

2

Sintaxe e estrutura

NESTE CAPÍTULO, APRESENTAREMOS A SINTAXE E ESTRUTURA DA LINGUAGEM JAVASCRIPT. O assunto apresentado aqui se aplica ao núcleo da linguagem JavaScript que independe de onde é usada: no browser, no servidor ou como linguagem independente. O núcleo da linguagem é especificado no padrão ECMA-262 [5].

Como a maior parte das linguagens de programação, o código JavaScript é expresso em formato texto. O texto do código pode representar *instruções*, grupos de instruções, organizadas em *blocos* e *comentários*. Dentro das instruções, pode-se manipular *valores* de diversos *tipos*, armazená-los em *variáveis* e realizar diversas de *operações* com eles.

Uma instrução JavaScript consiste de uma série de símbolos, organizados de forma significativa de acordo com as regras da linguagem, que ocupam uma única linha ou terminam em ponto-e-vírgula. Os caracteres de “retorno de carro” (<CR>) e “nova-linha” (<LF>) são considerados terminadores de linha em JavaScript. O interpretador automaticamente acrescenta um ponto-e-vírgula quando os encontra, terminando a instrução. Utilizar ponto-e-vírgula para terminar cada instrução, portanto, é opcional em JavaScript, já que o interpretador faz isto automaticamente, porém, trata-se de uma boa prática de programação.

Com exceção dos caracteres que provocam novas linhas, nenhum outro tipo de caractere que representa espaço em branco (espaço horizontal, nova-página ou tabulações) interferem no código. Onde se usa um espaço pode-se usar 200 espaços. O espaço em branco pode e deve ser utilizado para endentar blocos de código e torná-lo mais legível. Por exemplo, os dois trechos de código abaixo são equivalentes mas o primeiro é bem mais legível:

<pre> x = 5; function xis() { var x = 0; while (x < 10) { x = x + 1; } } xis(); document.write("x é " + x); </pre>	<pre> x=5;function xis() {var x=0;while(x<10) {x=x+1}} xis() document.write("x é " + x) </pre>
---	---

Instruções compostas (seqüências de instruções que devem ser tratadas como um grupo) são agrupadas em *blocos* delimitados por chaves ({ e }), como mostrado acima. Blocos são usados em JavaScript na definição de funções e estruturas de controle de fluxo. Blocos são tratados como uma única instrução e podem ser definidos dentro de outros blocos. No exemplo acima, o bloco da função `xis()` contém duas instruções. A segunda é um bloco (`while`) que contém uma instrução. As outras instruções não pertencem a bloco algum.

As chaves que definem um bloco são caracteres separadores. Podem ser colocadas em qualquer lugar após a declaração da estrutura que representam. Não precisam estar na mesma linha. Pode haver qualquer número de caracteres terminadores de linha antes ou depois:

```
function media(a, b)
{
    return (a + b)/2;
}
```

Os formatos maiúsculo e minúsculo de um caractere são considerados caracteres distintos em JavaScript. Por exemplo `function`, `Function` e `FUNCTION` são três nomes distintos e tratados diferentemente em JavaScript.

Há duas formas de incluir comentários em JavaScript. Qualquer texto que aparece depois de duas barras (//) é ignorado pelo interpretador até o final da linha. Quando o interpretador encontra os caracteres /*, ignora tudo o que aparecer pela frente, inclusive caracteres de nova-linha, até encontrar a seqüência */.

```
/* Esta função retorna a
 * média dos argumentos passados
 */
function media(a, b)
{
    return (a + b)/2;    // a e b devem ser números
}
```

Os comentários HTML (<!-- e -->) não podem ser usados dentro de código JavaScript. Se aparecerem, devem estar dentro de comentários JavaScript.

