

# 3

## Funções e objetos

NO ÚLTIMO CAPÍTULO APRESENTAMOS AS ESTRUTURAS FUNDAMENTAIS de JavaScript, que estão presentes em qualquer linguagem estruturada. Neste capítulo, apresentaremos o modelo de objetos JavaScript, que a diferencia das outras linguagens, caracterizando-a como uma linguagem baseada em objetos. Veremos o que são objetos e propriedades, como criar novas propriedades, novas funções e novos objetos.

Antes de explorarmos o modelo de objetos, devemos conhecer algumas funções úteis fornecidas em todas as implementações de client-side JavaScript. Também fazem parte da linguagem e são usadas para realizar operações úteis como conversão de dados e interpretação interativa de código.

### Funções nativas

JavaScript possui 6 funções nativas<sup>1</sup>. Essas funções são *procedimentos* que permitem realizar tarefas úteis e podem ou não retornar algum valor. Todas recebem *parâmetros* com os dados sobre os quais devem operar. Podem ser chamadas de qualquer lugar. Por exemplo:

```
ano = parseInt("1997");
```

chama a função `parseInt()` passando o string "1997" como argumento. A função `parseInt()` retorna um valor (tipo *number*) que atribuímos acima à variável `ano`. Se o valor passado não for conversível em número, `parseInt()` retorna o valor `NaN` (não é um número).

Os parâmetros (ou argumentos) de uma função são passados *por valor* entre parênteses que seguem ao nome da função. Algumas funções possuem mais de um argumento. Nesses casos, eles são separados por vírgulas:

```
cor = parseInt("0xff00d9", 16);
```

---

<sup>1</sup> A documentação JavaScript 1.1 da Netscape define 8 funções: `parseInt`, `parseFloat`, `isNaN`, `eval`, `escape`, `unescape`, `taint` e `untaint`. As funções `taint()` e `untaint()` são usadas no modelo de segurança data-tainting do browser Navigator 3.0 que foi tornado obsoleto em versões mais recentes. Outros browsers desconhecem essas funções.

Se uma função não retorna valor ou se não interessa guardar o valor de retorno, pode-se simplesmente chamá-la sem atribuí-la a qualquer variável. A função abaixo, simplesmente executa o código JavaScript que recebe como argumento:

```
eval("alert('Olá!')");
```

Além das 6 funções nativas, há muitos outros procedimentos na linguagem. A grande maioria, porém, não são rigorosamente *funções*, mas *métodos* – tipo especial de função associada a um objeto específico. As funções nativas do JavaScript estão listadas na tabela abaixo:

Função	O que faz
<code>parseInt(string)</code> ou <code>parseInt(string, base)</code>	Converte uma representação <i>String</i> de um número na sua representação <i>Number</i> . Ignora qualquer coisa depois do ponto decimal ou depois de um caractere que não é número. Se primeiro caractere não for número, retorna NaN (Not a Number). A base é a representação do <i>String</i> (2, 8, 10, 16)
<code>parseFloat(string)</code>	Converte uma representação <i>String</i> de um número na sua representação <i>Number</i> , levando em consideração o ponto decimal. Ignora qualquer coisa depois do segundo ponto decimal ou depois de um caractere que não é número. Se primeiro caractere não for número ou ponto decimal, retorna NaN (Not a Number)
<code>isNaN(valor)</code>	Retorna <code>true</code> se o valor passado não é um número.
<code>eval(string)</code>	Interpreta o string passado como parâmetro como código JavaScript e tenta interpretá-lo. <code>eval()</code> é uma função que oferece acesso direto ao interpretador JavaScript. Exemplo: <code>resultado = eval("5 + 6 / 19");</code>
<code>escape(string)</code>	Converte caracteres de 8 bits em uma representação de 7 bits compatível com o formato url-encoding. Útil na criação de cookies. Exemplo: <code>nome = escape("João");</code> // nome contém <code>Jo%E3o</code>
<code>unescape(string)</code>	Faz a operação inversão de <code>escape(string)</code> . Exemplo: <code>nome = unescape("Jo%E3o");</code> // nome contém <code>João</code>

A instrução `document.write()`, que usamos em alguns exemplos é um *método*. Métodos estão *sempre* associados a objetos (`write()`, por exemplo, opera sobre o objeto `document` – escreve na *página*). Métodos freqüentemente precisam de menos parâmetros que funções, pois obtêm todos ou parte dos dados que precisam para das propriedades do objeto ao qual pertencem. Já as funções independentes só têm os parâmetros para receber os dados que precisam.

## Funções definidas pelo usuário

Como vimos através de um exemplo no primeiro capítulo, JavaScript permite ao programador definir novas funções como uma seqüência de instruções dentro de um bloco iniciado com a

palavra-chave **function**. Uma vez criada uma função, ela pode ser usada globalmente (dentro da página onde foi definida), da mesma maneira que as funções globais do JavaScript.

