

Desenvolvendo
Websites Interativos
com

JavaScript

Helder da Rocha

Rev.: JS11-01-1999/01 A4 (AB05)
Janeiro de 1999

Copyright © 1997, 1998, 1999 por Helder Lima Santos da Rocha.

Este livro é parte integrante dos cursos da série “Web Sites Interativos” de propriedade de Helder Lima Santos da Rocha e não pode ser vendido separadamente.

Todos os direitos estão reservados. Nenhuma parte desta publicação poderá ser reproduzida ou utilizada em outros cursos sem a autorização, por escrito, do autor. Alunos, professores ou funcionários de instituições educacionais ou filantrópicas podem requerer autorização gratuita para a reprodução deste material para uso próprio, ou para uso em treinamentos sem fins lucrativos. O autor pode ser contatado através dos endereços eletrônicos hlsr@uol.com.br, helder@ibpinetsp.com.br ou através do pager (011) 866-4666 (107-4242)

Embora todos os cuidados tenham sido tomados na preparação deste livro, o autor não assume responsabilidade por erros e omissões, ou por quaisquer danos resultantes do uso das informações nele contidas.

Capa, editoração eletrônica e revisão: o autor.

Código desta edição: **JS11-01-1999/01** (Revisão Jan/1999)

Formato: A4 – apostila (numeração de páginas por capítulo).

Módulos opcionais: nenhum.

Responsável por esta tiragem: O autor. Esta é uma edição particular. Reprodução não-autorizada.

Tiragem desta edição: 1 cópia para biblioteca Itelcon.

da Rocha, Helder Lima Santos, 1968-

“Desenvolvendo Web Sites Interativos com JavaScript”. Quarta versão: jan/1999 (primeira versão em ago/1997). /192 páginas (A4). Disquete de 3 ¼” com código-fonte.

Helder da Rocha. – São Paulo, SP, 1999.

Inclui disquete de 3 ¼”.

1. JavaScript (linguagem de programação). 2. Web design. 2. HTML (linguagem de marcação de página). 4. World Wide Web (sistema de recuperação de informações). Internet (rede de computadores, meio de comunicações). I. Título

Netscape Navigator, Netscape Communicator, LiveWire, LiveConnect e JavaScript são marcas registradas da Netscape Communications Inc. ActiveX, ASP, Active Server Pages, Microsoft Internet Explorer, FrontPage e JScript e VBScript são marcas registradas da Microsoft Corp. Java é marca registrada da Sun Microsystems. Quaisquer outras marcas registradas citadas nesta obra pertencem aos seus respectivos proprietários.

Conteúdo

Prefácio

1. Introdução a JavaScript

O que é JavaScript?.....	1-2
JavaScript não é Java	1-3
Quem suporta JavaScript?.....	1-3
O que se pode fazer com JavaScript?	1-4
Como programar com JavaScript?	1-4
Formas de usar JavaScript.....	1-5
Blocos <SCRIPT> embutidos na página	1-5
Arquivos importados.....	1-6
Tratamento de eventos	1-7
Introdução prática	1-9
Exercício resolvido	1-9
Solução	1-10
Exercícios	1-13

2. Sintaxe e estrutura

Variáveis.....	2-2
Tipos de dados e literais.....	2-3
Caracteres especiais	2-6
Identificadores e palavras reservadas	2-6
Operadores e expressões.....	2-8
Estruturas de controle de fluxo.....	2-10
if... else.....	2-10
for.....	2-10
while.....	2-11
break e continue.....	2-12
for ... in e with	2-12
Exercícios	2-12

3. Funções e objetos

<i>Funções nativas</i>	3-1
<i>Funções definidas pelo usuário</i>	3-2
Exercícios.....	3-4
<i>Objetos</i>	3-4
Construtores e o operador "new"	3-5
Propriedades.....	3-7
Métodos	3-8
<i>Criação de novos tipos de objetos</i>	3-8
Exercício resolvido	3-9
Solução	3-10
A estrutura for...in	3-10
Referências e propriedades de propriedades	3-11
Exercícios.....	3-12
<i>Modelo de objetos do HTML</i>	3-12
Acesso a objetos do browser e da página.....	3-13
Manipulação de objetos do HTML.....	3-15
Exercício resolvido	3-16
Solução	16
<i>Estruturas e operadores utilizados com objetos</i>	3-17
this.....	3-17
with	3-17
typeof.....	3-18
void	3-19
delete.....	3-19
<i>Exercícios</i>	3-20

4. Objetos nativos embutidos

<i>Object</i>	4-2
<i>Number</i>	4-3
<i>Boolean</i>	4-3
<i>Function</i>	4-4
<i>String</i>	4-7
Exercícios.....	4-10
<i>Array</i>	4-10
Exercícios.....	4-12
<i>Math</i>	4-13
Exercícios.....	4-15
<i>Date</i>	4-15
Exercícios.....	4-17

5. As janelas do browser

<i>Objeto Window</i>	5-2
Janelas de diálogo.....	5-3
Métodos para manipular janelas	5-4
Janelas com aparência personalizada.....	5-5
Propriedades da barra de status	5-5
Eventos	5-6
<i>Comunicação entre janelas</i>	5-7
Exercício Resolvido.....	5-8
Solução	5-8
<i>Frames HTML</i>	5-10
Usando frames em JavaScript.....	5-13
<i>Exercícios</i>	5-15

6. O Browser

<i>Objeto Navigator</i>	6-1
Identificação do nome do fabricante.....	6-2
Identificação da versão	6-3
Identificação da plataforma.....	6-3
Exercício Resolvido.....	6-4
Solução	6-4
Métodos	6-5
<i>Plug-ins e tipos MIME</i>	6-6
MimeType.....	6-6
PlugIn	6-7
<i>Data-tainting</i>	6-8
<i>Exercício</i>	6-9

7. Navegação

<i>Objeto History</i>	7-1
Exercícios.....	7-2
<i>Objeto Location</i>	7-3
Exercícios.....	7-3
<i>Objetos Area e Link</i>	7-4
Eventos	7-5
<i>Objeto Anchor</i>	7-6
<i>Exercícios</i>	7-7

<i>8. A página HTML</i>	
<i> Objeto Document</i>	<i>8-1</i>
Métodos	8-3
<i> Geração de páginas on-the-fly</i>	<i>8-4</i>
Exercício Resolvido.....	8-5
Solução	8-6
Eventos	8-9
<i> Exercícios</i>	<i>8-9</i>

9. Imagens

<i>Image.....</i>	<i>9-1</i>
Eventos	9-5
Exercício Resolvido.....	9-6
Solução	9-6
<i>Exercícios.....</i>	<i>9-7</i>

10. Formulários

<i>Objeto Form</i>	<i>10-1</i>
Elementos de um formulário	10-3
Métodos	10-4
Eventos	10-4
<i>Objetos Button, Reset e Submit.....</i>	<i>10-5</i>
Eventos	10-7
<i>Objetos Password, Text e Textarea.....</i>	<i>10-7</i>
Eventos	10-9
<i>Objeto Hidden</i>	<i>10-11</i>
<i>Objeto Checkbox e Radio.....</i>	<i>10-12</i>
Eventos	10-14
<i>Objetos Select e Option.....</i>	<i>10-15</i>
Eventos	10-20
<i>Validação de formulários.....</i>	<i>10-20</i>
Exercício Resolvido.....	10-20
Solução	10-21
<i>Objeto FileUpload</i>	<i>10-24</i>
Eventos	10-25
<i>Exercícios</i>	<i>10-25</i>

11. Cookies

<i>Cookies em HTTP</i>	11-1
Criação de cookies via cabeçalhos HTTP.....	11-2
Criação de cookies via HTML.....	11-4
Espaço de nomes de um Cookie.....	11-5
Recuperação de cookies.....	11-5
<i>Cookies em JavaScript</i>	11-6
Carrinho de compras	11-8
Exercício Resolvido.....	11-8
Solução	11-10
<i>Exercícios</i>	11-13

12. JavaScript e Java

<i>Applets Java</i>	12-1
Objeto Applet.....	12-4
Controle de Applets via JavaScript.....	12-5
Exercício Resolvido.....	12-6
Solução	12-7
Exercícios.....	12-8
<i>Controle de JavaScript através de Applets</i>	12-9
Exercício Resolvido.....	12-12
Solução	12-12
Conversão de tipos	12-14
<i>Exercícios</i>	12-15

Apêndice A – Bibliografia

Prefácio

A INTERNET NUNCA MAIS FOI A MESMA DESDE QUE TIM BERNERS-LEE propôs em março de 1989, que a gerência do CERN adotasse um sistema de informações distribuído baseado em hipertexto, como solução para os problemas de comunicação da instituição. A CERN – Laboratório Europeu para Física de Partículas – é uma das maiores instituições científicas do mundo e seus laboratórios estão distribuídos por várias cidades localizadas em 19 países da Europa. Berners-Lee demonstrou como a informação se perdia diariamente no CERN, um ambiente que ele classificou como “um modelo em miniatura do resto do mundo em alguns anos”[1]. O sistema proposto, inicialmente chamado de “Mesh”, acabou por convencer seus gerentes e foi implantado no CERN no ano seguinte já com o nome de “World Wide Web”[2].

Berners-Lee estava certo. O CERN era uma miniatura do mundo. Hoje, 10 anos depois, a Internet não é mais a mesma. Hoje a Internet é a World Wide Web. Todos os serviços da Internet se renderam ao poder da Web e à linguagem HTML, que a sustenta. Até o serviço de correio eletrônico, campeão de tráfego na Internet por muitos anos, que por muito tempo exigia aplicações específicas, separadas do browser, hoje é lido dentro de um browser, através de páginas HTML.

A Web evoluiu e ocupou todos os espaços fazendo jus ao nome “World Wide”. Páginas interligadas por hipertexto não são mais novidade. Existem tantas hoje que é difícil separar o joio do trigo, e seria impossível encontrar alguma coisa se a Web não tivesse evoluído e se tornado mais interativa ainda. As páginas deixaram de ser meras páginas e passaram a se comportar como aplicações. O browser evoluiu junto e passou a ser tratado como uma interface universal, capaz de oferecer ao usuário acesso interativo e uniforme a programas remotos em diversas plataformas.

Todas essas mudanças impulsionaram o surgimento de novas tecnologias, pois o HTML era bastante limitado. HTML foi construído apenas para estruturar páginas de hipertexto. Como poderia realizar buscas na Web ou enviar e-mail? Esta necessidade impulsionou pesquisas por organizações abertas e fabricantes de produtos para a Web. Várias propostas surgiram. Algumas propunham até a substituição do HTML por outra linguagem. Poucas idéias, porém, tiveram aceitação tão ampla como a tecnologia CGI que ainda hoje é bastante popular. CGI tornou possível o surgimento das primeiras aplicações Web verdadeiras, permitindo que o cliente manipulasse aplicações remotas usando o seu browser como interface. Isto provocou uma revolução no desenvolvimento de aplicações distribuídas, pois HTML com CGI tornou possível a criação de interfaces baratas, fáceis de desenvolver e fáceis de usar.

Mas as interfaces Web, por dependerem de uma página, estática, não ofereciam a mesma interatividade do lado do cliente. Para fazer uma animação, por exemplo, era preciso fazer sucessivas requisições ao servidor, gerando tráfego de rede desnecessário. Qualquer tecnologia do cliente depende da capacidade do browser suportá-la. Muitos novos recursos foram introduzidos pela

Netscape por ser na época, líder absoluto do mercado de browsers. Inicialmente ofereceu suporte a Java, linguagem da Sun. Depois lançou LiveScript, posteriormente rebatizado de JavaScript. Assim, finalmente a programação de aplicações deixou de ser uma exclusividade do servidor e páginas Web deixaram de ser estáticas. As novas páginas “movidas a JavaScript” passaram a se comportar como componentes de aplicações distribuídas, e são hoje indispensáveis no desenvolvimento de Web sites interativos.

Objetivos

Este livro tem como objetivo apresentar e explorar a linguagem JavaScript – uma das linguagens mais populares do mundo e a mais utilizada na Internet para o desenvolvimento de Web sites interativos. Em 12 capítulos, apresentamos a estrutura e sintaxe da linguagem JavaScript e seus recursos de manipulação da página, formulários, janelas do browser, *frames*, imagens e applets; através de exemplos e exercícios resolvidos, que refletem aplicações práticas como comunicação entre frames, geração de documentos *on-the-fly*, validação de campos de formulários e a criação de carrinhos de compras virtuais.

Desde que foi criada em 1995 por Brendan Eich da Netscape[3], diversas implementações diferentes de JavaScript tem aparecido, na Web e fora dela, em browsers e servidores. Todas as implementações compartilham um núcleo comum (padronizado pela especificação ECMA-262[5]), e acrescentam estruturas específicas ao ambiente onde operam (um browser, um servidor, um sistema de arquivos). O objetivo deste livro é explorar apenas o JavaScript que opera nos browsers, chamado de *client-side* JavaScript.

O client-side JavaScript também não possui uma implementação padrão. Na época em que esta edição foi concluída (janeiro de 1999) havia duas versões recentes de JavaScript: a da Netscape, chamada de JavaScript 1.3, e a da Microsoft, chamada de JScript 5.0[4]. O núcleo das duas é semelhante e obedece ao ECMA-262. A implementação das características client-side é realizada através de um “modelo de objetos” que mapeia “objetos” JavaScript a propriedades do browser e da página HTML. As duas implementações obedecem ao W3C/DOM[6]. Porém, vários aspectos da sintaxe, implementação e extensões presentes nas duas implementações as fazem incompatíveis entre si. Uma das novas tecnologias suportadas por scripts é o Dynamic HTML (DHTML). Desenvolver páginas que usam DHTML hoje é duas vezes mais complexo do que deveria ser, já que é preciso levar em conta as diferenças do JavaScript de cada browser.

Este livro não abordará as versões mais recentes do JavaScript, nem o DHTML. Optamos por usar como base o JavaScript 1.1, que é a implementação mais estável, suportada pelo maior número de browsers. JavaScript 1.1 foi introduzida com a versão 3.0 do Netscape Navigator. É uma versão pequena, simples, útil e totalmente compatível com as novas versões da Netscape e da Microsoft. Foi base para o padrão ECMA-262. Neste livro, usamos JavaScript 1.1 como *referência*, abordando também alguns recursos que existem no Internet Explorer, mas deixando de fora recursos menos usados que existem somente nos browsers de um único fabricante, seja Netscape ou Microsoft. Cobrimos, assim, os aspectos fundamentais de programação do cliente, de forma independente de browser. Se no futuro você decidir usar DHTML e os novos recursos dos browsers, descobrirá que já conhece os fundamentos da linguagem, que são os mesmos do JavaScript apresentado aqui.

O que você já deve saber

Antes de aprender JavaScript, você já deve saber criar páginas Web com HTML. Muitas pessoas saber criar páginas Web mas nunca viram a “cara” do HTML, pois as mais sofisticadas aplicações de desenvolvimento Web escondem o código por trás das páginas. Para aprender JavaScript, porém, saber criar páginas dessa forma não basta. É preciso conhecer a *estrutura do código* que está por trás de sua aparência e saber criar parágrafos, listas, tabelas, formulários, *frames*, incluir links e imagens em uma página utilizando o código HTML. Este assunto não será abordado neste livro. Existem vários bons tutoriais e livros sobre o assunto, inclusive na Web. Alguns estão listados no apêndice A.

Conhecimento prévio de uma linguagem de programação é desejável, mas não essencial. É possível explorar JavaScript aos poucos, começando com recursos mais básicos e com aplicação imediata e ir avançando gradualmente até chegar em aplicações mais complexas. Mesmo que você nunca tenha programado em uma linguagem estruturada antes, acreditamos que será possível acompanhar todos os exemplos deste livro e no final saber desenvolver aplicações de média complexidade com JavaScript.

Descrição do conteúdo

O livro está organizado em 12 capítulos e pelo menos um apêndice¹ contendo as fontes de informação consultadas e recursos na Web. O código-fonte de quase todos os exemplos, exercícios propostos, soluções e exercícios resolvidos está em um disquete que o acompanha.

Ao final de cada capítulo, ou de seções de um capítulo, há uma lista de exercícios propostos, que aplicam os assuntos apresentados. Ao todo são mais de 40 exercícios propostos, a maior parte com solução em disquete. Muitos fornecem um esqueleto que o programador pode usar como base, para se concentrar apenas nos aspectos relevantes ao problema.

Além dos exercícios propostos, vários recursos do JavaScript são apresentados através de exemplos detalhados, na forma de exercícios resolvidos. Nestes exercícios, um problema proposto é solucionado expondo as etapas da resolução, decisões tomadas e o código utilizado, com comentários. São 11 exercícios resolvidos ao todo.

O primeiro capítulo, “Introdução a JavaScript”, tem como objetivo apresentar uma breve introdução e visão geral da linguagem e sua utilização no browser. Depois de apresentados alguns exemplos demonstrando pequenas aplicações e manuseio de eventos, um exercício completo é proposto, e resolvido em seguida, com o objetivo de familiarizar o programador com o código JavaScript e o modelo de objetos do browser.

Os capítulos 2 a 4 tratam do núcleo comum da linguagem JavaScript, assim como é definida na especificação ECMA-262, JavaScript 1.1 e JScript 3.1. O capítulo 2 apresenta a sintaxe e estruturas elementares da linguagem, o capítulo 3 introduz os conceitos de objetos, protótipos, funções, métodos e propriedades e o capítulo 4 apresenta os objetos nativos do JavaScript.

¹ Como este livro é utilizado como apostila em treinamentos abertos e fechados, o seu formato pode mudar de acordo com a carga horária e necessidades do contratante do treinamento. O formato de 12 capítulos e 1 apêndice refere-se à versão básica (B).

Os capítulos 5 a 12 tratam do *client-side* JavaScript e cobrem em detalhes o modelo de objetos do browser que os browsers Netscape dividem em duas hierarquias: *Window*, explorada no capítulo 5, e *Navigator*, explorada no capítulo 6. Nos browsers Microsoft só há uma hierarquia que inicia em *Window*. Sua propriedade *navigator* também é abordada no capítulo 6.

O capítulo 7 trata de objetos que controlam a navegação nas janelas do browser: *History*, que representa o histórico da janela, e *Location*, que representa a URL da janela. Aborda também a representação de vínculos (*links*) em um documento.

O capítulo 8 explora o objeto *Document*, que representa a página ou documento HTML. Os capítulos que seguem mostram como usar componentes da página como imagens (capítulo 9), formulários (capítulo 10), cookies (capítulo 11) e applets (capítulo 12).

A ordem dos capítulos não é rigorosa. A maior parte dos capítulos depende de informações que estão nos capítulos 1-5. Os capítulos 9-12 dependem também de informações que estão na primeira parte do capítulo 8. Portanto, os capítulos 6, 7 e 9 a 12 podem ser abordados em qualquer ordem, depois dos capítulos 1 a 5.

Vários capítulos e seções de capítulos tratam de assuntos pouco usados ou de uso restrito (suportado por um ou outro fabricante de browser) e podem ser eliminados. Se você não pretende manipular com cookies, applets ou imagens, por exemplo, pode pular os capítulos correspondentes.

Cursos baseados neste livro

Este livro é utilizado como apostila em cursos de JavaScript com carga-horária que pode variar de 16 a 40 horas. Em cursos práticos de carga horária inferior a 24 horas, vários tópicos contidos nesta apostila podem não ser abordados. Embora todos os tópicos possam ser apresentados em 16 horas, sobra muito pouco tempo para atividades práticas em cursos que utilizam laboratório.

Em situações onde a carga horária é insuficiente, capítulos e partes de capítulos que tratam de assuntos usados menos freqüentemente podem ser eliminados (veja seção anterior). Os capítulos 11 e 12, por exemplo, tratam de tópicos avançados que podem ser deixados de fora em um curso introdutório. Partes dos capítulos 6 e 8 e detalhes do capítulo 3 podem ser omitidos sem prejudicar o restante do curso.

Em todos os casos, é o *programa do curso*, e não o índice de tópicos deste livro, a referência absoluta sobre o programa a ser cumprido.

Mudanças em relação a edições anteriores

A primeira versão deste livro foi produzida em agosto de 1997. Desde então, temos acompanhado a evolução da linguagem a cada novo *release* dos browsers da Netscape e da Microsoft, tentando encontrar formas de tirar o melhor proveito das semelhanças e diferenças de cada implementação para que tivessem imediata aplicação prática. Na época, não havia ainda uma especificação formal da linguagem o que tornava difícil a tarefa de definir a estrutura da linguagem de forma consistente e ao mesmo tempo independente de browser. Hoje, as duas implementações já tentam se adequar a um padrão, definido na especificação ECMA-262. Portanto, procuramos nesta edição, definir as estruturas elementares (núcleo comum) da linguagem de acordo essa especificação.

ECMA-262 limita-se ao núcleo da linguagem, e não abrange o modelo de objetos do HTML e do browser. Mantivemos a hierarquia e nomes definidos no JavaScript da Netscape², mas utilizamos uma notação “estilo ECMA” para distinguir “tipo de um objeto” de um “objeto” (o que nem sempre ocorre na documentação da Netscape). Fizemos isto representando sempre que possível, *o tipo* em itálico com a primeira letra maiúscula (por exemplo, usamos *Window*, neste livro, para representar o *tipo de objeto* que representa janelas e frames, que são utilizados em *client-side* JavaScript através de nomes como `window`, `window.frames[0]`, `frames[0]` e `parent`).

Quatro novos capítulos nesta edição são resultantes da divisão de capítulos muito grandes existentes na versão anterior. A nova organização facilita o estudo dos assuntos e permite que a ordem dos capítulos possa ser alterada e certos capítulos possam ser eliminados em cursos com carga horária reduzida. Há um novo capítulo sobre comunicação Java com JavaScript (capítulo 12) que nas versões anteriores era um apêndice opcional. Vários exercícios propostos foram transformados em exercícios resolvidos.

Mídia eletrônica e atualizações

Todos os exemplos, exercícios resolvidos e soluções de alguns exercícios propostos estão em um disquete que acompanha este livro. Os nomes dos diretórios refletem os nomes dos capítulos, por exemplo, os arquivos do capítulo 3 podem ser encontrados no subdiretório `cap3/`.

Uma nova edição deste livro, no formato apostila (A4) é produzida a cada 6 ou 12 meses. Quaisquer atualizações neste intervalo podem estar presentes na forma de anexos, distribuídos separadamente ou não. Erratas e atualizações menores são geralmente introduzidas a cada nova reprodução.

A partir desta edição, um website estará disponível com recursos, atualizações e formulário para feedback. Até a data de conclusão deste prefácio, porém, o endereço ainda não estava disponível. Procure-o na página *ii* (*copyright*).

Convenções usadas no texto

As seguintes convenções tipográficas são utilizadas neste livro:

- *Garamond Italic* é usada para tipos de dados e tipos de objetos JavaScript (*Window*, *Frame*, *String*) e texto grifado.
- *Courier New* é usada para representar código JavaScript (`eval("x+y")`, `window.status`), descritores e atributos HTML (`<HEAD>`, `SRC`, `HREF`), URLs, nomes de arquivo e nomes de programas (`index.html`, `http://www.abc.com`).
- *Courier New Italic* é usada para representar propriedades e atributos que representam um valor definido pelo programador (`parseInt(string_com_numero, base)`).
- **Courier New Bold** é usada para trechos de código destacados por algum motivo, linhas de comando que devem ser digitadas verbatim na tela (`C:> dir *.html`).

² O DOM (Document Object Model) do W3C, padroniza o JavaScript do lado do cliente, mas é excessivamente extenso para os nossos objetivos (que não incluem DHTML).

Agradecimentos

Este livro começou após um curso que eu ministrei na IBPINET em São Paulo sobre Web Sites Interativos. Inicialmente, era apenas pouco mais que um guia de referência que eu compilei para uso próprio, com exemplos extraídos da documentação da Netscape. Após o curso, com o *feedback* dos alunos, resolvi reorganizar o assunto em um formato mais didático, com alguns exemplos novos, resultantes de questões surgidas em sala de aula. Isto se repetiu várias vezes até chegar à forma atual. Este livro, portanto, existe graças aos alunos dos cursos realizados na IBPINET e Itelcon, que interagiram de várias formas, revisando seu conteúdo, sugerindo mudanças na apresentação do assunto, apontando erros e propondo exemplos práticos.

Pela oportunidade de poder ministrar os cursos que deram forma a este livro, gostaria de agradecer também à Fábio Marinho e Adriana Guerra, diretores do IBPINET, e a Joberto Martins e William Giozza, diretores da Itelcon.

Críticas e sugestões

Este livro está sempre sendo revisado, atualizado e ampliado periodicamente e cada vez que é utilizado em um curso. Cuidados foram tomados para garantir a apresentação dos assuntos de forma clara, didática e precisa, mas eventualmente podem escapar erros, imprecisões e trechos obscuros. Sugestões, críticas e correções são sempre bem vindas e podem ser endereçadas por e-mail a hlsr@uol.com.br ou helder@ibpinetsp.com.br. Sua opinião é muito importante e contribuirá para que futuras edições deste livro e outros livros e apostilas possam ser ainda melhores.

Helder L. S. da Rocha

Campina Grande, PB, 23 de fevereiro de 1999.

1

Introdução a JavaScript

A LINGUAGEM HTML – HYPERTEXT MARKUP LANGUAGE, foi criada *exclusivamente* para definir a estrutura de uma página. Esforços para usar HTML como linguagem de formatação de página, visando uma melhor *apresentação gráfica* resultaram ineficazes¹. De forma semelhante, HTML não é linguagem de *programação*. Não possui as estruturas essenciais para realizar operações e controle de fluxo. É uma linguagem declarativa criada para *estruturar* páginas de hipertexto através de *marcadores* que descrevem a *função* de blocos de texto.

Com HTML, é fácil criar interfaces do usuário sofisticadas, usando recursos de formulário como botões, caixas de seleção, etc. A coleção de componentes de formulário conta com dois tipos de botões que respondem a eventos do usuário. Um dos botões, ao ser apertado, provoca um evento permite enviar os dados coletados no formulário para um programa no servidor (CGI²) para processamento remoto. Não há processamento local.

Os componentes de formulário existem desde HTML 1.1 (1994) e com eles surgiaram as primeiras “aplicações Web”. Essas aplicações sempre tinham que se comunicar com o servidor para realizar qualquer operação. Para fazer uma simples conta era necessário enviar os dados para o servidor, rodar um programa na máquina remota e aguardar uma nova página retornada com o resultado. Isso era necessário porque não havia como fazer contas simples em HTML.

Plug-ins proprietários foram os primeiros a introduzir aplicações Web executando do lado do cliente. Depois vieram os applets Java, que tiveram mais sucesso que os *plug-ins* por não se limitarem a uma única plataforma. Os *plug-ins* e os applets usam o HTML apenas como base para aparecerem no browser. Utilizam uma interface própria que ocupa uma subjanela, toda a janela ou parte de uma janela do browser. Não aproveitam os recursos do HTML. É preciso desenvolver sua interface usando uma outra linguagem, o que torna o desenvolvimento bem mais complexo que a criação de formulários HTML.

¹ Várias versões do HTML continham descritores de apresentação. Eles foram considerados obsoletos pela última recomendação do W3C: HTML 4. A linguagem CSS (folhas de estilo em cascata) é a atual recomendação do W3C para a formatação de páginas HTML.

² Common Gateway Interface – especificação que define o formato de programas cuja execução é iniciada por um servidor Web.

A primeira tecnologia proposta como extensão verdadeira do HTML foi JavaScript. Como é extensão, faz parte da *página* e pode interagir com todos os seus componentes, como formulários e imagens, inclusive com os *plug-ins* e applets. É a melhor solução para realizar tarefas simples que não compensam o esforço de desenvolver um applet ou *plug-in*. Em geral, apresenta um desempenho melhor que esses componentes de browser. Hoje, com mais recursos e mais estável, JavaScript tem ocupado lugares antes ocupados por applets e *plug-ins* em vários serviços on-line.

O que é JavaScript?

JAVASCRIPT É UMA LINGUAGEM de programação interpretada criada em 1995 por Brendan Eich da Netscape como uma *extensão* do HTML para o browser Navigator 2.0. Hoje existem implementações JavaScript nos browsers dos principais fabricantes. Mas o uso de JavaScript não tem se limitado aos browsers. Tem sido usado, em menor escala, como linguagem de suporte a tecnologias de *gateway* para servidores HTTP e até como linguagem de roteiro de propósito geral. Embora ainda seja mantida e estendida pela Netscape, parte da linguagem JavaScript já é padrão proposto pela ECMA – organização européia para padrões em comunicações, que visa transformá-la em padrão Web³.

JavaScript do lado do browser (*client-side* JavaScript) tem evoluído e alcançado uma estabilidade razoável como um padrão da Web. É hoje (1998), suportada pelas principais versões de browser que povoam a Web e é a linguagem de programação mais popular do mundo, com presença em 35 milhões de páginas Web⁴.

JavaScript no servidor (*server-side* JavaScript) é uma linguagem que possui o mesmo núcleo que o JavaScript do lado do cliente, mas acrescenta estruturas exclusivas para interação com entidades do servidor. Não tem ainda a estabilidade necessária para ser considerada um padrão pois suas implementações são praticamente restritas à extensões Netscape, como a tecnologia LiveWire. O núcleo da linguagem JavaScript também está presente na tecnologia ASP (Active Server Pages) da Microsoft, mas LiveWire e ASP são incompatíveis entre si.

Este curso trata exclusivamente do *client-side* JavaScript, que roda no browser. No restante deste livro, chamaremos *client-side* JavaScript simplesmente de “JavaScript”.

JavaScript é uma linguagem de programação *baseada* em objetos. Trata suas estruturas básicas, propriedades do browser e os elementos de uma página HTML como *objetos* (entidades com propriedades e comportamentos) e permite que sejam manipulados através de eventos do usuário programáveis, operadores e expressões. JavaScript oferece recursos interativos que faltam no HTML e permite a criação de páginas interativas e dinâmicas, que são interpretadas localmente pelo browser, sem precisar recorrer a execução remota de programas no servidor.

³ <http://www.ecma.ch/stand/ecma-262.htm>[5]

⁴ <http://home.netscape.com/newsref/pr/newsrelease599.html>

JavaScript não é Java

JavaScript freqüentemente é confundida com a linguagem Java, provavelmente devido à semelhança do nome. Há também algumas semelhanças na sintaxe. Tudo mais é diferente. O nome “script”, que quer dizer roteiro, já indica que se trata de uma linguagem interpretada. Além do nome, podemos apontar diversas diferenças:

- *Interpretada.* Programas em Java são compilados para um código de máquina, que é executado em uma plataforma própria (que pode ser fornecida pelo browser). Programas em JavaScript são interpretados linha-por-linha enquanto o browser carrega a página ou executa uma rotina.
- *Simples.* Programas em Java são bem mais poderosos que programas JavaScript e não estão limitados à página HTML. Por outro lado, são bem mais complexos.
- *Pequena.* JavaScript 1.1, abordado neste livro, consiste de umas 300 funções, objetos, métodos, eventos e propriedades. A API do Java 2 possui mais de 20000 estruturas.
- *Baseada em objetos.* O modelo de objetos e as estruturas das duas linguagens são completamente diferentes. Java é uma linguagem *orientada a objetos* que possui estruturas como classes, herança, polimorfismo, etc. que não existem em JavaScript.
- *Extensão do HTML.* Nunca se coloca Java em uma página Web. Pode-se incluir uma *applet* em uma página, que é um tipo de aplicação que pode ter sido escrito em Java, ou não. O browser freqüentemente tem capacidade de executar um applet, mas não de interpretar o seu código Java. O código JavaScript geralmente vem embutido dentro de uma página HTML. Não existe JavaScript (client-side) sem HTML.

Quem suporta JavaScript?

Somente os browsers compatíveis com a linguagem JavaScript conseguem executar os roteiros (scripts). Entre os mais populares, isto inclui o Netscape Navigator versões 2 em diante, o Microsoft Internet Explorer versões 3 em diante e o OperaSoftware Opera 3.5 em diante. O JavaScript suportado por um browser pode não funcionar em outro. Os principais motivos são incompatibilidades entre versões e plataformas.

A primeira versão de JavaScript, chamada de JavaScript 1.0, foi primeiro suportada pelo Navigator 2.0. A Microsoft desenvolveu sua primeira implementação do JavaScript 1.0 no Internet Explorer 3.0, onde se chamava JScript. JavaScript 1.0 tinha muitos bugs e poucos recursos. Com o Navigator 3.0 foi lançado o JavaScript 1.1, que teve uma implementação semelhante em versões do Internet Explorer 3 e 4. A versão 1.2 do JavaScript, introduzida com o Netscape 4.05 e suportada em parte pelo Internet Explorer 4, acrescenta alguns recursos como expressões regulares e suporte a camadas.

As implementações JavaScript em browsers de fabricantes diferentes são conflitantes. O uso de recursos exclusivos de um fabricante provocará erros quando a página for carregada por outro browser. Há várias formas de usar o próprio JavaScript para atenuar esse problema.

Para garantir uma maior segurança, todos os scripts devem sempre ser testados nos os browsers, versões e *plataformas* utilizadas pelo público-alvo de um site ou página.

O que se pode fazer com JavaScript?

Com JavaScript pode-se fazer diversas coisas que antes não era possível apenas com a limitada linguagem HTML como:

- Realizar operações matemáticas e computação.
- Gerar documentos com aparência definida na hora da visualização, com base em informações do cliente como versões do browser, *cookies* e outras propriedades.
- Abrir janelas do browser, trocar informações entre janelas, manipular com propriedades do browser como o histórico, barra de estado, plug-ins e applets.
- Interagir com o conteúdo do documento, alterando propriedades da página, dos elementos HTML e tratando toda a página como uma estrutura de objetos.
- Interagir com o usuário através do tratamento de eventos.

Como programar com JavaScript?

Nas seções seguintes, daremos início à apresentação da linguagem JavaScript. Para editar código HTML ou JavaScript, não é preciso mais que um simples editor de texto, como o Bloco de Notas (Windows) ou Vi (Unix). Pode-se também usar um editor HTML. Alguns editores colocam cores ou dão destaque ao código JavaScript. Outros até permitem a geração de código ou a verificação de sintaxe.