Variáveis

Variáveis são usadas para armazenar valores temporários na maior parte das instruções em JavaScript. Para *definir* uma variável, basta escolher um nome que não seja uma palavra reservada e lhe atribuir um valor:

```
preco = 12.6;
produto = "Livro";
```

Uma variável também pode ser declarada sem que receba um valor. Para isto é necessário usar a palavra-chave `var`:

```
var preco;
```

A variável `preco` acima possui o valor `undefined`. Este valor é usado sempre que uma variável não possui um valor definido.

O *escopo* ou alcance de uma variável depende do contexto onde é definida ou declarada. Uma variável declarada ou definida pela primeira vez dentro de um bloco tem escopo *local* ao bloco e não existe fora dele. Variáveis declaradas ou definidas fora de qualquer bloco são *globais* e são visíveis em todo o programa ou página HTML:

```
<script>
    global = 3; // escopo: toda a pagina
    function func() {
        local = 5;    // escopo: somente o bloco atual
        global = 10;
    }
    // local nao existe aqui.
    // global tem valor 10! (pode ser lida em qualquer lugar da pagina)
</script>
```

O uso de `var` é opcional na definição de variáveis globais. Variáveis locais devem ser definidas com `var` para garantir que são locais mesmo havendo uma variável global com o mesmo nome, por exemplo:

```
g = 3; // variável global
function func() {
    var g = 10;    // esta variável g é local!
}
// g (global) tem o valor 3!
```

Quando usadas dentro blocos `<SCRIPT>` de páginas HTML, variáveis globais são tratadas como *propriedades* da janela que contém a página e podem ser utilizadas a partir de outras janelas ou frames.

Tipos de dados e literais

JavaScript possui seis *tipos de dados fundamentais*. Cinco são considerados tipos primitivos e representam valores. Eles são *string*, *number*, *boolean*, *undefined* e *null*. O primeiro representa caracteres e cadeiras de caracteres. O tipo de dados *number* representa números. Os literais booleanos `true` e `false` são os únicos representantes do tipo de dados *boolean*. Os tipos *undefined* e *null* possuem apenas um valor cada: `undefined` e `null`, respectivamente. O primeiro é o valor de variáveis não definidas. O segundo representa um valor definido nulo.

O sexto tipo de dados é *object*, que representa uma *coleção de propriedades*. Os tipos primitivos definidos e não-nulos são também representados em JavaScript como *objetos*. Cada propriedade de um objeto pode assumir qualquer um dos cinco valores primitivos. Pode também conter um outro objeto que tenha suas próprias propriedades ou pode conter um objeto do tipo *function*, que define um método – função especial que atua sobre o objeto do qual é membro. A figura abaixo ilustra os tipos de dados e objetos nativos do JavaScript:

Tipos e objetos nativos ECMAScript

function

Tipo de objeto que representa funções, métodos e construtores

object

Tipo de dados nativo que representa coleções de propriedades contendo valores de tipos primitivos, function ou object

Objetos nativos embutidos

Object

Array

Global

Boolean

Number

String

Function

Date

Math

Tipos de dados primitivos (que representam valores)

undefined

representa valores ainda não definidos

undefined

null

representa o valor nulo

null

boolean

representa valores booleanos

true
false

number

representa números de ponto-flutuante IEEE 754 com precisão de 15 casas decimais (64 bits)

Min: $\pm 4.94065 \text{ e-}324$

Max: $\pm 1.79769 \text{ e+}308$

NaN

Infinity

-Infinity

string

representa cadeias ordenadas (e indexáveis) de caracteres Unicode.

"\u0000 - \uFFFF"

'\u0000 - \uFFFF'

' '

" "

"abcde012+\$_@..."

