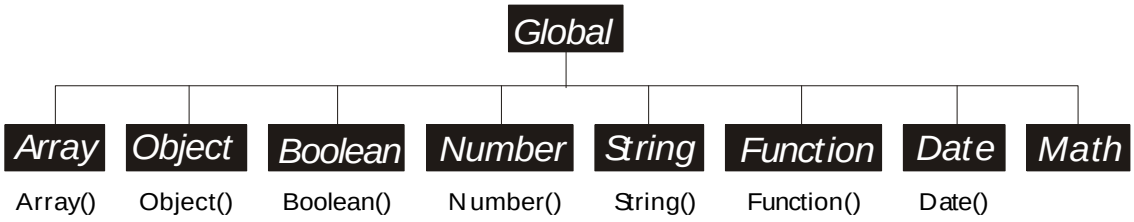
# 4

# *Objetos nativos embutidos*

OS OBJETOS NATIVOS EMBUTIDOS[1] NO JAVASCRIPT fazem parte do núcleo da linguagem. Existem em todas as implementações, até nas tecnologias proprietárias do servidor. Eles não são fornecidos pelo browser ou servidor, e, com exceção dos objetos *Global* e *Math*, é necessário criá-los explicitamente para poder usá-los.

No capítulo anterior conhecemos alguns dos objetos nativos do JavaScript e vimos como criá-los através de seus construtores. Nem todos os objetos nativos têm construtores. A figura abaixo mostra todos os objetos do JavaScript, indicando o construtor *default* de cada um quando houver.




O objeto *Global* representa o contexto global de execução. Não é possível criar um objeto *Global*. Ele é único e já existe antes que haja qualquer contexto de execução. Possui um conjunto de propriedades inicialmente que consistem dos objetos embutidos (*Array*, *Object*, *Boolean*, etc.), funções embutidas (`parseInt()`, `parseFloat()`, construtores, etc.). No client-side JavaScript, o objeto *Global* define a propriedade `window`, cujo valor é o próprio objeto *Global*.

Objetos de todos os tipos nativos embutidos podem ser criados usando o operador new. A exceção é *Math* que não possui construtor portanto não representa um *tipo de objeto* mas é um objeto em si próprio, criado pelo sistema quando o contexto global é inicializado. *Math* funciona apenas como repositório para agrupar funções e constantes matemáticas utilitárias.

---

[1] Esta terminologia é utilizada na especificação ECMA-262 [5].

Como *Math*, outros tipos também servem de repositório de *funções* e constantes úteis ao mesmo tempo em que possuem construtores que permitem a criação de objetos distintos. As funções, diferentemente dos métodos, não se aplicam a um objeto em especial. São *globais*, como as funções `parseInt()`, `eval()`, etc. mas precisam ser chamadas através do identificador do construtor (nome do tipo) do objeto, da forma:

```
Nome_do_tipo.função(parametros);
```

Essas funções e constantes são agrupadas de acordo com o sua finalidade. Exemplos são todas as funções e constantes de *Math*, *Number* e *Date*:

```
a = Math.random()*256;      // função que retorna valor aleatório
b = Number.MAX_VALUE;             // constante com maior número
representável
c = Date.parse(34532343);   // converte milissegundos em uma data
```

Nas seções a seguir, apresentaremos cada um dos objetos nativos embutidos de JavaScript, com suas propriedades, métodos e funções, além de exemplos de uso.

# *Object*

Trata-se de um tipo de objeto genérico usado para representar *qualquer* objeto criado com `new`. Seu construtor raramente é utilizado pelo programador JavaScript. Existe basicamente para dar suporte a operações internas.

Para criar um *Object*, pode-se fazer:

```
obj = new Object();
```

Os métodos de *Object* são três e são "herdados" por todos os objetos JavaScript, mas nem todos os definem. São usados pelo sistema para realizar as conversões entre tipos e operações de atribuição. O programador raramente precisa usá-los:

| **Método** | **Ação** |
|---|---|
| `toString()` | Transforma qualquer objeto em uma representação *string*. É usado automaticamente nas conversões de números em strings, por exemplo, durante a concatenação. |
| `valueOf()` | Converte um objeto em seu valor primitivo, se houver. |
| `assign(`*`valor`*`)` | Implementa o operador de atribuição (=). |

Dos três métodos acima, o mais usado é **`toString()`**. Ele pode ser chamado explicitamente sobre qualquer objeto para transformá-lo em uma representação *string*. É chamado automaticamente quando o objeto é usado em uma operação de concatenação.

