------------|---------------------------|
| <b>CSS</b>  | Cascading Style Sheets    |
| <b>DOM</b>  | Document Object Model     |
| <b>HTML</b> | HyperText Markup Language |



## Referências

- [1] ECMA International, *Standard ECMA-262 – ECMAScript Language Specification*, Padrão, dez. de 1999. URL: <http://www.ecma-international.org/publications/standards/Ecma-262.htm>.
- [2] W3C. (1999). HTML 4.01 Specification, URL: <http://www.w3.org/TR/1999/REC-html401-19991224/>.
- [3] —, (2001). Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, URL: <http://www.w3.org/TR/2011/REC-CSS2-20110607/>.
- [4] W3Schools, *JavaScript and HTML DOM Reference*, <http://www.w3schools.com/cssref/>, [Online; acedido em 8 de Novembro de 2018], 2013.