O editor HTML pode ser qualquer um, mas deve *expor o código HTML*. Editores que escondem e dificultam o acesso ao código HTML devem ser evitados. É preciso que o editor tenha pelo menos uma janela de edição de código. O ideal é usar um *editor de código* como o Allaire HomeSite, Sausage HotDog (para Windows), HotMetal (para Unix, Mac e Windows) e BBEdit (para Mac). Este livro é compatível com qualquer editor de texto ou de código.

Existem ferramentas que facilitam o desenvolvimento de JavaScript, porém elas não serão exploradas aqui. As mais conhecidas são a Microsoft Script Debugger, que funciona embutido no Microsoft Internet Explorer (é uma extensão com distribuição separada) e o Netscape Visual JavaScript. Ambos os produtos podem ser descarregados dos sites de seus respectivos fabricantes.

Os arquivos utilizados neste capítulo estão no subdiretório **cap1/**, do disquete distribuído com este livro. Para acompanhar e repetir os exemplos das seções seguintes, abra um arquivo HTML qualquer (ou use a página em branco **cap1/intro.html** disponível no disquete) e repita os exemplos, testando-os no seu browser. Qualquer editor de código ou de texto pode ser usado.

Formas de usar JavaScript

Há três⁵ maneiras de incluir JavaScript em uma página Web:

- *Dentro de blocos HTML <SCRIPT> ... </SCRIPT> em várias partes da página:* para definir funções usadas pela página, gerar HTML em novas páginas ou alterar o procedimento normal de interpretação do HTML da página pelo browser.
- *Em um arquivo externo, importado pela página:* para definir funções que serão usadas por várias páginas de um site.
- *Dentro de descritores HTML sensíveis a eventos:* para tratar eventos do usuário em links, botões e componentes de entrada de dados, durante a exibição da página.

As três formas podem ser usadas em uma mesma página.

Blocos <SCRIPT> embutidos na página

A forma mais prática de usar JavaScript é embutindo o código na página dentro de um bloco delimitado pelos descritores HTML <SCRIPT> e </SCRIPT>. Pode haver vários blocos <SCRIPT> em qualquer lugar da página.

```
<script>
    ...
</script>
```

O descritor <SCRIPT> possui um atributo LANGUAGE que informa o tipo e versão da linguagem utilizada. O atributo LANGUAGE é necessário para incluir blocos em outras linguagens como VBScript. É opcional para JavaScript:

```
<SCRIPT LANGUAGE="VBScript"> ... código em VBScript ... </SCRIPT>
<SCRIPT LANGUAGE="JavaScript"> ... código JavaScript ... </SCRIPT>
<SCRIPT> ... código JavaScript ... </SCRIPT>
```

A versão 1.1 de JavaScript possui estruturas inexistentes nas versões anteriores. Um browser Netscape 2.0 ou Internet Explorer 3.0 que tentar interpretar o código entre <script> e </script> pode provocar erros. O atributo LANGUAGE com o valor “JavaScript1.1” pode ser usado para identificar um bloco que só será usado por browsers que suportem JavaScript 1.1:

```
<BODY> <p>Última modificação:
<script language="JavaScript1.1"> <!--
    autor = "Cyrano de Bergerac";
    document.write("<b>" + document.lastModified + "</b>");
    document.write("<p>Autor: " + autor);
```

⁵ A especificação da Netscape permite ainda incluir JavaScript dentro de qualquer atributo HTML para passar valores ou expressões e alterar características da página. É um recurso disponível apenas nos browsers Netscape.

```
// --> </script>
</BODY>
```

Tudo o que está em negrito, na listagem acima, é JavaScript. O que não está em negrito é código HTML.

O código JavaScript foi colocado entre *comentários HTML* `<!--` e `-->`. Isto é usado para proteger contra browsers antigos, que não suportam JavaScript, e podem exibir o código na página em vez de executá-lo ou ignorá-lo. Browsers que suportam JavaScript ignoram os comentários HTML dentro de blocos `<SCRIPT>` e tentam interpretar o código. Browsers que suportam uma versão inferior a JavaScript 1.1 irão ignorar todo o bloco.

No código acima, **autor** é uma *variável* que recebe por atribuição o texto “*Cyrano de Bergerac*”; **document** é um *objeto*⁶ JavaScript que representa a página da janela atual do browser. **lastModified** é uma *propriedade*⁶ da página (texto contendo a data de última modificação do arquivo) e **write()** é um *método*⁶ que escreve o texto passado como parâmetro na página representada por **document**.

O ponto `(.)` é usado para que se possa ter acesso a propriedades e métodos de um objeto. O sinal `“+”` é usado para concatenar caracteres e *strings*. As duas barras `(//)` representam um comentário JavaScript.

Ao se carregar a página HTML contendo o código acima em um browser, obtém-se uma página com a informação:

Última modificação: **Quarta-feira, 2 de abril de 1998 13:11:47 –0300**

Autor: Cyrano de Bergerac

Arquivos importados

Muitas vezes é necessário realizar um mesmo tipo de tarefa mais de uma vez. Para esse tipo de problema JavaScript permite que o programador crie *funções* que podem ser chamadas de outras partes da página várias vezes. As funções geralmente ficam em um bloco `<SCRIPT>` separado, antes de todos os outros blocos, para que sejam carregadas antes que a página seja exibida. Se várias páginas usam as mesmas *funções* JavaScript definidas pelo autor da página, uma boa opção é colocá-las em um arquivo externo e importá-lo nas páginas que precisarem delas. Este arquivo deve ter a extensão “`.js`” e conter apenas código JavaScript (não deve ter descritores HTML, como `<SCRIPT>`). Por exemplo, suponha que o arquivo `biblio.js` possua o seguinte código JavaScript⁷:

⁶ O significado desses termos ficará mais claro nas seções posteriores.

⁷ As palavras `return` e `function` são reservadas em JavaScript. No exemplo acima definem a função `soma(a, b)` que retorna a soma de números que recebe como parâmetro.

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

Para carregar esta função e permitir que seja usada em outra página, usa-se o atributo **SRC** do descritor <SCRIPT>:

```
<script LANGUAGE=JavaScript SRC="biblio.js"></script>
(...)

<script>
    resultado = soma(5, 6);
    document.write("<P>A soma de 5 e 6 é " + resultado);
</script>
```

É preciso que o servidor Web esteja configurado para relacionar a extensão .js como o tipo MIME application/x-javascript para que a carga, de um arquivo externo seja possível.

Tratamento de eventos

A linguagem JavaScript introduziu no HTML como extensão 13 novos atributos⁸, que permitem a captura de eventos disparados pelo usuário, como o arrasto de um mouse, o clique de um botão, etc. Quando ocorre um evento, um procedimento de manuseio do evento é chamado. O que cada procedimento irá fazer pode ser determinado pelo programador.

A linguagem HTML já possui três eventos *nativos* não programáveis, que são:

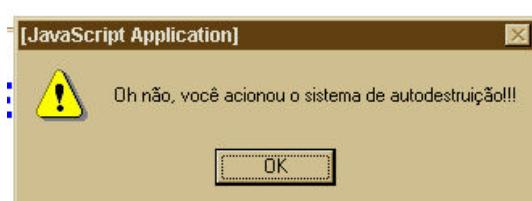
- clique sobre um **link** ou imagem mapeada
- clique em um botão **submit** (para envio de formulário ao CGI)
- clique sobre um botão **reset** (que limpa o formulário)

Em JavaScript 1.1, há 13 eventos adicionais programáveis através de atributos HTML especialmente criados para manuseio de eventos. No caso de conflito, eles têm precedência sobre os eventos do HTML. Os atributos de eventos se aplicam a elementos HTML específicos e sempre começam com o prefixo “ON”. Os valores recebidos por esses atributos é código JavaScript. Por exemplo:

```
<FORM>
<INPUT TYPE="button"
       ONCLICK="alert('Oh não, você acionou o sistema de
autodestruição!')"
       VALUE="Não aperte este botão">
</FORM>
```

mostra um trecho de código HTML que fará aparecer um botão na tela (figura).

Não aperte este botão



⁸ Refere-se ao JavaScript1.1/JScript 3.0

Tudo o que aparece entre as aspas duplas do atributo **ONCLICK** é JavaScript. **ONCLICK** é um atributo HTML, criado como extensão para dar suporte ao evento de clicar o botão.

O código JavaScript que está em negrito será interpretado quando o usuário apertar o botão com o mouse (**onclick**). A instrução **alert()** cria uma janela de alerta (acima) com a mensagem passada como parâmetro (entre parênteses e aspas no código em negrito). Observe que as aspas usadas dentro do método **alert()** são aspas simples já que aspas duplas já estão sendo usadas para representar o atributo HTML.

Código JavaScript também pode ser acionado através de eventos nativos do HTML, como links e botões de submissão de formulários usando uma URL “**javascript:**”:

```
<a href="javascript:alert('Tem Certeza?')"> link </a>
```

O código acima fará com que o evento HTML (clicar no link) provoque a execução do código JavaScript após o *prompt* “**javascript:**”. Este *prompt* também pode ser usado na barra de *Location* do browser. Oferece acesso direto ao interpretador.

Nem todos os elementos HTML suportam atributos de eventos. Também nem todas as operações JavaScript que são possíveis em blocos, como escrever na página, são possíveis após a carga completa da página, se acionados por um evento.

Os treze procedimentos de manuseio de eventos introduzidos por JavaScript são:

Atributo HTML	Quando o procedimento é executado	Descritores HTML onde é suportado
onclick	Quando um objeto é clicado pelo mouse	<a>, <input>
onselect	Quando um objeto é selecionado	<input type=text>, <textarea>
onchange	Quando uma seleção ou campo de texto tem seu conteúdo modificado	<input type=text>, <textarea>, <select>
onfocus	Quando um componente de formulário ou janela se torna ativa	<textarea>, <body>, <form>, <input>, <select>, <option>
onblur	Quando um componente de formulário ou janela se torna inativa	<textarea>, <body>, <form>, <input>, <select>, <option>
onmouseover	Quando o mouse está sobre um link	<a>, <area>
onmouseout	Quando o mouse deixa um link	<a>, <area>
onsubmit	Antes de enviar um formulário	<input type=submit>
onreset	Antes de limpar um formulário	<form>
onload	Após carregar uma página, janela ou frame	<body>
onunload	Ao deixar uma página, janela ou frame	<body>
onerror	Quando um erro ocorre durante a carga de uma imagem ou página	, <body>
onabort	Quando a carga de uma imagem é abortada	

Como procedimentos de eventos são atributos do HTML (e não do JavaScript), tanto faz escrevê-los em letras maiúsculas ou minúsculas. Usar `onclick`, `ONCLICK` ou `onClick` não faz diferença. Já o texto *dentro das aspas* é JavaScript, que é uma linguagem que diferencia letras maiúsculas de minúsculas, portanto `alert` não é a mesma coisa que `ALERT`.

Introdução prática

O objetivo desta seção é apresentar uma rápida introdução à linguagem JavaScript. Não são explorados assuntos relacionados à sintaxe da linguagem. O objetivo é dar uma visão geral da linguagem e facilitar a absorção das informações apresentadas nos capítulos posteriores.

A melhor forma de introduzir a linguagem é através de um exemplo. No exemplo a seguir, teremos contato com vários tópicos que veremos em detalhes nos capítulos a seguir, como: sintaxe de expressões, variáveis, objetos, métodos e propriedades, funções e eventos.

Exercício resolvido

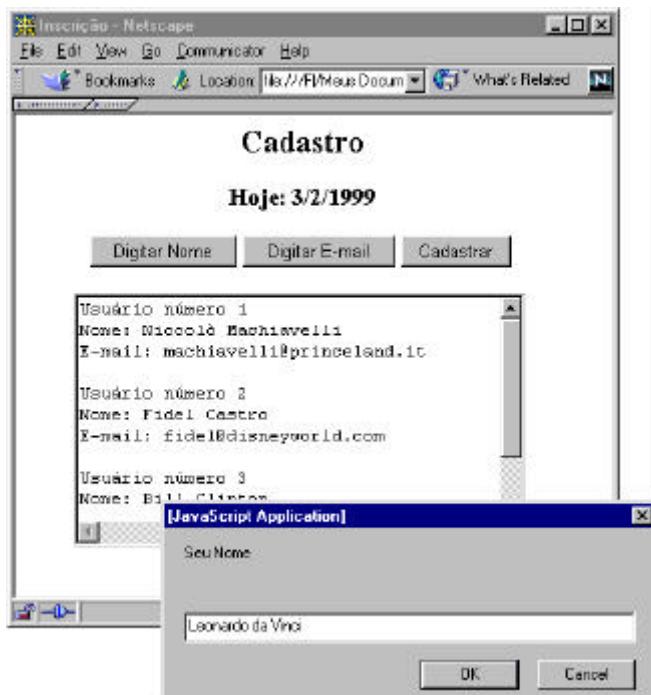
Construa uma aplicação Web de entrada de dados que ofereça uma interface semelhante à da figura ao lado. O objetivo é construir uma lista de usuários com seus e-mails para posterior envio pela rede.

Requisitos:

- Quando o usuário apertar no botão “Digitar Nome”, deve aparecer uma janela de diálogo como mostrada na figura para que ele possa digitar um nome. Apertando no botão “Digitar E-mail”, uma janela semelhante deverá aparecer, para recuperar o e-mail.
- Apertando o botão “Cadastrar”, os dados digitados mais recentemente devem ser “armazenados” no campo de texto no formato:

```
Usuário número <número>
Nome: <nome do usuário>
E-mail: <e-mail do usuário>
```

O número deve ser incrementado cada vez que um novo usuário for cadastrado. Cada novo usuário cadastrado não deve apagar os anteriores, mas aumentar a lista.



- c) A data de hoje deve ser mostrada na página no formato ilustrado na figura.

Não é preciso construir o HTML. Use o arquivo `cap1/formcad.html` que já contém um esqueleto da aplicação (somente os descritores HTML). A solução será apresentada na página seguinte.

Solução

O arquivo sem JavaScript está listado abaixo. Os botões estão presentes mas não respondem a eventos. Não aparece a data de hoje.

```
<html>
<head>
    <title>Inscrição</title>
</head>
<body bgcolor=white>
<form>
    <h2 align=center>Cadastro</h2>
    <div align=center>
        <p><input type=button value="Digitar Nome">
        <input type=button value="Digitar E-mail">
        <input type=button value="Cadastrar">
        <p><textarea rows=10 cols=40 name=Area></textarea>
    </div>
</form>
</body>
</html>
```

A primeira alteração, para cumprir o requisito (a), consiste na programação dos eventos `ONCLICK` dos dois primeiros botões. É preciso coletar uma linha de texto do usuário e armazená-la em uma variável global. Para *declarar* uma variável globalmente acessível em JavaScript, usamos a palavra-chave `var` antes do nome escolhido. As declarações devem estar em um bloco `<SCRIPT>` que seja interpretado antes que um botão seja interpretado, então as colocamos no `<head>`:

```
<head>
    <title>Inscrição</title>
    <script>
        var nome
        var email
    </script>
</head>
```

As alterações estão mostradas em negrito.

O próximo passo é programar o evento. A instrução

```
prompt("texto da janela", "texto inicial do campo de entrada")
```

é um *método* JavaScript que abre uma janela de diálogo contendo um campo de entrada de dados (como o da figura). O usuário pode digitar texto no campo disponível e este será devolvido como valor de retorno, caso o OK seja apertado. Para colocar o valor na variável nome, podemos fazer:

```
nome = prompt("Digite o Nome", "")
```

Fazemos isto dentro do atributo **ONCLICK** de cada botão, para os dois valores, para que o comando só seja executado quando o usuário apertar o botão:

```
<input type=button value="Digitar Nome"
       onclick="nome=prompt('Seu Nome', '')">
<input type=button value="Digitar E-mail"
       onclick="email=prompt('Email', '')">
```

O segundo requisito requer instruções para o atributo **ONCLICK** do *terceiro* botão. Mas é necessário realizar diversas operações:

- incrementar um número (outra variável global)
- construir uma linha de texto (*string*) com os dados lidos
- imprimir os dados em um campo de textos

O ideal, neste caso, é criar uma função que realize as operações acima e chamar esta função a partir do atributo **ONCLICK** do terceiro botão. Acrescentamos a função no bloco **<SCRIPT>** do início da página e construímos o string concatenando as variáveis:

```
<script>
    var nome;
    var email;
    var num = 0;

    function escrever() {
        info = "Usuário número " + (++num) + "\n";
        info += "Nome: " + nome + "\n";
        info += "E-mail: " + email + "\n\n";
    }
</script>
```

O “+”, como vimos antes, concatena strings. O valor da variável num é incrementado com “**++num**”, que é equivalente à expressão “**num = num + 1**”. A atribuição “**+=**” *acrescenta* o texto do lado direito à variável **info**, sem apagar o texto existente. “\n” representa uma quebra de linha. São armazenadas em **info** quatro linhas de informação, sendo a última em branco.

Falta ainda imprimir os resultados no campo de textos. Alteramos a função **escrever()** para que receba, como argumento, uma referência ao campo de textos (*quadro*), que será passado na chamada da função. Todo campo de textos **<TEXTAREA>** ou **<INPUT TYPE=TEXT>** em JavaScript tem uma propriedade **value**, que dá acesso à sua área editável. A

propriedade `value` é uma variável que pode receber ou conter texto. Dentro da função, concatenamos o texto (em `info`) com o texto já existente na caixa de texto (e visível na tela) em `value`:

```
function escrever(quadro) {
    info = "Usuário número " + (++num) + "\n";
    info += "Nome: " + nome + "\n";
    info += "E-mail: " + email + "\n\n";
    quadro.value += info;
}
```

Uma referência para o campo de textos (<TEXTAREA>) pode ser obtido a partir do formulário no qual está contido, através de seu nome. Os nomes dos componentes de um formulário são propriedades do formulário. *Dentro* de qualquer componente, pode-se obter uma referência ao formulário em que está contido usando `this.form`. A expressão:

```
this.form.Area
```

portanto, representa o campo de textos chamado `Area` (<TEXTAREA NAME="`Area`">), localizado *neste* formulário. A chamada acima é válida dentro de qualquer componente do formulário, como o terceiro botão. Assim, podemos chamar a função `escrever()` de dentro do botão e passar como argumento o campo de textos, que é representado pelo objeto de nome `Area` localizado neste formulário::

```
<input type=button value="Cadastrar"
       onclick="escrever(this.form.Area)">
```

Agora, ao se apertar o botão, a função `escrever` será chamada. Dentro dela, a variável `quadro` receberá o valor em `this.form.Area`, como se tivesse ocorrido uma atribuição do tipo:

```
quadro = this.form.Area
```

O último requisito pede para que a página exiba a data de hoje na página. A exibição não depende de eventos do usuário. Deve ser uma transformação realizada somente na carga e exibição da página, portanto, incluímos o código em um segundo bloco <SCRIPT> no corpo da página.

Utilizamos a instrução `new Date()` para obter a data de hoje e passamos para uma variável `hoje`, que criamos. Não é preciso usar a palavra `var` para definir variáveis:

```
hoje = new Date(); // armazena a data de hoje
```

A instrução “`new`” é um operador utilizado para criar novos objetos. `Date()` é uma função especial, chamada de construtora. Ela constrói o novo objeto e define métodos e propriedades que podem ser invocados a partir do objeto. `hoje`, portanto, é um *objeto* que representa a data de hoje e tem métodos, definidos pela função construtora `Date()`. Uma data é um *tipo de dados* abstrato que possui várias propriedades. Só precisamos de três: dia, mês e ano. A única forma de obtê-las em JavaScript é invocando *métodos* sobre `hoje`. Os métodos

retornam propriedades específicas dos objetos onde são invocados. Usamos o operador “.” para ter acesso a eles, assim como fizemos com as propriedades:

```
<div align=center>
<script>
    hoje = new Date()
    dia = hoje.getDate()
    mes = hoje.getMonth() + 1
    ano = hoje.getYear() + 1900
    document.write("<h3>Hoje: " + dia + "/" + mes + "/" + ano + "</h3>")
</script>
```

Tivemos que somar 1 ao valor retornado pelo método `getMonth()` porque ele retorna os meses contados a partir de 0 e terminando em 11. Somamos 1900 ao valor retornado por `getYear()` porque o método retorna o número de anos desde 1900. A última instrução, imprime os valores na página. Veja o código completo no arquivo `formcodsol.html`.

Exercícios

- 1.1 Faça com que a propriedade `window.status` (texto da barra de status do browser) seja redefinida com a string ‘Um Link’ quando o usuário passar o mouse sobre o link (use qualquer página HTML). Utilize os atributos eventos `onmouseover` e `onmouseout` em `<A HREF>`.
- 1.2 Altere o exercício resolvido para que os dados digitados sejam mostrados em uma janela de alerta (instrução `alert("string")`) em vez de serem mostrados no campo de texto.

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:

```
x = 5;

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

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

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 possuir 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

Global

Object

Boolean

Function

Array

Number

Date

String

Math

Tipos de dados primitivos (que representam valores)

undefined

representa valores undefined ainda não definidos

number

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

Min: ± 4.94065 e-324

Max: ±1.79769 e+308

NaN

Infinity

-Infinity

null

representa o valor nulo

null

boolean

representa valores booleanos

true

false

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. Freqüentemente 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:

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

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;     // atribuição com soma. Equivale a      y = y + 1 ou y++
z /= 5;     // atribuição com divisão. 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:

Associatividade	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 esquerda, 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`, `numero`, 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.

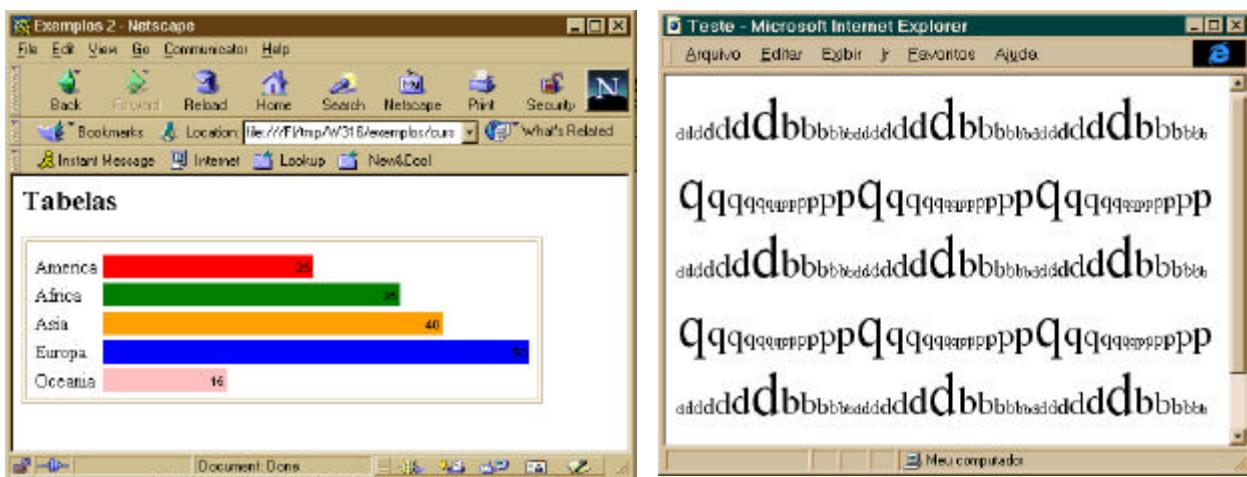
Exercícios

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 parâmetro `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 parâmetro `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.

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 nativas¹. 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);
```

¹ 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 <code>NaN</code> (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 <code>NaN</code> (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 <i>string</i> 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"); // nome contém Jo%EAo</code>
<code>unescape(string)</code>	Faz a operação inversão de <code>escape(string)</code> . Exemplo: <code>nome = unescape("Jo%EAo"); // nome contém 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 (sequência 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 ($n! = n(n-1)(n-2)\dots(2)(1)$). 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); ← Chama a função Date()
                                                (construtor) que retorna as
                                                informações necessárias
                                                para criar o objeto.
```

Variável do tipo *object* que armazena informações retornadas por `Date()`.

JS17-01-1999/01/44 - © 1999 Held 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 JavaScript²:

Construtor	Tipo de objeto construído
<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

² 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:

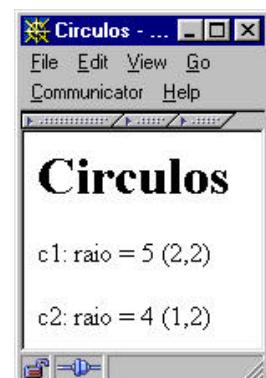
```
function metodo1() {
    document.write("Saldo: " + this.saldo);
}
function Conta(corr, saldo) {
    this.correntista = corr;
    this.saldo = saldo;
    this.imprimeSaldo = metodo1;
}
```

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 *Círculo* especificando um construtor da forma *Círculo(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>c2: 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 extraír 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 `variável` e seu valor poderá ser extraído usando vetores associativos, da forma:

```
objeto[variável]
```

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` (*Círculo*) 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 *Círculo*. É possível, através de um *Alvo*, ter acesso e propriedades do *Círculo* 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

objetos, diferentemente do que ocorre com valores primitivos, não faz uma cópia do objeto. Copia um *ponteiro* ou *referência* para eles. É preciso usar `new` para criar um novo objeto.

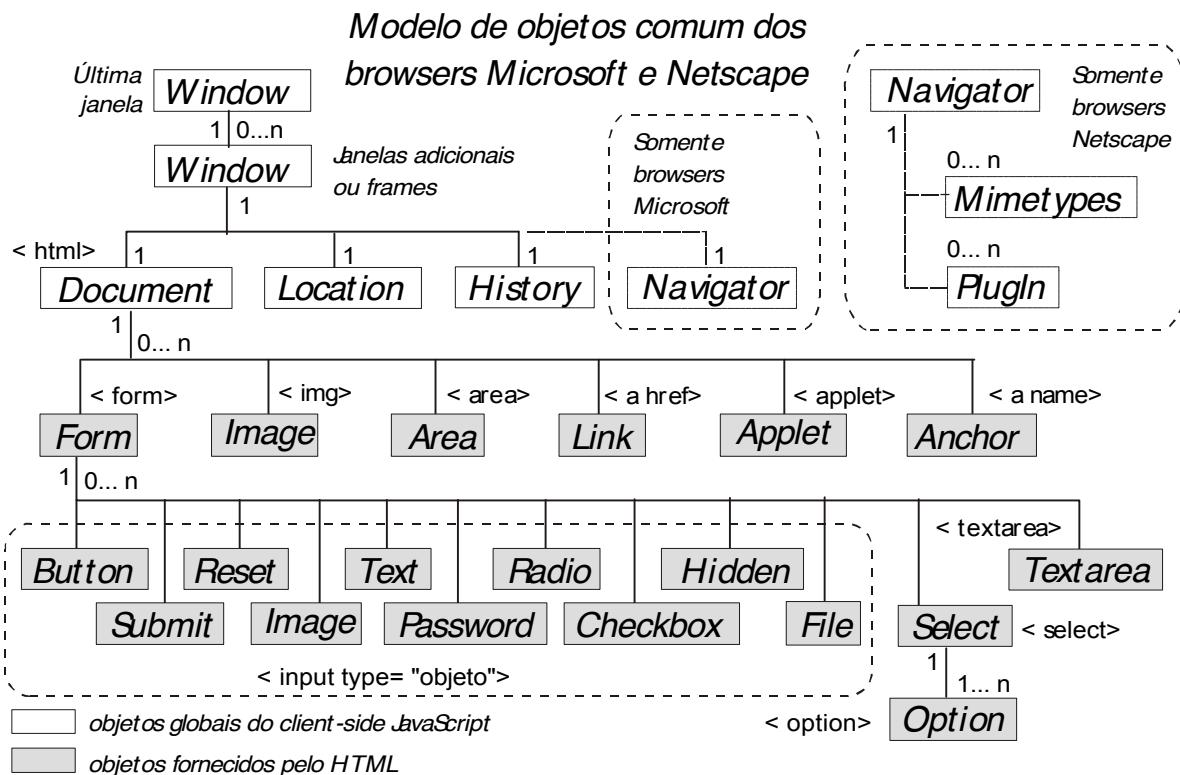
Exercícios

- 3.3 Crie um novo tipo *Cliente*, com as propriedades `nome`, `email` e `telefone`. Crie 5 objetos usando esse tipo e use `for...in` para listar e imprimir na página cada cliente e suas propriedades.
- 3.4 Crie um método para o tipo *Círculo* para que seja possível imprimir o raio e centro do círculo da mesma forma para todos os círculos. Use o formato: Raio: $r(x, y)$.

Modelo de objetos do HTML

A grande maioria dos objetos client-side JavaScript representam características do browser e do documento HTML. Dentro de um documento, a hierarquia de objetos e propriedades JavaScript reflete a hierarquia do HTML, através de um *modelo de objetos do documento* (Document Object Model - DOM) suportado pelo browser. O DOM relaciona cada elemento HTML, respeitando sua hierarquia, a um objeto JavaScript. Por exemplo, em HTML um bloco `<INPUT TYPE="text">` está relacionado a um objeto do tipo *Text*. O elemento deve estar dentro de um bloco `<FORM>`, representado por um objeto do tipo *Form*, que por sua vez, deve estar dentro de um bloco `<BODY>`, representado por um objeto do tipo *Document*.

Os modelos de objetos utilizados pelos browsers da Netscape e da Microsoft não são idênticos mas têm várias semelhanças. A figura abaixo mostra a hierarquia de objetos do JavaScript suportados por ambas as implementações.



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

objetos JavaScript como uma hierarquia. Todos os elementos que existirem na página podem ser objetos accessíveis como propriedades de `document`.

Dentro de uma página há vários elementos: imagens, formulários, parágrafos, tabelas. O modelo de objetos do JavaScript 1.1 permite apenas a manipulação de imagens, vínculos, âncoras, applets, formulários e seus componentes. O nome de um objeto associado a um elemento pode ser definido em HTML, através do atributo `NAME`:

```
<IMG SRC="zebra.gif" name="figura3">
```

Existindo o elemento acima, passa a existir também um objeto JavaScript, acessível através de uma nova propriedade do documento chamada `figura3`:

```
z = y.figura3 // window.document.figura3 é ref. do tipo Image
```

A variável `z` é um objeto do tipo `Image`, e pode ser manipulada como tal em JavaScript, ou seja, suas propriedades podem ser lidas e seus métodos podem ser invocados. Utilizando instruções JavaScript pode-se, por exemplo, trocar a imagem (“`zebra.gif`”) por outra:

```
z.src = "jegue.gif"; // src é propriedade (tipo String) de Image
```

Os objetos definidos através do HTML são objetos como quaisquer outros objetos JavaScript. Praticamente todos possuem *propriedades* e *métodos* mas a grande maioria não possui construtores utilizáveis pelo programador, já que são construídos automaticamente pelo browser a partir do HTML da página. Como grande parte dos objetos do HTML possui propriedades que são outros tipos de objetos, é freqüente surgirem expressões longas como:

```
comprimento = window.document.forms[2].campotexto.value.length;
           ↑          ↑          ↑          ↑          ↑          ↑          ↑
           Number      Window     Document   Form       Text      String    Number
```

A expressão acima mostra como a hierarquia de elementos do HTML se reflete em JavaScript através de propriedades. Para ler propriedades ou invocar métodos sobre um objeto do browser é necessário portanto citar toda a hierarquia de objetos que está acima dele. A única exceção à regra é a referência `window`, que sempre pode ser omitida. As instruções abaixo fazem a mesma coisa:

```
window.document.write("Tigres");
document.write("Tigres");
```

Quando a janela na qual estamos operando não é a janela atual, mas outra que foi aberta utilizando instruções JavaScript, é preciso utilizar o nome do objeto, só que não será `window`. Quando criamos uma nova janela, podemos batizá-la com um nome qualquer que servirá de referência para operar sobre ela. Essa referência é um *objeto* do tipo `Window`:

```
janela2 = window.open("página2.html"); // método open retorna referência
           ↑
           Window
           // do tipo Window que é propriedade
           // da janela atual (window)
```

```
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">
```



```

<input type=text name=val1 size=5> +
<input type=text name=val2 size=5>
<input type=button value=" = " onclick="soma()">
<input type=text name=val3 size=5>
</form>
</body> </html>

```

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.):

```
<input type=button value=" = " onclick="soma(this.form)">
```

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.val1.value;
    b = calc.val2.value;
    calc.val3.value = parseFloat(a) + parseFloat(b);
}
</script>

```

with

`with` é uma estrutura especial que 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

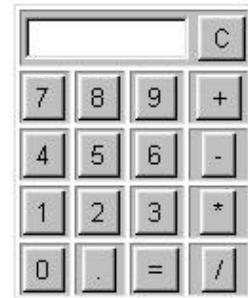
O operador **delete** não existe em JavaScript 1.1. Pode ser usado nos browsers que suportam implementações mais recentes para remover objetos, elementos de um vetor ou propriedades de um objeto. Não é possível remover variáveis declaradas com **var** ou propriedades e objetos pré-definidos. A sintaxe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

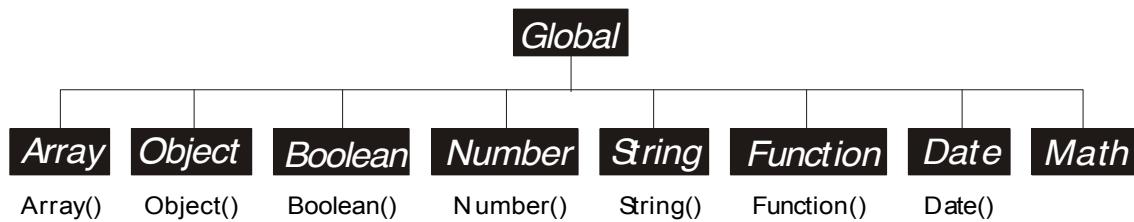


4

Objetos nativos embutidos

OS OBJETOS NATIVOS EMBUTIDOS¹ 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
<code>toString()</code>	Transforma qualquer objeto em uma representação <i>string</i> . É usado automaticamente nas conversões de números em strings, por exemplo, durante a concatenação.
<code>valueOf()</code>	Converte um objeto em seu valor primitivo, se houver.
<code>assign(valor)</code>	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 *Círculo*, 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 obtém 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 (<i>tipo da propriedade em itálico</i>)
<code>arguments[]</code>	<i>Array</i> . O vetor de argumentos da função
<code>arguments.length</code>	<i>Number</i> . O comprimento do vetor de argumentos (retorna o número de argumentos que a função tem)
<code>length</code>	<i>Number</i> . Mesma coisa que <code>arguments.length</code>
<code>caller</code>	<i>Function</i> . Uma referência para o objeto <i>Function</i> que chamou esta função, ou <code>null</code> 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.
<code>prototype</code>	<i>Object</i> . 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.getFullYear() + 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 **bissesto**) a uma nova propriedade do protótipo do objeto, que chamamos de **isLeapYear**:

```
Date.prototype.isLeapYear = bissesto;
```

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+')';");

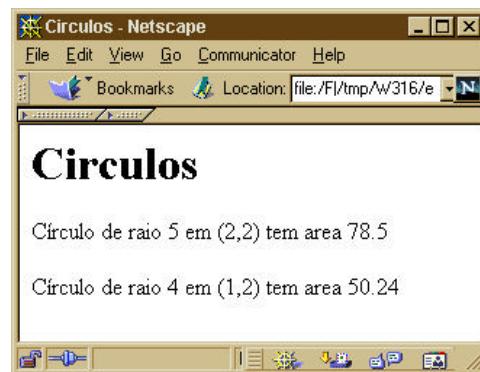
// criação de um método area para o Circulo
Circulo.prototype.area =
    new Function("return 3.14 * this.r * this.r;");
</script>
</HEAD>
<BODY>
<h1>Círculos</h1>
<script>
c1 = new Circulo(2,2,5); // uso da função construtora
c2 = new Circulo(1,2,4);

// 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;
```

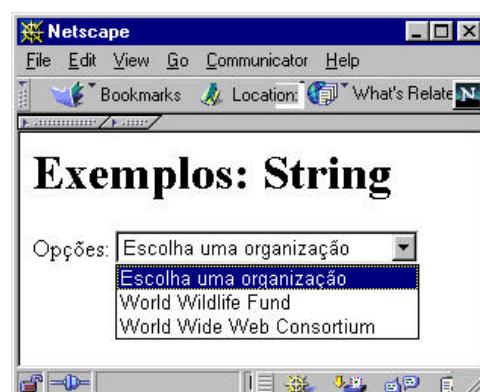
O construtor `String()` possui uma propriedade `prototype` que permite a definição de novos métodos e propriedades. A propriedade `prototype` não é uma propriedade de `String`, mas do construtor `String()`, que é `Function` (como são todos os construtores), portanto deve ser usada da forma:

```
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 método
        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 *Círculo*, 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")	 Texto <>
link("http://a.com")	 Texto
small()	<small> Texto </small>
big()	<big> Texto </big>
blink()	<blink> Texto </blink>
strike()	<strike> Texto </strike>
sub()	_{Texto}
sup()	^{Texto}
italics()	<i> Texto </i>
bold()	 Texto
fixed()	<tt> Texto </tt>
fontcolor("cor")	 Texto (cor pode ser um valor rrggbb hexadecimal ou nome de cor)
fontsize(7)	 Texto (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 <i>íntlico</i>)
charAt(<i>n</i>)	<i>String</i> . Retorna o caractere na posição <i>n</i> . A string <i>s</i> inicia na posição 0 e termina em <i>s.length</i> -1. Se for passado um valor de <i>n</i> maior que <i>s.length</i> -1, o método retorna uma string vazia.
indexOf("substring")	<i>Number</i> . Retorna um índice <i>n</i> referente à posição da primeira ocorrência de "substring" na string <i>s</i> .
indexOf("substring", <i>inicio</i>)	<i>Number</i> . Retorna um índice <i>n</i> referente à posição da primeira ocorrência de "substring" em <i>s</i> após o índice <i>inicio</i> . <i>inicio</i> é um valor entre 0 e <i>s.length</i> -1
lastIndexOf("substring")	<i>Number</i> . Retorna um índice <i>n</i> referente à posição da última ocorrência de "substring" na string <i>s</i> .
lastIndexOf("substring", <i>fim</i>)	<i>Number</i> . Retorna um índice <i>n</i> referente à posição da última ocorrência de "substring" em <i>s</i> antes do índice <i>fim</i> . <i>fim</i> é um valor entre 0 e <i>s.length</i> -1
split("delimitador")	<i>Array</i> . Converte o string em um vetor de strings

Método Invocado	Ação (tipo de retorno em <i>italico</i>)
	separando-os pelo "delimitador" especificado. O método <code>join()</code> de <code>Array</code> faz o oposto.
<code>substring(inicio, fim)</code>	<code>String</code> . Extrai uma substring de uma string <code>s</code> . <ul style="list-style-type: none"> • <code>inicio</code> é um valor entre 0 e <code>s.length-1</code>. • <code>fim</code> é um valor entre 1 e <code>s.length</code>. O caractere na posição <code>inicio</code> é <i>incluído</i> na string e o caractere na posição <code>fim</code> <i>não é incluído</i> . A string resultante contém caracteres de <code>inicio</code> a <code>fim -1</code> .