Todos os objetos também possuem uma propriedade **`constructor`** que contém uma string com sua função de construção. Por exemplo, suponha um objeto criado com a função *Circulo*, definida no capítulo anterior. O trecho de código:

```
c = new Circulo (13, 11, 5);
document.write("<p>Construtor: <pre>" + c.constructor + "</pre>");
```

imprime na página:

Construtor:
```
function Circulo(x, y, r) {
  this.x = x;
  this.y = y;
  this.r = r;
}
```

# *Number*

É um tipo de objeto usado para representar números (tipo primitivo *number*) como objetos. A criação de um número pode ser feita simplesmente através de uma atribuição. O número será transformado em objeto automaticamente quando for necessário. Um objeto *Number* pode ser criado explicitamente usando `new` e o construtor `Number()`:

```
n = new Number(12);
```

A principal utilidade do tipo *Number* é como repositório de constantes globais referentes à números JavaScript. Estas constantes só estão disponíveis através do nome do construtor (nome do tipo) e não através de objetos específicos do tipo *Number*, por exemplo:

```
maior = Number.MAX_value;   // forma CORRETA de recuperar maior
                            // número representável em JavaScript

n = 5;                    // ou n = new Number(5);
maior = n.MAX_value;      // ERRADO! Forma incorreta.
```

A tabela abaixo lista todas as constantes disponíveis através do tipo *Number*. As propriedades devem ser usadas da forma:

```
Number.propriedade;
```

| Constante | Significado |
|---|---|
| `MAX_value` | Maior valor numérico representável: `4,94065e-324` |
| `MIN_value` | Menor valor numérico representável: `1,79769e+308` |
| `NaN` | Não é um número: `NaN` |
| `NEGATIVE_INFINITY` | Infinito positivo: `+Infinity` |
| `POSITIVE_INFINITY` | Infinito negativo: `-Infinity` |

# *Boolean*

É um tipo de objeto usado para representar os literais `true` e `false` como objetos. Um valor booleano é criado sempre que há uma expressão de teste ou comparação sendo realizada. O valor será transformado automaticamente em objeto quando necessário. Todas as formas abaixo criam objetos *Boolean* ou valores *boolean*:

```
boo = new Boolean("");   // 0, números < 0, null e "": false
boo = new Object(true);
boo = true;
boo = 5 > 4;
```

# *Function*

Um objeto *Function* representa uma operação JavaScript, que pode ser uma função, método ou construtor. Para criar um objeto deste tipo, basta definir uma nova função com a palavra-chave `function`. Também é possível criar funções "anônimas" usando o construtor `Function()` e o operador `new`:

```
func = new Function("corpo_da_função"); // ou, ...
func = new Function(arg1, arg2, ..., argn, "corpo_da_função");
```

Por exemplo, considere a seguinte função:

```
function soma(calc) {
   a=calc.v1.value;
   b=calc.v2.value;
   calc.v3.value=a+b;
}
```

A função acima é um objeto do tipo *Function*. O código abaixo obtem o mesmo resultado, desta vez definindo uma variável que representa o objeto:

```
soma = new Function(calc,
              "a=calc.v1.value; b=calc.v2.value; calc.v3.value=a+b;");
```

O resultado do uso de `Function()` acima é um código mais complicado e difícil de entender que a forma usada anteriormente com `function`. Também é menos eficiente. As funções declaradas com `function` são interpretadas uma vez e compiladas. Quando forem chamadas, já estão na memória. As funções criadas com `Function()` são interpretadas todas as vezes que forem chamadas.

O objeto *Function* tem quatro propriedades que podem ser usadas por qualquer função (tenha sido definida com `function` ou com `new Function()`). Elas estão na tabela abaixo. As propriedades devem ser usadas usando-se o identificador da função (omitindo-se os parênteses e argumentos), da forma:

```
nome_da_função.propriedade;
```

| **Propriedade** | **Significado** *(tipo da propriedade em itálico)* |
|---|---|
| `arguments[]` | *Array*. O vetor de argumentos da função |
| `arguments.length` | *Number*. O comprimento do vetor de argumentos (retorna o número de argumentos que a função tem) |
| `length` | *Number*. Mesma coisa que `arguments.length` |
| `caller` | *Function*. Uma referência para o objeto *Function* que chamou esta função, ou `null` se o objeto que a invocou não é uma função. Só tem sentido quando uma função chama a outra. É uma forma da função atual se referir àquela que a chamou. |
| `prototype` | *Object*. Através desta propriedade, é possível definir novos métodos e propriedades para funções construtoras, que estarão disponíveis nos objetos criados com ela. |

Vimos no capítulo 3 como acrescentar propriedades temporárias a objetos. As propriedades podem ser permanentes se forem definidas dentro do construtor do objeto, mas nem sempre temos acesso ao construtor. Podemos criar novos métodos e propriedades e associá-las a um construtor qualquer usando a sua propriedade `prototype`. Assim a propriedade passa a ser permanente, e estará presente em todos os objetos.