A linguagem não é rigorosa em relação a tipos de dados e portanto não é preciso *declarar* os *tipos* das variáveis antes de usá-las, como ocorre em outras linguagens. Uma mesma variável que é usada para armazenar um *string* pode receber um número de ponto-flutuante e vice-versa. O tipo de dados é alocado no momento da inicialização, ou seja, se na definição de uma variável ela receber uma *string*, JavaScript a tratará como *string* até que ela receba um novo tipo através de outra atribuição.

```
s = "texto";           // s é string
y = 123;               // y é number
s = true;              // s agora é boolean
```

Os tipos primitivos *number*, *string* e *boolean* são representados pelos objetos nativos *Number*, *String* e *Boolean*. Cada objeto contém uma propriedade com o valor do tipo correspondente. A conversão de tipos primitivos em objetos é automática e totalmente transparente ao programador. Raramente é necessário fazer uma distinção, por exemplo, entre um *string* – tipo de dados primitivo e um *String* – tipo de objeto que contém um *string*.

JavaScript suporta números inteiros e de ponto flutuante mas todos os números em são representados internamente como ponto-flutuante de dupla-precisão. Podem ser usados através de representações decimais, hexadecimais ou octais. Números iniciados em “0” são considerados *octais* e os iniciados em “0x” são *hexadecimais*. Todos os outros são considerados

decimais, que é o formato *default*. Números de ponto-flutuante só podem ser representados na base decimal e obedecem ao padrão IEEE 754. A conversão entre representação de tipos numéricos é automática. Veja alguns exemplos de números e os caracteres utilizados para representá-los:

```
h = 0xffac;           // hexadecimal: 0123456789abcdef
flut = 1.78e-45;      // decimal ponto-flutuante: .0123456789e-
oct = 0677;           // octal: 01234567
soma = h + 12.3 + oct - 10;    // conversão automática
```

O tipo *number* também representa alguns valores especiais, que são infinito positivo (Infinity), infinito negativo (-Infinity) e indeterminação (NaN - Not a Number).

Booleans representam os estados de ligado e desligado através dos literais `true` e `false`. São obtidos geralmente como resultados de expressões condicionais.

Strings são identificados por literais contidos entre aspas duplas ("...") ou simples ('...'). O texto entre aspas pode ser qualquer caractere Unicode. Tanto faz usar aspas simples como aspas duplas. Frequentemente usa-se aspas simples quando um trecho de código JavaScript que requer aspas é embutido em um atributo HTML, que já utiliza aspas duplas:

```
<INPUT TYPE="button" ONCLICK="alert('Oh não!')" VALUE="Não aperte!">
```

A *concatenação* de strings é realizada através do operador "+". O operador "+=" (atribuição composta com concatenação) acrescenta texto a um string existente. Qualquer número ou valor booleano que fizer parte de uma operação de concatenação será automaticamente transformado em string.

```
str1 = "Eu sou uma string";
str2 = str1 + ' também!';
str3 = str2 + 1 + 2 + 3;    // str3 contém Eu sou uma string também!123
str1 += "!";               // mesmo que str1 = str1 + "!".
```

Qualquer valor entre aspas é uma *string* mesmo que represente um número. Qualquer valor lido a partir de um campo de formulário em uma página HTML ou janela de entrada de dados também é *string*. Para converter um número ou valor booleano em string basta utilizá-lo em uma operação de concatenação com uma string vazia:

```
a = 10;
b = "" + a; // b contém a string "10"
```

A conversão de strings em números não é tão simples. É preciso identificar a representação utilizada, se é ponto-flutuante, hexadecimal, etc. Para isto, JavaScript fornece duas *funções nativas*: `parseInt(string)` e `parseFloat(string)` que convertem strings em representações de número inteiro e ponto-flutuante respectivamente.

```
a = "10"; b = prompt("Digite um número"); // lê string
document.write(a + b); // imprime "105"
```

```
c = parseInt(a) + parseInt(b); // converte strings em inteiros decimais
document.write(c); // imprime "15"
```

Caracteres especiais

Se for necessário imprimir aspas dentro de aspas é preciso usar um caractere de escape. O caractere usado para este fim dentro de strings é a contra-barra (\). Use \' para produzir uma aspa simples e \" para produzir aspas duplas. A própria contra-barra pode ser impressa usando "\\". Outros caracteres são usados para finalidades especiais em JavaScript e não podem ser impressos da forma convencional. A contra-barra também é usada nesses casos. A tabela a seguir mostra um resumo desses caracteres especiais em JavaScript.