O identificador da função deve vir seguido de um par de parênteses e, entre eles, opcionalmente, uma lista de parâmetros, separados por vírgulas. A implementação (seqüência de instruções) da função deve vir dentro de um bloco entre chaves “{” e “}”.

```
function nomeDaFunção (param1, param2, ..., paramN) {
    ... implementação ...
}
```

Para retornar um valor, é preciso usar uma instrução **return**:

```
function soma () {
    a = 2; b = 3;
    return a + b;
}
```

Funções não precisam ter parâmetros. Funções que operam sobre variáveis globais ou simplesmente executam procedimentos têm todos os dados que precisam para funcionar à disposição. Não é o caso da função acima, que seria mais útil se os tivesse:

```
function soma (a, b) {
    return a + b;
}
```

Os parâmetros têm um *escopo local* ao bloco da função e *não são visíveis fora dele*. Variáveis utilizadas dentro da função podem ser locais ou não. Para garantir que o escopo de uma variável seja local a uma função, é necessário declará-las locais usando **var**:

```
x = 60; // este x é global
function soma(a, b) {
    var x = a; // este x é uma variável local
    var y = b;
    return x + y;
}
```

A função acima pode ser chamada de qualquer lugar na página HTML da forma:

```
resultado = soma(25, 40);
```

passando *valores* na chamada. Os valores são passados à função por atribuição. No exemplo acima, a variável local **a** recebe 25 e **b** recebe 40. A variável global **resultado** recebe 65 que é o valor retornado pela função.

Identificadores utilizados para nomes de função são *propriedades* do contexto onde foram definidos. Não pode haver, por exemplo, uma variável global com o mesmo nome que uma função. O uso do identificador de uma função (sem os parênteses ou argumentos) como argumento de uma atribuição, copia a *definição da função* para outra variável, por exemplo:

```
sum = soma;
```

copia a definição da função `soma()` para a variável `sum`, que agora é uma função. A nova variável pode então ser usada como função:

```
resultado = sum(25, 40);
```

## Exercícios

- 3.1 Escreva uma função recursiva ou iterativa `fatorial(n)` que retorne o fatorial de um número, passado como parâmetro ( $n! = n(n-1)(n-2)...(2)(1)$ ). Chame a função de outro bloco script no seu código usando-a para imprimir uma tabela de fatoriais de 0 a 10:

0!	1
1!	1
2!	2
3!	6

- 3.2 Escreva uma função `combinacao(n, k)` que receba dois parâmetros  $n$  e  $k$  (número e amostra) e retorne o número de combinações do número na amostra passada como parâmetro. Chame a função `fatorial()` do exercício 1.6 a partir desta função. A fórmula para calcular a combinação de  $n$  em amostras de  $k$  é:

$$C(n,k) = n! / (n - k)! * k!$$

## Objetos

A maior parte da programação em JavaScript é realizada através de objetos. Um objeto é uma estrutura mais elaborada que uma simples variável que representa tipos primitivos. Variáveis podem conter apenas um valor de cada vez. Objetos podem conter vários valores, de tipos diferentes, ao mesmo tempo.

Um objeto é, portanto, uma coleção de valores. Em várias situações necessitamos de tais coleções em vez de valores isolados. Considere uma data, que possui um dia, um mês e um ano. Para representá-la em JavaScript, podemos definir três variáveis contendo valores primitivos:

```
dia = 17;  
mes = "Fevereiro";  
ano = "1999";
```

Para manipular com uma única data não haveria problemas. Suponha agora que temos que realizar operações com umas 10 datas. Para fazer isto, teríamos que criar nomes significativos para cada grupo de dia/mes/ano e evitar que seus valores se misturassem.

A solução para este problema é usar um objeto, que trate cada coleção de dia, mes e ano como um grupo. Objetos são representados em JavaScript por variáveis do tipo *object*. Esse tipo é capaz de armazenar coleções de variáveis de tipos diferentes como sendo suas *propriedades*. Suponha então que a variável `dataHoje` é do tipo *object*, podemos definir as variáveis dia, mes e ano como suas propriedades, da forma:

```
dataHoje.dia = 17;
dataHoje.mes = "Fevereiro";
dataHoje.ano = "1999";
```

As propriedades de `dataHoje` são do tipo *number* e *string* mas poderiam ser de qualquer tipo, inclusive *object*. Se uma propriedade tem o tipo *object*, ela também pode ter suas próprias propriedades e assim formar uma hierarquia de objetos interligados pelas propriedades:

```
dataHoje.agora.minuto = 59; // agora: objeto que representa uma hora
```

E como fazemos para criar um objeto? Existem várias formas, mas nem sempre isto é necessário. Vários objetos já são fornecidos pela linguagem ou pela página HTML. O próprio contexto global onde criamos variáveis e definimos funções é tratado em JavaScript como um objeto, chamado de *Global*. As variáveis que definimos ou declaramos fora de qualquer bloco são as *propriedades* desse objeto. Os tipos primitivos em JavaScript também assumem um papel duplo, se comportando ora como tipo primitivo – com apenas um valor, ora como objeto – tendo o seu valor armazenado em uma propriedade. O programador não precisa se preocupar com os detalhes dessa *crise de identidade* das variáveis JavaScript. A conversão *objeto - tipo primitivo* e vice-versa é totalmente transparente.