Para acrescentar uma propriedade ao tipo *Date*, por exemplo, podemos fazer:

```
Date.prototype.ano = d.getYear() + 1900;
```

Agora *todos* os objetos criados com o construtor *Date* terão a propriedade ano:

```
d = new Date();
document.write("Estamos no ano de: " + d.ano);
```

Para acrescentar métodos a um tipo, a propriedade definida em prototype deve receber um objeto *Function*. Por exemplo, considere a função abaixo, que calcula se um número passado como argumento é um ano bissexto:

```
function bissexto(umAno) {
   if (((umAno % 4 == 0) && (umAno % 100 != 0)) || (umAno % 400 == 0))
      return true;
   else
      return false;
}
```

Podemos transformá-la em *método*. O primeiro passo é fazê-la operar sobre os dados do próprio objeto. O ano de quatro dígitos, na nossa data é representado pela propriedade `ano` (que definimos há pouco). Obtemos acesso ao objeto atual com `this`:

```
function bissexto() {     // método!
  if(((this.ano % 4 == 0) && (this.ano % 100 != 0)) || (this.ano % 400 ==
0))
     return true;
```

```
  else
     return false;
}
```

O segundo passo, é atribuir a nova função (um objeto *Function* chamado `bissexto`) a uma nova propriedade do protótipo do objeto, que chamamos de `isLeapYear`:

```
Date.prototype.isLeapYear = bissexto;
```

Agora, temos um *método* `isLeapYear()` que retorna `true` se a data no qual for invocado ocorrer em um ano bissexto, e `false`, caso contrário:

```
hoje = new Date();
if (hoje.isLeapYear())
     document.write("O ano " + hoje.ano + " é bissexto");
else
     document.write("O ano " + hoje.ano + " não é bissexto");
```

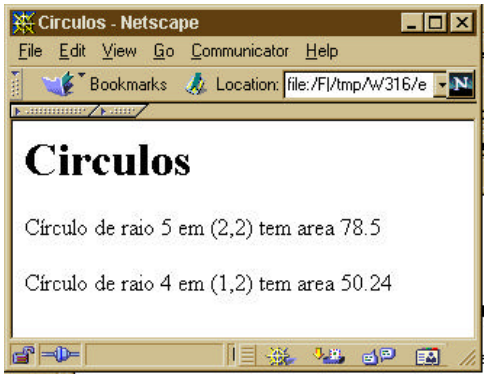
Veja abaixo um exemplo da especificação e construção do objeto *Círculo* (visto no capítulo anterior) com a definição de novos métodos usando a propriedade `prototype` e o construtor `Function()`:

```
<HEAD>
<script>
function Circulo(x, y, r) {      // função "construtora"
 this.x = x; // definição das propriedades deste objeto
 this.y = y;
 this.r = r;
}
// definição de um método toString para o Circulo
Circulo.prototype.toString =
    new Function("return 'Círculo de raio '+this.r+' em ('+this.x+','+this.y+')';");

// criação de um método area para o Circulo
Circulo.prototype.area =
    new Function("return 3.14 * this.r * this.r;");
</script>
</HEAD>
<BODY>
<h1>Circulos</h1>
<script>
c1 = new Circulo(2,2,5); // uso da função construtora
c2 = new Circulo(1,2,4);

// uso de métodos
document.write("<P>" + c1.toString() + " tem area " + c1.area());
document.write("<P>" + c2.toString() + " tem area " + c2.area());
</script>
```

```
</BODY>
```




O resultado da visualização da página acima em um browser é mostrado na figura ao lado.

Todas as funções definidas na página, são propriedades da janela (`window`). Outras janelas ou frames que tenham acesso a esta janela poderão usar o construtor `Circulo()` para criar objetos em outros lugares.

# *String*

O tipo *String* existe para dar suporte e permitir a invocação de métodos sobre cadeias de caracteres, representadas pelo tipo primitivo *string*. Pode-se criar um novo objeto *String* fazendo:

```
s = new String("string");
```

ou simplesmente:

```
s = "string";
```

que é bem mais simples.

Objetos *String* possuem apenas uma propriedade: `length`, que pode ser obtida a partir de qualquer objeto *string* e contém o comprimento da cadeia de caracteres:

```
cinco = "zebra".length;
seis = s.length;
```

O construtor `String()` possui uma propriedade `prototype` que permite a definição de novos métodos e propriedades. A propriedade `prototype` não é uma propriedade de *String*, mas do construtor `String()`, que é *Function* (como são todos os construtores), portanto deve ser usada da forma:

```
String.prototype;          // CERTO
```

e não

```
s = "ornitorrinco"; // ou s = new String("ornitorrinco");
s.prototype;        // ERRADO: Não é propriedade de String!
```

A página ao lado ilustra a utilização da propriedade `prototype` para acrescentar um novo método ao tipo *String* utilizado nos textos de uma página. O método, que chamamos de `endereco()`, serve para gerar o HTML das opções `<OPTION>` de uma caixa de seleção