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 (`[]`). 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 maneira. 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 *italico* na descrição de cada método:

Método	Ação
<code>join() ou join("separador")</code>	Retorna <i>String</i> . 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.
<code>reverse()</code>	<i>Array</i> . Inverte a ordem dos elementos de um vetor. Tanto o vetor retornado, quanto o vetor no qual o método é chamado são afetados.
<code>sort()</code>	<i>Array</i> . 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.
<code>sort(funcão_de_ordenação())</code>	<i>Array</i> . Ordena os elementos do vetor com base em uma função de ordenação. A função deve tomar dois valores a e b e deve retornar: <ul style="list-style-type: none"> • Menor que zero se a < b • Igual a zero se a = b • 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
}
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.constant;

```

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

Funções	Constantes
<code>acos(x)</code> cosseno ⁻¹	<code>E</code> e
<code>asin(x)</code> seno ⁻¹	<code>LN10</code> $\ln 10$
<code>atan(x)</code> tangente ⁻¹	<code>LN2</code> $\ln 2$
<code>atan2(x, y)</code> retorna o ângulo θ de um ponto (x,y)	<code>POW(x, y)</code> x^y
<code>ceil(x)</code> arredonda para cima (3.2 e 3.8 → 4)	<code>LOG10E</code> $\log_{10} e$
<code>cos(x)</code> cosseno	<code>LOG2E</code> $\log_2 e$
<code>exp(x)</code> e^x	<code>PI</code> π
<code>floor(x)</code> arredonda para baixo (3.2 e 3.8 → 3)	<code>SQRT1_2</code> $1/\sqrt{2}$
<code>random()</code> retorna um número pseudo-aleatório entre 0 e 1.	<code>SQRT2</code> $\sqrt{2}$

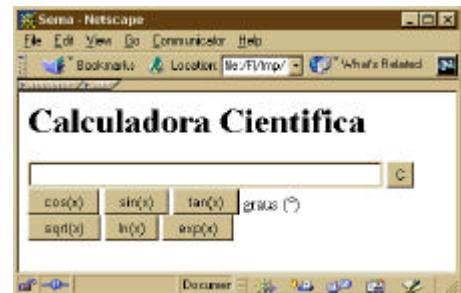
O programa a seguir utiliza algumas funções e constantes do tipo *Math* para implementar uma pequena calculadora científica.

```

<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 aleatório 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("");
</script>
</BODY> </HTML>
```

Exercícios

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

Método	Ação
<code>setDate(dia_do_mês)</code>	Acerta o dia do mês (1 a 31)
<code>setHours(hora)</code>	Acerta a hora (0 a 23)
<code>setMinutes(minuto)</code>	Acerta o minuto (0-59)
<code>setMonth(mês)</code>	Acerta o mês (0-11)
<code>setSeconds()</code>	Acerta o segundo (0-59)
<code>setTime()</code>	Acerta a hora em milissegundos desde 1-1-1970 0:0:0 GMT
<code>setYear()</code>	Acerta o ano (ano – 1900)
<code>toGMTString()</code>	<i>String.</i> Converte uma data em uma representação GMT
<code>toLocaleString()</code>	<i>String.</i> 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
<code>parse(string)</code>	Retorna <code>Date</code> . Converte uma data do sistema no formato IETF (usado por servidores de email, servidores HTTP, etc.) em milissegundos 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: <code>DataIETF = "Wed, 8 May 1996 22:44:53 -0200"; umaData = new Date(Date.parse(DataIETF));</code>
<code>UTC()</code>	Retorna <code>Number</code> . Converte uma data no formato UTC separado por vírgulas para a representação em milissegundos: <code>Date.UTC(ano, mês, dia [, horas[, minutos[, segundos]]]);</code> Exemplo: <code>millis = Date.UTC(75, 11, 13, 23, 30);</code>

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 quantos 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`.

5

As janelas do browser

A JANELA DO BROWSER é manipulável de várias formas através da linguagem JavaScript. Pode-se alterar dinamicamente várias de suas características como tamanho, aparência e posição, transferir informações entre janelas e frames, abrir e fechar novas janelas e criar janelas de diálogo.

Janelas do browser são representadas em JavaScript através de objetos do tipo *Window*. Pode-se classificar as janelas usadas em JavaScript em cinco categorias:

- *Janela da aplicação*: é um papel assumido pela última janela aberta do browser. Se esta janela for fechada, a aplicação é encerrada. Em JavaScript, métodos para fechar janelas (`close()`) não funcionam na última janela.
- *Janelas abertas através de instruções JavaScript*: são novas janelas abertas através de um método `open()`. Podem ter tamanho e características diferentes, ser manipuladas e manipular a janela que as criou, recebendo ou retornando dados, lendo ou alterando propriedades, invocando métodos, inclusive para fechar a outra janela.
- *Janelas abertas através de HTML*: são janelas abertas usando links com o descritor `target` (``). JavaScript pode carregar novas páginas nessas janelas, mas não pode manipular suas propriedades ou métodos.
- *Janelas estruturais*: são janelas ou *frames* que contém uma página HTML que estabelece uma estrutura que divide a janela em *frames* (contém bloco `<FRAMESET>` e não contém `<BODY>`). Possui referências para cada frame que contém.
- *Frames de informação*: são *frames* de uma janela pai que contém uma página HTML com informação (contém um bloco `<BODY>`). Este tipo de janela só possui referências para as janelas que as contém.

Além das janelas comuns, que contém páginas HTML, há três janelas de diálogo: *alerta*, *confirmação* e *entrada de dados*, que não têm propriedades manipuláveis. Todos os tipos de janelas são representadas através de propriedades do objeto `window`.

Objeto Window

O tipo *Window*¹ representa janelas. A propriedade global `window` representa *a janela do browser onde roda o script*. Através de `window`, têm-se acesso a outras propriedades que referenciam possíveis sub-janelas, a janela que a criou (se existir) ou *frames*. Também têm-se acesso a métodos que abrem caixas de diálogo de aviso, confirmação e entrada de dados.

As propriedades e métodos de *Window*, quando referentes à janela atual (objeto `window`), podem omitir o nome do objeto:

```
window.status = "oye!";           // ou status = "oye!";
window.open("documento.html"); // ou open("documento.html");
```

Mas isto só vale se a janela na qual se deseja invocar o método ou a propriedade for a janela atual, onde roda o script. A propriedade `window` refere-se sempre à janela atual.

A tabela abaixo relaciona as propriedades dos objetos do tipo *Window*. Observe que muitos são objetos *Window* e, como consequência, têm as mesmas propriedades:

Propriedade	Acesso	Função
<code>defaultStatus</code>	read / write	Contém <i>String</i> . Texto que aparece por <i>default</i> na barra de status da janela.
<code>status</code>	r / w	Contém <i>String</i> . Define texto que aparecerá na barra de status.
<code>name</code>	r / w	Contém <i>String</i> . Contém nome da janela. Este nome é utilizável em HTML no atributo <code>TARGET</code> em <code></code> e em <code><BASE TARGET="nome"></code> . Em <i>frames</i> , retorna uma referência <i>Window</i> .
<code>document</code>	r	Contém <i>Document</i> . Referência à página contida na janela.
<code>history</code>	r	Contém <i>History</i> . Referência ao histórico da janela.
<code>location</code>	r	Contém <i>Location</i> . Referência à URL exibida na janela.
<code>navigator</code>	r	Contém <i>Navigator</i> . Referência a string de identificação do <i>browser</i> .
<code>opener</code>	r	Contém <i>Window</i> . Refere-se a janela que abriu esta janela
<code>self</code>	r	Contém <i>Window</i> . Referência à própria janela. Mesmo que <code>window</code>
<code>window</code>	r	Contém <i>Window</i> . Sinônimo de <code>self</code> .
<code>frames</code>	r	Contém <i>Array</i> de <i>Window</i> . Vetor dos frames contidos na janela.
<code>length</code>	r	Contém <i>Number</i> . Número de elementos <i>Window</i> no vetor <code>frames</code> (mesma coisa que <code>window.frames.length</code>)
<code>parent</code>	r	Contém <i>Window</i> . Referência à janela que contém esta janela (só existe quando a janela atual é um frame)
<code>top</code>	r	Contém <i>Window</i> . Referência à janela que não é frame que contém a janela atual (só existe quando a janela atual é um frame)

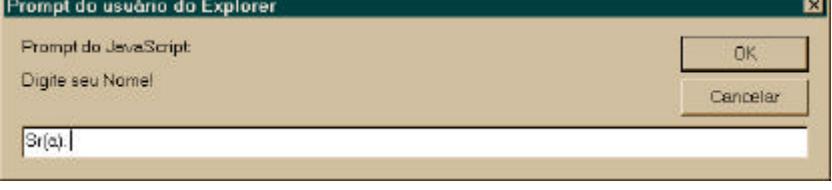
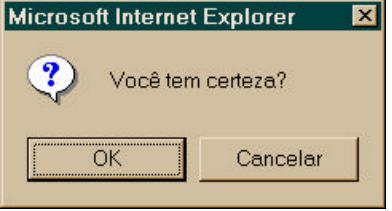
¹ 'Window' é um nome genérico que usamos para qualificar janelas. Não há construtor ou qualquer propriedade com este nome. Existe sim, a propriedade `window` (com "w" minúsculo), que representa a janela atual.

As propriedades `top`, `frames`, `length` e `parent` só têm sentido em janelas que são frames ou que estão dentro de frames. A propriedade `opener` só existe em janelas que foram abertas por outras janelas. É uma forma da ‘janela filha’ ter acesso à sua ‘janela mãe’.

Além das propriedades acima, *Window* possui vários métodos com finalidades bem diferentes. Com eles é possível criar de janelas de diálogo e janelas do browser com aparência personalizada, manipular janelas e realizar tarefas pouco relacionadas com janelas como rolagem de páginas e temporização.

Janelas de diálogo

Três métodos de *Window* são usados apenas para criar janelas de diálogo. Eles são: `alert()`, `confirm()` e `prompt()` e estão listados na tabela abaixo. Não é possível retornar à controle da janela (de onde foram chamados) sem que os diálogos sejam fechados.

Método	Exemplo
<code>alert("msg")</code>	<pre>window.alert("Tenha Cuidado!");</pre> 
<code>prompt("msg")</code> ou <code>prompt("msg", "texto inicial")</code>	<pre>nome = window.prompt("Digite seu Nome!", "Sr(a). ");</pre>  <p>Retorna <i>String</i>. Devolve o string digitado caso o usuário clique em OK e um string nulo caso o usuário clique em Cancelar.</p>
<code>confirm("msg")</code>	<pre>if (window.confirm("Você tem certeza?")) { ... }</pre>  <p>Retorna <i>Boolean</i>: true caso o usuário clique em OK e false caso o usuário clique em Cancelar.</p>

Nos exemplos acima, a referência `window` pode ser omitida ou substituída por outra referência caso os diálogos estejam sendo abertos em outras janelas.

Métodos para manipular janelas

Os métodos restantes definidos para os objetos `Window`, manipulam parâmetros das janelas, abrem e fecham novas janelas, rolam a página e definem funções de temporização. Estão listados na tabela abaixo.

Método	Ação
<code>open("URL")</code> ou <code>open("URL", "nome")</code> ou <code>open("URL", "nome", "características")</code>	Abre uma nova janela contendo um documento indicado pela URL. Opcionalmente, a janela pode ter um <i>nome</i> que pode ser usado em HTML, ou ter alteradas características como tamanho, layout, etc. (veja tabela abaixo). Retorna uma referência do tipo <code>Window</code> para a janela criada: <code>filha = window.open("http://a.com/abc.html");</code>
<code>close()</code>	Fecha uma janela (não vale para frames para a última janela da aplicação).
<code>blur()</code>	Torna uma janela inativa
<code>focus()</code>	Torna uma janela ativa (traz para a frente das outras, se for uma janela independente).
<code>scroll(x, y)</code>	Rola o documento dentro de uma janela de forma que as coordenadas <i>x</i> e <i>y</i> (em pixels) da página apareçam no canto superior esquerdo da área útil da janela, se possível.
<code>setTimeout("instruções", atraso)</code>	Executa uma ou mais instruções JavaScript após um período de atraso em milissegundos. Este método é parecido com a função <code>eval()</code> , mas com temporização. O código continua a ser interpretado imediatamente após o <code>setTimeout()</code> . A espera ocorre em um <i>thread</i> paralelo. Retorna <code>Number</code> : um número de identificação que pode ser passado como argumento do método <code>clearTimeout()</code> para executar a operação imediatamente, ignorando o tempo que falta.
<code>clearTimeout(id)</code>	Cancela a temporização de uma operação <code>setTimeout()</code> cujo número de identificação foi passado como parâmetro, e faz com que as instruções do <code>setTimeout()</code> sejam interpretadas e executadas imediatamente.

Uma janela pode ser aberta em qualquer lugar. Basta fazer:

```
window.open("documento.html"); // ou simplesmente open("documento.html");
```

Janelas com aparéncia personalizada

As janelas abertas podem ter várias de suas características alteradas no momento em que são abertas. Estas características deverão vir em uma string com uma lista de opções separadas por vírgulas, como o terceiro argumento opcional do método `open()`. Cada característica pode ou não ter um valor. *Não deverá haver espaços* em qualquer lugar da lista. Por exemplo:

```
window.open("enter.html", "j2", "height=200,width=400,status");
```

abre uma janela de 200 pixels de altura por 400 de largura *sem* barra de ferramentas, *sem* barra de diretórios, *sem* campo de entrada de URLs, *sem* barra de menus, não-redimensionável e com barra de status. As características estão na tabela abaixo:

Característica	Resultado
<code>height=h</code>	h é a altura da janela em pixels: <code>height=150</code>
<code>width=w</code>	w é a largura da janela em pixels: <code>width=300</code>
<code>resizable</code>	Se estiver presente permite redimensionar a janela
<code>toolbar</code>	Se estiver presente, mostra a barra de ferramentas do browser
<code>directories</code>	Se estiver presente, mostra a barra de diretórios do browser
<code>menubar</code>	Se estiver presente, mostra a barra de menus do browser
<code>location</code>	Se estiver presente, mostra o campo para entrada de URLs
<code>status</code>	Se estiver presente, mostra a barra de status

Se for utilizado o método `open()` com três argumentos, qualquer característica acima que não apareça listada no string passado como terceiro argumento, não estará presente.

Propriedades da barra de status

A propriedade `defaultStatus` determina o valor *default* do texto que é exibido na barra de status do browser. Geralmente este valor é um string vazio ("") mas pode ser alterado. A propriedade `status` é usada para mudar o valor da barra de status no momento em que um novo valor é atribuído. Para fazer links informativos, que apresentam uma mensagem na barra de status quando o mouse passa sobre eles, pode-se usar:

```
<script>
    window.defaultStatus="";
</script>

<a href="resultados.html" onmouseover="window.status='Resultados '"
   onmouseout="window.status = window.defaultStatus">
Clique Aqui!</a>
```

Uma aplicação comum para o `window.status` é colocar uma mensagem rolando na barra de status. O processo é semelhante ao proposto para um campo de textos, no exercício 4.1. Consiste em colocar o primeiro caractere no final de uma string e escrevê-lo no `window.status`.

Eventos

Vários eventos do JavaScript estão relacionados com janelas. Estes eventos são chamados a partir dos atributos HTML listados abaixo, que são aplicáveis aos descritores HTML `<BODY>` e `<FRAME>`:

- `ONBLUR` – quando a janela deixa de ser a janela ativa
- `ONERROR` – quando ocorre um erro (uma janela deixa de ser carregada totalmente)
- `ONFOCUS` – quando a janela passa a ser a janela ativa
- `ONLOAD` – depois que a página é carregada na janela
- `ONUNLOAD` – antes que a página seja substituída por outra ou a janela fechada.

Por exemplo, o código abaixo em uma página carregada em uma janela do browser impedirá que qualquer outra janela esteja ativa até que a janela atual seja fechada. Qualquer tentativa de minimizar a janela, ou de selecionar outra causará o evento tratado por `ONBLUR`, que chamará o método `focus()`, reestabelecendo o estado ativo da janela.

```
<body onblur="focus()"> ... </body>
```

Este outro exemplo, mostra uma o uso do atributo de evento `ONUNLOAD` para criar uma página que só permite uma única saída, ou seja, só é possível sair da janela atual para entrar em outra definida pelo autor da página. Qualquer tentativa de escolher uma outra URL será sobreposta:

```
<body onunload="location.href='pagina2.html';"> ... </body>
```

Para iniciar um programa, ou rodar uma função, ou executar qualquer procedimento logo depois que *todo o HTML* de uma página tiver sido carregado na janela do browser, pode-se usar o atributo de evento `ONLOAD`:

```
<body onload="iniciarAnimacao()"> ... </body>
```

Todos os atributos de evento também podem ser usados em conjunto:

```
<body onload="iniciar()"  
onunload="parar()"  
onblur="parar()"  
onfocus="iniciar()"  
onerror="location.href=document.location"> ... </body>
```

O manuseador de evento `ONERROR` poderá não funcionar se o erro ocorrer antes que o descritor `<BODY>` que o contém seja carregado.

Comunicação entre janelas

Para passar informações para uma janela recém criada, é necessário obter *uma referência* para a janela. Isto só é possível se a nova janela for criada usando JavaScript. Não funciona para janelas criadas usando HTML. A referência é obtida como valor de retorno do método `open()`:

```
novaJanela = window.open("pg2.html");
```

Com a referência `novaJanela`, que é *Window*, é possível ter acesso a qualquer propriedade da nova janela e invocar seus métodos, por exemplo:

```
novaJanela.document.write(""); //acrescenta texto à página da janela
novaJanela.focus();           // torna a janela ativa
novaJanela.close();          // fecha a janela
```

Se uma janela é criada usando `open()`, mas o seu valor de retorno não é armazenado em uma variável, não será possível ter acesso às propriedades da *janela filha*. Mas a nova janela sempre pode ter acesso à janela que a criou, manipular suas propriedades e até fechá-la. Toda *janela filha* possui uma propriedade `opener`, que é uma referência à sua *janela mãe*. Para manipular propriedades e invocar métodos ela poderá fazer:

```
opener.focus();           // torna a janela mãe ativa
opener.document.forms[0].elements[2].value = "Oi mãe!";
opener.close();          // mata a mãe
```

É importante verificar que uma propriedade existe, antes de tentar usá-la. Quando se trabalha com múltiplas janelas, é comum uma janela tentar usar uma propriedade que não existe em outra (ou que *ainda* não existe). Se uma página procura um formulário em outra janela e a outra janela não mais apresenta a página que tinha o formulário, o browser acusará um erro, informando a inexistência do objeto.

A tentativa de acessar propriedades inexistentes provoca erros feios em JavaScript. Os browsers mais novos já escondem as janelas de aviso, mas muitos ainda não o fazem. Uma forma de evitá-los é sempre verificar se um objeto está definido, antes de usá-lo. Isto pode ser feito em JavaScript usando a palavra-chave `null`:

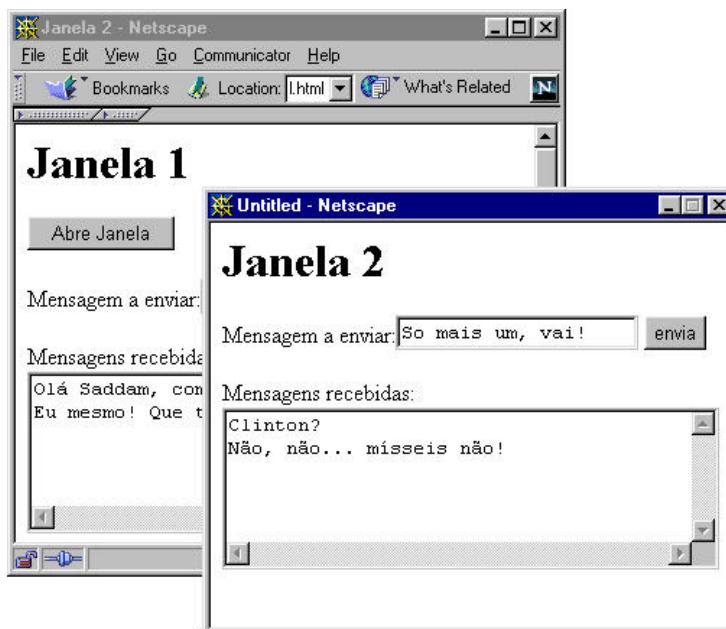
```
if (janela != null) {      // verifica se janela existe
    janela.focus();       // coloca na frente
    if (janela.document.forms[0] != null) { // formulario existe?
        if (campotexto != null) {
            janela.document.forms[0].campotexto.value = "OK";
        }
    }
} else {
    janela = open("pagina.html");
    janela.document.forms[0].elements[0].value = "OK";
}
```

Exercício Resolvido

Monte duas páginas HTML como mostrado na figura abaixo. A primeira página deve ter um botão “Abre Janela” que, quando apertado, deve abrir uma nova janela nas dimensões 360x280 (pixels). Depois de aberta, a nova janela deverá estar na frente da antiga (use `focus()`).

Depois que as duas janelas estiverem abertas, o texto digitado no campo `enviar`, da janela menor, deve aparecer na caixa de mensagens da janela maior, logo que o botão envia for pressionado. Em seguida, a janela maior deverá tornar-se ativa. Pode-se fazer o mesmo na janela maior e passar informações para o campo de mensagens da janela menor.

Use os esqueletos `jan1.html` e `jan2.html` disponíveis no subdiretório `cap5/`. A solução é mostrada a seguir e está nos arquivos `jan1sol.html` e `jan2sol.html`.



Solução

O exemplo a seguir ilustra a comunicação entre janelas. São duas listagens. A primeira é o arquivo para a primeira janela e a segunda o arquivo para a sub-janela. Observe o nome do arquivo “`jan2.html`”. Deve ser idêntico ao primeiro parâmetro do método `open()` na página abaixo.

A página principal contém um botão que permite criar uma nova janela. A partir daí, escreva algo no primeiro campo da nova janela, clique na primeira e veja os dados serem copiados de uma janela para outra.

Este é o código para a janela maior “jan1.html”:

```
<HTML>
<HEAD>
    <TITLE>Janela 2</TITLE>
    <SCRIPT LANGUAGE=JavaScript>

        var janela2; // global

        function abreJanela() {
            if (janela2 != null) { // janela já está aberta
                janela2.focus();
            } else {
                janela2 = open("jan2.html", "", "height=280,width=360");
            }
        }

        function envia() {
            janela2.document.f1.mensagens.value += document.f1.enviar.value + "\n";
            document.f1.enviar.value = "";
            janela2.focus();
        }
    </SCRIPT>
</HEAD>

<BODY>
<H1>Janela 1</H1>

<FORM NAME=f1>
<INPUT TYPE=button VALUE="Abre Janela" ONCLICK="abreJanela()">
<P>Mensagem a enviar:<INPUT TYPE=text NAME="enviar">
<INPUT TYPE=button VALUE="envia" onclick="envia()">
<p>Mensagens recebidas: <br>
<TEXTAREA NAME="mensagens" COLS=40 ROWS=5></TEXTAREA>
</FORM>
</BODY>
```

Este é o arquivo para a janela menor: “jan2.html”

```
<HTML>
<HEAD>
    <TITLE>Janela 2</TITLE>
    <SCRIPT LANGUAGE=JavaScript>
        function envia() {
            opener.document.f1.mensagens.value += document.f1.enviar.value + "\n";
            document.f1.enviar.value = "";
            opener.focus();
        }
    </SCRIPT>
```

```
</HEAD>

<BODY>
<H1>Janela 1</H1>
<FORM NAME=f1>
<P>Mensagem a enviar:<INPUT TYPE=text NAME="enviar">
<INPUT TYPE=button VALUE="envia" onclick="envia()">
<p>Mensagens recebidas: <br>
<TEXTAREA NAME="mensagens" COLS=40 ROWS=5>
</TEXTAREA>
</FORM>

</BODY>
</HTML>
```

Frames HTML

Frames são janelas que estão limitadas dentro de outras janelas. Através de referências especiais, é possível, usando JavaScript, manipular as propriedades de qualquer frame dentro de uma janela ou em outra janela. Antes de apresentar, porém, como é possível manipular frames em JavaScript, vejamos como os frames podem ser construídos em HTML.

Para dividir uma janela em frames, é preciso criar uma página HTML especificando as dimensões relativas ou absolutas das subjanelas em relação à janela que irá conter a página. Uma página de frames não é um documento HTML, pois não contém informação. Todo documento HTML deve ter a forma:

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

O bloco **<body>** contém a informação da página. O bloco **<head>**, contém meta-informação, ou seja, informação sobre a página. Páginas de frames têm uma estrutura diferente:

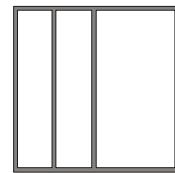
```
<html>
  <head> ... </head>
  <frameset atributos> ... </frameset>
</html>
```

e não podem conter blocos **<body>**².

² Até podem conter blocos **<BODY>**, mas isto ora os transforma em páginas de informação, ora não causa efeito algum. Um bloco **<BODY>** antes do **<FRAMESET>** faz com que o browser ignore o **<FRAMESET>**. Um bloco **<BODY>** após o **<FRAMESET>** será ignorado por browsers que suportam frames, mas será lido por browsers antigos que não os suportam.

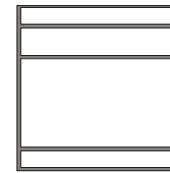
O bloco `<frameset>` define a divisão da janela em linhas (usando o atributo `rows`) ou colunas (usando o atributo `cols`). Os atributos especificam a largura ou altura de cada frame usando valores absoltos, em pixels, ou relativos, em percentagens da largura ou altura da janela principal. Por exemplo, um `<FRAMESET>` da forma (figura ao lado):

```
<FRAMESET COLS="25%, 25%, 50%"> ... </FRAMESET>
```



divide a janela principal em três colunas, tendo as duas primeiras $\frac{1}{4}$ da largura total, e a última, metade da largura total. De forma semelhante pode-se dividir a janela em linhas. Neste outro exemplo (figura ao lado):

```
<FRAMESET ROWS="100, 200, *, 100"> ... </FRAMESET>
```



a janela foi dividida em quatro linhas, tendo a primeira e quarta 100 pixels cada de altura, a segunda 200 pixels e a terceira, o espaço restante.

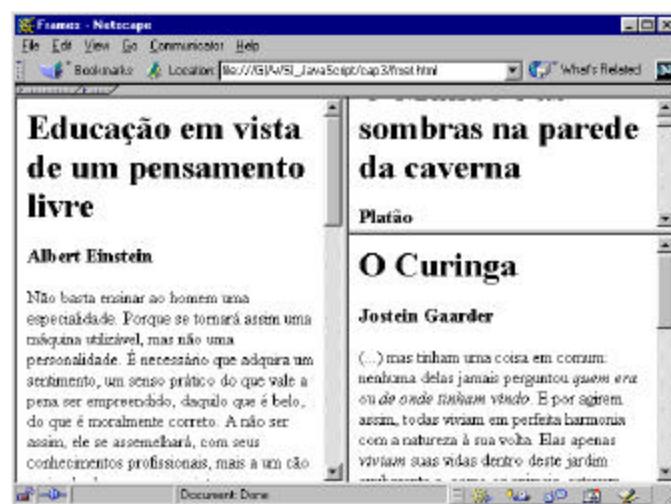
Um bloco `<FRAMESET>...</FRAMESET>` só pode conter dois tipos de elementos:

- descritores `<FRAME>`, que definem a página HTML que ocupará uma janela. A página HTML poderá ser uma página de informação comum ou outra página de frames que dividirá a sub-janela novamente em linhas ou colunas.
- sub-blocos `<FRAMESET> ... </FRAMESET>` que dividirão outra vez a subjanela (em linhas ou colunas) e poderão conter descritores `<FRAME>` e novos sub-blocos `<FRAMESET>`.

O número de sub-blocos para cada `<FRAMESET>` dependerá do número de linhas (ou colunas) definidas. Para dividir uma janela em linhas e colunas ou de forma irregular, pode-se proceder de duas formas:

- usar um único `<FRAMESET>`, contendo elementos `<FRAME>` que referem-se a páginas de frames (páginas que definem um `<FRAMESET>`), ou
- usar vários `<FRAMESET>` em cascata na mesma página.

Usaremos as duas formas para montar a janela ao lado. Na primeira versão, utilizaremos *dois* arquivos de frames: `frset1.html` dividirá a janela principal em duas colunas, e `frset2.html` dividirá a segunda coluna em duas linhas. Na segunda versão, precisaremos de apenas *um* arquivo de frames (`frset.html`). As duas versões utilizarão três arquivos de



informação: `um.html`, `dois.html` e `tres.html`. O resultado final é o mesmo, mas as duas formas podem ser manipuladas de forma diferente em JavaScript.

Na primeira versão temos dois arquivos. Os trechos em negrito indicam as ligações entre eles. O primeiro é `frset1.html`, que referencia uma página de informação:

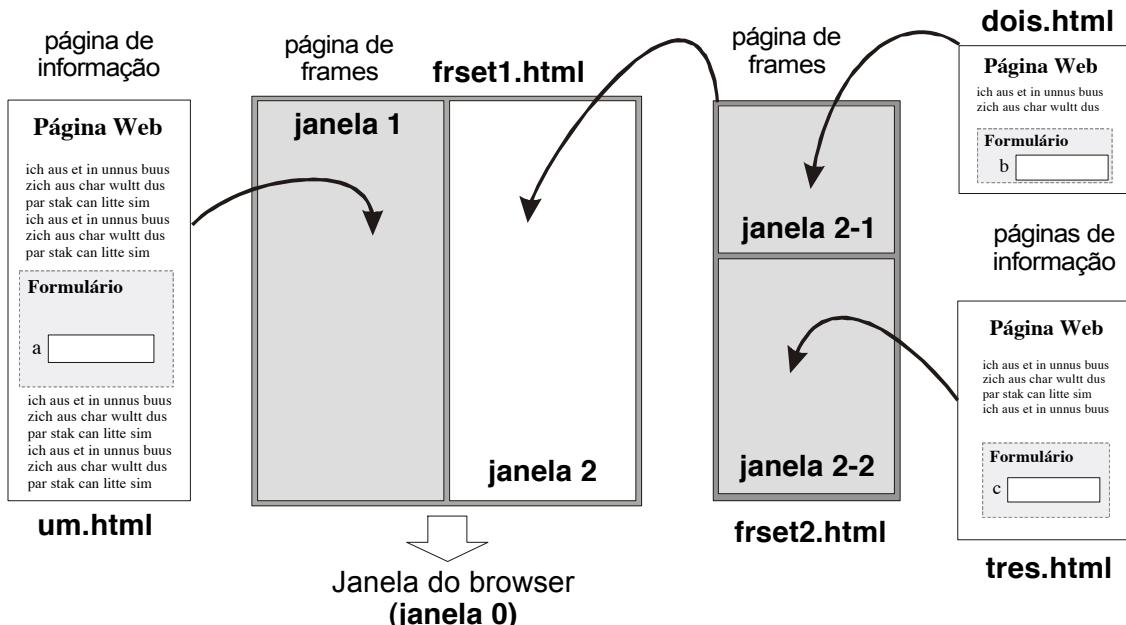
```
<html>
<head> ... </head>