Caractere especial	Função
\"	Aspas duplas (")
\'	Aspas simples(')
\\	Contra-barra (\)
\n	Nova linha (<i>line feed</i>)
\r	Retorno de carro (<i>carriage return</i>)
\f	Avança página (<i>form feed</i>)
\t	Tabulação horizontal (<i>horizontal tab</i>)
\b	Retrocesso (<i>backspace</i>)

Usando JavaScript em HTML é importante lembrar que HTML ignora completamente espaços em branco extras, novas-linhas, etc. que não sejam provocadas por descritores HTML (como
, <P>, etc.). Portanto os escapes acima que provocam espaços em branco não aparecerão na página a menos que o texto esteja dentro de um bloco <PRE>.

Identificadores e palavras reservadas

Identificadores JavaScript são os nomes que o programador pode escolher para variáveis e funções definidas por ele. Esses nomes podem ter qualquer tamanho e só podem conter caracteres que sejam:

- números (0-9)
- letras (A-Z e a-z)
- caractere de sublinhado (_)

Além disso, embora identificadores JavaScript possam conter números, não podem *começar* com número. Por exemplo, os identificadores abaixo são ilegais:

```
ping-pong, 5abc, Me&You
```

Mas os identificadores

`inicio, indice, abc5, _Me, ping_pong`

são legais e podem ser usados como nome de funções e variáveis.

As palavras listadas abaixo são utilizadas pela linguagem JavaScript e por isto são consideradas *reservadas*, não sendo permitido usá-las para definir identificadores para métodos, variáveis ou funções:

- **break** – sai de um loop sem completá-lo
- **continue** – pula uma iteração de um loop
- **delete** – operador que apaga um objeto (não existe em JavaScript 1.1)
- **false** – literal booleano
- **for** – estrutura de repetição *for*
- **function** – declara uma função JavaScript
- **if, else** – estrutura de controle condicional *if-else*
- **in** – usado dentro de um loop *for* para iterar pelas propriedades de um objeto
- **new** – cria uma nova cópia de um objeto a partir de um protótipo
- **null** – literal do tipo *null*
- **return** – retorna de uma função (pode retornar um valor)
- **this** – ponteiro para o objeto atual (ou elemento HTML atual)
- **true** – literal booleano
- **typeof** – operador que identifica o tipo de uma variável
- **undefined** – literal do tipo *undefined*
- **var** – declara uma variável
- **void** – operador que executa funções sem retornar valor
- **while** – estrutura de repetição *while*
- **with** – estabelece o objeto como default dentro de um bloco

Como o formato de caixa-alta e caixa-baixa em JavaScript é significativo, palavras como `This`, `Null`, `TRUE` são identificadores legais (embora não sejam aconselháveis já que podem confundir). Nomes de objetos nativos, como *String*, *Number*, *Date* e propriedades globais do *client-side* JavaScript como `window`, `document`, `location`, `parent`, etc. não são consideradas palavras reservadas em JavaScript, mas seu uso deve ser evitado, uma vez que podem, além de confundir, provocar erros em programas que fazem uso desses objetos.

Várias outras palavras também são reservadas em JavaScript, embora não tenham significado algum na linguagem (ECMA-262). São reservadas para uso futuro:

<code>abstract</code>	<code>do</code>	<code>import</code>	<code>short</code>
<code>boolean</code>	<code>double</code>	<code>instanceof</code>	<code>static</code>
<code>byte</code>	<code>enum</code>	<code>int</code>	<code>super</code>
<code>case</code>	<code>export</code>	<code>interface</code>	<code>switch</code>
<code>catch</code>	<code>extends</code>	<code>long</code>	<code>synchronized</code>
<code>char</code>	<code>final</code>	<code>native</code>	<code>throw</code>
<code>class</code>	<code>finally</code>	<code>package</code>	<code>throws</code>
<code>const</code>	<code>float</code>	<code>private</code>	<code>transient</code>
<code>debugger</code>	<code>goto</code>	<code>protected</code>	<code>try</code>