<SELECT>. A sua utilização economiza digitação e torna a página menor, para transferência mais eficiente na Web.

```
<HTML> <HEAD>
<SCRIPT>
  function Endereco(url) {     // função para definir metodo
     return "<OPTION VALUE='" + url + "'>" + this.toString()+ "</OPTION>";
  }

  String.prototype.endereco = Endereco; // cria método: endereco()
</SCRIPT>
</HEAD>

<BODY> <FORM>
<h1>Exemplos: String</h1>
<p>Opções:
<SELECT ONCHANGE='location.href=this.options[this.selectedIndex].value'>
<SCRIPT>
  wwf = "World Wildlife Fund";      // todos objetos do tipo String
  w3c = "World Wide Web Consortium";

  document.write("Escolha uma organização".endereco(document.location) );
  document.write( wwf.endereco("http://www.wwf.org") );
  document.write( w3c.endereco("http://www.w3c.org") );
</SCRIPT>
</SELECT>
</FORM> </BODY> </HTML>
```

A função `Endereco()` acima poderia ter sido definida anonimamente com `new Function()`, como fizemos na definição dos dois métodos que criamos para o tipo *Circulo*, na seção anterior. Utilizamos a sintaxe baseada na palavra-chave `function` por ser mais clara e eficiente.

Raramente é preciso definir métodos da forma mostrada acima. O tipo *String* já possui uma coleção de métodos úteis, aplicáveis diretamente à qualquer cadeia de caracteres em JavaScript. Podem ser divididos em três tipos:

- os que retornam o string original marcado com descritores HTML,
- os que retornam transformações sobre os caracteres e
- os que permitem realizar operações com caracteres individuais.

Os primeiros estão relacionados nas tabelas abaixo, juntamente com dois métodos que fazem conversões de formato. Supondo que o string usado pelos métodos abaixo é:

```
s = "Texto";
```

a invocação do método (`s.`*`método()`*) na primeira coluna retorna como resultado, o conteúdo da segunda. O string original não é afetado. Todos os métodos retornam *String*.

| **Método Invocado** | **Retorna** |
|---|---|
| `anchor("âncora")` | `<a name="âncora">`**Texto**`</a><>` |
| `link("http://a.com")` | `<a href="http://a.com">`**Texto**`</a>` |
| `small()` | `<small>`**Texto**`</small>` |
| `big()` | `<big>`**Texto**`</big>` |
| `blink()` | `<blink>`**Texto**`</blink>` |
| `strike()` | `<strike>`**Texto**`</strike>` |
| `sub()` | `<sub>`**Texto**`</sub>` |
| `sup()` | `<sup>`**Texto**`</sup>` |
| `italics()` | `<i>`**Texto**`</i>` |
| `bold()` | `<b>`**Texto**`</b>` |
| `fixed()` | `<tt>`**Texto**`</tt>` |
| `fontcolor("cor")` | `<font color="cor">`**Texto**`</font>` (cor pode ser um valor rrggbb hexadecimal ou nome de cor) |
| `fontsize(7)` | `<font size="7">`**Texto**`</font>` (o número representa o tamanho e pode ser um número de 1 a 7) |

Os dois métodos a seguir realizam transformações no formato dos caracteres. São extremamente úteis em comparações e rotinas de validação. Retornam *String*.

| **Método Invocado** | **Retorna** |
|---|---|
| `toLowerCase()` | **`texto`** (converte para caixa-baixa) |
| `toUpperCase()` | **`TEXTO`** (converte para caixa-alta) |

Os métodos seguintes realizam operações baseados nos caracteres individuais de uma *string*. Permitem, por exemplo, localizar caracteres e separar tokens com base em delimitadores. Não afetam os strings originais. As transformações são retornadas:

| **Método Invocado** | **Ação** (tipo de retorno em *itálico*) |
|---|---|
| `charAt(`*`n`*`)` | *String*. Retorna o caractere na posição *n*. A string `s` inicia na posição 0 e termina em `s.length–1`. Se for passado um valor de *n* maior que `s.length-1`, o método retorna uma string vazia. |
| `indexOf("substring")` | *Number*. Retorna um índice *n* referente à posição da primeira ocorrência de `"substring"` na string `s`. |
| `indexOf("substring",` *`inicio`*`)` | *Number*. Retorna um índice *n* referente à posição da primeira ocorrência de `"substring"` em `s` após o índice *`inicio`*. *`inicio`* é um valor entre 0 e `s.length-1` |
| `lastIndexOf("substring")` | *Number*. Retorna um índice n referente à posição da última ocorrência de `"substring"` na string `s`. |
| `lastIndexOf("substring",` *`fim`*`)` | *Number*. Retorna um índice *n* referente à posição da última ocorrência de `"substring"` em `s` antes do índice *`fim`*. *`fim`* é um valor entre 0 e `s.length-1` |
| `split("delimitador")` | *Array*. Converte o string em um vetor de strings |