<frameset cols="50%,50%">
    <frame name="janela1" src="um.html">
    <frame name="janela2" src="frset2.html">
</frameset>
</html>
```

e chama `frset2.html`, com mais duas páginas de informação, listado abaixo:

```
<html>
<head> ... </head>
<frameset rows="35%,65%">
    <frame name="janela2_1" src="dois.html">
    <frame name="janela2_2" src="tres.html">
</frameset>
</html>
```

A figura abaixo mostra a organização das páginas de informação e das páginas de frames na janela do browser.



Observe que há três níveis de páginas. No nível mais alto está a página `frset1.html`, que ocupa toda a janela do browser. No segundo nível estão os arquivos `um.html` e `frset2.html`. E no terceiro nível, encontramos os arquivos `dois.html` e `tres.html`.

Na segunda versão, temos apenas um arquivo de frames contendo referências para os três arquivos de informação. Em negrito está mostrado o segundo *frame-set*:

```
<html>
<head> ... </head>
<frameset cols="50%,50%">
    <frame name="janela1" src="um.html">
    <bframeset rows="35%,65%">
        <frame name="janela2_1" src="dois.html">
        <frame name="janela2_2" src="tres.html">
    </bframeset>
</frameset>
</html>
```

Esta segunda versão, possui apenas dois níveis. No primeiro, a página de frames frset.html, no segundo, as páginas de informação. A aparência final é a mesma, nas duas versões, mas na primeira versão há uma janela a mais (*janela2*) que pode ser manipulada em JavaScript e em HTML. Se a *janela2* for utilizada como alvo de um link HTML:

```
<a href="pagina.html" TARGET="janela2"> link </A>
```

os frames *janela2_1* e *janela2_2*, que estão em um nível abaixo de *janela2* deixarão de existir e *página.html* ocupará toda a segunda coluna da janela do browser. Isto não poderá ser feito na segunda versão, pois ela só possui dois níveis.

Se o link estiver dentro da página *dois.html* ou *tres.html*, a sintaxe abaixo, usando o nome especial *_parent* causará um resultado equivalente:

```
<a href="pagina.html" TARGET="_parent"> link </A>
```

Usando frames em JavaScript

Em JavaScript, frames podem ser manipulados por referências (objetos) que indicam relações hierárquicas, posição dos frames ou seus nomes. Toda página de frames possui um vetor **frames** que contém referências para os frames, na ordem em que aparecem no **<FRAMESET>**.

Suponha a seguinte estrutura de frames

```
<html>
<head> ... </head>
<frameset cols="50%,50%">
    <frame name="janela1" src="um.html"<!-- frames[0] --&gt;
    &lt;frameset rows="35%,65%"&gt;
        &lt;frame name="janela2_1" src="dois.html"<!-- frames[1] --&gt;
        &lt;frame name="janela2_2" src="tres.html"<!-- frames[2] --&gt;
    &lt;/frameset&gt;
&lt;/frameset&gt;
&lt;/html&gt;</pre>

```

Um script nesta página pode manipular os seus frames de duas formas: pelo nome ou através do vetor **frames**. O código abaixo mostra duas maneiras diferentes de mudar a cor de fundo das páginas do primeiro e do último frame:

```
frames[0].document.bgColor = "red";
frames[2].document.bgColor = "blue"; // ... é a mesma coisa que...
janela1.document.bgColor = "red";
janela2_2.document.bgColor = "blue";
```

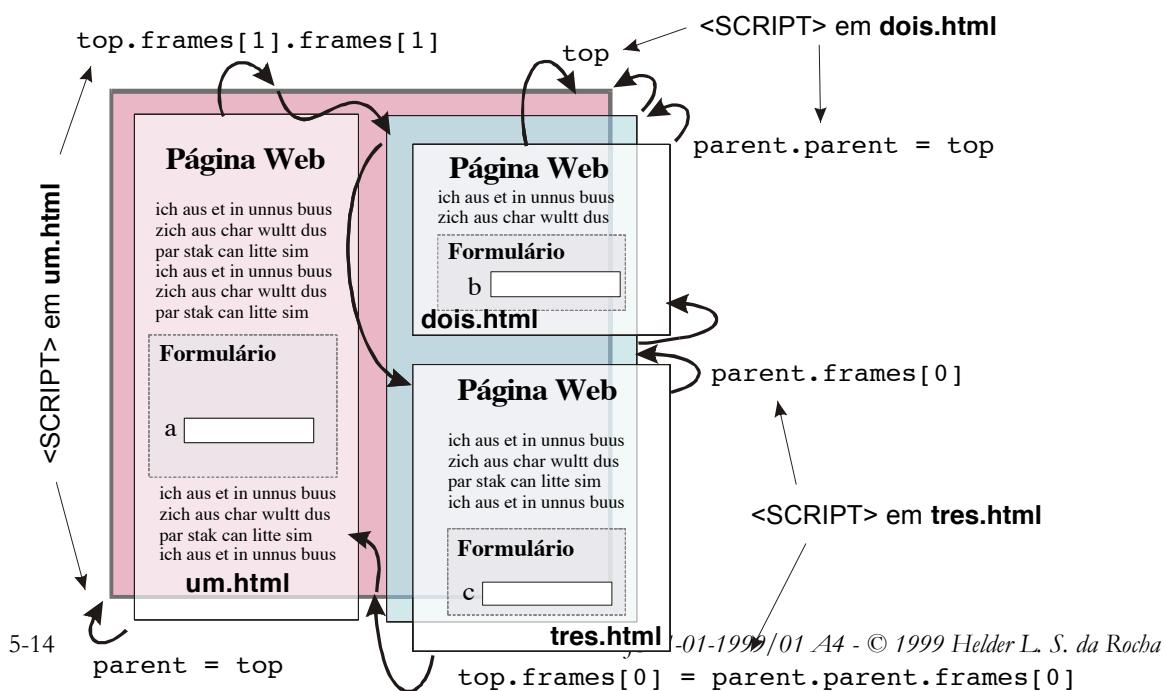
Geralmente não há informação alguma nas páginas de frames, muito menos scripts. O mais comum é existirem scripts nas páginas de informação contidas nos frames. Sendo assim, é necessário haver uma referência para a página que contém o frame. Em JavaScript, esta referência é a propriedade **parent**. Para mudar a cor da página do primeiro frame a partir de um script rodando no último, pode-se fazer:

```
parent.frames[0].document.bgColor = "red"; // ... ou
parent.janela1.document.bgColor = "red";
```

Isto funciona porque **parent** é *Window*, possui a propriedade **frames**, e conhece o nome **janela1**, que está definido no código HTML da página que contém. O código acima não funcionaria se tivéssemos usado a estrutura de frames com três níveis, como o primeiro exemplo da seção anterior. Para ter acesso ao primeiro frame, teríamos que subir dois níveis, até o nível mais alto, para então descer um nível até **frames[0]**. Poderíamos usar **parent** duas vezes ou a propriedade **top**, que representa o nível mais alto:

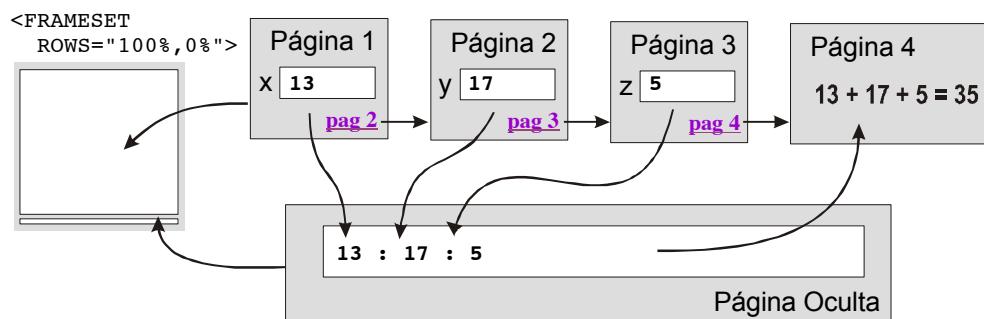
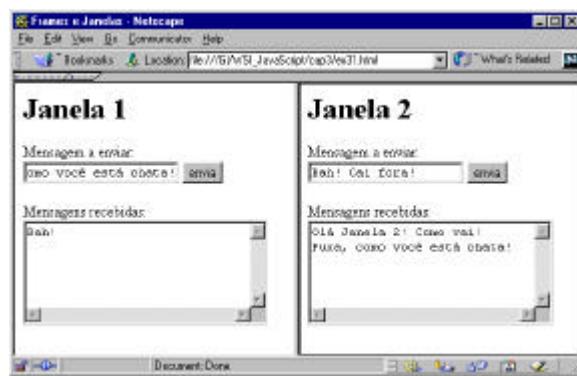
```
parent.parent.frames[0].document.bgColor = "red";
top.janela1.document.bgColor = "red";
```

A partir de **top** pode-se chegar a qualquer frame, usando seu nome ou o vetor **frames**. Nos casos onde existem apenas dois níveis de frames, **top** é sinônimo de **parent**. A figura abaixo mostra várias formas de comunicação entre frames:



Exercícios

- 5.1 Repita o exercício resolvido neste capítulo criando uma página de frames e posicionando as duas janelas nesta estrutura. Altere as páginas de forma que elas possam trocar valores entre frames (veja a figura ao lado).
- 5.2 Este exercício usa frames para passar informações entre páginas. Divida a janela em dois frames, sendo um frame fixo, com altura zero (deverá ficar escondido na parte de baixo da página) e outro, ocupando toda a página. Crie uma página HTML contendo apenas um formulário e um elemento `<textarea>`. Crie mais quatro páginas HTML. A primeira delas deverá ser carregada no frame maior. As três primeiras são idênticas e deverão ter, cada uma, uma caixa de texto, onde o usuário deverá digitar um número, e um link, para a página seguinte. Quando o usuário decidir seguir para a página seguinte, o texto digitado deverá ser copiado para o `<textarea>` da página escondida. Ao chegar na quarta página, esta deverá exibir os números digitados em cada página e a sua soma. (em vez de `<textarea>`, pode-se usar `<input type=hidden>`, e manter os dados temporários invisíveis. Veja o diagrama da aplicação na figura abaixo. Esta é uma forma de passar informações entre páginas sem usar *cookies*.



- 5.3 Use `setTimeout()` e o tipo `Date` para implementar um relógio como o mostrado na figura abaixo. O relógio deverá ser iniciado logo que a página for carregada e atualizado a cada segundo. Implemente um mecanismo para recarregar a página caso o dia mude (use `location.reload()`).



6

O browser

COM O GRANDE NÚMERO DE VERSÕES, PLATAFORMAS E FABRICANTES de browsers, cada um suportando extensões proprietárias e introduzindo recursos incompatíveis, é útil poder indentificar o browser que está carregando uma determinada página. Com esta informação, pode-se tomar a decisão utilizar instruções que só existem naquela versão de browser, ou de redirecionar a janela para outra página. Informações sobre o cliente que acessa uma página são mantidas pela propriedade global `navigator`.

Objeto Navigator

O objeto `Navigator`¹ representa as propriedades do browser. Usando suas propriedades e métodos booleanos (que retornam `true` ou `false`) é possível identificar as possibilidades de um cliente e desenvolver páginas personalizadas com conteúdo específico para aproveitar ao máximo os recursos existentes.

`Navigator` define as características de um único objeto, representado pela propriedade global² `navigator`. Todas as suas propriedades são somente-leitura. Todas as cópias de `navigator` em uma mesma aplicação são idênticas e possuem as mesmas propriedades.

As informações que se pode obter através da propriedade `navigator` são:

- Marca, nome, plataforma e versão do browser do cliente
- Plug-ins instalados no cliente (em browsers Netscape).
- Tipos de dados MIME suportados pelo browser e pela plataforma do cliente, através de plug-ins e programas externos ao browser habilitados a funcionarem como aplicações auxiliares para tipos desconhecidos do browser (Netscape).

¹ Assim como `Window`, não existe na documentação do JavaScript 1.1 um objeto ou construtor chamado ‘Navigator’. Usamos este nome apenas para referirmos ao tipo que define os métodos e propriedades do objeto `navigator` e manter a consistência com os outros objetos.

² No Internet Explorer, `navigator` não é global, mas é propriedade de `window`.

Quatro propriedades contém informações do fabricante do browser, e outras duas, são vetores com objetos do tipo *PlugIn* e *MimeType*, usados para identificar plug-ins e tipos suportados. Não estão disponíveis em todos os browsers. As propriedades de *Navigator* estão listadas na tabela abaixo. Para utilizá-las, é preciso usar o objeto *navigator* da forma:

`navigator.propriedade`

Propriedade	Descrição
<code>userAgent</code>	Contém <i>String</i> . Informação contida no cabeçalho HTTP <i>User-Agent</i> . Esta propriedade é a combinação das propriedades <code>appCodeName</code> e <code>appVersion</code> . <i>Exemplos:</i> Mozilla/4.0 (compatible; MSIE 4.0; Windows 95) Mozilla/4.5 [en] (Win95; I)
<code>appCodeName</code>	Contém <i>String</i> . Contém o nome interno do browser. <i>Exemplo:</i> Mozilla
<code>appVersion</code>	Contém <i>String</i> . Contém informações sobre a versão. <i>Exemplos:</i> 4.0 (compatible; MSIE 4.0; Windows 95) 4.5 [en] (Win95; I)
<code>appName</code>	Contém <i>String</i> . Contém o nome oficial do browser. <i>Exemplos:</i> Microsoft Internet Explorer Netscape
<code>mimeTypes</code>	Contém <i>Array</i> . Um vetor de tipos MIME que descrevem os tipos MIME reconhecidos e suportados pelo browser, internamente, via plug-ins ou através de aplicações auxiliares (do sistema operacional).
<code>plugins</code>	Contém <i>Array</i> . Um vetor com todos os plug-ins instalados no cliente.

Com as propriedades `userAgent`, e `appName`, obtemos diversas informações sobre o browser. Para utilizá-las é preciso isolar o sub-string com a informação correspondente. Infelizmente os formatos diferem entre os principais fabricantes, mas é possível identificar as três informações mais importantes, onde ocorrem as maiores diferenças: nome e fabricante, versão e plataforma.

Identificação do nome do fabricante

O nome e fabricante do produto é fácil de obter. Usa-se a propriedade `appName`. Sabendo-se de antemão o nome utilizado pelo fabricante em cada tipo de browser, é possível usá-lo para comparar com o string contido em `appName`, e verificar se o browser atual é um deles:

```
if (navigator.appName == "Microsoft Internet Explorer") {
    // código que só será executado em browsers Internet Explorer
}
```

Identificação da versão

Obter a versão do browser é uma tarefa mais complicada. Ela está disponível tanto na propriedade `userAgent`, como na propriedade `appVersion`. A versão aqui refere-se ao *browser de referência* “Mozilla”, tanto para Netscape como para o Internet Explorer. O Internet Explorer 3, por exemplo, é compatível com o Netscape Navigator 2.0, portanto a versão que aparece para o Internet Explorer 3 é 2.0 e não 3.0³.

Usar `appVersion` é mais fácil pois a versão está logo no início do string. Tanto nos browsers da Microsoft quanto nos browsers da Netscape a primeira coisa após a versão é um espaço. Portanto, basta identificar o espaço, e recuperar o substring que está antes dele. Eis duas formas de fazer isto:

```
espaco = navigator.appVersion.indexOf(" ");
versao = parseFloat(navigator.appVersion.substring(0, espaco));

versao = parseInt( navigator.appVersion.split(" ")[0] );
```

Se apenas interessa o valor maior da versão, pode-se truncar o que está depois do ponto usando simplesmente:

```
versao = parseInt(navigator.appVersion);
```

Depois de extraída a versão, ela pode ser usada para executar trechos de código dependentes de browser:

```
if (versao < 3.0 && navigator.appName == "Netscape") {
    // código para browsers Netscape de versões inferiores a 3
}
```

Identificação da plataforma

A posição e o tamanho da informação sobre a plataforma diferem nos browsers Internet Explorer e Netscape Navigator. O ideal, portanto, é identificar primeiro o browser, para depois identificar a plataforma. Strings como “Win”, “Mac”, “95” e “NT” estão presentes em ambos os browsers. Pode-se então localizar esses strings em `appVersion`:

```
if (navigator.appVersion.lastIndexOf('Win') != -1) {
    // é Windows... que tipo?
if (navigator.appVersion.lastIndexOf('NT') != -1)
    // é Windows NT
else if (navigator.appVersion.lastIndexOf('Mac') != -1) {
    // é Macintosh
} else {
    // é outra plataforma... Unix talvez
}
```

³ Para obter a versão verdadeira do Internet Explorer, é preciso extrair a informação localizada entre parênteses no meio do string `appVersion` do Internet Explorer (`compatible; MSIE 4.0; Windows 95`)

Exercício Resolvido

Escreva um programa JavaScript que redirecione a janela do browser de acordo com o tipo de browser que a carregar. Para redirecionar, use a instrução:

```
location.href = "url destino";
```

As páginas destino estão localizadas no diretório `cap6/`. Os arquivos destino são:

- a página `msie.html`, se o browser for Microsoft Internet Explorer 4, ou superior
- a página `netscape.html`, se o browser for Netscape 3 ou superior.
- a página `outro.html` se o browser não estiver entre os tipos acima mas suportar JavaScript 1.1.

Se o browser não suportar JavaScript 1.1, deve permanecer na página.

Solução

A solução está mostrada no código a seguir (arquivo `redir.html`) e deve estar em um bloco `<SCRIPT>` no início da página.

```
<html> <head>
<script language="JavaScript1.1">
<!--
    browser = navigator.appName;
    versao = parseInt(navigator.appVersion);
    netscape = "Netscape";
    explorer = "Microsoft Internet Explorer";

    if (browser == netscape && versao >= 3) {
        location.href = "netscape.html";
    } else if (browser == explorer && versao >= 4) {
        location.href = "msie.html";
    } else {
        location.href = "outro.html";
    }
// -->
</script>
</head>
<!-- Somente browsers que não suportam JavaScript 1.1 continuarão -->
(...)
```

Browsers que são exceções (não foram previstos pelo código) sempre devem permanecer na página ou ficar com parâmetros *default* (que não dependam do código) pois sempre existe a possibilidade do browser não entender JavaScript de forma alguma. O bloco `<SCRIPT>` com o atributo `LANGUAGE=JavaScript1.1` garante que browsers que *suportam* versões de JavaScript inferiores a 1.1 não irão tentar interpretar o código.

Métodos

Os métodos de navigator são apenas dois:

Método	Ação
<code>javaEnabled()</code>	Retorna <code>true</code> se o suporte a Java está habilitado.
<code>taintEnabled()</code>	Retorna <code>true</code> se o modelo de segurança data-tainting está habilitado.

Se o usuário não suporta data-tainting (o que é comum), certas operações como a chamada de métodos em outras janelas poderá não funcionar, se contiverem arquivos de servidores diferentes. Se o suporte a Java não estiver ativado, o browser não será capaz de executar applets. Sabendo disso, o programador poderá oferecer uma alternativa.

Suponha, por exemplo, que uma página use formulários HTML para oferecer uma interface de conexão a um serviço. O formulário é simples, consiste de duas caixas de texto (para login e senha) e um botão “Conectar”:

```
<form action="auth.exe?verhtml" method=POST>
<p>User ID <input type=text name=usuario size=14>
    Senha <input type=text name=senha size=14 maxlength=10>
    <input type=submit value="Conectar">
</form>
```

Suponha agora que o autor do site decida substituí-la por um applet, oferecendo a mesma interface, mas aproveitando os recursos de segurança da linguagem Java. No lugar do formulário, a página teria um descritor HTML do tipo:

```
<applet code=PainelCon.class height=80 width=450></applet>
```

Se o usuário não suporta Java, não poderá executar o painel, e o site perderá vários de seus clientes. Se ele voltar a usar somente HTML, deixará de aproveitar os recursos mais modernos presentes nos browsers de um número crescente de clientes, que também usam serviços de seus concorrentes. Uma solução é verificar se o browser do cliente suporta Java e, caso positivo, carregar o applet; caso contrário, carregar o formulário antigo:

```
<script language=JavaScript1.1>      <!--
if (navigator.javaEnabled()) {
    document.write("<applet code=PainelCon.class height=80 width=450>");
    document.write("</applet>");
} else {
    document.write("<form action=\"auth.exe?verhtml\" method=POST>");
    document.write("<p>User ID <input type=text name=usuario size=14>");
    document.write(" Senha <input type=text name=senha size=14>");
    document.write(" <input type=submit value=\"Conectar\">");
    document.write("</form>");
}
//-->
</script>
```

```
<noscript> <!-- browsers que tem JavaScript ativado, ignoram -->
<form action="auth.exe?verhtml" method=POST>
<p>User ID <input type=text name=usuario size=14>
    Senha <input type=text name=senha size=14 maxlength=10>
    <input type=submit value="Conectar">
</form>
</noscript>
</BODY>
```

O código acima só instala o applet se o browser suportar JavaScript e Java e tiver os dois habilitados. Browsers que suportam JavaScript e não suportam Java ou não habilitam Java no browser, interpretarão o bloco `else { ... }`, e receberão um formulário HTML. Browsers que suportam JavaScript, mas estão com o mesmo desabilitado, interpretarão o código em `<noscript>` que também constrói o mesmo formulário HTML. Browsers que não suportam JavaScript, e que ignoram o significado de `<script>` e `<noscript>`, não imprimirão na tela o código em `<script>`, por estar entre comentários HTML (`<-- e -->`) mas *interpretarão* o HTML dentro de `<noscript>` e `</noscript>` por *não* estar comentado.

Plug-ins e tipos MIME

Assim como fizemos com os applets Java, podemos usar JavaScript para verificar se o browser do usuário possui um determinado plug-in instalado. Também é possível verificar se a plataforma do cliente é capaz de executar ou interpretar algum tipo de conteúdo como formatos de imagens, vídeos, textos em Word, arquivos executáveis, etc. identificados através de tipos MIME⁴. A maior parte desses recursos está disponível apenas para browsers Netscape, portanto eles não serão explorados aqui em detalhe.

Dois tipos de objetos são acessíveis através de propriedades do objeto *Navigator* que fornecem essas informações: *PlugIn* e *MimeTypes*. Como pode haver vários plug-ins instalados e geralmente um cliente suporta vários tipos MIME, esses objetos são obtidos através de vetores.

MimeType

Cada objeto do vetor `navigator.mimetypes` é um objeto *MimeType*, que possui propriedades de tipo, descrição, extensões de arquivo e plug-ins habilitados para suportar o tipo. Para obter uma lista de todos os tipos MIME suportados em um browser Netscape Navigator 3.0 ou superior, pode-se fazer:

⁴ Multipart Internet Mail Extension. Padrão Internet para identificação de conteúdo baseado em um par de identificadores que representam um tipo genérico (imagem, texto, aplicação, etc.) e um tipo específico (JPEG, GIF, EXE, HTML, TEX, etc.). O formato é tipo_genérico/tipo_específico. Exemplos: image/jpeg, image/gif, text/html, text/plain, application/msword, application/exe.

```

for (i=0; i < navigator.mimeTypes.length; i++) {
    document.write("<br>" + navigator.mimeTypes[i].type)
}

```

O número de tipos suportados geralmente é extenso, pois não se limita ao browser. Todas as aplicações que registram um tipo de dados no sistema operacional estão disponíveis como aplicações auxiliares e seus tipos MIME são levados em consideração. A melhor forma de ter acesso aos objetos é utilizando o nome do tipo entre colchetes (vetor associativo), em vez do índice:

```
tipo = navigator.mimetypes["tipo/subtipo"];
```

A tabela abaixo lista as propriedades do tipo *MimeType* e a informação que contém. Todas são *read-only*. O exemplo apresentado em cada mostra em negrito os valores de cada propriedade para o tipo `text/html`:

Propriedade	Descrição (e tipo de dados)
<code>name</code>	<i>String</i> . Tipo MIME no formato tipo/subtipo. Exemplo: <code>navigator.mimetypes["text/html"].name</code> contém “ text/html ”
<code>description</code>	<i>String</i> . Descrição em inglês do tipo de conteúdo representado pelo tipo MIME. Exemplo: <code>navigator.mimetypes["text/html"].description</code> contém “ Hypertext Markup Language ”
<code>suffixes</code>	<i>String</i> . Lista de extensões comuns de arquivos associados com este tipo MIME. Exemplo: <code>navigator.mimetypes["text/html"].suffixes</code> contém “ html, htm, shtml ”
<code>enabledPlugin</code>	<i>PlugIn</i> . Referência ao objeto <i>PlugIn</i> que suporta este tipo, ou <code>null</code> se não é suportado por um plug-in (é nativo ou suportado por aplicação externa). Se um tipo é suportado por um plug-in, um arquivo contendo este tipo de dados poderá ser incluído na página através de um descritor <code><EMBED></code> . Exemplo: <code>navigator.mimetypes["text/html"].enabledPlugin</code> contém null

PlugIn

Cada plug-in instalado no browser (Netscape) do cliente é um objeto *PlugIn* que possui propriedades contendo o seu nome, nome de arquivo, descrição e vetor de tipos MIME suportados pelo plug-in. Um vetor de todos os plug-ins é obtido através da propriedade `navigator.plugins`. Se um plug-in existe, ele pode ser incluído na página usando um descritor `<EMBED>`.

As propriedades do tipo *PlugIn* estão relacionadas na tabela abaixo:

Propriedade	Descrição (e tipo de dados)
<code>name</code>	<i>String</i> . Nome em inglês descrevendo o plug-in.
<code>description</code>	<i>String</i> . Descrição detalhada do plug-in.
<code>filename</code>	<i>String</i> . Arquivo do sistema que suporta o plug-in.
<code>mimetypes</code>	<i>Array</i> de <i>MimeType</i> . Vetor com os tipos MIME suportados pelo plug-in.
<code>length</code>	<i>Number</i> . Quantidade de tipos MIME suportados. Mesmo que <code>mimetypes.length</code>

O trecho de código abaixo (arquivo `plugins.html`) pode ser usado para imprimir uma lista com todos os plug-ins instalados em um browser (somente Netscape):

```
<script>
numPlugins = navigator.plugins.length;
document.write("<p>Plug-ins instalados: " + numPlugins);

if (numPlugins > 0) {
    for (i = 0; i < numPlugins; i++) {
        document.write("<p><b>Nome: </b>" + navigator.plugins[i].name);

        document.writeln("<br><b>Arquivo: </b>");
        document.write(navigator.plugins[i].filename);
        document.write("<br><b>Descrição: </b>");
        document.write(navigator.plugins[i].description);
        document.write("<br><b>Qte. de tipos MIME suportados: </b>");
        document.write(navigator.plugins[i].length);
    }
}
</script>
```

Data-tainting

O modelo de segurança do JavaScript impede que programas enviados por um servidor tenham acesso a propriedades de documentos enviados por outro servidor e assim possam utilizar informação privativa. Com esse modelo, é impossível que uma janela leia, por exemplo, o histórico (objeto `window.history`) de outra janela, caso o documento tenha originado de um servidor diferente.

O browsers Netscape 3.0 em diante, suportam um modelo segurança conhecido como *data-tainting*⁵ (marcação de dados). Com ele é possível flexibilizar estas restrições de forma segura, mas isto depende do cliente, que deve ativar o recurso no seu browser, já que ele não é ativado por default. Com o *data-tainting* ativado, uma janela poderá ter acesso a propriedades de outra janela não importando de onde veio o documento, mas o autor da

⁵ taint é mancha, nódoa ou marca.

outra janela pode “manchar” (*taint*) certos valores de propriedades que devem ser mantidas privativas e impedir que essas informações sejam passadas sem a permissão do usuário.

Na prática, data-tainting é pouco útil pois os browsers Netscape não são instalados com o recurso habilitado. Para instalá-lo é preciso que o cliente tome a iniciativa de definir uma variável de ambiente no seu sistema. Os browsers Internet Explorer também não suportam o recurso, embora sejam mais flexíveis em relação à segurança. Um programador pode verificar se data-tainting está habilitado, usando o método `taintEnabled()` de `navigator`:

```
if (navigator.taintEnabled()) {
    // instruções se taint está habilitado
}
```

Se tiver a sorte de encontrar um browser com o recurso ativado, poderá passar propriedades entre janelas, tirando as “manchas” de segurança com `untaint(objeto)` ou evitar que outras propriedades sejam lidas sem autorização com `taint(objeto)`. O objeto passado como argumento não é alterado, mas a função retorna um objeto alterado.

Exercício

- 6.1 Escreva um programa de diagnóstico que imprima na tela um relatório completo sobre o browser do cliente. Primeiro, o programa deverá identificar o fabricante e versão do browser. Se for Netscape 3.0 em diante, deve imprimir uma lista de plug-ins instalados e todos os tipos MIME suportados. Se for outro browser (Netscape 2.0, Internet Explorer ou outro), deve imprimir apenas as informações de fabricante, versão e plataforma.

7

Navegação

CINCO OBJETOS JAVASCRIPT ESTÃO RELACIONADOS COM A NAVEGAÇÃO em hipertexto. Com eles é possível ler e alterar as localidades representadas pelos links, redirecionar as janelas do browser para outras páginas e controlar as informações contidas no histórico de navegação de uma janela.

Area, *Link* e *Anchor* permitem manipular com as propriedades dos elementos HTML `<AREA>`, `<A HREF>` e `<A NAME>` contidos em uma página HTML. Os objetos *History* e *Location* permitem mudar o conteúdo das janelas dinamicamente.

Objeto History

O objeto *History* é um vetor de strings somente-leitura usado por uma janela do browser para armazenar os lugares já visitados durante uma sessão. O conteúdo da lista é o equivalente ao encontrado nas opções “Histórico”, “History” ou “Go” dos browsers Microsoft Internet Explorer e Netscape Navigator. Os botões “Back” ou “Voltar” e “Forward” ou “Avançar” usam as informações no histórico para navegar através dele.

History é uma característica da janela. Todo objeto *Window* possui uma propriedade *history*. Na janela atual, pode ser usado também como uma referência global, usando simplesmente o nome *history*.

As propriedades de *History* são quatro mas apenas uma é utilizável na prática, que é *length*. As outras só são suportadas em browsers Netscape e com várias restrições:

Propriedade	Descrição
<code>length</code>	<i>Number</i> . Contém o número de itens do histórico do browser
<code>current</code>	<i>String</i> . Contém uma string com a URL da página atual.
<code>next</code>	<i>String</i> . Contém uma string com a URL da próxima página do histórico
<code>previous</code>	<i>String</i> . Contém uma string com a URL da página anterior do histórico.

Em JavaScript 1.1, o acesso às propriedades acima, exceto `length`, só é possível se o modelo de segurança data-tainting estiver ativado no browser do cliente¹. Tendo o acesso às URLs do histórico, pode-se redirecionar a janela do browser até qualquer página já visitada, , fazer buscas e imprimir o histórico. Tudo isso pode ser feito de forma simples, sem as restrições do data-tainting, usando os métodos de *History*:

Método	Ação
<code>go(<i>&#pm;n</i>)</code> ou <code>go("string")</code>	Avança ou volta <i>n</i> páginas no histórico. A segunda forma procura no histórico até encontrar a primeira página que tenha a string especificada no título do documento ou nas palavras da sua URL.
<code>back()</code>	Volta uma página no histórico (simula o botão “Back” ou “Voltar” do browser).
<code>forward()</code>	Avança uma página no histórico (simula o botão “Forward” ou “Avançar” do browser).
<code>toString()</code>	Retorna <i>String</i> . Converte o histórico em uma tabela HTML de URLs, cada uma com seu link. Pode ser impressa usando <code>document.write()</code> . Só funciona se o modelo de segurança data-tainting estiver ativado.

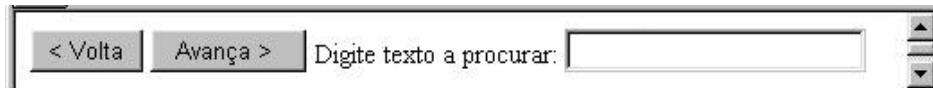
O trecho abaixo simula um painel de navegação em uma janela filha que controla o histórico da janela que a criou² usando métodos de *History*:

```
<form>
<input type=button value="< Volta"
       onclick="opener.history.back()">
<input type=button value="Avança >"
       onclick="opener.history.forward()">
</form>
```



Exercícios

- 7.1 Utilize o painel de navegação apresentado como exemplo nesta seção dentro de um frame. Acrescente um mecanismo de busca no histórico usando o método `go("string")`. Utilize uma caixa de textos (`<input type=text>`) para que o usuário possa entrar com uma palavra para procurar.



¹ O que, na maioria dos casos, não ocorre. Data-tainting (veja capítulo 6) é suportado por browsers Netscape, apenas, e mesmo assim, exige que o usuário habilite a opção em seu browser (não é default).

² O painel de navegação funcionará enquanto o usuário se mantiver no mesmo site (a não ser que data-tainting esteja ativado).

Objeto Location

Location é uma propriedade das janelas que representa a URL cujo documento está sendo exibido³. Todo objeto *Window* possui uma propriedade *location*. As propriedades de *Location* são strings com partes da URL atual. Se forem alteradas, a URL atual irá mudar e o browser tentará imediatamente carregar o recurso localizado pela nova URL na janela atual.

A propriedade mais usada de *Location* é *location.href*, que contém a URL completa. Mudar o valor de *location.href* é uma forma de causar o redirecionamento dinâmico:

```
location.href = "http://www.aeiouy.com/pag2.html"
```

carrega a página localizada por `http://www.aeiouy.com/pag2.html` no momento em que o browser interpretar a linha acima.

Todas as outras propriedades de *location* são substrings do string que contém a URL completa contida na propriedade *href*. Todas podem ser lidas e alteradas:

Propriedade	Descrição
<code>href</code>	A URL completa. Exemplo: <code>http://www.abc.com:80/sub/dir/index.html?name=Cookie1#parte2</code>
<code>protocol</code>	O protocolo da URL. Ex: <code>http:</code>
<code>host</code>	O nome da máquina. Ex: <code>//www.abc.com</code>
<code>port</code>	A porta do servidor. Ex: <code>80</code>
<code>hostname</code>	O nome do servidor. Ex: <code>//www.abc.com:80</code>
<code>pathname</code>	O caminho. Ex: <code>/sub/dir/index.html</code>
<code>hash</code>	O fragmento. Ex: <code>#parte2</code>
<code>search</code>	O string de busca. Ex: <code>?name=Cookie1</code>

Os métodos de *location* são dois: *reload()* é usado para fazer uma página ser recarregada e *replace()* apaga a página anterior do histórico, substituindo-a com uma nova:

Método	Ação
<code>reload()</code> ou <code>reload(true)</code>	Sem argumentos, recarrega a página atual caso não tenha sido modificada. Com o argumento <code>true</code> , carrega a página incondicionalmente.
<code>replace("URL")</code>	Carrega a página especificada pela URL e substitui o registro anterior do histórico com o registro atual.

³ Tem o mesmo significado que o cabeçalho HTTP “Location”

Exercícios

- 7.2 Crie uma “roleta” que jogue o usuário em um site escolhido aleatoriamente a partir de uma lista armazenada em um vetor.
- 7.3 Crie uma janela pequena, sem barra de menus, status, ou scrollbars, para servir de barra de navegação flutuante para um site. Ela deve abrir quando o usuário clicar em um link “SiteMap” e ficar sempre na frente das outras janelas. Todas as URLs das páginas do site devem estar em um componente `<select>` que, ao ter uma opção escolhida pelo usuário, deve fazer com que a janela que o abriu (se ela ainda existir) passe a exibir a nova URL. Se a janela não mais existir, uma nova deverá ser criada.
- 7.4 Usando `setTimeout()` (método de `Window`), escreva uma rotina que faça com que uma página carregue outra que a substitua na janela do browser em 30 segundos.

Objetos Area e Link

Os objetos *Area* e *Link* são a mesma coisa. Ambos representam *vínculos* (referências de hipertexto). O objeto *Area* representa o vínculo acionado a partir de uma imagem mapeada do lado do cliente (client-side *imagemap*) pelo do descritor `<AREA>` e *Link* representa o vínculo criado a partir de um ``.

O acesso a todos os vínculos de um documento é obtido a partir da propriedade `links` de `document`. *Link* e *Area* são objetos do tipo *Location*, mas não são a mesma coisa que a propriedade `location` de `window`. A alteração da propriedade `href` ou qualquer outra, muda a URL à qual o vínculo se refere e não interfere na URL do documento exibido na janela. O efeito da mudança só será notado quando o usuário clicar sobre o vínculo.

Todos os objetos *Link* e *Area* de uma página estão disponíveis através do vetor `document.links`. Eles podem ser acessados pelo índice do vetor correspondente à ordem em que aparecem no código HTML da página ou pelo nome (se houver). Por exemplo, suponha que uma página tenha o seguinte código HTML:

```
<BODY>
<h1><a name="top"></a>Mapa Interativo</h1>

<map name="brasil">
  <area href="norte.html" shape="poly" coords="..." name="n">
  <area href="nordeste.html" shape="poly" coords="..." name="ne">
  <area href="centroeste.html" shape="poly" coords="..." name="co">
  <area href="sudeste.html" shape="poly" coords="..." name="se">
  <area href="sul.html" shape="poly" coords="..." name="s">
</map>
<p align=center>
<a href="index.html">Volta para Revendedores</a>
```

```
<a href="../index.html" name="hp">Home Page</a>
</BODY>
```

Todos os elementos HTML marcados em negrito acima fazem parte do vetor `document.links`. Para saber quantos vínculos *Area* e *Link* existem em uma página, pode-se usar a propriedade `length` do vetor:

```
numLinks = document.links.length; // numLinks contém 7
```

O elemento ``, no código acima, não é um objeto *Link*, pois não contém o atributo `HREF`. É apenas um objeto *Anchor*. *Links* que têm o atributo `NAME` e serão tratados ao mesmo tempo como objetos *Link* e *Anchor*.

Há duas maneiras para fazer com que o objeto *Area*

```
<area href="sudeste.html" shape="poly" coords="..." name="se">
```

do exemplo acima tenha sua URL destino alterada para “`sudeste/index.html`”: através do índice do vetor `links` ou através do nome:

```
document.links[3].href = "sudeste/index.html";
document.se.href = "sudeste/index.html";
```

As propriedades de *Link* e *Area* são praticamente as mesmas de *Location*. A única diferença é a propriedade `target`, que não existe em *Location*.

Propriedade	Descrição
<code>href</code>	A URL completa. Exemplo: <code>http://www.abc.com:80/sub/dir/index.html?name=Cookie1#parte2</code>
<code>protocol</code>	O protocolo da URL. Ex: <code>http</code> :
<code>host</code>	O nome da máquina. Ex: <code>//www.abc.com</code>
<code>port</code>	A porta do servidor. Ex: <code>80</code>
<code>hostname</code>	O nome do servidor. Ex: <code>//www.abc.com:80</code>
<code>pathname</code>	O caminho. Ex: <code>/sub/dir/index.html</code>
<code>hash</code>	O fragmento. Ex: <code>#parte2</code>
<code>search</code>	O string de busca. Ex: <code>?name=Cookie1</code>
<code>target</code>	O nome da janela ou frame onde a URL será exibida.

Eventos

Não há métodos definidos para os objetos *Link* e *Area*. Existem, porém, três eventos manuseados por atributos dos elementos HTML que representam esses objetos. Os dois primeiros atributos aplicam-se tanto a elementos `<A HREF>` como a elementos `<AREA>`:

- `ONMOUSEOVER` – quando o usuário move o mouse sobre o vínculo ou imagem.
- `ONMOUSEOUT` – quando o usuário afasta o mouse que antes estava sobre o vínculo ou imagem.

O terceiro, só produz efeito em elementos <A>. É ignorado em elementos <AREA>:

- ONCLICK – quando o usuário clica o mouse sobre o vínculo.

Todos os eventos são tratados *antes* que o browser siga o vínculo do atributo HREF, por exemplo, no código abaixo, a URL no atributo HREF do vínculo abaixo nunca será carregada pois a janela será redirecionada para outra localidade assim que o usuário passar o mouse sobre o link:

```
<a href="http://www.sao.nunca.org" onmouseover="http://www.eh.aqui.com">  
    Não chegue perto deste link!  </a>
```

Objeto Anchor

O objeto *Anchor* representa uma âncora fixa. Âncoras podem ser referenciadas como URLs destino localizando partes de um documento. Em HTML, qualquer elemento <A> que tiver um atributo NAME pode ser usado como âncora:

```
<a name="aqui"></a>
```

A âncora não precisa conter texto. Marca uma posição que pode ser localizada a partir de um vínculo local à página ou não. Dentro da página, pode-se criar um link para a âncora usando:

```
<a href="#aqui">Rolar a página até chegar lá</a>
```

Em páginas externas, o fragmento “#aqui” deve ser acrescentado ao link, logo após o nome do arquivo. No trecho de código abaixo, há dois objetos *Anchor*, destacados em negrito:

```
<BODY>  
<h1><a name="top"></a>Mapa Interativo</h1>  
(...)  
<p align=center>  
<a href="index.html">Volta para Revendedores</a>  
<a href="..//index.html" name="hp">Home Page</a>  
</BODY>
```

Todas as âncoras de uma página estão na propriedade anchors, de document. Para saber quantos objetos *Anchor* existem em uma página, pode-se usar sua propriedade length:

```
numAncoras = document.anchors.length;           // numAncoras contém 2
```

O primeiro <A> do código é *Anchor* e não é *Link* porque não tem o atributo HREF. O segundo é *Link* e não é *Anchor*, porque não têm o atributo NAME. O terceiro é ao mesmo tempo um *Link* e um *Anchor* e aparece tanto no vetor links como no vetor anchors.

Objetos *Anchor* podem ser referenciados pelo nome ou pelo índice do vetor correspondente à ordem em que aparecem no código. As formas abaixo são equivalentes:

```
location.href = "#" + document.anchors[1].name;
location.href = "#" + document.hp.name;
```

Objetos *Anchor* não possuem métodos ou eventos associados. Têm apenas uma propriedade *read-only*:

Propriedade	Descrição
name	Nome da âncora (texto que está no seu atributo NAME do HTML)

Exercícios

- 7.5 Uma grande página HTML contém um glossário, organizado em ordem alfabética. No início da lista de palavras de cada letra há uma âncora do tipo:

```
<h2><a name="M"></a> M </h2>
```

Escreva um programa JavaScript que construa, no final da página, uma tabela HTML com links para todas as âncoras contidas na página.

- 7.6 Uma página possui 5 vínculos para páginas de um site. São páginas dependentes de browser. Se o browser for Netscape, os vínculos devem apontar para páginas localizadas em um subdiretório `./netscape/`. Se for Microsoft, devem apontar para um subdiretório `./microsoft/`. Se for outro browser, as páginas devem ser encontradas no diretório atual (`./`). Use JavaScript para identificar o browser e alterar as propriedades de *todos os links existentes* para que carreguem as páginas corretas quando o usuário as requisitar.

8

A página HTML

AS PROPRIEDADES DE UMA PÁGINA HTML incluem seus elementos, representados pelos descritores HTML, atributos como cor de fundo ou cor dos links, e informações enviadas pelo servidor como cookies, URL, referenciador e data da última modificação. Todas essas propriedades são acessíveis através de JavaScript, e várias podem ser alteradas.

Além de permitir o acesso às propriedades, JavaScript também define vários métodos para gerar HTML e criar páginas novas, em tempo de exibição.

A única forma de ter acesso a uma página é através da propriedade `document`, que qualquer objeto do tipo *Window* possui. A página da janela onde roda o script pode ser acessada diretamente pela propriedade `window.document`, ou simplesmente `document`. Esta propriedade possui métodos e propriedades definidos pelo tipo *Document*, apresentado neste capítulo.

Objeto Document

O objeto `document` representa o documento HTML atual. `document` é uma propriedade de `window` e, portanto, pode ser usado sem fazer referência a `window`:

```
window.document      // ou simplesmente document
```

Para ter acesso a páginas de outras janelas, é preciso citar a referência *Window* que possui a propriedade `document`:

```
janela = open("nova.html");
janela.document.bgColor = "green";
parent.fr1.fr1_2.document.forms[0].b1.click();
```

No restante deste capítulo, usaremos `document` (e não `window.document`), como fizemos com as propriedades de `window` nos capítulos anteriores, para se referir à página da janela atual.

As propriedades do tipo *Document* estão listadas na tabela abaixo, indicando quais as propriedades que podem ser alteradas, e o tipo de dados que contém.

Propriedade	Acesso	Função
<code>bgColor</code>	<code>read / write</code>	Contém <i>String</i> . Define ou recupera cor de fundo da página. Pode ser um string contendo código hexadecimal do tipo #rrggbb ou nome da cor (red, blue, navy, etc.)
<code>fgColor</code>	<code>r / w</code>	Contém <i>String</i> . Define ou recupera cor do texto na página.
<code>linkColor</code>	<code>r / w</code>	Contém <i>String</i> . Define ou recupera cor de links na página.
<code>alinkColor</code>	<code>r / w</code>	Contém <i>String</i> . Define ou recupera cor de links ativos.
<code>vlinkColor</code>	<code>r / w</code>	Contém <i>String</i> . Define ou recupera cor de links visitados.
<code>title</code>	<code>r</code>	Contém <i>String</i> . Recupera o título (<TITLE>) do documento.
<code>links</code>	<code>r</code>	Contém <i>Array</i> de objetos <i>Link</i> . Para obter a quantidade de links <A HREF> no documento: <code>document.links.length</code>
<code>applets</code>	<code>r</code>	Contém <i>Array</i> de objetos <i>Applet</i> . Para obter a quantidade de applets <APPLET> no documento: <code>document.applets.length</code>
<code>anchors</code>	<code>r</code>	Contém <i>Array</i> de objetos <i>Anchor</i> . Para obter a quantidade de âncoras <A NAME> no documento: <code>document.anchors.length</code>
<code>embeds</code>	<code>r</code>	Contém <i>Array</i> de objetos <i>PlugIn</i> . Para obter a quantidade de plugins <EMBED> no documento: <code>document.plugins.length</code>
<code>plugins</code>	<code>r</code>	Contém <i>Array</i> de objetos <i>PlugIn</i> . Mesma coisa que <code>embeds</code>
<code>images</code>	<code>r</code>	Contém <i>Array</i> de objetos <i>Image</i> . Para obter a quantidade de imagens no documento: <code>document.images.length</code>
<code>location</code>	<code>r</code>	Contém <i>String</i> com URL do documento.
<code>URL</code>	<code>r</code>	Mesma coisa que <code>location</code> .
<code>referrer</code>	<code>r</code>	Contém <i>String</i> com URL do documento que contém um link para o documento atual.
<code>lastModified</code>	<code>r</code>	Contém <i>String</i> . A string recebida informa a data da última modificação do arquivo. Está no formato de data do sistema. Pode ser convertida usando <code>Date.parse()</code> e transformada em objeto ou automaticamente em <i>String</i> .
<code>domain</code>	<code>r / w</code>	Contém <i>String</i> com o domínio dos arquivos referenciados.
<code>cookie</code>	<code>r / w</code>	Contém <i>String</i> . Usado para ler e armazenar preferencias do usuário no computador do cliente.

As propriedades `bgColor`, `fgColor`, `linkColor`, `vlinkColor` e `alinkColor` alteram a aparência da página. Correspondem aos atributos `BGCOLOR`, `TEXT`, `LINK`, `VLINK` e `ALINK` do descriptor HTML <BODY>, respectivamente. Existem desde as primeiras versões do JavaScript, mas só podiam ser alteradas antes que a página fosse montada. Nos browsers modernos, é possível mudar essas propriedades em tempo de exibição. Pode-se ter, por exemplo, um link que ‘apaga a luz’ quando o mouse passa sobre ele:

```
<a href="..."  
onmouseover="document.bgColor='black'"  
onmouseout=""document.bgColor='white'"> Não se aproxime! </a>
```

As seis últimas propriedades da lista acima possibilitam a obtenção de informações do arquivo HTML e do domínio onde reside. Uma das propriedades mais úteis de `document` é `lastModified`, que retorna a data de última modificação do arquivo. Para imprimi-la na página:

```
document.write("<p>Última modificação: " + document.lastModified);
```

As propriedades `location`, `domain` e `referrer` contém informações sobre a localização da página. A propriedade `document.referrer` contém a URL da página que contém um link para o documento atual. `document.referrer` pode apresentar o valor `null` se a carga da página não foi resultante da seleção de um link:

```
<p>Você acaba de vir de  
<script language=JavaScript>  
    document.write(document.referrer);  
</script>. Seja bem vindo!
```

A propriedade `document.domain` representa o domínio da página atual. É a mesma informação que existe em um `<BASE HREF="url">`. `document.location` representa o endereço da página atual. É útil quando se precisa gerar uma página nova com um link para a página atual (que a criou).

Métodos

Os métodos de `Document` são usados principalmente para gerar HTML e criar novas páginas em tempo de exibição e de carga. Os métodos tanto podem ser aplicados na janela atual ou em outras janelas de forma a gerar documentos novos.

Método	Ação
<code>write("string")</code> ou <code>writeln()</code> <code>write("arg1", "arg2", ... , "argn")</code>	Recebe e concatena zero ou mais argumentos separados por vírgulas e os escreve na página atual.
<code>open()</code> ou <code>open("tipo/subtipo")</code>	Abre um canal de fluxo de dados para <code>document</code> de forma que as chamadas <code>document.write()</code> subsequentes possam acrescentar dados ao documento. O <code>tipo/subtipo</code> é por <i>default</i> <code>text/html</code> .
<code>close()</code>	Imprime na página qualquer texto não impresso enviado à <code>document</code> e fecha o fluxo de saída de dados.
<code>clear()</code>	limpa a janela ou frame que contém <code>document</code> .

O método `write()` é provavelmente o método mais usado em JavaScript. Sua principal aplicação é a geração de HTML durante a carga da página. Uma chamada ao

método `write()`, depois que a carga e exibição da página foi concluída, não funciona, pois o canal de gravação do arquivo já foi fechado. Mas o canal pode ser reaberto com uma chamada à `document.open()`:

```
document.open();
document.write("Esta é a última linha do arquivo");
document.close();
```

A linha `document.close()` é essencial para que o texto seja exibido. Uma chamada ao método `document.clear()`, depois de um `document.open()`, limpa a tela, permitindo que o texto apareça no início da tela.

Com exceção de `write()`, porém, há poucas aplicações para os outros métodos no documento atual. Imprimir no final do arquivo não é a melhor forma de criar páginas dinâmicas. As aplicações mais interessantes são a geração dinâmica de páginas novas. Mostraremos, na seção a seguir, um exemplo de geração de páginas *on-the-fly*.

Geração de páginas *on-the-fly*

A melhor forma de gerar uma nova página é começar com uma nova janela. Através da referência da nova janela, pode-se abrir então um fluxo de dados para escrever nela.

O primeiro passo, portanto, é criar a nova janela:

```
w1 = open(""); // abre janela vazia, sem arquivo
```

Em seguida, abrir o documento para que possa receber dados enviados por instruções `write()`, posteriores:

```
w1.document.open(); // abre documento para gravação
w1.document.write("<html><head>\n<title>Nova Página</title>\n");
w1.document.write("</head>\n<body bgcolor=white>\n");
w1.document.write("<h1 align=center>Nova Página</h1>");
```

Para que o documento completo seja exibido, é preciso que o fluxo de dados seja fechado, depois que todos as instruções `write()` tenham sido enviadas. Quando isto ocorre, quaisquer linhas que estejam na fila são impressas, e o documento é fechado.

```
w1.document.close(); // imprime documento na nova janela
```

Ao se visualizar o código-fonte do documento gerado, encontra-se:

```
<html><head>
<title>Nova Página</title>
</head>
<body bgcolor=white>
<h1 align=center>Nova Página</h1>
(...)
```

O exercício resolvido a seguir apresenta um exemplo completo. Veja mais exemplos no diretório `cap8/`.

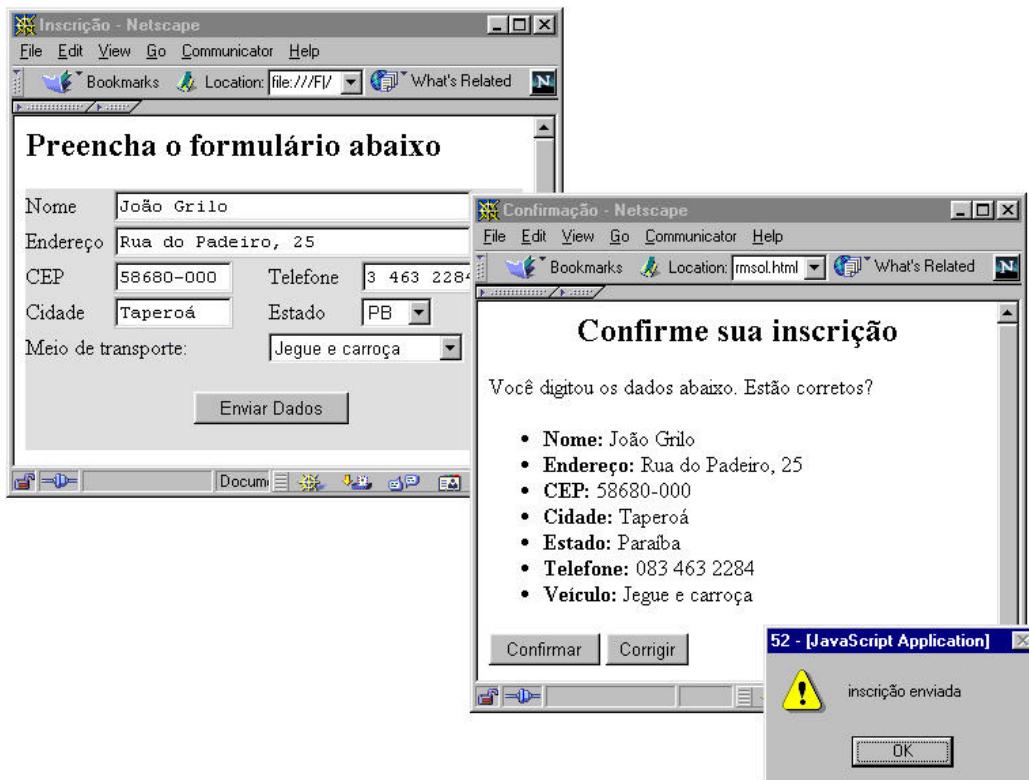
Exercício Resolvido

Este exercício consiste na construção de uma janela de confirmação para dados digitados em um formulário, antes do envio ao servidor, permitindo que o usuário verifique se os dados que selecionou estão corretos antes de enviá-los.

Crie um formulário como o mostrado na figura (use o esqueleto `form.html` disponível). Quando o usuário apertar o botão “Enviar Dados”, uma nova janela deverá ser criada contendo a lista das informações que ele selecionou e dois botões “Corrigir” e “Confirmar”.

Caso o usuário decida fazer uma correção, deve apertar o botão “Corrigir” que tornará ativa (`focus()`) a janela do formulário. O usuário pode então alterar quaisquer informações e enviar novamente. Após novo envio, a janela de confirmação deverá ser re-ativada ou criada caso tenha sido fechada (verifique se é preciso criar a janela ou não).

Caso o usuário confirme, os dados devem ser enviados para o servidor. Os dados digitados na primeira janela, portanto, devem estar presentes também na segunda janela em campos de formulário (`<hidden>`) para que o programa CGI no servidor possa recuperá-los. Como este exemplo é apenas uma simulação, o envio pode ser sinalizado através de uma janela de alerta.



A solução está na página seguinte.

Solução

O primeiro passo é a criação do formulário. Nós já fornecemos um esqueleto com um formulário (`cap8/form.html`) pronto. Precisaremos utilizar os nomes dos campos (atributo `NAME`) e saber o tipo de dispositivo de entrada usado. A listagem abaixo destaca os nomes dos campos e seus tipos em negrito:

```

<form>
  (...)

  <td>Nome</td>      <td colspan=3><input type=text name=Nome></td><tr>
  <td>Endereço</td> <td colspan=3><input type=text name=Endereco></td></tr>
  <tr><td>CEP</td>   <td><input type=text name=CEP size=10></td>
  <td>Telefone</td>  <td><input type=text name=Telefone></td></tr><tr>
  <td>Cidade</td>   <td><input type=text name=Cidade size=10></td>
  <td>Estado</td>    <td><select name=UF size=1 >
                        <option value="Acre">          AC </option>
                        <option value="Alagoas">        AL </option>
  (...)

                        <option value="Santa Catarina"> SC </option>
                        <option value="Tocantins">       TO </option>
                    </select></td></tr>
  <tr><td colspan=2>Meio de transporte:</td>
      <td colspan=2><select name=Veiculobutton value="Enviar Dados" onclick="enviar(this.form)">
  <br>&nbsp;</td></tr></table>
</form>
```

No nosso formulário só há dois tipos de dispositivos de entrada: caixas de texto (`<input type=text>`) e caixas de seleção (`<select>`). No evento `onclick` do botão, um método `enviar()` é chamado, que passa o próprio formulário como argumento (para que possamos, dentro da função, referenciar o formulário através de uma variável local).

Para ler um valor da caixa de texto, utilizamos a propriedade `value`, do objeto `Text`. A leitura dos valores de objetos `Select` é mais complicada, pois as informações são armazenadas em objetos `Option`, de duas formas diferentes. O tipo `Select` têm uma propriedade `options`, que retorna um `Array` de objetos `Option`. Outra propriedade, `selectedIndex`, retorna o índice atualmente selecionado (veja mais sobre `Select` e `Options` no capítulo 10). Combinando os dois, obtemos o item selecionado. Veja como obter o ítem selecionado pelo usuário na caixa de seleção `UF`:

```
indice = document.forms[0].UF.selectedIndex; // obtém índice
item = document.UF.options[indice];
```

As informações do item selecionado podem estar em dois lugares:

- entre <option> e </option> (siglas dos estados) e
- no atributo **VALUE** de cada <option> (nomes dos estados por extenso)

O primeiro valor é obtido através da propriedade **text**, o segundo, através da propriedade **value**, por exemplo:

```
sigla = document.UF.options[indice].text; // PR, BA, AC
estado = document.UF.options[indice].value; // Paraná, Bahia, Acre
```

Para evitar escrever sempre este longo *string* todas as vezes que uma variável do formulário for usada, criamos novas variáveis dentro da função **enviar()**:

```
function enviar(dados) { // dados = document.forms[0]
    nome      = dados.Nome.value;
    endereco = dados.Endereco.value;
    cep       = dados.CEP.value;
    cidade   = dados.Cidade.value;
    uf        = dados.UF.options[dados.UF.selectedIndex].value;
    tel       = dados.Telefone.value;
    carro    = dados.Veiculo.options[dados.Opcão_1.selectedIndex].text;
    (... )
}
```

Antes de gerar uma nova página, o ideal é criá-la da forma convencional, diretamente em HTML como uma página estática. Desta forma, é fácil verificar sua aparência e até acrescentar mais recursos visuais. Nesta versão temporária, preenchemos os espaços onde estarão os dados a serem gerados dinamicamente com os nomes das variáveis (em negrito). Observe que eles também são enviados em campos <hidden>, para que possam ser repassados ao servidor (programa CGI). Esta é a página que queremos gerar:

```
<HTML>
<HEAD>
<TITLE>Confirmação</TITLE>
</HEAD>
<BODY bgcolor=white>
<H2 align=center>Confirme sua inscrição</H2>
<FORM action="/cgi-local/inscricao.pl"> <!-- programa CGI -->
<P>Você digitou os dados abaixo. Estão corretos?
<UL>
<LI><B>Nome:</B> nome </LI>
<LI><B>Endereço:</B> endereco </LI>
<LI><B>CEP:</B> cep </LI>
<LI><B>Cidade:</B> cidade </LI>
<LI><B>Estado:</B> estado </LI>
```

```

<LI><B>Telefone:</B> telefone </LI>
<LI><B>Veículo:</B> carro </LI>
</UL>
<!-- Campos necessarios para enviar os dados ao servidor -->
<INPUT type=hidden name=Nome value="nome">
<INPUT type=hidden name=Endereco value="endereco">
<INPUT type=hidden name=CEP value="cep">
<INPUT type=hidden name=Cidade value="cidade">
<INPUT type=hidden name=Estado value="estado">
<INPUT type=hidden name=Telefone value="telefone">
<INPUT type=hidden name=Veiculo value="carro">
<P>
<INPUT type=button onclick="alert('inscrição enviada')"
           value="Confirmar">
<INPUT type=button value="Corrigir"
           onclick="opener.focus()">
</FORM>
</BODY></HTML>

```

Agora é só colocar todo o código acima dentro de instruções `document.write()`, isolando as variáveis. As aspas precisam ser precedidas de um escape (\") para que possam ser impressas. Primeiro precisamos abrir uma nova janela:

```

if (w1 != null) { // verifica se a janela já está aberta!
    w1.focus(); // ... se estiver... simplesmente ative-a
} else w1 = open("", "janconf"); // ... se não, abra uma nova!

```

Depois abrimos um canal para escrever no documento da janela (`w1.document`), através do método `open()` e enviamos uma seqüência de instruções `write()`. Usamos as variáveis definidas antes para passar os dados à nova página:

```

w1.document.open();
w1.document.write("<html><head>\n<title>Confirmação</title>\n");
w1.document.write("</head>\n<body bgcolor=white>\n");
w1.document.write("<h2 align=center>Confirme sua inscrição</h2>\n");
w1.document.write("<form action=\""/cgi-local/inscricao.pl\">\n");
w1.document.write("<p>Você digitou os dados abaixo. Estão corretos?");
w1.document.write("<ul><li><b>Nome:</b> " + nome + "</li>\n");
w1.document.write("<li><b>Endereço:</b> " + endereco + "</li>\n");
w1.document.write("<li><b>CEP:</b> " + cep + "</li>\n");
w1.document.write("<li><b>Cidade:</b> " + cidade + "</li>\n");
w1.document.write("<li><b>Estado:</b> " + uf + "</li>\n");
w1.document.write("<li><b>Telefone:</b> " + tel + "</li>\n");
w1.document.write("<li><b>Veículo:</b> " + carro + "</li></ul>\n");

w1.document.write("<input type=hidden name=Nome value=\"" + nome + "\">\n");
w1.document.write("<input type=hidden name=Endereco value=\"" + endereco + "\">\n");
w1.document.write("<input type=hidden name=CEP value=\"" + cep + "\">\n");

```

```
w1.document.write("<input type=hidden name=Cidade value=\"\" + cidade + \">\n");
w1.document.write("<input type=hidden name=Estado value=\"\" + uf + \">\n");
w1.document.write("<input type=hidden name=Telefone value=\"\" + tel + \">\n");
w1.document.write("<input type=hidden name=Veiculo value=\"\" + carro + \">\n");

w1.document.write("<p><input type=button onclick=\"alert('inscrição enviada')\""
                 "value=\"Confirmar\">\n");
w1.document.write("<input type=button value=\"Corrigir\""
                 "onclick=\"opener.focus();\">\n");
w1.document.write("</form></body></html>\n");

w1.document.close();
w1.focus();

}
```

A solução com o código completo da página acima está em `cap8/formsol.html`. Veja também outros exemplos, como `job_form1.html`, no mesmo diretório.

Eventos

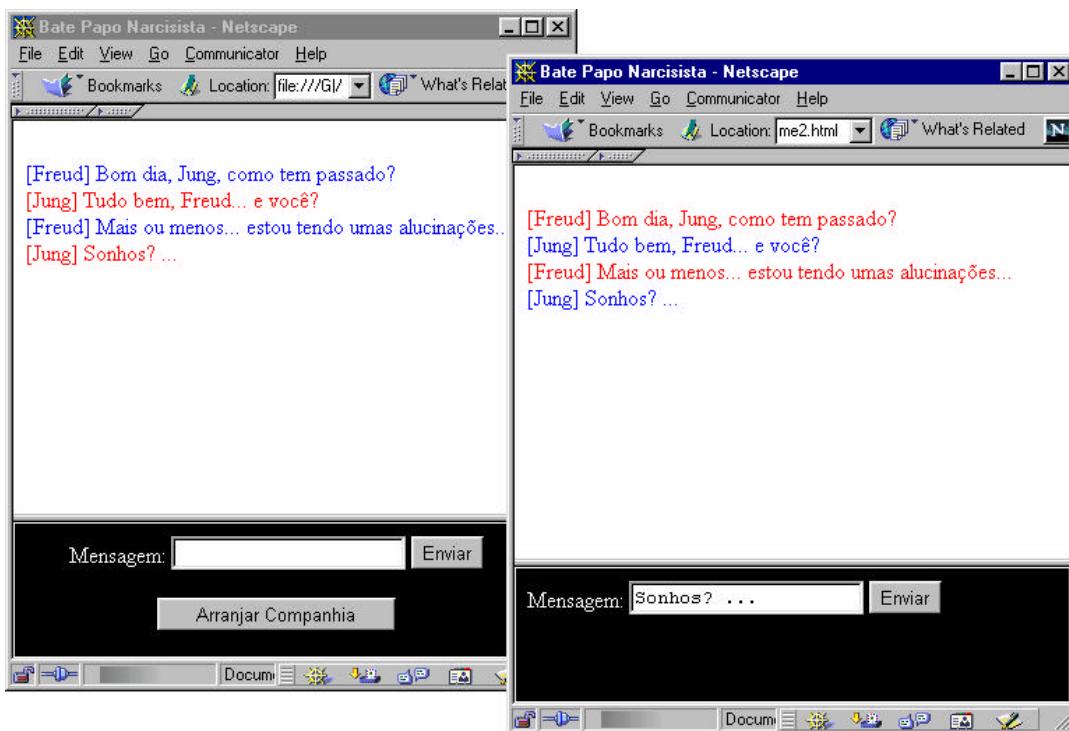
Os eventos do JavaScript que afetam a página também afetam as janelas. Estes eventos são chamados a partir dos atributos HTML listados abaixo, e são aplicáveis aos descritores `<BODY>` e `<FRAME>`. Já os vimos no capítulo anterior:

- `ONLOAD` – depois que a página é totalmente carregada
- `ONUNLOAD` – antes que a página seja substituída por outra

Exercícios

- 8.1 Escreva uma aplicação onde o usuário possa escolher três cores de uma lista (use um `<select>` ou `<input type=radio>`): uma cor de fundo, uma cor do texto e uma cor do link. Após os seletores, o documento deve conter um título, alguns parágrafos e links. Depois que o usuário clicar um botão “Visualizar”, a mesma página deve ser recarregada mostrando a combinação que ele escolheu.
- 8.2 Escreva uma aplicação que, através de uma interface de formulários, permita que o usuário monte uma página HTML sem precisar saber HTML. O usuário deverá poder escolher o título, dois subtítulos e os textos de duas seções. Também deve poder escolher cores (veja exercício anterior). A página deve oferecer duas opções:
 - Um botão de “preview”, que irá abrir uma nova janela com o resultado esperado.

- Um botão “gerar página”, que irá colocar o código HTML correspondente à escolha do usuário em um <textarea>, para que possa copiar e colar em um arquivo de texto.
- 8.3 Escreva uma aplicação de ‘bate-papo narcisista’, utilizando duas janelas de browser com frames. A parte superior de cada janela é um documento em branco (use **blank.html**, no diretório **cap8/**) que irá crescer a medida em que linhas de texto são enviadas a partir dos frames inferiores de cada janela. Cada vez que uma mensagem for digitada no campo de texto do frame inferior e o botão “Enviar” apertado, ela deve aparecer na parte superior de ambas as janelas. A primeira janela deverá ter um botão “Arranjar Companhia” para que outra janela seja criada, tornando a conversa possível. Veja as figuras abaixo. A solução está no diretório **cap8/** (veja no arquivo **papoframe1.html**)



9

Imagens

EM JAVASCRIPT, É POSSÍVEL MANIPULAR COM AS IMAGENS DE UMA PÁGINA, alterando a URL que localiza o arquivo de imagem. Assim, pode-se trocar a imagem que está sendo exibida por outra durante a exibição da página. Também é possível criar novos objetos representando imagens que inicialmente não aparecem na página e transferir seus arquivos previamente em *background*, para que estejam disponíveis na memória na hora da substituição. Com esses recursos, pode-se incrementar a página com recursos dinâmicos, como ícones que mudam de aparência quando ocorre um evento, animações e *banners*.

As imagens utilizadas em JavaScript podem ser carregadas de duas formas: através do HTML e através de instruções JavaScript. As *imagens estáticas*, fornecidas pela página HTML através do descritor ``, são representadas como objetos da página (`document`), acessíveis através da sua propriedade `images`: um vetor que contém referências para todas as imagens do documento. As *imagens dinâmicas*, que não são fornecidas pelo HTML, podem ser criadas como objetos JavaScript dentro de qualquer bloco `<SCRIPT>` ou atributo HTML de eventos usando o operador ‘`new`’ e o construtor `Image()`.

Neste capítulo, conheceremos as duas formas de manipular imagens em JavaScript, e como utilizá-las para criar páginas dinâmicas eficientes.

Image

Tanto uma imagem visível em uma página HTML como uma imagem carregada na memória, porém invisível, podem ser representadas em JavaScript por um objeto do tipo `Image..`. Para criar uma referência para uma imagem que não existe na página, é preciso usar `new`:

```
figura5 = new Image(50, 100);
```

onde os números passados como parâmetros (opcionais) correspondem respectivamente à largura e altura da imagem na página em pixels. Pode-se também usar:

```
figura6 = new Image()
```

que calculará o tamanho da imagem quando ela for carregada.

Depois que um objeto do tipo *Image* é criado, e suas dimensões definidas, seu tamanho não pode mais ser alterado. Todas as imagens passadas para a referência **figura5** abaixo serão redimensionadas para 50x100 pixels:

```
figura5.src = "square.gif"; // square.gif agora tem 50x100 pixels
```

A propriedade **src** tem a mesma função do atributo **SRC** do descritor HTML ****: indicar a URL do arquivo-fonte da imagem.

Toda página que possui o descritor HTML **** já possui um objeto *Image* que pode ser manipulado através da sua propriedade **document.images** (do tipo *Array*). Para criar uma nova imagem no documento, é preciso usar HTML e o descritor ****, cuja sintaxe geral está mostrada abaixo:

```
<IMG SRC="URL do arquivo-fonte da imagem"
      NAME="nome_do_objeto"
      ALT="texto alternativo (descrição da imagem)"
      LOWSRC="URL de arquivo-fonte de baixa-resolução"
      HEIGHT="altura em pixels"
      WIDTH="largura em pixels"
      HSPACE="margens externas laterais em pixels"
      VSPACE="margens externas verticais em pixels"
      BORDER="largura da borda de contorno em pixels"
      ALIGN="left" ou "right" ou "top" ou "middle" ou "bottom" ou
            "texttop" ou "absmiddle" ou "absbottom" ou "baseline"
      ISMAP           <!-- é imagem mapeada do lado do servidor -->
      USEMAP="#mapa" <!-- é imagem mapeada por 'mapa' no cliente -->
      ONABORT="Código JavaScript"
      ONERROR="Código JavaScript"
      ONLOAD="Código JavaScript" >
```

Todos os atributos, com exceção de **SRC**, são opcionais. Para manipular uma imagem do HTML em JavaScript, é preciso usar o vetor **images** que contém referências para cada uma das imagens do documento, na ordem em que elas aparecem no código HTML:

```
prima = document.images[0]; // primeira imagem da página atual
nona = document.images[8]; // nona imagem da página atual
```

Assim como formulários e *frames*, que são acessíveis através de vetores ou nomes, as imagens podem receber um nome, para tornar o seu acesso mais fácil. O atributo HTML opcional **NAME**, se presente, pode ser usado pelo JavaScript para fazer referência à imagem, em vez de usar o vetor **images**. É uma boa idéia, pois torna o código mais legível e independente da ordem e número de imagens na página. Por exemplo, a imagem:

```

```

pode ser referenciada do JavaScript da forma:

```
prima = document.anota;
```

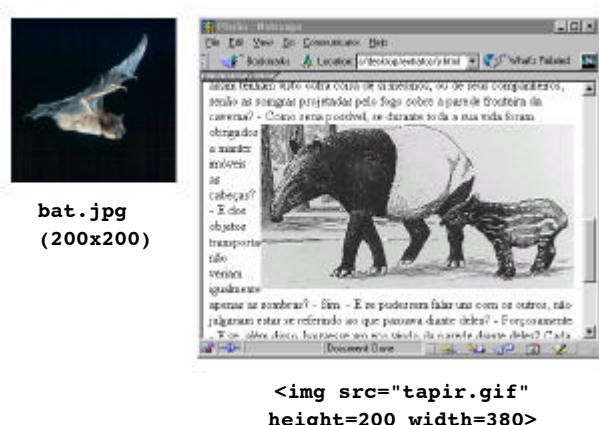
Para manipular com a fonte da imagem, ou acompanhar o status de seu carregamento na página, é preciso recorrer às propriedades definidas para o tipo *Image*, listadas abaixo. Com exceção de `complete` têm o mesmo nome que os atributos HTML do descritor ``. Com exceção de `src` e `lowsrc`, todas são *somente-leitura*.

Propriedade	Descrição
<code>complete</code>	<i>Boolean</i> . Contém <code>true</code> se a imagem foi carregada completamente.
<code>border</code>	<i>String</i> . Reflete o valor do atributo HTML <code>BORDER</code>
<code>height</code>	<i>String</i> . Reflete o valor do atributo HTML <code>HEIGHT</code>
<code>name</code>	<i>String</i> . Reflete o valor do atributo HTML <code>NAME</code>
<code>src</code>	<i>String</i> . Reflete o valor do atributo HTML <code>SRC</code> se for uma imagem da página HTML, e permite redefini-lo. Sempre indica o arquivo-fonte ou URL da imagem.
<code>lowsrc</code>	<i>String</i> . Reflete o valor do atributo HTML <code>LOWSRC</code> e permite redefini-lo. Indica o arquivo-fonte de baixa-resolução temporário da imagem, que é carregado antes do arquivo em <code>SRC</code> .
<code>hspace</code>	<i>String</i> . Reflete o valor do atributo HTML <code>HSPACE</code>
<code>vspace</code>	<i>String</i> . Reflete o valor do atributo HTML <code>VSPACE</code>
<code>width</code>	Retorna o valor do atributo HTML <code>WIDTH</code>

Das propriedades acima, `src` é a mais importante. É ela quem indica o arquivo-fonte da imagem através de uma URL (pode ser uma URL relativa, contendo apenas o nome de um arquivo localizado no mesmo diretório que a página). Um string contendo essa URL, atribuído à propriedade `src` de um objeto *Image*, fará com que o browser tente (imediatamente) carregar o arquivo-fonte da imagem:

```
animal = new Image();
animal.src = ".../figuras/bat.jpg";
```

As imagens carregadas através das instruções JavaScript acima não são exibidas automaticamente quando a página é carregada. Precisam um contexto gráfico na página onde possam ser exibidas. Esse contexto é fornecido pelo descritor HTML ``. Não é possível exibir imagens em páginas que não tenham pelo menos um descritor ``, mesmo que essas



imagens tenham sido carregadas¹. O descritor funciona como uma janela onde a imagem pode ser exibida. Havendo um descritor na página, sua imagem original poderá ser substituída por qualquer imagem, cujo arquivo tenha sido carregado dinamicamente, através da propriedade **src**. Por exemplo:

```
document.images[0].src = animal.src;
```

fará com que a imagem **bat.jpg** ocupe o lugar da primeira imagem da página (**tapir.gif**, na figura acima). O contexto gráfico não pode ser redimensionado², então a imagem **bat.jpg**, que tem dimensões 200x200 será redimensionada e ocupará o espaço antes ocupado por **tapir.gif** (380x200). O resultado será uma imagem “esticada”. Veja a figura ao lado.

A imagem também poderia ter sido substituída diretamente, sem precisar criar um novo objeto do tipo *Image*:

```
document.images[0].src = "../figuras/bat.jpg";
```

mas isto faria com que o browser tentasse carregar a imagem no momento em que a instrução acima fosse interpretada. Se uma página já foi completamente carregada e um evento dispara a instrução acima, o usuário teria que esperar que a imagem fosse carregada através da rede. No outro exemplo, a carga da imagem poderia ter sido feita antes. Quando o evento causasse a troca das imagens, ela estaria disponível no cache e seria substituiria a antiga imediatamente.

Quando se utiliza várias imagens, é útil carregá-las todas antes do uso. Isto pode ser feito colocando instruções em um bloco <SCRIPT> dentro do <HEAD> de uma página, o que garante que será executado antes de qualquer outra instrução no <BODY>. O código abaixo carrega 10 imagens chamadas **tela1.gif**, ..., **tela10.gif** e as armazena em um vetor **telas**:

```
<head>
<script>
    telas = new Array(5);
    for (i = 0; i < imagens.length ; i++) {
        telas[i] = new Image();
        telas[i].src = "tela" + (i+1) + ".gif";
    }
</script>
(...)
```



¹ É possível, porém, inserir dinamicamente um descritor na página, usando `document.write()`.

² JavaScript 1.1

As 10 imagens podem ser usadas para substituir outras imagens da página ou para fazer uma animação. Usando vetores, fica fácil manipular toda a coleção através de seus índices. Quando uma substituição ocorrer:

```
document.images[2].src = telas[5];
```

o arquivo será trocado imediatamente, pois está disponível localmente.

Quando se têm muitas imagens, o browser poderá demorar para mostrar a página, enquanto executa as instruções no <HEAD>. Uma forma eficiente de evitar esse problema, é colocar as instruções dentro de uma função global, e chamá-la quando a página tiver terminado de carregar, usando <BODY ONLOAD="nomeFuncao()">. Por exemplo:

```
<head>
<script>
    var telas;          // variável global, para que possa ser usada em
                        // outras partes da página

    function carregaImagens() {
        telas = new Array(10);
        for (i = 0; i < imagens.length ; i++) {
            telas[i] = new Image();
            telas[i].src = "tela" + (i+1) + ".gif";
        }
    }
    ...
</script>
</head>
```

poderá ser chamada depois que a página tiver sido carregada com:

```
<BODY ONLOAD="carregaImagens()">
...
</BODY>
```

Na transferência de imagens através da rede é comum acontecerem erros, provocando a interrupção da transferência ou a carga incorreta da imagem. A propriedade **complete** pode ser usada para verificar se uma imagem já foi carregada totalmente, antes de utilizá-la. **complete** contém o valor **true** somente se a imagem já foi carregada totalmente. Se ocorrer um erro ou a carga da imagem for interrompida, **complete** irá conter o valor **false**:

```
if (telas[9].complete) {
    iniciarAnimacao();
}
```

Eventos

Os atributos HTML de que respondem a eventos associados com imagens são:

- **ONERROR** – executa quando acontece um erro (arquivo não encontrado, conexão perdida). Uma imagem com o atributo ONERROR usado da forma:

```

```

fará com que a transferência da imagem seja reiniciada se houver erro.

- **ONABORT** – executa quando o usuário solicita a interrupção da transferência da imagem. Uma imagem com o atributo ONABORT usado da forma:

```

```

fará com que a transferência da imagem seja reiniciada cada vez que o usuário tentar interrompê-la.

- **ONLOAD** – executa quando a imagem termina de ser carregada.

Freqüentemente, imagens são colocadas dentro de links, e os eventos aplicáveis a links podem ser usados para realizar a substituição de imagens. Isto é usado, por exemplo, em ícones dinâmicos, assunto do exercício a seguir.

Exercício Resolvido

Crie um ícone ativo (link em torno de uma imagem) em uma página HTML que muda de cor quando o mouse passa sobre ele. Utilize duas imagens disponíveis no diretório **cap9/**. A primeira, **dullbart.gif**, mais apagada, deve estar presente na página quando ela for carregada; a outra **brightbart.gif**, deve substituir a primeira quando o mouse estiver sobre ela (evento **ONMOUSEOVER** do link **<A>**). A primeira imagem deverá voltar a ocupar a sua posição quando o mouse deixar a imagem (evento **ONMOUSEOUT** do link **<A>**). Há um esqueleto disponível em **bart.html**.

A substituição deve ser imediata. As duas imagens devem estar carregadas na memória antes de haver qualquer substituição.



Solução

A listagem a seguir apresenta uma possível solução ao problema proposto (está no arquivo **bartsol.html**).

```
<html>
<head>
<title>Imagens</title>
<script>
```

```

apagado = new Image();
aceso   = new Image();
apagado.src = "dullbart.gif";
aceso.src  = "brightbart.gif";

function apaga() {
    document.images[0].src = apagado.src;
}

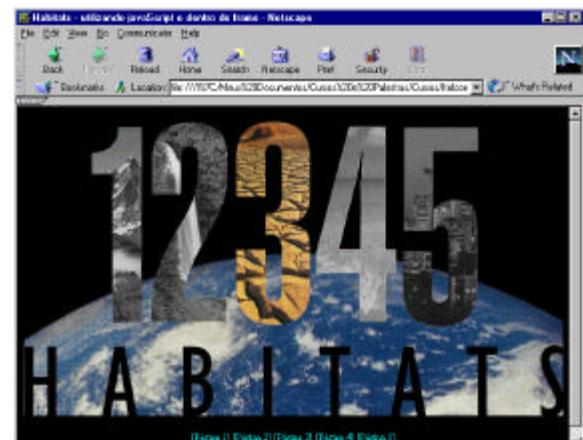
function acende() {
    document.images[0].src = aceso.src;
}

</script>
</head><body>
<a href="" onmouseover="acende()" onmouseout="apaga()">
    
</a>
</body>
</html>

```

Exercícios

- 9.1 Crie um banner animado usando JavaScript que mostre várias imagens em seqüência (use as imagens `im-01.jpg` a `im-05.jpg` (figura ao lado) disponíveis no diretório `cap9/`. As imagens devem ser carregadas previamente, logo após a carga da página (use `ONLOAD`). O intervalo de tempo entre cada imagem deve ser de um segundo (solução em `cap9/anima.html`).
- 9.2 Na página `uma.html` (figura abaixo) há 5 imagens preto-e-branco. Faça as seguintes alterações para que sua aparência seja mudada de acordo com o movimento do mouse do usuário sobre as imagens (use os arquivos disponíveis no diretório `cap9/`):
- Faça com que cada imagem `impb-nn.jpg` (onde nn é 01, 02, 03, 04 ou 05) seja trocada por sua correspondente a cores `im-nn.jpg` quando o mouse passar sobre a ela.
 - Quando o mouse deixar a imagem e passar para outra área da página,



- a imagem original (em preto e branco) deve ser restaurada.
- c) Faça com que cada imagem seja um link ativo para as páginas pag-01.html a pag-05.html.
- 9.3 Inclua as páginas uma.html e menu.html em uma estrutura de frames como ilustrado abaixo (use o arquivo frame.html). Clicar em uma das imagens deve fazer com que a janela principal seja ocupada pela página referenciada pelo link (pag-01.html a pag-05.html) e que a *imagem* (não a página) da janela secundária seja trocada por uma imagem correspondente (menu01.gif a menu02.gif).



10

Formulários

OS COMPONENTES DE FORMULÁRIO SÃO OS OBJETOS HTML mais utilizados em JavaScript. Por componentes de formulário nos referimos a qualquer campo de entrada de dados dentro de um bloco HTML `<FORM>`, como botões, caixas de seleção, caixas de texto e botões de “rádio”. Eles são a principal forma de entrada de dados disponível no HTML.

Os objetos de formulário consistem de doze objetos, situados abaixo de `Form`, no modelo de objetos do documento JavaScript. São referências aos elementos HTML `<INPUT>`, `<SELECT>`, `<OPTION>` e `<TEXTAREA>`.

Uma das principais aplicações do JavaScript, é a validação de dados em aplicações Web. Verificar se os campos de um formulário foram preenchidos corretamente antes de enviar os dados a um programa no servidor é uma tarefa realizada eficientemente com JavaScript. Na maior parte das aplicações, JavaScript é capaz de fazer toda a verificação localmente, economizando conexões de rede desnecessárias.

Neste capítulo, apresentaremos cada um dos objetos JavaScript relacionados a componentes de formulário, como criá-los em HTML e como manipular os dados recebidos por eles.

Objeto `Form`

O objeto `Form` é o mais alto da hierarquia dos componentes de formulários. Através dele se tem acesso aos componentes existentes dentro de um bloco HTML `<form>`, que podem ser botões, caixas de texto, caixas de seleção, etc.

Não é possível criar um objeto `Form` em JavaScript. Ele precisa existir no código HTML da página através de um bloco `<form> ... </form>`. Este bloco não é visível na página. Serve apenas para agrupar componentes de entrada de dados, torná-los visíveis e associar seus dados a um programa no servidor. A sintaxe do objeto `Form` em HTML está mostrada abaixo. Todos os atributos (em *ítlico*) são opcionais para uso em JavaScript:

```

<FORM
    NAME="nome_do_formulario (usado por JavaScript)"
    ACTION="url para onde será enviado o formulário"
    METHOD="método HTTP (pode ser GET ou POST)"
    ENCTYPE="formato de codificação"
    TARGET="janela alvo de exibição da resposta do formulário"
    ONRESET="código JavaScript"
    ONSUBMIT="código JavaScript" >
    ... corpo do formulário ...
</FORM>

```

Um bloco `<FORM>` deve estar dentro de um bloco `<BODY>`, em HTML. Para ter acesso à propriedades de um objeto *Form*, que é um reflexo de `<FORM>` em JavaScript, é necessário ter acesso ao documento que o contém. Um documento pode ter vários formulários. A propriedade `document`, portanto, possui um vetor com referências a todos os formulários do documento, na ordem em que eles aparecem no código. Este vetor está na propriedade `document.forms`. Para ter acesso ao primeiro formulário de um documento (se ele existir), pode-se usar:

```
objForm = document.forms[0]
```

Pode-se também dar um *nome* ao formulário. Todo componente de `<BODY>` que recebe um nome passa a ser uma propriedade de `document`. O nome é criado através do atributo `NAME` do HTML:

```
<form name="f1"> ... </form>
```

Criado o formulário em HTML, podemos ter acesso a seus métodos e propriedades através do operador ponto (`.`) usando seu nome ou posição no vetor `document.forms`:

```
x = document.forms[0].propriedade;
document.f1.método();
```

A maior parte das propriedades de *Form* são strings que permitem ler e alterar atributos do formulário definidos no elemento HTML `<FORM>`. A propriedade `elements` é a exceção. Ela contém um vetor com referências para todos os elementos contidos no formulário, na ordem em que aparecem no código.

Propriedade	Descrição
<code>elements</code>	<i>Array</i> . Vetor de elementos do formulário (<i>read-only</i>).
<code>elements.length</code>	<i>Number</i> . Número de elementos do formulário (<i>read-only</i>).
<code>name</code>	<i>String</i> . Contém o valor do atributo HTML <code>NAME</code> (<i>read-only</i>).
<code>action</code>	<i>String</i> . Contém o valor do atributo HTML <code>ACTION</code> .
<code>encoding</code>	<i>String</i> . Contém o valor do atributo HTML <code>ENCTYPE</code> .
<code>method</code>	<i>String</i> . Contém o valor do atributo HTML <code>METHOD</code> .
<code>target</code>	<i>String</i> . Contém o valor do atributo HTML <code>TARGET</code> .

Elementos de um formulário

As propriedades do objeto *Form* incluem todos os elementos de formulário e imagens que estão *dentro* do bloco <FORM> na página HTML. Estes objetos podem ser referenciados pelos seus nomes – propriedades de *Form* criadas com o atributo NAME de cada componente, ou através da propriedade **elements** – vetor que contém *todos* os elementos contidos no bloco <FORM> na ordem em que aparecem no HTML. Por exemplo, os componentes

```
<form name="f1">
    <input type="text" name="campoTexto">
    <input type="button" name="botao">
</form>
```

podem ser acessados usando o vetor **elements** da forma:

```
txt = document.f1.elements[0]; // primeiro elemento (Text)
bot = document.f1.elements[1]; // segundo elemento (Button)
```

ou usando o seu nome, como propriedade de *Form*:

```
txt = document.f1.campoTexto; // primeiro elemento (Text)
bot = document.f1.botao; // segundo elemento (Button)
```

Usar **elements** em vez do nome de um componente exige uma atenção maior pois a reorganização do formulário irá afetar a ordem dos componentes. Componentes diferentes possuem propriedades diferentes. Se **elements[0]** tem uma propriedade que **elements[1]** não tem, a tentativa de utilizar a propriedade em **elements[1]** causará um erro.

Existem três propriedades que são comuns a *todos* os elementos. Podem ser usadas para identificar o tipo do componente, seu nome e para obter uma referência ao formulário que o contém. Essas propriedades são:

Propriedade	Descrição
form	<i>Form</i> . Referência ao formulário que contém este botão.
name	<i>String</i> . Contém o valor do atributo HTML NAME do componente.
type	<i>String</i> . Contém o tipo do elemento. Pode ter um dos seguintes valores: select-one , select-multiple , textarea , text , password , checkbox , radio , button , submit , reset , file , hidden

Uma forma de evitar o uso acidental de um tipo de componente em lugar de outro, é testando sua propriedade **type**. Antes de usar a propriedade de um componente obtido através de **elements**, teste-o para verificar se realmente se trata do componente desejado:

```
var info;
elemento = document.forms[0].elements[5];
if (elemento.type == "textarea")
    info = elemento.value;
```

```
else if (elemento.type == "select-one")
    info = elemento.options[elemento.selectedIndex].text;
```

O objeto `type` de cada objeto geralmente contém o string contido no atributo `TYPE` do elemento HTML em caixa-baixa. A única exceção é o objeto do tipo *Select* que pode ter dois valores: “`select-multiple`” ou “`select-one`”, identificando listas de seleção múltipla e simples.

Métodos

Os métodos do objeto *Form* são usados para submeter os dados ao programa no servidor ou reinicializar o formulário com os seus valores originais. Estes métodos podem habilitar outros botões (tipo *Button*) ou qualquer objeto do HTML que capture eventos a implementar a mesma funcionalidade dos botões `<SUBMIT>` ou `<RESET>`:

Método	Ação
<code>reset()</code>	Reinicializa o formulário
<code>submit()</code>	Envia o formulário
<code>focus()</code>	Seleciona o formulário
<code>blur()</code>	Tira a seleção do formulário

Por exemplo, o código abaixo irá enviar os dados do primeiro formulário de uma página para o servidor:

```
<script>
    document.forms[0].submit();
</script>
(...)
```

Eventos

Os eventos relacionados ao objeto *Form* são a reinicialização do formulário e o envio dos dados ao servidor, desencadeados por botões *Reset* e *Submit* ou por instruções JavaScript. São interceptados pelo código contido nos atributos HTML abaixo:

- `ONSUBMIT` – antes de enviar o formulário ao servidor
- `ONRESET` – antes de inicializar o formulário com os valores *default*

Os eventos acima são sempre tratados *antes* das ações correspondentes acontecerem. O código em `ONSUBMIT`, por exemplo, será interpretado antes que o envio dos dados seja realizado. Se o código JavaScript dentro do `ONSUBMIT` produzir o valor `false`, os dados não serão enviados. Isto permite que os dados sejam verificados antes de enviados.

Estes eventos apresentam várias incompatibilidades entre browsers. O **ONSUBMIT** por exemplo chega a ser inútil tanto em versões do Netscape quanto no Internet Explorer¹.

Objetos *Button*, *Reset* e *Submit*

Button, *Reset* e *Submit* são todos botões criados em HTML. Têm a mesma aparência na tela, porém efeitos diferentes quando apertados. *Submit* e *Reset* têm eventos já definidos pelo HTML para enviar e limpar o formulário ao qual pertencem. *Button* é inoperante a menos que possua um atributo de eventos **ONCLICK**, com o código que será interpretado quando o usuário apertá-lo. A sintaxe do objeto *Button* em HTML está mostrada abaixo. Apenas os atributos em negrito são obrigatórios:

```
<INPUT TYPE="button"
       NAME="nome do botão"
       VALUE="rótulo do botão"
       ONCLICK="Código JavaScript"
       ONFOCUS="Código JavaScript"
       ONBLUR="Código JavaScript">
```

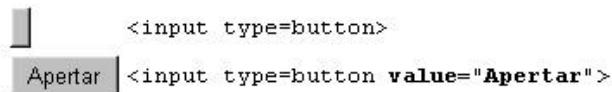
Objetos *Submit* e *Reset* são idênticos. A única diferença é o valor do atributo **TYPE**:

```
<INPUT TYPE="submit" ... >      <!-- Objeto Submit -->
<INPUT TYPE="reset" ... >      <!-- Objeto Reset -->
```

Os botões *Submit* e *Reset* também respondem ao evento de se apertar o botão caso tenham um atributo **ONCLICK**. Mas é importante lembrar que *eles já têm eventos* atribuídos pelo HTML, que sempre ocorrerão *depois* dos eventos JavaScript. Por exemplo, qualquer alteração no formulário realizada por um programa no **ONCLICK** de um objeto *Reset*, será anulada pelo evento nativo do *Reset* que reinicializa os campos do formulário aos seus valores originais.

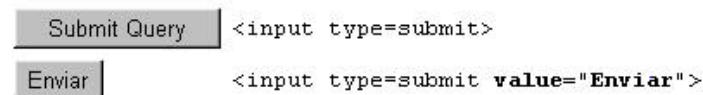
O atributo **VALUE** é opcional em todos os botões. Permite alterar o texto que aparece dentro do mesmo. Em objetos *Reset* e *Submit*, **VALUE** possui um valor *default*. Em objetos *Button*, o *default* para **VALUE** é um string vazio, portanto, a menos que este atributo seja definido, o botão

Button



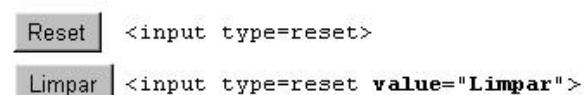
```
<input type=button>
Apertar <input type=button value="Apertar">
```

Submit



```
Submit Query <input type=submit>
Enviar <input type=submit value="Enviar">
```

Reset



```
Reset <input type=reset>
Limpar <input type=reset value="Limpar">
```

¹ A documentação JavaScript 1.1 também de referência, aparece como evento de *F* **TYPE=submit**.

aparecerá vazio. A figura ao lado mostra a aparência dos botões em um browser Netscape Navigator rodando em Windows 95 com e sem o atributo **VALUE**.

Qualquer objeto <INPUT> é um elemento de formulário e precisa estar dentro de um bloco <FORM>...</FORM>. para que seja visível na tela², possa receber dados e ser manipulado como um objeto JavaScript. Os botões podem ser acessados através do vetor **elements**, na ordem em que aparecem no código, ou através do seu nome, especificado pelo atributo **NAME**:

```
<form>
  <input type=button name=b1>  <!--objeto tipo Button -->
  <input type=submit name=b2>  <!--objeto tipo Submit -->
  <input type=reset name=b3>  <!--objeto tipo Reset -->
</form>
```

Se os três botões acima estiverem dentro do primeiro formulário de uma página HTML, o segundo botão pode ser referenciado da forma:

```
but2 = document.forms[0].b2          // ou ...
but2 = document.forms[0].elements[1]
```

Os botões possuem quatro propriedades. Três refletem atributos do HTML e uma é um ponteiro para o formulário que contém o botão. A propriedade **value** é a única que pode ser alterada dinamicamente.

Propriedade	Descrição
form	<i>Form.</i> Referência ao formulário que contém este botão.
value	<i>String.</i> Contém o valor do atributo HTML VALUE que especifica o texto que aparece no botão. Pode ser lida ou alterada.
name	<i>String.</i> Contém o valor do atributo HTML NAME . (<i>read-only</i>)
type	<i>String.</i> Contém o valor do atributo HTML TYPE . (<i>read-only</i>)

Com a propriedade **form**, um botão pode subir a hierarquia e ter acesso a outros elementos do formulário no qual está contido. Por exemplo, no código abaixo, o primeiro botão altera o rótulo do segundo botão, ao ser apertado. Para ter acesso a ele, obtém uma referência ao formulário em que está contido através de sua propriedade **form**:

```
<form>
  <input type=button name=b1 value="Editar"
         onclick="this.form.b2.value='Alterar'">
  <input type=submit name=b2 value="Criar">
</form>
```

² O Internet Explorer mostra na tela componentes que não estão dentro de <form>, mas eles não têm utilidade alguma pois não podem enviar dados ao servidor nem serem manipulados em JavaScript.

Os métodos dos objetos *Button*, *Submit* e *Reset* estão associados a eventos. Permitem simular o evento que ocorre ao clique do mouse, ao ativar e desativar um botão.

Método	Ação
<code>click()</code>	Realiza as tarefas programadas para o clique do botão (executa o código JavaScript contido no atributo <code>ONCLICK</code> sem que o botão precise ser apertado).
<code>focus()</code>	Ativa o botão.
<code>blur()</code>	Desativa o botão.

Eventos

Os eventos suportados por botões são três. Os atributos HTML abaixo respondem a eventos de botão interpretando o código JavaScript contido neles:

- `ONCLICK` – quando o usuário aperta o botão
- `ONFOCUS` – quando o usuário seleciona o botão.
- `ONBLUR` – quando o usuário seleciona outro componente.

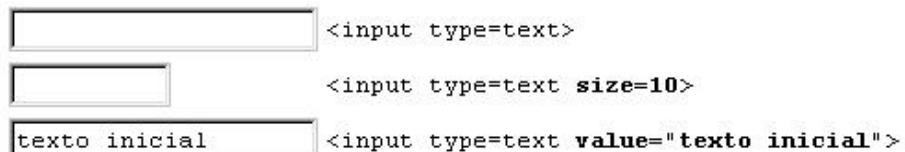
Objetos *Password*, *Text* e *Textarea*

Os objetos do tipo *Password*, *Text* e *Textarea* são usados para entrada de texto. Possuem as mesmas propriedades. Objetos *Text* e *Password* definem caixas de texto de uma única linha enquanto que os objetos *Textarea* entendem quebras de linha.

Objetos *Text* e *Password* são criados com elementos HTML `<INPUT>` e têm a mesma aparência, mas o texto dos objetos *Password* não é exibido na tela. A sintaxe de um objeto *Text* em HTML é a seguinte:

```
<INPUT TYPE="text"
      NAME="nome_do_campo_de_texto"
      VALUE="texto inicial do campo de textos"
      SIZE="número de caracteres visíveis"
      MAXLENGTH="número máximo de caracteres permitido"
      ONBLUR="código JavaScript"
      ONFOCUS="código JavaScript"
      ONCHANGE="código JavaScript"
      ONSELECT="código JavaScript" >
```

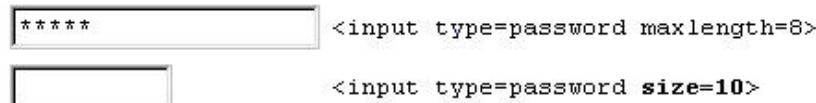
Todos os atributos, exceto o atributo `TYPE` são opcionais. Se `SIZE` não for definido, a caixa de texto terá 20 caracteres de largura. Se `MAXLENGTH` não for definido, não haverá limite para o número de caracteres digitado no campo de textos. A figura abaixo ilustra a aparência de objetos *Text* em um browser Netscape 4.5 rodando em Windows 95.



O objeto *Password* é criado da mesma forma, mas com um atributo **TYPE** diferente:

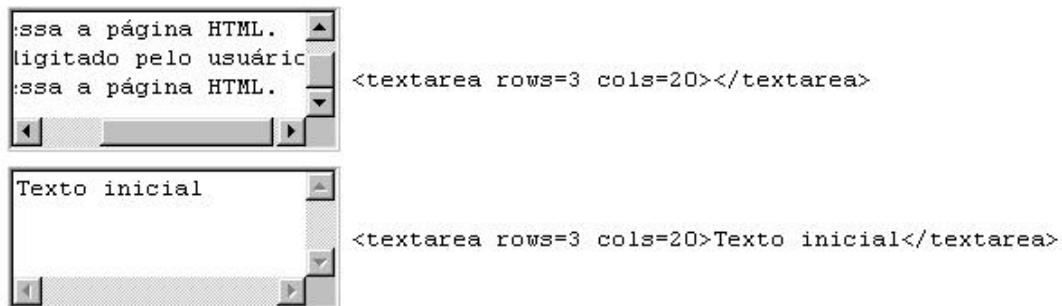
```
<INPUT TYPE="password" ... >
```

Os caracteres do texto digitado em objetos *Password* não aparecem na tela, como mostrado na figura abaixo (Windows95):



A sintaxe para criar um objeto *Textarea* em HTML é a seguinte. Os atributos em negrito são obrigatórios:

```
<TEXTAREA  
    ROWS="número de linhas visíveis"  
    COLS="número de colunas visíveis"  
    NAME="nome_do_campo_de_texto"  
    ONBLUR="handlerText"  
    ONFOCUS="handlerText"  
    ONCHANGE="handlerText"  
    ONSELECT="handlerText" >  
    Texto inicial  
</TEXTAREA>
```



A figura abaixo mostra a aparência de objetos *Textarea*:

Como qualquer outro elemento de formulário, é preciso que os blocos **<TEXTAREA>** e descritores **<INPUT>** estejam dentro de um bloco **<FORM>...</FORM>** para que sejam visíveis na tela, possam receber dados e serem manipulados como objetos JavaScript. Os campos de texto podem ser acessados através do vetor **elements**, na ordem em que aparecem no código, ou através do seu nome, especificado pelo atributo **NAME**:

```
<form>  
    <input type=text name=userid>  
    <input type=password name=senha>  
    <textarea rows=2 cols=10 name=codigos>
```

```
</textare>
</form>
```

Se os três campos acima estiverem dentro do primeiro formulário de uma página HTML, o segundo campo pode ser referenciado da forma:

```
texto2 = document.forms[0].senha           // ou ...
texto2 = document.forms[0].elements[1]
```

As propriedades dos campos de texto são as mesmas para *Text*, *Textarea* e *Password*. São todas *read-only* com exceção da propriedade **value**, que pode ser alterada, mudando o conteúdo dos campos de texto dinamicamente.

Propriedade	Descrição
form	<i>Form</i> . Referência ao formulário no qual este elemento está contido.
type	<i>String</i> . Valor do atributo TYPE do HTML.
name	<i>String</i> . Valor do atributo NAME do HTML.
defaultValue	<i>String</i> . Valor default previamente definido no campo VALUE do HTML.
value	<i>String</i> . Conteúdo do campo de texto. Valor que <i>pode ser redefinido</i> .

Um objeto *Textarea* pode ter várias linhas de texto e armazenar os caracteres “\n” passados através de sua propriedade **value**, o que não ocorre com objetos *Text* ou *Password*. Se o usuário digitar [ENTER], em um objeto *Textarea* o cursor pulará para a próxima linha. Isto é diferente do que acontece em um objeto *Text*, onde [ENTER] provoca um evento que interpreta o código disponível no atributo **ONCHANGE**.

Os métodos dos objetos *Text*, *Password* e *Textarea* são utilizados para selecionar os componentes e o seu conteúdo. Todos provocam eventos.

Método	Ação
focus()	Ativa o componente.
blur()	Desativa o componente.
select()	Seleciona o campo editável do componente (faz o cursor aparecer piscando dentro do campo de texto ou seleciona o texto nele contido).

Eventos

Os eventos suportados por objetos *Text*, *TextArea* e *Password* são quatro. Os atributos HTML abaixo respondem aos eventos interpretando o código JavaScript contido neles:

- **ONFOCUS** – quando o usuário seleciona o componente.
- **ONBLUR** – quando o usuário, que tinha o componente selecionado, seleciona outro componente.
- **ONCHANGE** – em *Textarea*, quando o usuário deixa o componente e o valor seu valor difere daquele existente antes da sua seleção; em *Text* e *Password* quando o usuário deixa o componente com valor diferente ou aperta a tecla [ENTER].

- **ONSELECT** – quando o usuário seleciona a área editável do componente.

Para validar dados digitados em campos de texto, é preciso ler sua propriedade **value**, que é um objeto *String*. Qualquer operação que possa ser realizada sobre strings pode ser realizada sobre **value**. Por exemplo, *String* possui uma propriedade **length**, que informa quantos caracteres possui. Através dela pode-se verificar se um campo de textos está vazio usando:

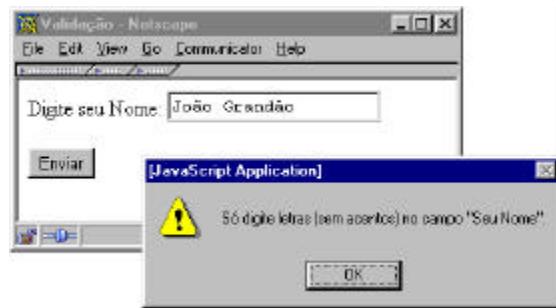
```
if (document.forms[0].senha.length == 0) {
    alert("É preciso digitar uma senha");
}
```

Os métodos **select()** e **focus()** são úteis para rolar a tela e posicionar o cursor em um campo de textos específico, facilitando o acesso a campos que precisam ser corrigidos. O exemplo a seguir mostra uma aplicação desses métodos na validação de um campo de textos. Considere o seguinte formulário HTML que têm a aparência ao lado:

```
<FORM METHOD="POST" ACTION="/cgi-bin/mailForm.pl">
<P>Digite seu Nome:
<INPUT TYPE="TEXT" SIZE="20" NAME="nome">
<P><INPUT TYPE="button" VALUE="Enviar" ONCLICK="valida(this.form)">
</FORM>
```

O botão chama a função **valida()**, passando como argumento uma referência para o formulário, que verifica se o texto digitado no campo do nome contém apenas letras do alfabeto ASCII. Para comparar, usamos os métodos **toLowerCase()** e **charAt()** de *String*:

```
<script>
function valida(f) {
    var valid = true;
    if (f.nome.value.length <= 0) {      // verifica se string está vazio
        valid = false;
    } else {                  // verifica se usuário digitou caracteres ilegais
        for (i = 0; i < f.nome.value.length; i++) {
            ch = f.nome.value.charAt(i).toLowerCase();
            if (ch < 'a' || ch > 'z') valid = false;
        }
    }
    if (!valid) {
        alert("Só digite letras (sem acentos) no campo \"Seu Nome\".");
        f.nome.focus();
        f.nome.select();
    } else {
        f.submit();   // envia o formulário
    }
}
```



```

        }
    }
</script>

```

Objeto Hidden

O objeto *Hidden* é um campo de entrada de dados invisível, que o usuário da página não tem acesso. Serve para que o *programador* passe informações ao servidor, ocultando-as no código HTML da página. É bastante útil na transferência de informações entre formulários distribuídos em mais de uma página. Vários desses campos foram usados no capítulo 8 para transferir dados digitados em um formulário para uma página gerada *on-the-fly*. Sua sintaxe é a seguinte:

```

<INPUT TYPE="hidden"
       NAME="nome_do_campo_oculto"
       VALUE="valor_armazenado" >

```

Os atributos **NAME** e **VALUE** são opcionais se o campo oculto for manipulado em JavaScript³. São elementos de formulário e devem estar dentro de um bloco **<FORM>**. Os campos ocultos podem ser acessados através do vetor **elements**, na ordem em que aparecem no código, ou através do seu nome, especificado pelo atributo **NAME**:

```

<form>
    <input type=text name=userid>
    <input type=password name=senha>
    <input type=hidden name=email value="admin@root.where.org">
    <textarea rows=2 cols=10 name=codigos></textarea>
</form>

```

Se os quatro campos acima estiverem dentro do primeiro formulário de uma página HTML, o valor do campo oculto pode ser obtido de qualquer uma das forma abaixo:

```

valorOculto = document.forms[0].email.value      // ou ...
valorOculto = document.forms[0].elements[2].value

```

As propriedades do objeto *Hidden*, com exceção de **form**, são todas relacionadas aos atributos HTML correspondentes. A única que pode ser alterada é **value**.

Propriedade	Descrição
form	<i>Form.</i> Referência ao formulário no qual este elemento está contido.
name	<i>String.</i> Valor do atributo NAME do HTML (<i>read-only</i>).
type	<i>String.</i> Valor do atributo TYPE do HTML (<i>read-only</i>).
value	<i>String.</i> Valor do atributo VALUE do HTML. Esta propriedade pode ser redefinida e usada como meio de passar informações entre páginas.

³ Sem JavaScript, um campo hidden sem os atributos NAME e VALUE é inútil.

Não há novos métodos nem eventos associados ao objeto *Hidden*.

Objeto *Checkbox* e *Radio*

Os objetos *Checkbox* e *Radio* representam dispositivos de entrada booleanos cuja informação relevante consiste em saber se uma opção foi selecionada ou não. As únicas diferenças entre os objetos *Checkbox* e *Radio* são a sua aparência na tela do browser e a quantidade de opções que podem conter para cada grupo de dispositivos.

Objetos *Radio* são organizados em grupos de descritores com o mesmo nome (atributo **NAME**). Cada componente aparece na tela como um botão ou caixa de dois estados: *ligado* ou *desligado*. Dentro de um grupo de componentes (todos com o *mesmo* atributo **NAME**), somente um deles poderá estar ligado ao mesmo tempo. A sintaxe de um objeto *Radio* em HTML é a seguinte:

```
<INPUT TYPE="radio"
       NAME="nome_do_componente"
       VALUE="valor (o valor que será enviado ao servidor)"
       CHECKED      <!-- previamente marcado -->
       ONBLUR="código JavaScript"
       ONFOCUS="código JavaScript"
       ONCLICK="código JavaScript" > Rótulo do componente
```

A figura abaixo mostra dois grupos de botões de rádio (em um browser Netscape rodando em Windows95). Observe que os atributos **NAME** distinguem um grupo do outro. O atributo **CHECKED** indica um botão previamente ligado mas que pode ser desligado pelo usuário ao clicar em outro botão.

GRUPO I

- pela manhã <input type=radio name=turno value="Manhã"> pela manhã
- à tarde <input type=radio name=turno value="Tarde"> à tarde
- à noite <input type=radio name=turno value="Noite" checked> à noite

GRUPO II

- Masculino <input type=radio name=sexo value="M"> Masculino
- Feminino <input type=radio name=sexo value="F"> Feminino

Um grupo não é tratado como um elemento de formulário, mas os nomes dos grupos são referências do tipo *Array* em JavaScript. Se os elementos acima fossem os únicos elementos do primeiro formulário de uma página, o primeiro elemento do segundo grupo poderia ser acessado de qualquer uma das formas abaixo:

```
fem = document.forms[0].elements[4];
fem = document.forms[0].sexo[1];
```

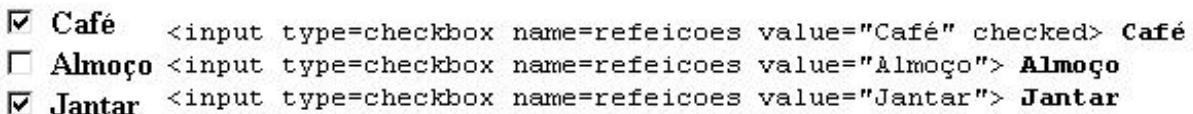
O código acima não inclui as informações que mais interessam, que são: 1) o usuário selecionou que opção? e 2) a opção *x* está ou não selecionada? Para responder à essas

questões, precisamos usar as propriedades do objeto *Radio*, que são as mesmas do objeto *Checkbox*, que veremos a seguir. Permitem manipular elementos individuais e grupos.

Cada *Checkbox* aparece na tela como um botão ou caixa que pode assumir dois estados: *ligado* ou *desligado*. Diferentemente dos objetos *Radio*, vários objetos *Checkbox* de um mesmo grupo podem estar ligados ao mesmo tempo, não havendo, portanto, necessidade de organizar tais objetos em um grupo. A sintaxe de um objeto *Checkbox* em HTML é praticamente idêntica à de *Radio*, mudando apenas o valor do atributo **TYPE**:

```
<INPUT TYPE="checkbox" ... > Rótulo do componente
```

A figura abaixo mostra um grupo de caixas de checagem (em um browser Netscape rodando em Windows95. O atributo **CHECKED** indica um botão previamente ligado mas que pode ser desligado pelo usuário ao clicar em outro botão.



```


```

Assim como objetos *Radio*, elementos *Checkbox* podem ser manipulados em grupo, embora isto seja desnecessário, já que mais de um valor pode estar associado ao mesmo grupo, como mostrado acima. Se os elementos acima fossem os únicos elementos do primeiro formulário de uma página, o segundo elemento poderia ser acessado de qualquer uma das formas abaixo:

```
almoco = document.forms[0].elements[2];
almoco = document.forms[0].refeicoes[2];
```

Os *Checkboxes* acima poderiam também ser implementados da forma seguinte, sem organizar seus elementos em grupos, com os mesmos resultados:

```
<input type=checkbox name=cafe checked> Café<br>
<input type=checkbox name=almoco> Almoço<br>
<input type=checkbox name=jantar> Jantar
```

O acesso ao segundo elemento agora pode ser feito da forma:

```
almoco = document.forms[0].almoco;
```

As propriedades de *Checkbox* e *Radio* são as mesmas e estão listadas abaixo.

Propriedade	Read/Write	Descrição
type	r	<i>String.</i> Conteúdo do atributo HTML TYPE (<i>read-only</i>).
name	r	<i>String.</i> Conteúdo do atributo HTML NAME (<i>read-only</i>).
defaultChecked	r	<i>Boolean.</i> Retorna true se o elemento HTML que representa o objeto contém o atributo CHECKED .
checked	r / w	<i>Boolean.</i> Retorna true se um <i>Checkbox</i> ou <i>Radio</i> está <i>atualmente</i> ligado. O valor desta propriedade pode ser

Propriedade	Read/Write	Descrição
		alterado dinamicamente em JavaScript para ligar ou desligar os componentes.
value	r / w	<i>String.</i> Conteúdo do atributo HTML VALUE. O valor desta propriedade pode ser alterado dinamicamente.
length	r	<i>Number.</i> Comprimento do vetor de objetos <i>Radio</i> ou <i>Checkbox</i> . Aplica-se apenas a grupos de elementos identificados pelo nome (não pode ser usado no vetor elements , que refere-se a objetos individuais). <i>Exemplo:</i> <code>document.forms[0].nomegrupo.length</code>

A validação de botões de rádio e caixas de checagem consiste em verificar se uma ou mais opções foram selecionadas. Para isto, basta testar a propriedade **checked** e verificar se ela é **true** nos campos existentes. Veja um exemplo:

```
opcao = null;
turnoArray = document.forms[0].turno; // vetor de botoes de radio
for (i = 0; i < turnoArray.length; i++) {
    if (turnoArray[i].checked) {
        opcao = turnoArray[i];
    }
}

if (opcao == null) {
    alert("É preciso escolher Manhã, Tarde ou Noite!");
}
```

Os métodos suportados pelos objetos *Radio* e *Checkbox*, como os outros elementos de formulário, provocam eventos. Estão listados na tabela abaixo:

Método	Ação
click()	Marca (seleciona) o componente.
focus()	Ativa o componente.
blur()	Desativa o componente.

Eventos

Os eventos suportados são três. Os atributos HTML abaixo respondem aos eventos interpretando o código JavaScript contido neles:

- **ONCLICK** – quando o usuário liga ou desliga o componente
- **ONFOCUS** – quando o usuário seleciona o componente.

- **ONBLUR** – quando o usuário, que tinha o componente selecionado, seleciona outro componente.

Objetos Select e Option

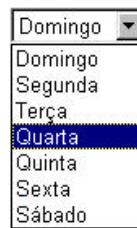
Caixas e listas de seleção como as mostradas nas figuras ao lado são representadas em JavaScript por objetos *Select*, que refletem o elemento HTML **<SELECT>**. Um objeto *Select* pode ter uma ou mais objetos *Option*, que refletem elementos **<OPTION>**.

Os objetos *Select* podem ter uma aparência e comportamento diferente dependendo se possuem ou não os atributos **SIZE** e **MULTIPLE**. A figura ao lado ilustra o efeito desses atributos, transformando uma caixa de seleção em uma lista que permite seleção de múltiplas opções.

Objetos *Select* só podem ser criados em HTML. Objetos *Option* podem tanto ser criados em HTML como em JavaScript através do construtor **Option()**. Desta forma, é possível ter listas que crescem (ou diminuem) dinamicamente. A sintaxe de um objeto *Select* em HTML está mostrada abaixo:

```
<SELECT
  NAME="nome_do_componente"
  SIZE="número de opções visíveis"
  MULTIPLE      <!-- Suporta seleção múltipla --&gt;
  ONBLUR="código JavaScript"
  ONCHANGE="código JavaScript"
  ONFOCUS="código JavaScript" &gt;
    &lt;OPTION ...&gt; Opção 1 &lt;/OPTION&gt;
    ...
    &lt;OPTION ...&gt; Opção n &lt;/OPTION&gt;
<b></SELECT>
```

Todos os atributos são opcionais. A existência do atributo **NAME** é obrigatória em formulários que terão dados enviados ao servidor. Os objetos *Option* não são elementos do formulário (não podem ser acessados através do vetor **elements**) mas são elementos do objeto *Select* e podem ser acessados pelo seu vetor **options**. A sintaxe de cada objeto *Option* está mostrada abaixo.



```
<select name=umdia>
  <option value="D">Domingo</option>
  <option value="S">Segunda</option>
  <option value="T">Terça</option>
  <option value="Q">Quarta</option>
  <option value="I">Quinta</option>
  <option value="X">Sexta</option>
  <option value="B">Sábado</option>
</select>
```



```
<select name=variosdias size=4 multiple>
  <option value="D">Domingo</option>
  <option value="S">Segunda</option>
  <option value="T">Terça</option>
  <option value="Q">Quarta</option>
  <option value="I">Quinta</option>
  <option value="X">Sexta</option>
  <option value="B">Sábado</option>
</select>
```

```
<OPTION  
    VALUE="Valor da opção"  
    SELECTED >  
    Texto descrevendo a opção  
</OPTION>
```

O atributo **VALUE** é opcional. Se os dados forem enviados ao servidor, o texto contido entre **<OPTION>** e **</OPTION>** é enviado somente se um atributo **VALUE** não tiver sido definido. Em JavaScript, ambos podem ser usados para armazenar informações ao mesmo tempo.

Um bloco **<SELECT>** é elemento de formulário e deve estar sempre dentro de um bloco **<FORM>**. Caixas e listas de seleção podem ser acessadas através do vetor **elements**, na ordem em que aparecem no código, ou através do seu nome, especificado pelo atributo **NAME**, por exemplo, considere o código HTML:

```
<form>  
    <input type=text name=nome1>  
    <select name=umdia>  
        <option value="D">Domingo</option>  
        <option value="S">Segunda</option>  
        <option value="T">Terça</option>  
        <option value="Q">Quarta</option>  
    </select>  
    <input type=text name=nome2>  
</form>
```

No código acima há apenas três elementos de formulário. Se o bloco **<form>** acima for o primeiro de um documento, a caixa de seleção poderá ser referenciada da forma:

```
dia = document.forms[0].umdia           // ou ...  
dia = document.forms[0].elements[1]
```

Os quatro elementos **<option>** são elementos do objeto *Select*. Para ter acesso ao terceiro objeto **<option>**, pode-se usar o vetor **options**, que é uma propriedade dos objetos *Select*. As duas formas abaixo são equivalentes e armazenam um objeto do tipo *Options* na variável **terca**:

```
terca = document.forms[0].umdia.options[2] // ou ...  
terca = document.forms[0].elements[1].options[2]
```

Para ter acesso aos dados armazenados pelo objeto recuperado, é preciso usar propriedades do objeto *Option*. A propriedade **text** recupera o texto entre os descritores **<option>** e **</option>**. A propriedade **value** recupera o texto armazenado no atributo HTML **<VALUE>**:

```
textoVisivel = terca.text;  
textoUtil = terca.value;
```

O código acima não obtém as informações que mais interessam, que são: 1) o usuário selecionou qual opção? e 2) a opção de índice *x* ou que contém o texto *y* está ou não selecionada? Para responder à essas questões, e poder realizar outras tarefas, como selecionar opções dinamicamente, precisamos conhecer as propriedades do objeto *Select* e dos seus objetos *Option*. Elas estão listadas nas tabelas abaixo.

A primeira tabela lista as propriedades do objeto *Select*, que são:

Propriedade	Read/Write	Descrição
<code>name</code>	r	<i>String</i> . Equivalente ao atributo HTML NAME.
<code>form</code>	r	<i>String</i> . Referência ao formulário que contém este objeto.
<code>type</code>	r	<i>String</i> . Informa o tipo de lista: <i>select-one</i> , se o elemento HTML não tiver o atributo MULTIPLE, ou <i>select-multiple</i> , se tiver.
<code>options</code>	r	<i>Array</i> . Vetor de objetos <i>Option</i> contidos no objeto <i>Select</i> .
<code>length</code>	r	<i>Number</i> . Número de objetos do vetor <code>options</code> .
<code>selectedIndex</code>	r/w	<i>Number</i> . Índice da opção atualmente selecionada. Para listas múltiplas, contém o <i>primeiro</i> índice selecionado.
<code>options.length</code>	r	<i>Number</i> . Mesmo que <code>length</code> .
<code>options.selectedIndex</code>	r/w	<i>Number</i> . Mesma coisa que <code>selectedIndex</code> .

As propriedades `options` e `selectedIndex` permitem a manipulação dos objetos *Options* contidos no *Select*. As propriedades de *Option* são:

Propriedade	Read/Write	Descrição
<code>index</code>	r	<i>Number</i> . Contém o índice desta opção dentro do vetor <code>options</code> do objeto <i>Select</i> ao qual pertence.
<code>defaultSelected</code>	r	<i>Boolean</i> . Retorna <code>true</code> se o elemento HTML que representa o objeto contém o atributo <code>SELECTED</code> .
<code>selected</code>	r/w	<i>Boolean</i> . Retorna <code>true</code> se objeto está selecionado. Pode ser alterado para selecionar ou deseletar o objeto dinamicamente.
<code>value</code>	r/w	<i>String</i> . Contém o conteúdo do atributo HTML <code>VALUE</code> (que contém os dados que serão enviados ao servidor).
<code>text</code>	r/w	<i>String</i> . O texto dentro de <code><option>...</option></code> , que aparece na lista de opções. Pode ser alterado. Este texto não será enviado ao servidor se existir um atributo <code>VALUE</code> .

A propriedade `selectedIndex` de *Select* contém o índice correspondente à opção atualmente selecionada. Muitas vezes, esta informação é suficiente para uma aplicação de validação de dados, por exemplo, que precisa verificar apenas que o usuário selecionou (ou não selecionou) uma determinada opção.

No trecho de código abaixo, é verificado o valor do índice selecionado de um objeto *Select*, cuja primeira opção não tem valor (apenas informa que o usuário deve selecionar uma opção). As opções válidas, portanto, são as opções selecionadas de índice maior ou igual a 1. Se o valor for zero, significa que o usuário não fez uma seleção válida:

```
if(document.forms[0].sele1.selectedIndex == 0) {
    alert("É preciso selecionar uma opção!");
}
```

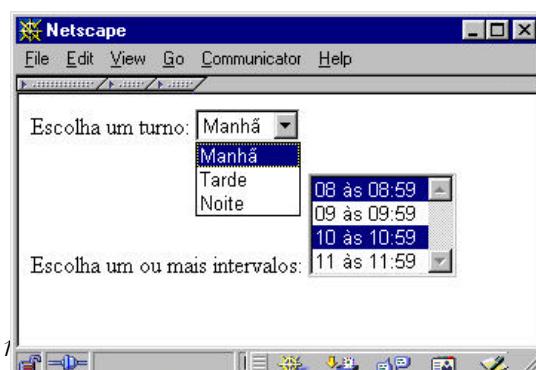
Em outros casos, informações relevantes precisam ser recuperadas do objeto *Option* atualmente selecionado. Se o *Select* não permite seleção múltipla, o número armazenado na propriedade **selectedIndex** pode ser usado para recuperar o objeto correspondente no vetor **options**, e a partir daí obter os dados que estão na propriedade **value** ou **text**. Os exemplos abaixo operam sobre os objetos *Select* apresentados no início desta seção:

```
// obtenção do índice atualmente selecionado
indice = document.forms[0].umdia.selectedIndex;
// valor enviado ao servidor: D, S, T, etc.
valorUtil = document.forms[0].umdia.options[indice].value;
// valor mostrado na lista de opções: Domingo, Segunda, Terça, etc.
valorVisivel = document.forms[0].umdia.options[indice].text;
```

Quando o objeto *Select* é uma lista de seleção múltipla, que pode ou não ter mais de um ítem selecionado, **selectedIndex** retorna apenas o índice do primeiro item selecionado. Neste caso, é preciso verificar uma a uma quais opções estão selecionadas através da propriedade **selected** de cada objeto *Options*. O código abaixo recupera o texto visível de cada uma das opções selecionadas de um objeto *Select* e as armazena em um vetor:

```
objSel = document.forms[1].variosdias;
opcoes = "";
if (objSel.type == "select-one") { // se for caixa de seleção
    opcoes = objSel.options[objSel.selectedIndex].text;
} else { // se for lista de múltiplas seleções
    for (i = 0; i < objSel.length; i++) {
        if (objSel.options[i].selected) {
            opcoes += objSel.options[i].text + "; "
        }
    }
}
opcoes = opcoes.split(";;"); // armazena em vetor
```

É possível mudar o texto de uma opção dinamicamente alterando o valor da propriedade **text**. O novo texto ocupará o lugar do antigo, mas o espaço disponível não será aumentado se o texto for maior, e será truncado. No exemplo a seguir (ilustrado na figura ao lado), há dois



objetos *Select*. As opções disponíveis na segunda lista dependem da opção escolhida pelo usuário na primeira lista:

```

<head>
<script>
intervalos = new Array(3);      // vetor 2D c/ intervalos/turno
intervalos[0] = new Array("08 às 08:59", "09 às 09:59", "10 às 10:59", "11 às 11:59");
intervalos[1] = new Array("13 às 13:59", "14 às 14:59", "15 às 15:59", "16 às 16:59");
intervalos[2] = new Array("18 às 18:59", "19 às 19:59", "20 às 20:59", "21 às 21:59");

function setIntervalos(f) {
    idx = f.turno.options.selectedIndex;
    for (i = 0; i < f.horario.length; i++) {
        f.horario.options[i].text = intervalos[idx][i];
    }
}
</script> </head>
<body>
<form> <p>Escolha um turno:
<select name=turno onchange="setIntervalos(this.form)">
    <option>Manhã</option>
    <option>Tarde</option>
    <option>Noite</option>
</select>
<p>Escolha um ou mais intervalos:
<select name=horario size=4 multiple>
    <option> Intervalo 01 </option> <!-- Estes valores irão mudar -->
    <option> Intervalo 02 </option>
    <option> Intervalo 03 </option>
    <option> Intervalo 04 </option>
</select> </form>
</body>
```

É possível acrescentar e remover opções de uma lista *Select* criando e removendo objetos *Option*. Para criar um novo objeto *Option*, usa-se o construtor *Option()* (veja 1 no código abaixo). Depois, pode-se definir valores de interesse, através das propriedades *text*, *value* e *selected* (2). Finalmente, cada objeto *Option* precisa ser colocado na lista, ocupando o final do vetor *options* (3). A função abaixo acrescenta novas opções à lista do exemplo acima:

```

function mais(f) { // f é o formulário onde o Select está contido
    selObj = f.horario;
    novaOp = new Option();                                // (1)
    novaOp.text = "Intervalo 0" + (selObj.length + 1);   // (2)
    selObj.options[selObj.length] = novaOp;               // (3)
}
```

Para remover uma opção de uma lista, basta encurtar o vetor *options*. A função abaixo redefine a propriedade *length* do vetor, encurtando-o cada vez que é chamada:

```

function menos(f) {
    selObj = f.horario;
    if (selObj.length > 0) {
        len = selObj.length - 1;
        selObj.length = len;           // length = length - 1
    }
}

```

O objeto *Select* possui dois métodos. Ambos provocam eventos e são usados para ativar ou desativar o componente:

Método	Ação
<code>focus()</code>	Ativa o componente.
<code>blur()</code>	Desativa o componente.

Eventos

Os eventos suportados por objetos *Select* são três. Os atributos HTML abaixo respondem aos eventos interpretando o código JavaScript contido neles:

- **ONFOCUS** – quando o usuário seleciona o componente.
- **ONBLUR** – quando o usuário, que tinha o componente selecionado, seleciona outro componente.
- **ONCHANGE** – quando o usuário seleciona uma opção diferente da que estava previamente selecionada no componente.

Validação de formulários

Nesta seção mostraremos uma das aplicações mais freqüentes de JavaScript: a verificação dos dados em um formulário antes do envio ao servidor. O exercício resolvido a seguir utiliza todas as categorias de componentes vistos neste capítulo para desenvolver um formulário de inscrição em um evento, a validadr as informações digitadas pelo usuário.

Exercício Resolvido

Para este exercício, utilize o esqueleto contido no arquivo `validform.html` (disponível no diretório `cap10/`) que já contém todo o formulário montado em HTML (figura ao

FORMULÁRIO DE RESERVA		
Preencha o formulário abaixo (um para cada participante):		
Participante	Viley E. Coyote	
Empresa	Exterminadores de Papagaios S/A	
Endereço	Rua Boemia, 235	
CEP	01423-002	Bairro Jd. Paulista
Cidade	São Paulo	Estado DF
Telefone	(011) 814123	Fax (011) 81623
E-mail	coyote@acme.com	
Dia 9/11/98 (2ª feira) - Selecione os minicursos abaixo de acordo com a sua ordem de preferência:		
<input type="checkbox"/> Minicurso 1 - Migrando de C/C++ para Java R\$ 170,00 <input type="checkbox"/> Minicurso 2 - Usando Java em Ambientes Distribuídos R\$ 170,00 <input type="checkbox"/> Minicurso 3 - Desenvolvimento em Java com JavaBeans® R\$ 170,00		
<input checked="" type="checkbox"/> Dia 10/11/98 (3ª feira) R\$ 160,00 <input checked="" type="checkbox"/> Dia 11/11/98 (4ª feira) R\$ 110,00 <input type="checkbox"/> Dia 12/11/98 (5ª feira) R\$ 160,00		
<input type="checkbox"/> Participação em todo o Congresso R\$ 450,00 <small>Nota: é necessário selecionar um minicurso acima!</small>		
<input type="button" value="Inscrire-se"/> <input type="button" value="Congresso"/>		

lado). Escreva uma rotina JavaScript que verifique os campos de texto, campos numéricos, caixas de checagem e listas de seleção para que estejam de acordo com as regras abaixo:

- a) Os únicos campos que podem permanecer vazios são **Empresa**, **Bairro** e **Fax**.
- b) Os campos **CEP** e **Telefone** só podem conter caracteres numéricos (0 a 9), traço “-”, espaço “ ” e parênteses “(” e “)”
- c) O campo **E-mail** deve necessariamente conter um caractere “@”
- d) Se o usuário escolher um minicurso (primeira lista de seleção) e marcar os três dias do congresso (três caixas de checagem), a opção “Participação em todo o congresso” (caixa de checagem) deverá ser selecionada, e as outras três, desmarcadas. Se o usuário marcar “Participação em todo o congresso”, as outras três opções devem ser desmarcadas.
- e) Se o usuário decidir participar de todo o evento, ele deve escolher um minicurso. Se escolher uma segunda ou terceira opção, deve necessariamente escolher uma primeira.
- f) Se tudo estiver OK, uma janela de alerta deverá ser apresentada ao usuário informando-o que os dados foram digitados corretamente.

A solução é apresentada na seção seguinte e pode ser encontrada no arquivo **validformsol.html**. Uma outra versão disponível no arquivo **job_form.html** gera uma nova página *on-the-fly* para confirmação e envio ao servidor (em vez de uma janela de alerta), caso os dados sejam digitados corretamente. É proposta como exercício.

Solução

Toda a validação dos dados ocorre no método **validar(dados)**. O formulário é passado como argumento da função e passa a ser representado pela variável local **dados**. Dentro da função **validar()**, cinco outras funções são chamadas várias vezes. Cada uma realiza uma tarefa solicitada nos requisitos (a) a (e) do exercício, e retorna **true** se os dados estavam corretos. No final da função **validar()**, se nenhuma das chamadas de função retornou **false**, um diálogo de alerta é mostrado avisando que os dados estão corretos:

```
function validar(dados) {
    // requisito (a) - verifica campos vazios
    if (!checkVazios(dados.Nome, "Participante")) return;
    if (!checkVazios(dados.Endereco, "Endereço")) return;
    if (!checkVazios(dados.Cidade, "Cidade")) return;
    if (!checkVazios(dados.CEP, "CEP")) return;
    if (!checkVazios(dados.Telefone, "Telefone")) return;
    if (!checkVazios(dados.Email, "Email")) return;

    // requisito (b) - verifica campos numéricos
    if (!checkNumericos(dados.Telefone)) return;
    if (!checkNumericos(dados.CEP)) return;
```

```

//requisito (c) - verifica campo de email
    if (!checkEmail(dados.Email) ) return;
// requisito (d) - organiza selecoes
    checkDias(dados.Tudo, dados.Dia10, dados.Dia11, dados.Dia12, dados.Opcão_1);
// requisito (e) - garante selecao de minicursos
    if (!checkMinis(dados.Tudo, dados.Opcão_1, dados.Opcão_2, dados.Opcão_3))
        return;
// requisito (f) -se chegou até aqui, tudo eh true: diga OK!
    alert("Dados digitados corretamente!");
}

```

O primeiro requisito (a) deve verificar se seis campos não estão vazios. A função `checkVazios(elemento, "string")` verifica o comprimento do valor de cada elemento *Text*. Se o texto na propriedade `value` do objeto tiver comprimento inferior a um caractere, a função retorna `false`:

```

function checkVazios(elemento, nome) {
    if (elemento.value.length < 1) {
        alert("O campo " + nome + " não pode ser vazio!");
        elemento.focus();
        elemento.select();
        return false;
    } else return true;
}

```

O segundo requisito (b) é cumprido pela função `checkNumericos()` que verifica se os campos numéricos contém caracteres ilegais. Os caracteres legais são “ ” (espaço), “-”, “(”, “)” e os dígitos de 0 a 9. Os elementos `Telefone` e `CEP` são passados para a função `checkNumericos()` que faz a verificação. Se houver qualquer outro caractere que não os citados no campo de textos, o formulário não será enviado e a função retornará `false`:

```

function checkNumericos(elemento) {
    for (i = 0; i < elemento.value.length; i++) {
        ch = elemento.value.charAt(i);
        if ((ch < '0' || ch > '9') && ch != '-'
            && ch != ' ' && ch != ')' && ch != '(') {
            alert("O campo " + elemento.name +
                  " só aceita números, parênteses, espaço e '-'");
            elemento.focus();
            elemento.select();
            return false;
        }
    }
    return true;
}

```

O requisito (c) pede para verificar se a informação de email contém o caractere “@”. Se o índice do substring do string `value` contendo o “@” for “-1”, o caractere não existe no

campo `value` do objeto, portanto não é um e-mail válido e a função `checkEmail()` retornará `false`:

```
function checkEmail(elemento) {
    if (elemento.value.lastIndexOf("@") == -1) {
        alert("Formato ilegal para E-mail");
        elemento.focus();
        elemento.select();
        return false;
    } else return true;
}
```

O requisito (d) pede duas coisas: para deselecionar as três caixas de checagem intermediárias, se a caixa “Tudo” estiver selecionada e para marcar a caixa “Tudo” se o usuário tiver escolhido um minicurso e marcado todas as caixas intermediárias. A função `checkDias()` recebe como argumentos 4 objetos *Checkbox* (correspondentes às caixas intermediárias e a caixa “Tudo”) e um objeto *Select* (correspondente à seleção do primeiro minicurso) e faz todas as alterações.