Uma simples atribuição, portanto, é suficiente para criar variáveis que podem se comportar como objetos ou valores primitivos:

```
num = 5; // num é tipo primitivo number e objeto do tipo Number
boo = true; // boo é tipo primitivo boolean e objeto do tipo Boolean
str = "Abc"; // str é tipo primitivo string e objeto do tipo String
```

Objetos podem ser de vários tipos (não confunda tipo de *objeto* com tipo de *dados*), de acordo com as propriedades que possuem. Um objeto que representa uma data, por exemplo, é diferente de um objeto que representa uma página HTML, com imagens, formulários, etc. A linguagem JavaScript define nove tipos de objetos nativos embutidos. Quatro representam tipos primitivos: *Number*, *String*, *Boolean* e *Object* (usamos a primeira letra maiúscula para distinguir o tipo de objeto do tipo de dados).

## Construtores e o operador “new”

Para criar novos objetos é preciso usar um *construtor*. O construtor é uma função especial associada ao *tipo do objeto* que define todas as características que os objetos criados terão. O construtor só criará um novo objeto se for chamado através do operador `new`. Este operador cria um novo objeto de acordo com as características definidas no construtor. Atribuindo o objeto criado a uma variável, esta terá o tipo de dados *object*:

```
dataViagem = new Date(1999, 16, 01);
```

Variável do tipo *object* que armazena um objeto *Date*  
 Utiliza as informações retornadas por *Date()*  
 Chama a função *Date()* (construtor) que retorna as informações necessárias para criar o objeto.

Os tipos de objetos nativos *Object*, *Number*, *String*, *Boolean*, *Function*, *Date* e *Array* (veja figura na página 2-4) todos possuem construtores definidos em JavaScript. Os construtores são funções globais e devem ser chamadas através do operador `new` para que um objeto seja retornado. A tabela abaixo relaciona os construtores nativos do JavaScript<sup>2</sup>:

<b>Construtor</b>	<b>Tipo de objeto construído</b>
<code>Object()</code> <code>Object(valor)</code>	Constrói objeto genérico do tipo <i>Object</i> . Dependendo do tipo do valor primitivo passado, o resultado pode ainda ser um objeto <i>String</i> , <i>Number</i> ou <i>Boolean</i> .
<code>Number()</code> <code>Number(valor)</code>	Constrói um objeto do tipo <i>Number</i> com valor inicial zero, se for chamada sem argumentos ou com o valor especificado.
<code>Boolean()</code> <code>Boolean(valor)</code>	Constrói um objeto do tipo <i>Boolean</i> com valor inicial <code>false</code> , se for chamada sem argumentos ou com o valor especificado.
<code>String()</code> <code>String(valor)</code>	Constrói um objeto do tipo <i>String</i> com valor inicial "", se for chamada sem argumentos ou com o valor especificado.
<code>Array()</code> <code>Array(tamanho)</code>	Constrói um objeto do tipo <i>Array</i> , que representa uma coleção ordenada (vetor) de <i>tamanho</i> inicial zero ou definido através de parâmetro.
<code>Function()</code> <code>Function(corpo)</code> <code>Function(arg1, arg2, ..., corpo)</code>	Constrói um objeto do tipo <i>Function</i> com <i>corpo</i> da função vazio, com uma string contendo o código JavaScript que compõe o corpo da função, e com argumentos.
<code>Date()</code> <code>Date(ano, mes, dia)</code> <code>Date(ano, mes, dia, hora, min, seg)</code> <code>Date(string)</code> <code>Date(milissegundos)</code>	Constrói um objeto do tipo <i>Date</i> . O primeiro construtor constrói um objeto que representa a data e hora atuais. Os outros são formas diferentes de construir datas no futuro ou no passado.

Tipos primitivos podem assumir o papel de objetos. A conversão é feita automaticamente mas também pode ser feita explicitamente através de um construtor. Há duas formas, portanto, de criar um número contendo o valor 13:

```
n = 13; // valor primitivo
m = new Number(13); // objeto
```

A primeira cria uma variável que contém o valor primitivo 13. A segunda forma, cria um objeto explicitamente. A qualquer momento, porém, dentro de um programa JavaScript, as representações podem ser trocadas. Os construtores de objetos que representam tipos

<sup>2</sup> Os construtores `Image()` e `Option()` também fazem parte do JavaScript, mas não são "nativos".

primitivos são chamados automaticamente pelo sistema durante a conversão de um tipo de dados primitivo em objeto. A conversão inversa também é realizada automaticamente através de métodos do objeto.