| **Método Invocado** | **Ação** (tipo de retorno em *itálico*) |
|---|---|
| | separando-os pelo `"delimitador"` especificado. O método `join()` de `Array` faz o oposto. |
| `substring(`*`inicio, fim`*`)` | *String.* Extrai uma substring de uma string `s`.<br>• *`inicio`* é um valor entre 0 e `s.length-1`.<br>• *`fim`* é um valor entre 1 e `s.length`.<br>O caractere na posição *`inicio`* é *incluído* na string e o caractere na posição *`fim`* *não é incluído*. A string resultante contém caracteres de *`inicio`* a *`fim -1`*. |

Há várias aplicações para os métodos acima. O método `split()`, que retorna um objeto do tipo *Array*, é uma forma prática de separar um texto em *tokens*, para posterior manipulação. Por exemplo, considere o string:

```
data = "Sexta-feira, 13 de Agosto de 1999";
```

Fazendo

```
sexta = data.split(",");                // separa pela vírgula
```

obtemos `sexta[0] = "Sexta-feira"` e `sexta[1] = "13 de Agosto de 1999"`. Separamos agora o string `sexta[1]`, desta vez, pelo substring `" de "` :

```
diad = sexta[1].split(" de ");    // separa por <espaço> + de + <espaço>
```

obtendo `diad[0] = 13`, `diad[1] = Agosto`, `diad[2] = 1999`. Podemos agora imprimir a frase "**Válido até 13/Ago/1999**" usando:

```
diad[1] = diad[1].substring(0,3); // diad1[1] agora é "Ago"
document.write("Válido até " + diad[0] + "/" + diad[1] + "/" + diad[2]);
```

## *Exercícios*

4.1 Escreva uma função que faça uma mensagem rolar dentro de um campo `<INPUT TYPE=TEXT>`. Deve ter um *loop*. Use o método `substring()` para extrair um caractere do início de uma *String* e colocá-lo no final, atualizando em seguida o conteúdo (propriedade `value`) do campo de texto. Crie botões para iniciar e interromper o rolamento da mensagem.

# *Array*

O tipo *Array* representa coleções de qualquer tipo, na forma de vetores ordenados e indexados. Para criar um novo vetor em JavaScript, é preciso usar o operador `new` e o construtor `Array()`:

```
direcao = new Array(4);
```

Vetores começam em `0` e terminam em *`length-1`*. `length` é a única propriedade do tipo *Array*. Contém um número com o comprimento do vetor. Os elementos do vetor são acessíveis através de índices passados entre colchetes (`[` e `]`). Para acessar qualquer um dos elementos do vetor `direcao`, por exemplo, usa-se o nome da variável seguida do índice do elemento entre colchetes:

```
x = direcao[2]; // copia o conteúdo do terceiro elemento de direcao em
x
```

Os elementos do vetor são suas propriedades. A construção do vetor acima com 4 elementos cria inicialmente 4 propriedades no objeto e as inicializa com o valor `undefined`. Portanto, no exemplo acima, `x` terá o valor `undefined` pois o vetor foi criado mas não foi preenchido. O vetor pode ser povoado de mais de uma maneiro. Uma das formas é definir seus termos um a um:

```
direcao[0] = "Norte";
direcao[1] = "Sul";
direcao[2] = "Leste";
direcao[3] = "Oeste";
```

Outra forma é povoá-lo durante a criação:

```
direcao = new Array("Norte","Sul","Leste","Oeste");
```

Para recuperar o tamanho do vetor, usa-se a propriedade `length` que também pode ser redefinida com valores maiores ou menores para expandir ou reduzir o vetor:

```
tamanho = direcao.length; // direcao possui 4 elementos
direcao.length--;    // agora só possui 3
direcao.length++;    // agora possui 4 novamente, mas o último é
undefined
```

O vetor acima foi inicializado com quatro elementos, através do seu construtor, mas isto não é necessário. Ele pode ser inicializado com zero elementos e ter novos elementos adicionados a qualquer hora. Existirá sempre uma seqüência ordenada entre os elementos de um vetor. Não é possível ter índices avulsos. Se uma propriedade de índice 6 for definida:

```
direcao[6] = "Centro";
```

o novo vetor `direcao` será atualizado e passará a ter 7 elementos, que terão os valores:

```
("Norte","Sul","Leste","Oeste",undefined,undefined,"Centro")
```

Os campos intermediários foram “preenchidos” com os valores primitivos `undefined`, que representam valores indeterminados.