default

implements

public

volatile

Operadores e expressões

JavaScript possui várias classes de operadores. Operações de atribuição, aritméticas, booleanas, comparativas e binárias em JavaScript são realizadas da mesma forma que em outras linguagens estruturadas como C++ ou em Java. As estruturas de controle de fluxo também são as mesmas. Algumas outras operações são mais específicas à linguagem, como concatenação, criação e eliminação de objetos. A tabela abaixo relaciona todos os operadores de JavaScript:

Operadores aritméticos		Operadores lógicos		Operadores de bits	
- n	negação	!=	diferente de	&	and
++n, n++	incremento	==	igual a		or
--n, n--	decremento	>	maior que	^	xor
*	multiplicação	<	menor que	~	not
/	divisão	>=	maior ou igual a	<<	desloc. à esquerda
%	resto	<=	menor ou igual a	>>	desloc. à direita
+	adição e conc.		or	>>>	desloc. à dir. s/ sinal
-	subtração	&&	and	Operadores de objetos	
Operadores de atribuição		!	not	new	criação
=	atribuição	?:	condicional	delete	remoção
op=	atribuição com operação op	,	vírgula	typeof	tipo do objeto
				void	descarta o tipo

A *atribuição* é usada para armazenar valores em variáveis. Ocorre da esquerda para a direita, ou seja, quaisquer operações do lado direito, mesmo de atribuição, são realizadas antes que a operação de atribuição à esquerda seja efetuada. O operador “=” representa a operação de atribuição. É possível também realizar a atribuição ao mesmo tempo em que se realiza uma outra operação aritmética ou binária usando os operadores compostos.

```
x = 1;    // atribuição simples
y += 1;   // atribuicao com soma. Equivale a      y = y + 1 ou y++
z /= 5;   // atribuicao com divisao. Equivale a    z = z / 5
```

O operador “+” tanto é usado para adição de números como para a concatenação de strings. Quando ambas as operações ocorrem em uma mesma instrução, a precedência é a mesma mas a associatividade (esquerda para a direita) beneficia a concatenação. Se ocorrer pelo menos uma concatenação à esquerda, todas as operações seguintes à direita serão concatenações mesmo que envolvam números:

```
texto = 4 + 5 + ":" + 4 + 5;    // texto contém 9:45
```


No exemplo acima, só há uma adição, que está em negrito. A segunda operação “+” é concatenação porque ocorre com uma string. Pelo mesmo motivo, todas as operações seguintes são concatenações. É preciso usar parênteses para mudar a precedência.

As operações em JavaScript obedecem a determinadas regras de precedência. Multiplicações e divisões, por exemplo, sempre são realizadas antes de somas e subtrações, a não ser que existam parênteses (que possuem a maior precedência) em volta da expressão. A tabela abaixo relaciona os operadores JavaScript e sua precedência:

Associatividades	Tipo de Operador	Operador
Direita à Esquerda	separadores	[] . ()
Esquerda à Direita	operadores unários e de objetos	expr++ expr-- ++expr --expr +expr -expr ~ ! new delete void typeof
E a D	multiplicação/divisão	* / %
E a D	adição/sub./concat.	+ - +
E a D	deslocamento	<< >> >>>
E a D	relacional	< > >= <=
E a D	igualdade	== !=
E a D	AND	&
E a D	XOR	^
E a D	OR	
E a D	E lógico	&&
E a D	OU lógico	
D a E	condicional	? :
D a E	atribuição	= += -= *= /= %= >>= <<= >>>= &= ^=

Os parênteses sempre podem ser usados para sobrepor a precedência original. Eles são necessários em diversas ocasiões como, por exemplo, para evitar a concatenação em expressões que misturam strings com números:

```
texto = (4 + 5) + ":" + (4 + 5);      // texto contém 45:45
```