Para o programador, tanto faz usar um procedimento como outro. A primeira forma é sempre mais simples e mais clara. Para outros tipos de objetos, como *Date*, não existe atalho simples e é preciso criar o objeto explicitamente.

## Propriedades

Cada objeto pode ter uma coleção de propriedades, organizadas através de *índices* ou de *nomes* e acessadas através de colchetes [ e ]. Para criar novas propriedades para um objeto, basta defini-las através de uma atribuição:

```
zebra = "Zebra"; // variável zebra é do tipo primitivo string ...
zebra[0] = true; // ... agora assume o papel de objeto do tipo String
zebra[1] = "brancas"; // para que possa ter propriedades.
zebra[2] = 6;
```

As propriedades acima foram definidas através de um índice. Índices geralmente são indicados quando a ordem das propriedades têm algum significado. No exemplo acima, as propriedades seriam melhor definidas através de nomes:

```
zebra ["domesticada"] = true;
zebra ["listras"] = "brancas";
zebra ["idade"] = 6;
```

Os nomes das propriedades também podem ser usadas como variáveis associadas ao objeto, como temos feito até agora. Para indicar as variáveis que pertencem ao objeto, e não a um contexto global ou local, é preciso *ligá-la* ao objeto através do operador ponto “.”:

```
zebra.domesticada = true;
zebra.listras = "brancas";
zebra.idade = 6;
```

Várias propriedades estão documentadas e estão disponíveis para todos os objetos dos tipos nativos. Qualquer valor primitivo *string*, por exemplo, é um objeto *String*, e possui uma propriedade `length` que contém um número com a quantidade de caracteres que possui:

```
tamanho = zebra.length; // propriedade length de str contém 5 (Number)
```

Diferentemente das propriedades que definimos para *zebra*, `length` existe em qualquer *String* pois está associada ao *tipo do objeto*. O tipo do objeto é representado pelo seu construtor e define as características de todos os objetos criados com o construtor. As propriedades que nós criamos (*domesticada*, *listras*, *idade*) pertencem ao objeto *zebra* apenas. Para acrescentar propriedades ao tipo *String*, precisamos usar uma propriedade especial dos objetos chamada de *prototype*. Veremos como fazer isto no próximo capítulo.

## Métodos

As propriedades de um objeto podem conter tipos primitivos, objetos ou funções. As funções são objetos do tipo *Function*. Funções que são associadas a objetos são chamadas de *métodos*. Todos os objetos nativos do JavaScript possuem métodos. Pode-se ter acesso aos métodos da mesma maneira que se tem acesso às propriedades:

```
letra = zebra.charAt(0); // método charAt(0) retorna "Z" (String)
```

Também é possível acrescentar métodos aos objetos e ao tipo dos objetos. Para acrescentar um método ao objeto *zebra*, basta criar uma função e atribuir o identificador da função a uma propriedade do objeto:

```
function falar() {  
    alert("Rinch, rinch!");  
}  
  
zebra.rinchar = falar;
```

A instrução acima copia a *definição* de *falar()* para a propriedade *rinchar*, de *zebra*. A propriedade *rinchar* agora é *método* de *zebra* e pode ser usado da forma:

```
zebra.rinchar();
```

Os métodos, porém, são mais úteis quando atuam sobre um objeto alterando ou usando suas propriedades. Na seção seguinte veremos alguns exemplos de métodos desse tipo além de como criar novos tipos de objetos.

## Criação de novos tipos de objetos

A especificação de um novo tipo de objeto é útil quando precisamos representar tipos de dados abstratos não disponíveis em JavaScript. Um novo tipo de objeto pode ser especificado simplesmente definindo um construtor:

```
function Conta() { }
```

A função *Conta*, acima, nada mais é que uma função comum. O que a transforma em construtor é a forma como é chamada, usando *new*. Tendo-se a função, é possível criar objetos com o novo tipo e atribuir-lhes propriedades:

```
cc1 = new Conta(); // cc1 é do tipo object  
cc1.correntista = "Aldebaran";  
cc1.saldo = 100.0;
```

As propriedades *correntista* e *saldo* acima existem apenas no objeto *cc1*, e não em outros objetos *Conta*. Isto porque foram definidas como propriedades do *objeto* (como as propriedades que definimos para *zebra*), e não *do tipo de objeto*. Se ela for definida dentro da definição do construtor *Conta()*, valerá para todos os objetos criados com o construtor:

```
function Conta() {
    this.correntista = "Não identificado";
    this.saldo = 0.0;
}
```