Os objetos *Array* possuem três métodos listados na tabela a seguir. Os tipos de retorno variam de acordo com o método. Estão indicados em *itálico* na descrição de cada método:

| **Método** | **Ação** |
|---|---|
| `join()` ou `join("separador")` | Retorna *String*. Converte os elementos de um vetor em uma string e os concatena. Se um string for passado como argumento, o utiliza para separar os elementos concatenados. |
| `reverse()` | *Array*. Inverte a ordem dos elementos de um vetor. Tanto o vetor retornado, quanto o vetor no qual o método é chamado são afetados. |
| `sort()` | *Array*. Ordena os elementos do vetor com base no código do caractere. Tanto o vetor retornado, quanto o vetor no qual o método é chamado são afetados. |
| `sort(`*`função_de_ordenação()`*`)` | *Array*. Ordena os elementos do vetor com base em uma função de ordenação. A função deve tomar dois valores a e b e deve retornar:<br>• Menor que zero se a < b<br>• Igual a zero se a = b<br>• Maior que zero se a > b |

O método `join()` tem várias aplicações, principalmente quando usado em conjunto com o método `split()`, de *String*. Uma aplicação é a conversão de valores separados por delimitadores em tabelas HTML:


```
dados = "Norte; Sul; Leste; Oeste"; // String
vetor = dados.split(";");
s = "<tr><td>";
s += vetor.join("</td><td>");
s += "</td></tr>";
document.write("<table border>" + s + "</table>");
```

Qualquer tipo de dados pode ser contido em vetores. Vetores multidimensionais podem ser definidos como vetores de vetores. Veja um exemplo:

```
uf = new Array(new Array("São Paulo", "SP"), new Array("Paraíba", "PB"));
// uf[0] é o Array ("São Paulo", "SP")
// uf[1][1] é o String "PB"
```

Uma invocação de `split()` sobre um string cria um vetor de vários strings. Uma nova invocação de `split()` sobre um desses strings, cria um novo vetor, que pode ser atribuído à mesma variável que lhe forneceu o string, resultando em um vetor bidimensional:

```
produtosStr = "arroz: 12.5; feijão: 14.9; açucar: 9.90; sal: 2.40";
cestaVet = produtosStr.split(";"); // separa produtos; cestaVet[i] é String
for (i = 0; i < cestaVet.length; i++) {
```

```
    cestaVet[i] = cestaVet[i].split(":");        // cestaVet[i] agora é vetor
1D
}                                              // e cestaVet é vetor 2D
prod = cestaVet[2][0];      // prod contém o String "açucar"
qte = cestaVet[2][1];             // qte contém o String "9.90"
```

## *Exercícios*

4.2 Escreva uma página contendo dois campos de texto <TEXTAREA> e um botão, com o rótulo "inverter". O primeiro campo de texto deverá receber uma string de informação digitada pelo usuário. Quando o botão inverter for apertado, todo o conteúdo do primeiro campo deverá ser copiado no outro <TEXTAREA> começando pela última palavra e terminando na primeira. Dica: use o método `reverse()` de *Array*.

# *Math*

O objeto *Math* não é um tipo de objeto. É na verdade uma propriedade global *read-only*. Serve apenas de repositório de constantes e funções matemáticas. Não é possível criar objetos do tipo *Math* (com new) e não há, rigorosamente, *métodos* definidos em *Math* mas apenas *funções*. Para ter acesso a suas funções e constantes, deve-se usar a sintaxe:

```
Math.função();
Math.constante;
```

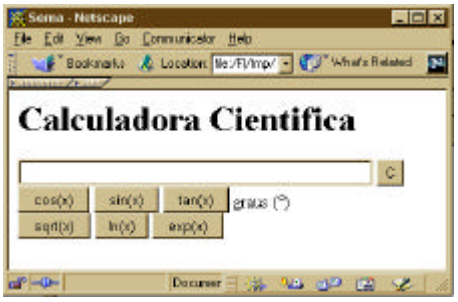
As funções e constantes do tipo *Math* estão listados na tabela a seguir.

| **Funções** | | | | **Constantes** | |
|---|---|---|---|---|---|
| `acos(x)` | cosseno$^{-1}$ | `abs(x)` | absoluto | `E` | $e$ |
| `asin(x)` | seno$^{-1}$ | `max(a, b)` | máximo | `LN10` | ln 10 |
| `atan(x)` | tangente$^{-1}$ | `min(a, b)` | mínimo | `LN2` | ln 2 |
| `atan2(x, y)` | retorna o ângulo θ de um ponto ($x$,$y$) | `pow(x, y)` | $x^y$ | `LOG10E` | $\log_{10} e$ |
| | | `sin(x)` | seno | `LOG2E` | $\log_2 e$ |
| `ceil(x)` | arredonda para cima (3.2 e 3.8 → 4) | `round(x)` | arredonda (3.49 → 3 e 3.5 → 4) | `PI` | π |
| | | | | `SQRT1_2` | 1/sqrt(2) |
| `cos(x)` | cosseno | `tan(x)` | tangente | `SQRT2` | sqrt(2) |
| `exp(x)` | $e^x$ | `sqrt(x)` | raiz quadrada | | |
| `floor(x)` | arredonda para baixo (3.2 e 3.8 → 3) | `log(x)` | logarítmo natural | | |
| `random()` | retorna um número pseudo-aleatório entre 0 e 1. | | | | |