Os “++” e “--” (incremento e decremento) acrescentam ou subtraem 1 da variável antes ou depois do uso, respectivamente. Se o operador ocorrer à esquerda, o incremento ou decremento ocorre *antes* do uso. Se o operador ocorrer à direita, o incremento ou decremento ocorre *após* o uso da variável.

```
x = 5;
z = ++x;           // z = 6, x = 6; Incrementa x primeiro, depois
atribui a z
z = x++;           // z = 5, x = 6; Atribui x a z, depois incrementa
```

Todas as expressões JavaScript possuem um *valor*, que pode ser `undefined`, `null`, número, booleano ou string. Expressões condicionais e comparativas sempre resultam em valor booleano (`true` ou `false`).

O operador “=” é utilizado somente para atribuição. A comparação de igualdade é feita exclusivamente com o operador “==”.

Estruturas de controle de fluxo

As estruturas de controle de fluxo são praticamente as mesmas utilizadas em outras linguagens estruturadas populares. A sintaxe das principais estruturas em JavaScript é idêntica à sintaxe usada em C, C++ e Java.

if... else

A estrutura **if... else** é utilizada para realizar controle de fluxo baseado em expressões condicionais:

```
if (condição) {  
    // instruções caso condição == true  
} else if (condição 2) {  
    // instruções caso condição 2 == true  
} else {  
    // instruções caso ambas as condições sejam false  
}
```

Exemplo:

```
if (ano < 0) {  
    alert("Digite um ano D.C.");  
} else if ( ((ano % 4 == 0) && (ano % 100 != 0)) || (ano % 400 == 0) ) {  
    alert(ano + " é bissexto!");  
} else {  
    alert(ano + " não é bissexto!");  
}
```

A operação do **if...else** pode ser realizada também de uma forma mais compacta (e geralmente menos legível) através do *operador condicional*. A sua sintaxe é

expressão ? instruções se expressão=true : instruções se expressão=false

Exemplo:

```
ano = 1994;  
teste = ((ano % 4 == 0) && (ano % 100 != 0)) || (ano % 400 == 0);  
alert ( teste ? ano + " não é bissexto!" : ano + " é bissexto!" );
```

for

As estruturas **for** e **while** são usadas para repetições baseadas em condições. O bloco **for** contém de três parâmetros opcionais: uma inicialização, uma condição e um incremento. A sintaxe é a seguinte:

```
for(inicialização; condição; incremento) {
    // instruções a serem realizadas enquanto condição for true
}
```

Por exemplo:

```
for (i = 0; i < 10; i = i + 1) {
    document.write("<p>Linha " + i);
}
```

Neste exemplo, a variável *i* é *local* ao bloco **for** (ela não é conhecida fora do bloco. Para que ela seja visível fora do bloco é preciso que ela seja declarada fora dele.

A primeira coisa realizada no bloco **for** é a inicialização. É feita uma vez apenas. A condição é testada cada vez que o loop é reiniciado. O incremento é realizado no final de cada loop. Os elementos do **for** são todos opcionais. A mesma expressão acima poderia ser realizada da maneira abaixo:

```
i = 0;
for (; i < 10;) {
    document.write("<p>Linha " + i);
    i++;
}
```

A única diferença entre esta forma e a anterior é que a variável *i* agora é visível fora do bloco **for** (não é mais uma variável local ao bloco):

Uma instrução do tipo:

```
for (;;) {
    document.write("<p>Linha");
}
```

é interpretada como um loop infinito. Loops infinitos em blocos `<SCRIPT>` de páginas HTML fazem com que a carga de uma página nunca termine. Em alguns browsers, o texto acima nunca será exibido. Em outros, pode fazer o browser travar.

while

O mesmo que foi realizado com **for** pode ser realizado com uma estrutura **while**, da forma:

```
inicialização;
while(condição) {
    // instruções a serem realizadas enquanto condição for true
    incremento;
}
```