Agora todo objeto *Conta* terá propriedades iniciais definidas. A palavra-chave `this` é um ponteiro para o próprio objeto. Dentro do construtor, o objeto não tem nome. Quando o construtor é invocado, `this`, que significa “este”, se aplica ao objeto que está sendo criado. Podemos usar `this` para criar um outro construtor, mais útil, que receba argumentos:

```
function Conta(corr, saldo) {
    this.correntista = corr;
    this.saldo = saldo;
}
```

Não há conflito entre a *variável local* `saldo` e a *propriedade* `saldo` do objeto *Conta* pois elas existem em contextos diferentes. Com o novo construtor, é possível criar contas da forma:

```
cc2 = new Conta("Sirius", 326.50);
cc1 = new Conta("Aldebaran", 100.0);
```

Para definir métodos para o novo tipo, basta criar uma função e copiá-la para uma propriedade do construtor, por exemplo:

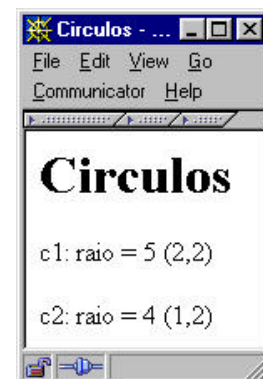
```
function metodo1() {
    document.write("Saldo: " + this.saldo);
}
function Conta(corr, saldo) {
    this.correntista = corr;
    this.saldo = saldo;
    this.imprimeSaldo = metodo1;
}
```

Agora qualquer objeto criado com o construtor `Conta()` possui um método que imprime na página o valor da propriedade `saldo`:

```
cc3 = new Conta("", 566.99);
cc3.imprimeSaldo(); // imprime da página: "Saldo: 566.99"
```

## Exercício resolvido

Crie um novo tipo *Circulo* especificando um construtor da forma `Circulo(x, y, r)` onde `x` e `y` são as coordenadas cartesianas do círculo e `r` é o seu raio. Utilize o construtor para criar dois objetos `c1` e `c2` e imprimir seus valores na tela do browser da forma mostrada na figura ao lado.



## Solução

Uma possível solução completa está mostrada na listagem a seguir. Poderíamos ter evitado o código repetitivo ao imprimir os valores criando um método para círculo que fizesse isto. Esse método é proposto como exercício.

```
<HTML> <HEAD>
<TITLE>Circulos</TITLE>
<script>
function Circulo(x, y, r) {          // função "construtora"
    this.x = x; // definição das propriedades deste objeto
    this.y = y; // a referência this é ponteiro para o próprio objeto
    this.r = r;
}
</script>
</HEAD>

<BODY>
<h1>Circulos</h1>
<script>
c1 = new Circulo(2,2,5); // uso da função construtora
c2 = new Circulo(0,0,4); // para criar dois circulos

c2.x = 1;          // definicao de propriedades ...usando o operador "."
c2["y"] = 2;      // ... usando o vetor associativo

// uso de propriedades
document.write("<P>c1: raio=" + c1.r + " (" + c1.x + "," + c1.y + ")");
document.write("<P>c1: raio=" + c2.r + " (" + c2.x + "," + c2.y + ")");
</script>

</BODY>
</HTML>
```

No browser, os novos objetos *Circulo* (*c1* e *c2*) são propriedades da janela onde a função foi definida e a função construtora *Circulo()* se comporta como um método dessa janela, podendo ser usado de outras janelas ou frames.

## A estrutura *for...in*

JavaScript possui uma estrutura de repetição especial que permite refletir as propriedades de um objeto: a estrutura **for...in**, que pode ser usada para ler todas as propriedades de um objeto, e extrair os seus valores. A sintaxe é

```
for (variavel in nome_do_objeto) {
    // declarações usando variavel
}
```

onde `variável` é o nome da variável que será usada para indexar as propriedades do objeto. O bloco será repetido até não haver mais propriedades. Em cada iteração, uma propriedade estará disponível em `variavel` e seu valor poderá ser extraído usando vetores associativos, da forma:

```
objeto[variavel]
```

Veja como exemplo a função abaixo, que retorna todas as propriedades de um objeto:

```
function mostraProps(objeto) {
  props = "";
  for (idx in objeto) {
    props += idx + " = " + objeto[idx] + "\n";
  }
  return props;
}
```

Se passássemos como argumento à função acima o objeto `c2` (*Circulo*) criado no exercício resolvido:

```
document.write("<pre>" + mostraProps(c2) + "</pre>");
```

teríamos os valores seguintes impressos na página:

```
x = 1
y = 2
r = 4
```

## *Referências e propriedades de propriedades*

Nos exemplos que vimos até agora, as propriedades de um objeto ou eram valores primitivos ou funções. Propriedades podem ser definidas também como objetos, que por sua vez podem conter outras propriedades. Suponha um objeto definido pelo tipo *Alvo*:

```
function Alvo(circ) {
  this.circ = circ;
}

c1 = new Circulo(3, 3, 6);
a1 = new Alvo(c1);
```