O programa a seguir utiliza algumas funções e constantes do tipo *Math* para implementar uma pequena calculadora científica.

```
<html> <head>
<script language=JavaScript>
<!--
function cos() {
    a = parseInt(document.f1.val1.value) * (Math.PI / 180);
    document.f1.val1.value = Math.cos(a);
}
function sin() {
    a = parseInt(document.f1.val1.value) * (Math.PI / 180);
    document.f1.val1.value = Math.sin(a);
}
function tan() {
    a = parseInt(document.f1.val1.value) * (Math.PI / 180);
    document.f1.val1.value = Math.tan(a);
}
function sqrt() {
    a = document.f1.val1.value;
    document.f1.val1.value = Math.sqrt(parseInt(a));
}
function log() {
    a = document.f1.val1.value;
    document.f1.val1.value = Math.log(parseInt(a));
}
function exp() {
    a = document.f1.val1.value;
    document.f1.val1.value = Math.exp(parseInt(a
}
//--></script>
</head>

<body>
<h1>Calculadora Cientifica</h1>
<form name="f1">
   <input type=text name=val1 size=40>
   <input type=button value=" C " onclick="this.form.val1.value=''"><br>
   <input type=button value=" cos(x) " onclick="cos()">
   <input type=button value=" sin(x) " onclick="sin()">
   <input type=button value=" tan(x) " onclick="tan()"> graus (°)<br>
   <input type=button value=" sqrt(x) " onclick="sqrt()">
   <input type=button value=" ln(x) " onclick="log()">
   <input type=button value=" exp(x) " onclick="exp()">
</form>
</body>  </html>
```



A página HTML a seguir usa o método `random()` para devolver um número aleatorio entre 0 e um limite estabelecido em uma chamada de função. Este número é então usado para carregar imagens (ou outro arquivo) aleatoriamente na página.

```
<HTML> <HEAD>
   <SCRIPT LANGUAGE="JavaScript">
      function loadFile(name, ext, number) {
              return name + Math.floor(Math.random() * lim) + "." + ext;
      }
   </SCRIPT>
</HEAD>

<BODY>
<h1 align=center>Imagens Aleatórias</h1>
<p align=center>
Atenção... eis a imagem da hora!
<!-- imagens: figura-0.gif, figura-2.gif, ..., figura-4.gif -->
<br><script language="JavaScript">
    document.write("<img src=" + loadFile("figura-", "gif", 5) + ">");
</script>
</BODY> </HTML>
```

## *Exercícios*

4.3 Incremente a calculadora desenvolvida no exercício 3.5 para que suporte funções de uma calculadora científica. Use o esqueleto disponível no arquivo `cap4/ex43.html`. Implemente uma tecla de função (`inv`) que permita usar a mesma tecla usada para cossenos, tangentes, etc. no cálculo dos seus inversos (funções `atan()`, `acos()` e `asin()`).

4.4 Crie um jogo onde o usuário deve adivinhar um número entre 0 e 99 em 5 tentativas. A página deverá gerar um número aleatório ao ser carregada (crie uma função e faça com que seja chamada da forma: `<BODY ONLOAD="geraNumero()">`). Depois, forneça uma caixa de textos ou diálogo do tipo `prompt('mensagem')` para que o usuário faça as suas apostas. Exiba uma janela de alerta informando, no final, se o usuário acertou ou não, e em quantas tentativas.

# *Date*

O tipo *Date* é um tipo de objeto usado para representar datas. Para criar data que represente a data e hora *atuais*, chame-o usando `new`, da forma:

```
aquiAgora = new Date();
```

Além da data e hora atuais, *Date* é usado para representar datas arbitrárias. Para representar uma data e hora específica, pode-se usar funções ou um de seus construtores:

```
new Date(ano, mes, dia);
                  // Ex: umDia = new Date(97, 11, 19);

new Date(ano, mes, dia, hora, minuto, segundo);
                  // Ex: outroDia = new Date(98, 10, 11, 23, 59, 59);

new Date(Data em forma de string: "Mes dd, aa hh:mm:ss");
                  // Ex: aqueleDia = new Date("October 25, 97
23:59:15");

new Date(milissegundos desde 0:0:0 do dia 1o. de Janeiro de 1970);
                  // Ex: oDia = new Date(86730923892832);
```

O não é representado em um campo fixo de dois dígitos, mas como (1900 – *ano*). O ano 2005, por exemplo, seria representado como 105. Os meses e dias da semana começam em zero.