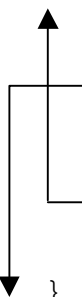
Veja como fica o mesmo exemplo acima usando **while**:

```
i = 0
while (i < 10) {
    document.write("<p>Linha " + i);
    i++;
}
```

break e continue

Para sair a força de loops em cascata existem ainda as instruções **break** e **continue**. **break** sai da estrutura de loops e prossegue com a instrução seguinte. **continue** deixa a execução atual do loop e reinicia com a passagem seguinte.

```
function leiaRevista() {
    while (!terminado) {
        passePagina();
        if (alguemChamou) {
            break; // caia fora deste loop (pare de ler)
        }
        if (paginaDePropaganda) {
            continue; // pule esta iteração (pule a pagina e nao leia)
        }
        leia();
    }
}
...
```



for ... in e with

As estruturas **for...in** e **with** são exclusivas do JavaScript e servem para manipulação de propriedades de objetos. Deixaremos para discuti-las quando apresentarmos o modelo de objetos do JavaScript, no próximo capítulo.

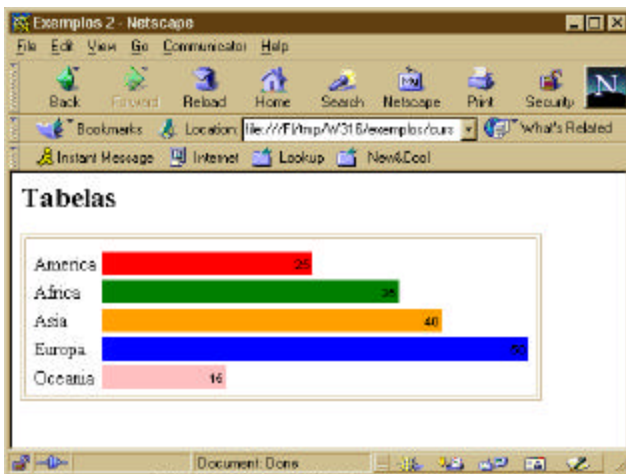
Exercícios

Os exercícios¹ abaixo têm a finalidade de explorar a sintaxe JavaScript apresentada neste capítulo. Devem ser executados em uma página HTML. O assunto visto até aqui, com o auxílio do método `document.write()`, (para imprimir HTML na página) é o suficiente para resolvê-los.

- 2.1 Escreva um programa que imprima uma tabela de conversão entre graus Celsius e graus Fahrenheit ($fahr = (cels * 9/5) - 32$) entre -40 e 100 °C, com incrementos de 10

¹ As soluções de alguns exercícios, deste e de outros capítulos encontram-se nos subdiretórios correspondentes (cap1/ a cap12/).

- em 10. (Use o método `document.write(string)` e para imprimir os resultados dentro de uma tabela HTML) .
- 2.2 Imprima um histograma (gráfico de barras) horizontal para representar uma tabela de cinco valores quaisquer (entre 0 e 100, por exemplo). Use um caractere como “#” ou “*” para desenhar as barras, através de estruturas de repetição (`for` ou `while`).
 - 2.3 Repita o problema anterior usando tabelas HTML. Cada barra deverá ter uma cor diferente (use tabelas `<TABLE>` com células de cores diferentes `<TD BGCOLOR="cor">` e repita os blocos `<TD>` para cada barra). Veja a figura abaixo (à esquerda). Para uma solução usando vetores (que serão apresentados no próximo capítulo), veja o arquivo `exemplos2.html`.
 - 2.4 Escreva uma aplicação que imprima o desenho abaixo, à direita, no browser (use blocos `for` em cascata e varie o parametro `SIZE` de `` ou use folhas de estilo²).
 - 2.5 Repita o exercício anterior fazendo com que cada letra seja de uma cor diferente (varie o parametro `COLOR` de `` ou use folhas de estilo³).



² `...` muda o tamanho da fonte para 24 pontos.

³ `...` muda a cor do texto para vermelho. Pode-se também usar nomes de cores.