Os dois objetos acima possuem uma relação hierárquica: Um *Alvo* contém um *Circulo*. É possível, através de um *Alvo*, ter acesso e propriedades do *Circulo* que ele contém:

```
a1.circ.x = 20;
a1.circ.y = 10;
```

As instruções acima alteram os valores do círculo do objeto `a1`, e *também* os valores do círculo `c1`, que são *o mesmo objeto*! Isto acontece porque o *Alvo* foi criado usando um círculo já existente, passado por *referência* e não por *valor*. Não é uma cópia. A atribuição simples de



O objeto *Window* é o mais importante da hierarquia do browser. É representado através da referência global `window` que representa a janela atual. A hierarquia da figura identifica objetos que podem ser interligados pelas suas propriedades. O tipo *Window* possui uma propriedade `document` que representa a página HTML que está sendo exibida na janela. No diagrama a propriedade é representada pelo tipo *Document*, abaixo de *Window* na hierarquia.

A outra raiz na hierarquia do browser é *Navigator*, que representa o próprio browser. É utilizado principalmente para extrair informações de identificação do browser, permitindo que programas JavaScript possam identificá-lo e tomar decisões baseado nas informações obtidas. Nos browsers Microsoft, *Navigator* não é raiz de hierarquia mas uma propriedade de *Window*, chamada `navigator`.

Todas as *variáveis globais* criadas em um programa JavaScript em HTML são propriedades temporárias do objeto *Global* e da janela do browser onde o programa está sendo interpretado. Por exemplo, a variável:

```
<script>
  var nome;
</script>
```

é propriedade de `window`, e pode ser utilizada na página, das duas formas:

```
nome = "Saddam";
window.nome = "Saddam";
```

pois o nome `window`, que representa a janela ativa do browser, sempre pode ser omitido quando o script roda dentro dessa janela.

## *Acesso a objetos do browser e da página*

Cada componente de uma página HTML, seja imagem, formulário, botão, applet ou link, define um *objeto* que poderá ser manipulado em JavaScript e agir como *referência* ao componente da página. Os nomes desses objetos não podem ser criados aleatoriamente em JavaScript mas dependem do *modelo de objetos do documento*, adotado pelo browser. Cada nome tem uma ligação com o elemento HTML ou propriedade do browser que representa. Por exemplo `window` é o nome usado para representar um objeto que dá acesso à janela do browser atual:

```
x = window;           // x é uma referência 'Window'
```

Todos os outros elementos da janela são obtidos a partir de propriedades do objeto `window`. Por exemplo, a propriedade `document`, que todo objeto do tipo *Window* possui, refere-se à página contida na janela atual:

```
y = x.document;      // window.document é referencia 'Document'
```

Há pelo menos uma propriedade em cada objeto do HTML que se refere a objetos que ele pode conter ou a um objeto no qual está contido. É essa característica permite organizar os



```
janela2.document.open(); // ou window.janela2.document.open()
janela2.document.write("Eu sou texto na Janela 2");
```

Este tipo de relação (janelas que têm janelas como propriedades) é ilustrado no diagrama de objetos da página 3-12. A última janela aberta tem um *status* especial pois representa a aplicação. Frames são outro exemplo de janelas dentro de janelas. As janelas de frames têm propriedades que permitem o acesso bidirecional.

## Manipulação de objetos do HTML

Todos os objetos criados em HTML estão automaticamente disponíveis em JavaScript, mesmo que um nome não seja atribuído a eles. Por exemplo, se há três blocos `<FORM>...</FORM>` em uma página, há três objetos do tipo *Form* no JavaScript. Se eles não tem nome, pode-se ter acesso a eles através da propriedade `'forms'` definida em *Document*. Essa propriedade armazena os objetos *Form* em uma coleção ordenada de propriedades (vetor). Cada formulário pode então ser recuperado através de seu índice:

```
frm1 = document.forms[0]; // mesma coisa que window.document.forms[0]
frm2 = document.forms[1];
```

Todos os índices usados nos vetores em JavaScript iniciam a contagem em 0, portanto, `document.forms[0]`, refere-se ao primeiro formulário de uma página.