Para utilizar as informações de um *Date*, invoca-se os seus métodos sobre o objeto criado. Há métodos para alterar e recuperar informações relativas à data e hora, além de métodos para formatar datas em formatos como UTC, GMT e fuso horário local. Métodos podem ser invocados a partir de um objeto *Date* como no exemplo a seguir:

```
dia = umDia.getDay();
hora = umDia.getHours();
ano = umDia.getYear();
document.writeln("Horário de Greenwich: " + umDia.toGMTString());
```

A tabela a seguir relaciona os métodos dos objetos do tipo *Date*, os tipos de retorno (se houver) e suas ações. Não há propriedades definidas no tipo *Date*.

| **Método** | **Ação** |
|---|---|
| `getDate()` | Retorna *Number*. Recupera o dia do mês (1 a 31) |
| `getDay()` | *Number*. Recupera o dia da semana (0 a 6) |
| `getHours()` | *Number*. Recupera a hora (0 a 23) |
| `getMinutes()` | *Number*. Recupera o minuto (0 a 59) |
| `getMonth()` | *Number*. Recupera o mês (0 a 11) |
| `getSeconds()` | *Number*. Recupera o segundo (0 a 59) |
| `getTime()` | *Number*. Recupera a representação em milissegundos desde 1-1-1970 0:0:0 GMT |
| `getTimezoneOffset()` | *Number*. Recupera a diferença em minutos entre a data no fuso horário local e GMT (não afeta o objeto no qual atua) |
| `getYear()` | *Number*. Recupera ano menos 1900 (1997 → 97) |

| **Método** | **Ação** |
|---|---|
| `setDate(dia_do_mês)` | Acerta o dia do mês (1 a 31) |
| `setHours(hora)` | Acerta a hora (0 a 23) |
| `setMinutes(minuto)` | Acerta o minuto (0-59) |
| `setMonth(mês)` | Acerta o mês (0-11) |
| `setSeconds()` | Acerta o segundo (0-59) |
| `setTime()` | Acerta a hora em milissegundos desde 1-1-1970 0:0:0 GMT |
| `setYear()` | Acerta o ano (ano – 1900) |
| `toGMTString()` | *String*. Converte uma data em uma representação GMT |
| `toLocaleString()` | *String*. Converte a data na representação local do sistema |

Além dos métodos, que devem ser aplicados sobre objetos individuais criados com o tipo *Date*, *Date* também serve de repositório para duas funções: `Date.parse(`*string*`)` e `Date.UTC()`. Elas oferecem formas alternativas de criar objetos *Date*:

Essas funções, listadas na tabela abaixo, não são métodos de objetos *Date*, mas do construtor `Date()` e devem ser chamadas usando-se o identificador `Date` e não usando o nome de um objeto específico, por exemplo:

```
Date d = new Date();
d.parse("Jan 13, 1998 0:0:0 GMT");          // ERRADO!

d = Date.parse("Jan 13, 1998 0:0:0 GMT");   // CORRETO!
```

| **Função** | **Ação** |
|---|---|
| `parse(string)` | Retorna *Date*. Converte uma data do sistema no formato IETF (usado por servidores de email, servidores HTTP, etc.) em milisegundos desde 1/1/1970 0:0:0 GMT (UTC). O valor de retorno pode ser usado para criar uma nova data no formato JavaScript. Exemplo:<br>`DataIETF = "Wed, 8 May 1996 22:44:53 –0200";`<br>`umaData = new Date(Date.parse(DataIETF));` |
| `UTC()` | Retorna *Number*. Converte uma data no formato UTC separado por vírgulas para a representação em milisegundos:<br>`Date.UTC(ano, mês, dia [, horas[, minutos[, segundos]]]);`<br>Exemplo:<br>`millis = Date.UTC(75, 11, 13, 23, 30);` |

# *Exercícios*

4.5 Escreva um programa que receba uma data através de um campo de textos (`prompt()`) no formato **dd/mm/aaaa**. O programa deve reclamar (use `alert()`) se o formato digitado for incorreto e dar uma nova chance ao usuário. Recebido o string, ele deve ser interpretado pelo programa que deverá imprimir na página quandos dias, meses e anos faltam para a data digitada.

4.6 Crie uma página que mude de aparência de acordo com a hora do dia. Se for de manhã (entre 6 e 12 horas), a página deverá ter fundo branco e letras pretas. Se for tarde (entre 12 e 18 horas), a página deverá ter fundo amarelo e letras pretas. Se for noite (entre 18 e 24 horas), o fundo deve ser escuro com letras brancas e se for madrugada (entre 0 e 6 horas), o fundo deve ser azul, com letras brancas. Para mudar a cor de fundo, use a propriedade `document.bgColor`, passando um string com o nome da cor como argumento:

```
document.bgColor = "blue";
```

A cor do texto pode ser alterada através da propriedade `document.fgColor`.