O diagrama de objetos da página 3-12 mostra *Form* como raiz de uma grande hierarquia de objetos. Se houver, por exemplo, dentro de um bloco `<FORM>...</FORM>` 5 componentes, entre botões, campos de texto e caixas de seleção, existirão 5 objetos em JavaScript dos tipos *Text*, *Button* e *Select*. Independente do tipo de componente de formulário, eles podem ser acessados na ordem em que aparecem no código, através da propriedade `elements`, de *Form*:

```
texto = document.forms[0].elements[1]; // qual será o componente?
```

Os vetores são necessários apenas quando um objeto não tem nome. Se tiver um nome (definido no código HTML, através do atributo `NAME` do descritor correspondente), o ideal é usá-lo já que independe da ordem dos componentes, e pode fornecer mais informações como por exemplo, o tipo do objeto (é um botão, um campo de textos?):

```
<form name="f1">
  <input type=button name="botao1" value="Botão 1">
  <input type=text name="campoTexto" value="Texto Muito Velho">
</form>
```

Agora é possível ter acesso ao campo de textos em JavaScript usando nomes, em vez de índices de vetores:

```
texto = document.f1.campoTexto;
textoVelho = texto.value; // lendo a propriedade value...
texto.value = "Novo Texto"; // redefinindo a propriedade value
```

O código acima também poderia ter sido escrito da forma, com os mesmos resultados:

```
textoVelho = document.f1.campoTexto.value;
document.f1.campoTexto.value = "Novo Texto";
```

## Exercício resolvido

Implemente o somador mostrado na figura ao lado em JavaScript. Deve ser possível digitar números nos dois campos de texto iniciais, apertar o botão “=” e obter a soma dos valores no terceiro campo de texto.

Para ler um campo de texto, você vai ter que ter acesso à propriedade `value` dos campos de texto (objeto do tipo *Text*). A propriedade `value` é um *String* que pode ser lido e pode ser alterado. Os campos de texto são acessíveis de duas formas:

- através do vetor `elements`, que é uma propriedade de todos os componentes do formulário (use `elements[0]`, `elements[1]`, etc.)
- através do nome do elemento (atributo `NAME` do HTML).

Quando ao botão, é preciso que no seu evento `ONCLICK`, ele chame uma função capaz de recuperar os dois valores e colocar sua soma na terceira caixa de texto. Este exercício está resolvido. Tente fazê-lo e depois veja uma das possíveis soluções na próxima seção.



## Solução

Observe a utilização de toda a hierarquia de objetos para ler os campos do formulário, a conversão de string para inteiro usando a função `parseFloat()` e a chamada à função `soma()` através do evento `ONCLICK` do botão.

```
<html> <head>
  <script language=JavaScript>
    function soma() {
      a = document.f1.val1.value;
      b = document.f1.val2.value;
      document.f1.val3.value = parseFloat(a) + parseFloat(b);
    }
  </script>
</head>
<body>
  <h1>Somador JavaScript</h1>
  <form name="f1">
```

```



```

Observe no código acima que a função `soma()` foi definida no `<HEAD>`. Isto é para garantir que ela já esteja carregada quando for chamada pelo evento. É uma boa prática definir sempre as funções dentro de um bloco `<SCRIPT>` situado no bloco `<HEAD>` da página.

## *Estruturas e operadores utilizados com objetos*

JavaScript possui várias estruturas e operadores criados especificamente para manipulação de objetos. Já vimos o uso do operador `new`, da estrutura `for...in` e da referência `this`. Nesta seção conheceremos aplicações de `this` em HTML, a estrutura `with` e os operadores `typeof`, `void` e `delete`.

### *this*

A palavra-chave `this` é usada como referência ao objeto no qual se está operando. A palavra-chave `this` pode ser usada apenas quando se está *dentro de um objeto*. Em objetos criados em JavaScript, só usamos `this` dentro de funções construtoras e métodos. No caso dos objetos HTML, `this` só faz sentido quando é usada dentro de um dos atributos de eventos (`ONCLICK`, `ONMOUSEOVER`, `HREF`, etc.):

```



```

Na linha acima, `this` refere-se ao objeto *Button*. A propriedade de *Button* chamada `form` é uma referência ao formulário no qual o botão está contido (subindo a hierarquia). Usando o código acima, podemos reescrever o script do somador para que receba uma referência para o formulário (que chamamos localmente de `calc`):

```

<script>
function soma(calc) {
    a = calc.va11.value;
    b = calc.va12.value;
    calc.va13.value = parseFloat(a) + parseFloat(b);
}
</script>

```

## *with*

`with` é uma estrutura especial eu permite agrupar propriedades de objetos, dispensando a necessidade de chamá-las pelo nome completo. É útil principalmente quando se trabalha repetidamente com hierarquias de objetos. Veja um exemplo. Em vez de usar:

```
objeto.propriedade1 = 12;
objeto.propriedade2 = true;
objeto.propriedade3 = "informação";
```

use

```
with(objeto) {
  propriedade1 = 12;
  propriedade2 = true;
  propriedade3 = "informação";
}
```

Veja uma aplicação, novamente relacionada ao somador:

```
<script>
function soma() {
  with(document.f1) {
    a = val1.value;
    b = val2.value;
    val3.value = parseFloat(a) + parseFloat(b);
  }
}</script>
```

## *typeof*

Uma das maneiras de identificar o tipo de um objeto, é através do operador `typeof`. Este operador retorna um *String* que indica o tipo de dados primitivo (*object*, *number*, *string*, *boolean* ou *undefined*) do operando ou se é um objeto do tipo *Function*. O operando que pode ser uma variável, uma expressão, um valor, identificador de função ou método. A sintaxe é:

```
typeof operando // ou typeof (operando)
```

O conteúdo da string retornada por `typeof` é uma das seguintes: `undefined` (se o objeto ainda não tiver sido definido), `boolean`, `function`, `number`, `object` ou `string`. Veja alguns exemplos:

```
var coisa; // typeof coisa: undefined
var outraCoisa = new Object(); // typeof outraCoisa: object
var texto = "Era uma vez..."; // typeof texto: string
var numero = 13; // typeof numero: number
var hoje = new Date(); // typeof hoje: object
var c = new Circulo(3, 4, 5); // typeof c: object
var boo = true; // typeof boo: boolean
```

O operador `typeof` retorna o tipo `function` para qualquer tipo de procedimento, seja método, construtor ou função. Deve-se usar apenas o *identificador* do método ou função, eliminando os parênteses e argumentos:

```
typeof Circulo      // function
typeof eval        // function
typeof document.write // function
typeof Document    // function
typeof Window      // undefined (nao ha construtor p/ o tipo Window)
typeof window      // object
typeof Math        // object (Math nao é tipo... é objeto)
```

O uso de `typeof` é útil em decisões para identificar o tipo de um objeto primitivo, mas não serve para diferenciar por exemplo, um objeto `Date` de um `Array`, ou um `document` de um objeto `Circulo`. São todos identificados como `object`.

Uma forma mais precisa para identificar o tipo do objeto, é identificar seu construtor. Toda a definição do construtor de um objeto pode ser obtida através da propriedade `constructor`, que todos os objetos possuem. Por exemplo, `c.constructor` (veja exemplos na página anterior) contém toda a função `Circulo(x, y, r)`. Para obter só o *nome* do construtor, pode-se usar a propriedade `name` de `constructor`:

```
document.write(c.constructor.name);      // imprime Circulo
document.write(hoje.constructor.name);   // imprime Date
```

Assim, pode-se realizar testes para identificar o tipo de um objeto:

```
if (c.constructor.name == "Circulo") {
    ...
}
```

## *void*

O operador `void` é usado para executar uma expressão JavaScript, mas jogar fora o seu valor. É útil em situações onde o valor de uma expressão não deve ser utilizado pelo programa. A sintaxe está mostrada abaixo (os parênteses são opcionais):

```
void (expressão);
```

O operador `void` é útil onde o valor retornado por uma expressão pode causar um efeito não desejado. Por exemplo, na programação do evento de clique do vínculo de hipertexto (`HREF`), o valor de retorno de uma função poderia fazer com que a janela fosse direcionada a uma página inexistente.

Considere o exemplo abaixo. Suponha que no exemplo acima, `enviaFormulario()` retorne o texto “enviado”. Este valor poderia fazer com que a janela tentasse carregar uma suposta página chamada “enviado”:

```
<a href="javascript: enviaFormulario()">Enviar formulário</a>
```

Para evitar que o valor de retorno interfira no código, e ainda assim poder executar a função, usamos `void` que descarta o valor de retorno:

```
<a href="javascript: void(enviaFormulario())">Enviar formulário</a>
```

## *delete*

O operador `delete` não existe em JavaScript 1.1. Pode ser usado nos browsers que suportam implementações mais recentes para remover objetos, elementos de um vetor ou propriedades de um objeto. Não é possível remover variáveis declaradas com `var` ou propriedades e objetos pré-definidos. A sintaxe é a seguinte:

```
delete objeto;  
delete objeto.propriedade;  
delete objeto[índice];
```

Se a operação `delete` obtém sucesso, ela muda o valor da propriedade para `undefined`. A operação retorna `false` se a remoção não for possível.

## *Exercícios*

- 3.5 Com base no somador mostrado no exercício resolvido, implemente uma calculadora simples que realize as funções de soma, subtração, divisão e multiplicação. A calculadora deverá utilizar a mesma janela para mostrar os operandos e o resultado final (figura ao lado). O resultado parcial deverá ser armazenado em uma variável global e exibida quando for apertado o botão “=”. *Dica:* aproveite o esqueleto montado no arquivo `cap3/ex35.html` (que monta o HTML da figura mostrada) e use `eval()` para realizar o cálculo do valor armazenado.