```
function checkDias(ck, ck10, ck11, ck12, m1) {
    if (ck.checked) { // caixa "Tudo" está marcada...
        ck10.checked = false; // desmarque as intermediárias...
        ck11.checked = false;
        ck12.checked = false;
    }
    // todas as intermediarias ligadas e minicurso escolhido...
    if (ck10.checked && ck11.checked && ck12.checked && m1.selectedIndex != 0) {
        ck10.checked = false;
        ck11.checked = false; // desmarque todas e...
        ck12.checked = false;
        ck.checked = true; // ... marque a caixa "Tudo"
    }
}
```

Finalmente, a função `checkMinis()` cumpre o requisito (e) recebendo como argumentos o *Checkbox* que representa a opção de participar de todo o evento e os três componentes *Select*. Verifica se um minicurso foi selecionado (`m1`), quando o usuário selecionou `ckTudo` e garante que uma primeira opção de minicurso foi selecionada (`m1`) quando há uma segunda (`m2`) ou terceira (`m3`) seleção. O índice zero em um *Select* corresponde a uma opção não selecionada:

```
function checkMinis(ckTudo, m1, m2, m3) {
    if (m1.selectedIndex == 0 && ckTudo.checked) {
        alert("Por favor, selecione um minicurso!");
        m1.focus();
        return false;
    }
}
```

```

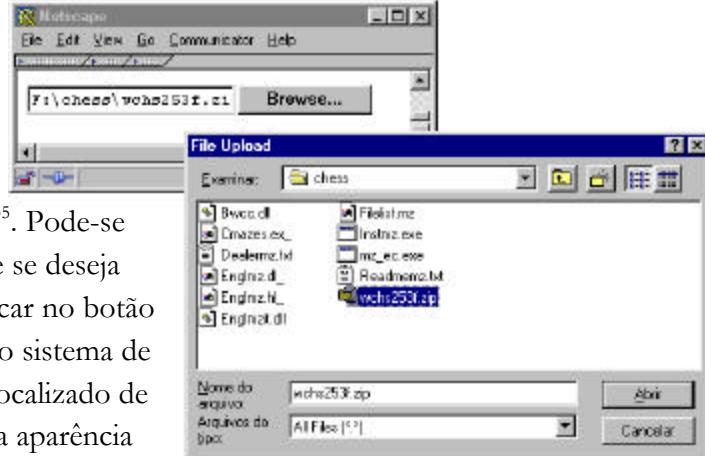
        else if ((m2.selectedIndex != 0 && m1.selectedIndex == 0) ||
                  (m3.selectedIndex != 0 && m1.selectedIndex == 0)) {
            alert("Por favor, selecione um minicurso como primeira opção!");
            m1.focus();
            return false;
        } else return true;
    }
}

```

Vários melhoramentos podem ser introduzidos nesta aplicação. Mudando a ordem de alguns testes na função `valida()`, por exemplo (que foram agrupados de acordo com os requisitos) pode tornar a interface mais eficiente. Algumas outras modificações são propostas em exercícios. A solução completa deste exercício está no arquivo `validformsol.html`.

Objeto File

File (ou *FileUpload*⁴) representa um dispositivo de entrada que permite o envio de um arquivo no cliente ao servidor. Na tela do browser, aparece como uma caixa de texto e um botão com o rótulo “Browse...” ou “Procurar...”⁵. Pode-se digitar o caminho absoluto ao arquivo que se deseja enviar ao servidor na caixa de texto ou clicar no botão “Browse...” e fazer aparecer um diálogo do sistema de arquivos, que permite que o arquivo seja localizado de forma interativa. A figura ao lado mostra a aparência do objeto *File* no browser Netscape em Windows 95 e a janela de diálogo que aparece ao se apertar o botão “Browse...”.



O objeto *File* é criado em HTML através de um elemento `<INPUT>`. A sintaxe geral do componente está mostrada abaixo:

```

<INPUT TYPE="file"
       NAME="nome_do_componente"
       ONBLUR="código JavaScript"
       ONFOCUS="código JavaScript"
       ONCHANGE="código JavaScript" >

```

Um objeto *File* só poderá ser manipulado em JavaScript se seu descritor `<INPUT>` estiver dentro de um bloco `<FORM>...</FORM>`. Como qualquer outro componente, é um elemento do formulário e pode ser acessado através do vetor `elements`, ou através do seu

⁴ Nomenclatura usada pela Netscape.

⁵ Depende da versão e fabricante do browser. Este rótulo não pode ser mudado.

nome, especificado pelo atributo **NAME**. O trecho de código abaixo mostra como acessar um elemento *File* chamado **nomeFup** e que é o sétimo elemento do primeiro formulário de uma página:

```
fup = document.forms[0].nomeFup      // ou ...
fup = document.forms[0].elements[6]
```

Para que seja possível a transferência de arquivos ao servidor, o servidor deverá permitir o recebimento de dados. O browser também deve passar os dados ao servidor usando o método de requisição **POST** e no formato **multipart/form-data**, que pode ser definido através do atributo **ENCTYPE** de **<FORM>** ou na propriedade **encoding**, de **Form**:

```
<FORM ENCTYPE="multipart/form-data" ACTION="..." METHOD="POST">
    <INPUT TYPE="file">
</FORM>
```

Todas as propriedades dos objetos *File* são somente-leitura. Estão listadas na tabela abaixo:

Propriedade	Descrição
form	<i>Form.</i> Referência ao formulário que contém este componente.
name	<i>String.</i> Contém o valor do atributo HTML NAME . (<i>read-only</i>)
type	<i>String.</i> Contém o valor do atributo HTML TYPE . (<i>read-only</i>)
value	<i>String.</i> Contém o texto no campo de textos do objeto, que corresponde ao arquivo a ser enviado ao servidor. É <i>read-only</i> por questões de segurança.

File só possui os dois métodos listados abaixo. Ambos provocam eventos de ativação/desativação do componente.

Método	Ação
focus()	Ativa o botão.
blur()	Desativa o botão.

Eventos

Os eventos suportados são dois. Os atributos HTML abaixo respondem aos eventos interpretando o código JavaScript contido neles:

- **ONFOCUS** – quando o usuário seleciona a caixa de textos ou o botão de *File*.
- **ONBLUR** – quando o usuário, que tinha o componente selecionado, seleciona outro componente.

Exercícios

- 10.1 Após a validação dos dados no exercício resolvido, uma janela de alerta aparece na tela informando que os dados foram digitados corretamente. Acrescente uma nova função para substituir o alerta. Esta nova função deverá gerar uma nova página *on-the-fly* (veja o exercício resolvido do capítulo 8) com os dados que o usuário digitou. Deve dois botões na página de confirmação: um para voltar e alterar os dados e outro para enviar os dados ao servidor. Garanta que os dados sejam preservados em campos *Hidden* na nova página para que o programa no servidor possa usá-los.
- 10.2 No formulário do exercício resolvido, é preciso escolher três minicursos. Altere o código de forma que não seja possível o usuário definir o segundo ou terceiro sem antes definir o primeiro (na versão atual isto só ocorre quando o formulário está para ser enviado). Garanta também que: a) quando o usuário fizer uma seleção na primeira lista, ele não consiga fazer a mesma seleção nas outras listas, e b) que a terceira opção seja automaticamente escolhida assim que o usuário fizer duas seleções.

11

Cookies

UM COOKIE É UMA PEQUENA QUANTIDADE DE INFORMAÇÃO que pode ser armazenada na máquina do cliente através de instruções enviadas pelo servidor ou contidas em uma página HTML. É uma informação que pode persistir por toda uma sessão do cliente em um site, ou por mais tempo ainda. Um cookie pode ser gravado em uma página e recuperado em outra, permitindo o acesso a propriedades, informações ou preferências do usuário a qualquer momento, de qualquer página do site.

Um cookie está sempre associado a um browser e a um domínio. Não é um padrão formal ou especificação, e a implementação dos cookies é dependente de browser e fabricante. A sua manipulação, porém, é baseada em padrões HTTP (cabeçalhos) e tem amplo suporte tanto de tecnologias client-side, como JavaScript, como de tecnologias server-side como ASP, Servlets, LiveWire e CGI.

O objetivo deste capítulo é demonstrar o uso de cookies em JavaScript, e apresentar algumas aplicações como o “Carrinho de Compras”. A próxima seção, introduz a arquitetura de cookies cujo conhecimento é essencial para o uso eficiente de cookies com JavaScript.

Cookies em HTTP

A tecnologia conhecida como HTTP Cookies, surgiu em 1995 como um recurso proprietário do browser Netscape, que permitia que programas CGI gravassem informações em um arquivo de textos controlado pelo browser na máquina do cliente. Por oferecer uma solução simples para resolver uma das maiores limitações do HTTP – a incapacidade de preservar o estado das propriedades dos documentos em uma mesma sessão – os cookies logo passaram a ser suportados em outros browsers e por linguagens e tecnologias de suporte a operações no cliente e servidor. Hoje, embora não seja ainda um padrão formal, é um padrão de fato adotado pela indústria de software voltada à Web e Internet.

Um cookie não é um programa de computador, portanto não pode conter um vírus executável ou qualquer outro tipo de conteúdo ativo. Pode ocupar no máximo 4 kB de espaço no computador do cliente. Um servidor pode definir no máximo 20 cookies por domínio (endereço de rede) e o browser pode armazenar no máximo 300 cookies. Estas restrições referem-se ao browser Netscape e podem ser diferentes em outros browsers.

Há várias formas de manipular cookies:

- Através de CGI ou outra tecnologia de servidor, como LiveWire, ASP ou Servlets, pode-se criar ou recuperar cookies.
- Através de JavaScript também pode-se criar ou recuperar cookies.
- Através do descritor <META> em HTML, pode-se criar novos cookies ou redefinir cookies existentes, mas não recuperá-los.

Um cookie é enviado para um cliente no cabeçalho HTTP de uma resposta do servidor. Além da informação útil do cookie, que consiste de um par nome/valor, o servidor também inclui um informações sobre o domínio onde o cookie é válido, e o tempo de validade do mesmo.

Criação de cookies via cabeçalhos HTTP

Cookies podem ser criados através de um cabeçalho HTTP usando CGI. Toda resposta do servidor a uma requisição do browser sempre contém um conjunto de cabeçalhos com informações sobre os dados enviados. Essas informações são essenciais para que o browser consiga decodificar os dados, que ele recebe em pedaços, como um fluxo irregular de bytes. Os cabeçalhos trazem o comprimento total dos dados, o tipo dos dados (se é imagem, página HTML, som, etc.), a versão e nome do servidor, entre outras informações. Um exemplo típico de resposta de um servidor Web a um browser que solicita uma página HTML está mostrada abaixo:

```
HTTP/1.0 200 OK
Date: Friday, June 13, 1997
Server: Apache 1.02
Content-type: text/html

<HTML><HEAD>
<TITLE> Capítulo 11</TITLE>
(...)
```

A primeira linha acima não é um cabeçalho, mas o *status* da resposta do servidor. No caso acima, indica sucesso através do código 200. Um outro código freqüente na Web é o código 404, correspondente à recurso não encontrado.

Toda linha de cabeçalho HTTP tem a forma:

NomeDoCabeçalho: Valores ↴

onde ↵ corresponde a uma quebra de linha. O nome do cabeçalho será ignorado pelo browser, se ele não souber o seu significado. Os valores têm um formato específico para cada cabeçalho. O conjunto de caracteres suportado é ASCII de 7 bits, portanto, é necessário converter acentos e outros caracteres antes de usá-los como cabeçalhos.

O bloco de cabeçalhos é separado dos dados por uma linha em branco (dois caracteres de nova-linha seguidos). Ao receber a resposta, o browser separa o cabeçalho do restante da informação, identifica o formato e comprimento dos dados (que vêm depois da linha em branco) e os formata na sua área de visualização, se o seu tipo de dados for suportado.

Um bloco de cabeçalhos de resposta é gerado pelo servidor Web sempre que o browser solicita uma página estática. Parte ou todo o bloco de cabeçalhos também pode ser gerado por um programa CGI ou equivalente. Quando um programa CGI gera um cabeçalho, pode incluir outros campos de informação sobre a página que o servidor não inclui por *default*. Pode, por exemplo, definir um ou mais cabeçalhos **Set-Cookie**, que irão fazer com que o browser guarde a informação passada em cookies:

```
(... outros cabeçalhos ...)
Set-Cookie: cliente=jan0017
Set-Cookie: nomeclt=Marie
Content-type: text/html

(... dados ...)
```

Quando receber a resposta do servidor e interpretar os cabeçalhos acima, o browser irá gravar dois novos cookies na memória contendo as informações **cliente=jan0017** e **nomeclt=Marie**. Essas informações podem ser recuperadas em qualquer página que tenha origem no servidor que definiu os cookies enquanto a presente sessão do browser estiver aberta.

Um cabeçalho **Set-Cookie** pode conter muito mais informações que alteram a forma como o cookie é tratado pelo browser. Por exemplo, se o cookie tiver um campo **expires** com uma data no futuro, as informações do cookie serão gravadas em arquivo e persistirão além da sessão atual do browser:

```
Set-Cookie: nomeclt=Marie; expires=Monday, 15-Jan-99 13:02:55 GMT
```

A sintaxe completa do cabeçalho **Set-Cookie** está mostrada abaixo. Os campos são separados por ponto-e-vírgula. Todos, exceto o primeiro campo que define o nome do cookie, são opcionais.

```
Set-Cookie: nome_do_cookie=valor_do_cookie;
expires=data no formato GMT;
domain=domínio onde o cookie é válido;
path=caminho dentro do domínio onde o cookie é válido;
secure
```

Os campos do cabeçalho **Set-Cookie** são usados na definição de cookies tanto em CGI quanto em JavaScript. O significado dos campos está relacionado na tabela abaixo:

Campo	Descrição
<code>nome=valor</code>	<i>Este campo é obrigatório.</i> Seqüência de caracteres que não incluem acentos, ponto-e-vírgula, percentagem, vírgula ou espaço em branco. Para incluir esses caracteres é preciso usar um formato de codificação estilo URL. Em JavaScript, a função <code>escape()</code> codifica informações nesse formato e a função <code>unescape()</code> as decodifica.
<code>expires=data</code>	<i>Opcional.</i> Se presente, define uma data com o período de validade do cookie. Após esta data, o cookie deixará de existir. Se este campo não estiver presente, o cookie só existe enquanto durar a sessão do browser. A data deve estar no seguinte formato: <code>DiaDaSemana, dd-mes-aa hh:mm:ss GMT</code> Por exemplo: <code>Monday, 15-Jan-99 13:02:55 GMT</code> O método <code>toGMTString()</code> dos objetos <code>Date</code> gera uma data compatível com este formato.
<code>domain=domínio</code>	<i>Opcional.</i> Se presente, define um <code>domínio</code> onde o cookie atual é válido. Se este campo não existir, o cookie será válido em todo o domínio onde o cookie foi criado.
<code>path=caminho</code>	<i>Opcional.</i> Se presente, define o <code>caminho</code> onde um cookie é válido em um domínio. Se este campo não existir, será usado o caminho do documento que criou o cookie.
<code>secure</code>	<i>Opcional.</i> Se presente, impede que o cookie seja transmitido a menos que a transmissão seja segura (baseada em SSL ou SHTTP).

Criação de cookies via HTML

Um cookie pode ser criado através de HTML usando o descritor `<META>` e seu atributo `HTTP-EQUIV`. O atributo `HTTP-EQUIV` deve conter um cabeçalho HTTP. O valor do cabeçalho deve estar presente no seu atributo `CONTENT`. A presença de um descritor `<META>` dentro de um bloco `<HEAD>` de uma página HTML, criará um cookie no cliente quando este for interpretar a página.

```

<HEAD>
<META HTTP-EQUIV="Set-Cookie"
      CONTENT="nomeclt=Marie; expires=Monday, 15-Jan-99 13:02:55 GMT">
(...)
</HEAD>

```

Espaço de nomes de um Cookie

Várias páginas de um site podem definir cookies. O espaço de nomes de um cookie é determinado através de seu domínio e caminho. Em um mesmo espaço de nomes, só pode haver um cookie com um determinado nome. A definição de um cookie de mesmo nome que um cookie já existente no mesmo espaço, sobrepõe o cookie antigo.

Por *default*, o espaço de nomes de um cookie é todo o domínio onde foi criado. Para definir um novo domínio, mais restritivo, é preciso definir o campo `domain`. Por exemplo, se o domínio de um cookie é `.biscoitos.com`, ele pode ser lido nas máquinas `agua.biscoitos.com` e `chocolate.biscoitos.com`. Para restringi-lo à máquina `chocolate.biscoitos.com`, o campo `domain` deve ser especificado da forma:

```
domain=chocolate.biscoitos.com
```

Somente máquinas dentro do domínio `.biscoitos.com` podem redefinir o domínio. Ele necessariamente tem que ser mais restritivo que o *default*.

O caminho dentro do domínio onde o cookie é válido é o mesmo caminho onde foi criado. O caminho pode ser alterado de forma que tenha um valor mais restritivo definindo o campo `path`. Por exemplo, se um cookie é válido em todos os subdiretórios a partir da raiz, seu path é `/`. Para que só exista dentro de `/bolachas/`, o campo `path` pode ser especificado da forma:

```
path=/bolachas/
```

Um cookie chamado `bis` definido em `/` não colide com um cookie também chamado `bis` definido em `/bolachas/`.

Um cookie pode ser apagado se for definido um novo cookie com o mesmo nome e caminho que ele e com data de vencimento (campo `expires`) no passado.

Recuperação de cookies

Toda requisição de um browser ao servidor consiste de uma linha que contém o método de requisição, URL destino e protocolo, seguida de várias linhas de cabeçalho. É através de cabeçalhos que o cliente passa informações ao servidor, como, por exemplo, o nome do browser que enviou o pedido. Uma requisição HTTP típica tem a forma:

```
GET /index.html HTTP/1.0
User-Agent: Mozilla/4.5 (WinNT; I) [en]
Host: www.alnitak.org.br
Accept: image/gif, image/jpeg, */*
```

Quando um cookie é recuperado pelo browser, ele é enviado em todas as requisições à URLs que fazem parte do seu espaço de nomes, através do cabeçalho do cliente `Cookie`. Apenas o par nome/valor é armazenado no cabeçalho. As informações dos campos `expires`, `path`, e `domain` não aparecem:

Cookie: cliente=jan0017; nomeclt=Marie

O servidor pode recuperar as informações do cookie através do cabeçalho ou através da variável de ambiente `HTTP_COOKIE`, definida na maior parte dos servidores.

Cookies em JavaScript

Cookies podem ser manipulados em JavaScript através da propriedade `document.cookie`. Esta propriedade contém uma *String* com o valor de todos os cookies que pertencem ao espaço de nomes (domínio/caminho) do documento que possui a propriedade. A propriedade `document.cookie` é usada tanto para criar como para ler cookies.

Para definir um novo cookie, basta atribuir um string em um formato válido para o cabeçalho HTTP `Set-Cookie` à propriedade `document.cookie`. Como cookies não podem ter espaços, ponto-e-virgula e outros caracteres especiais, pode-se usar a função `escape(String)` antes de armazenar o cookie, para garantir que tais caracteres serão preservados em códigos hexadecimais:

```
nome="usuario";
valor=escape("João Grandão"); // converte para Jo%E3o%20Grand%E3o
vencimento="Monday, 22-Feb-99 00:00:00 GMT";
document.cookie = nome + "=" + valor + "expires=" + vencimento;
```

A propriedade `document.cookie` não é um tipo *string* convencional. Não é possível apagar os valores armazenados simplesmente atribuindo um novo valor à propriedade. Novos valores passados à `document.cookie` criam novos cookies e aumentam a *string*:

```
document.cookie = "vidacurta=" + escape("É só por hoje!");
document.cookie = "vidalonga=" + escape("É por duas semanas!") +
    "expires=" + vencimento;
```

Os cookies estão todos armazenados na propriedade `document.cookie`, em um string que separa os cookies pelo “;”. Se o valor de `document.cookie` for impresso agora:

```
document.write(document.cookie);
```

O texto a seguir será mostrado na página, com os três cookies separados por “;”:

```
usuario=Jo%E3o%20Grand%E3o; vidacurta=%C9%20s%F3%20por%20hoje%21;
vidalonga=%C9%20por%20duas%20semanas%21
```

As letras acentuadas, espaços e outros caracteres especiais foram substituídos pelo seu código hexadecimal, após o “%”. Para decodificar o texto, pode-se usar `unescape()`:

```
document.write(unescape(document.cookie));
```

Se não for definido um campo `expires` com uma data no futuro, o cookie só sobreviverá à sessão atual do browser. Assim que o browser for fechado, ele se perderá. Por exemplo, se a sessão atual do browser for encerrada e o browser for novamente iniciado,

carregando a página que imprime `document.cookie`, teremos apenas dois cookies, e não três como antes. Isto porque o cookie `vidacurta` foi definido *sem* o campo `expires`:

```
usuario=Jo%E3o%20Grand%E3o; vidalonga=%C9%20por%20duas%20semanas%21
```

Geralmente queremos definir o tempo de validade de um cookie de acordo com um intervalo relativo de tempo e não uma data absoluta. O formato de data gerado pelo método `toGMTString()` de `Date` é compatível com o formato aceito pelos cookies. Sendo assim, podemos criar, por exemplo, uma função para definir um cookie com validade baseada em um número de dias a partir do momento atual:

```
function setCookie(nome, valor, dias) {
    diasms = (new Date()).getTime() + 1000 * 3600 * 24 * dias;
    dias = new Date(diasms);
    expires = dias.toGMTString();
    document.cookie = escape(nome) + "=" +
                      escape(valor) + "; expires=" + expires;
}
```

A função acima pode ser chamada tanto para criar cookies como para *matar* cookies no mesmo espaço de nomes. Para criar um novo cookie, com duração de 12 horas:

```
setCookie("cook", "Um, dois, tres", 0.5);
```

Para matar o cookie criado com a instrução acima, basta criar um homônimo com data de vencimento no passado. Podemos fazer isto passando um *número negativo* como tempo de validade em `setCookie()`:

```
setCookie("cook", "", -365);
```

Para extrair as informações úteis de um cookie, usamos os métodos de `String` que realizam operações com caracteres (`indexOf()`, `substring()`, `split()`). Uma invocação do método `split(";")` coloca todos os pares nome/valor em células individuais de um vetor.

```
cookies = document.cookie.split(";");
// cookies[0] = "usuario=Jo%E3o%20Grand%E3o"
// cookies[1] = "vidacurta=%C9%20s%F3%20por%20hoje%21"
// cookies[2] = "vidalonga=%C9%20por%20duas%20semanas%21"
```

Uma segunda invocação de `split()`, desta vez sobre cada um dos pares nome/valor obtidos acima, separando-os pelo “=”, cria um vetor bidimensional. O string `cookies[i]` se transforma em um vetor para receber o retorno de `split("=")`. Criamos então duas novas propriedades: `name` e `value` para cada cookie, que contém respectivamente, o nome e valor, já devidamente decodificados:

```
for (i = 0; i < cookies.length; i++) {
    cookies[i] = cookies[i].split("=");
    cookies[i].name = unescape(cookies[i][0]);
    cookies[i].value = unescape(cookies[i][1]);
}
```

Carrinho de compras

Os cookies são essenciais na construção de aplicações de comércio eletrônico, pois permitem passar informações de uma página para outra e manter os dados persistentes na máquina do cliente por mais de uma sessão.

O carrinho de compras virtual consiste de um ou mais cookies que guardam as preferências do usuário enquanto ele faz compras pelo site. No final, quando o usuário decide fechar a conta, o(s) cookie(s) são lido(s), os dados são extraídos, formatados e mostrados na tela ou enviados para o servidor. Mesmo que a conexão caia ou que ele decida continuar a compra no dia seguinte, os dados podem ser preservados, se os cookies forem persistentes (terem um campo *expires* com uma data de vencimento no futuro). No final, depois que o usuário terminar a transação, o cookie não é mais necessário e é descartado.

No exercício a seguir, desenvolveremos uma pequena aplicação de comércio eletrônico usando cookies e JavaScript.

Exercício Resolvido

A Loja XYZ S/A deseja vender seus produtos esquisitos na Web. A implantação do serviço consiste de duas etapas. A primeira é a criação de um carrinho de compras virtual para que os clientes possam selecionar seus produtos. A segunda etapa envolve questões relacionadas ao servidor, como o acesso ao banco de dados da empresa, segurança, etc. Ficamos encarregados de desenvolver a primeira etapa e decidimos usar JavaScript.

A sua tarefa é desenvolver os requisitos mínimos para lançar a versão 0.1 da aplicação. Esta versão ainda não é adequada à publicação no site do cliente, mas já possui as características mínimas para demonstrar os principais recursos do site. Os arquivos HTML já estão prontos no subdiretório **cap11/carrinho/**.

Várias páginas da aplicação estão mostradas na figura abaixo. A primeira é a home

The figure consists of three screenshots of a web browser window showing a shopping cart application. The browser title bar reads "Loja XYZ - Netscape".

Screenshot 1: Loja XYZ produtos raros
The page title is "Loja XYZ produtos raros". It features a heading "Vende-se Qualquer Coisa" and a sub-section "Escolha a categoria de produtos que você deseja comprar:". Below this is a list of categories:

- Dinossauros
- Livros
- Invertebrados
- Outros Produtos

A message at the bottom says "Veja aqui, o conteúdo do seu carrinho de co...".

Screenshot 2: Livros: Lançamentos
The page title is "LIVROS: Lançamentos". It shows a book cover for "Zen e a Arte de Criar Cupins" by Prof. Termitum Isopterus (Entomologista). The details include:

- 340 pp. Ilustrado com fotos.
- Editora O Cupim de Asa, 1977
- ISBN: 0101673227
- Preço: R\$25,00

A "Colocar no carrinho" button is present.

Screenshot 3: Carrinho de Compras
The page title is "Carrinho de Compras". It displays a table of items:

Seleção	Código	Produto	Preço
<input checked="" type="checkbox"/>	B999	Zen e a Arte de Criar Cupins	R\$25,00
<input checked="" type="checkbox"/>	D449	Velocíssimo (Jávor)	R\$60,00
<input checked="" type="checkbox"/>	A572	Tarântulas	R\$35,00

Total a Pagar: R\$590,00

Buttons at the bottom include "Atualizar", "Escolher Carrinho", "Enviar Fatura", and "Voltar | Dinossauros | Livros | Invertebrados | Outros".

- a) Implementar os botões “Colocar no Carrinho” de cada produto para que armazenem um cookie contendo:

- um código de uma letra e três dígitos (por exemplo, X123, F555)
- uma descrição do produto
- preço do produto (por exemplo 5.99, 123.25)

O código e o preço podem ter valores arbitrários. O cookie deve durar 5 dias.

- b) Implementar a página carrinho.html para exibir os cookies armazenados. Esta página pode ser alcançada a partir de qualquer página do site (através de links comuns). Quando for carregada, deve mostrar na tela uma tabela com as informações armazenadas nos cookies. Se não houver cookies, ela deve ter a aparência da figura ao lado. A tabela deve ter o seguinte formato:

- *Primeira coluna:* Checkbox marcado. Se o usuário atualizar a página, o valor deste componente deve ser verificado. Se for `false`, o cookie correspondente deve ser eliminado.
 - *Segunda e terceira colunas:* código do produto e descrição obtidas a partir do cookie armazenado.
 - *Quarta coluna:* preço do produto obtido a partir do cookie. Todos os valores desta coluna devem ser somados e o total exibido no final da tabela.
- c) Implementar o botão “Atualizar”, que deverá recarregar a página e eliminar os cookies que não estiverem marcados.
- d) Implementar os botões “Esvaziar Carrinho”, que deverá eliminar todos os cookies e atualizar a página, e “Enviar Fatura”, que deverá eliminar todos os cookies e mostrar na tela um diálogo de alerta informando a ocorrência.



Solução

A solução do problema consiste de três partes: a criação dos cookies (a colocação dos produtos no carrinho), a leitura dos cookies (a visualização do conteúdo do carrinho) e a remoção dos cookies (o esvaziamento do carrinho).

Para criar os cookies (a), usamos a função `setCookie()` abaixo. Ela pode estar presente em todas as páginas de produtos ou em um arquivo externo (`.js`), importado em

cada página. A função recebe três argumentos apenas (estamos supondo que este domínio não terá outros cookies) que são um nome, um valor e um período de validade em dias:

```
<script>
    function setCookie(nome, valor, dias) {
        diasms = (new Date()).getTime() + 1000 * 3600 * 24 * dias;
        dias = new Date(diasms);
        expires = dias.toGMTString();
        document.cookie = escape(nome) + "=" +
                          escape(valor) + "; expires=" + expires;
    }
</script>
```

Precisamos armazenar três informações por produto. Se usássemos três cookies para cada produto, em pouco tempo ficaríamos sem cookies, pois o browser limita o número de cookies em 20 por domínio. Precisamos, portanto, armazenar as informações em o mínimo de cookies possível. Optamos, nesta primeira versão, por definir um cookie por produto¹. Para separar os dados, usamos o “&” como delimitador do *string*:

```
<form>
    <input type=button value="Colocar no carrinho"
           onclick="setCookie('dino1','D372&Brontossauro&1500.00',5)">
</form>
```

Os códigos, nomes de cookie e preços escolhidos são totalmente arbitrários. Tendo todos os botões implementados da forma acima, com nomes distintos para cada cookie, podemos armazenar as informações no carrinho de compras apertando o botão ao lado de cada produto.

Para recuperar os cookies (b), precisamos alterar apenas a página `carrinho.html`. Nesta página, também iremos precisar da função `setCookie()` para apagar os cookies, como foi pedido no requisito (c). Para ler os cookies, definimos duas funções: `getCookies()`, retorna um vetor bidimensional com todos os cookies disponíveis:

```
<script>
    // devolve todos os cookies em uma matriz (num_cookies x 2)
    function getCookies() {
        cookies = document.cookie.split(";");
        for (i = 0; i < cookies.length; i++) {
            cookies[i] = cookies[i].split("=");
            cookies[i][0] = unescape(cookies[i][0]); // nome
            cookies[i][1] = unescape(cookies[i][1]); // valor
        }
        return cookies; // retorna matriz[n][2]
    }

```

¹ Ainda não é a melhor idéia, pois aproveitamos pouco dos 4kB permitidos a cada cookie. Idealmente colocaríamos vários produtos em um único cookie e evitariam armazenar informações como descrição de produtos que poderiam ser recuperadas do banco de dados.

e `getCookie(nome)`, que chama `getCookies()` e retorna um cookie pelo nome:

```
// retorna o valor de um cookie fornecendo-se o nome
function getCookie(name) {
    cookies = getCookies();
    for (i = 0; i < cookies.length; i++) {
        if(cookies[i][0] == name) {
            return cookies[i][1];      // valor
        }
    }
    return null;
}
</script>
```

Com essas funções, temos condições de montar a tabela com os produtos. Como estamos supondo que não há outros cookies neste domínio² podemos verificar se o carrinho está vazio, simplesmente verificando se o string `document.cookie` está vazio:

```
<table border><tr>
<th>Seleção</th><th>Código</th><th>Produto</th><th>Preço</th></tr>
<script>
if (!document.cookie)
    document.write("<tr><td colspan=3>Seu carrinho está vazio!</td></tr>");
(...)
```

Caso existam cookies, os recuperamos e colocamos na variável `cooks`. Partimos as informações no valor de cada cookie (`cooks[i][1]`) pelo “&” e utilizamos a informação de preço (última das três informações armazenadas em cada cookie) para montar o total. Usamos o nome de cada cookie para definir o nome de um *Checkbox*, que é criado previamente ligado (contém atributo `CHECKED`):

```
else {
    cooks = getCookies();
    total = 0;
    for (i = 0; i < cooks.length; i++) {
        partes = cooks[i][1].split("&"); // partes eh Array
        total += parseFloat(partes[partes.length-1]);
        partes = partes.join("</td><td>"); // agora partes eh String
        partes = "</td><td>" + partes + "</td></tr>";
        partes = "<tr><td><input type=checkbox name='"
            + cooks[i][0] + "' checked size=3>" + partes;
        document.write(partes + "\n");
    }
    document.write("<tr><td colspan=4 align=right><b>TOTAL A PAGAR</b> R$"
        + formata(total) + "</td></tr>");
}
```

² Teremos que levar em conta a possibilidade de haver outros cookies neste domínio em uma versão definitiva.

```
</script>
</table>
```

Utilizamos a função `formata(numero)` para formatar o total (não está mostrada aqui³), para que ele seja exibido com duas casas após o ponto decimal.

O requisito (c) pede para implementar uma função que atualize o carrinho de compra, verificando se algum cookie deve ser removido. O botão atualizar chama a função `atualiza()`, mostrada abaixo, que localiza todos os *Checkbox* da página e extrai o seu nome (que corresponde ao nome de um cookie existente). Se algum *Checkbox* está desligado, um cookie com o seu nome será definido com data de vencimento no passado, o que provocará a morte do cookie:

```
function atualiza() {
    for(i = 0; i < document.forms[0].elements.length; i++) {
        if(document.forms[0].elements[i].type == "checkbox") {
            chkbox = document.forms[0].elements[i];
            nome = chkbox.name;
            if(!chkbox.checked) {
                setCookie(nome, "nada", -365); // mata cookie
            }
        }
    }
    location.reload(true); // atualiza a página
}
```

O último requisito (d) consiste da programação dos botões “Esvaziar Carrinho” e “Enviar Fatura”, que nesta versão, fazem praticamente a mesma coisa: matam todos os cookies. Em uma versão definitiva desta aplicação, o botão enviar fatura deverá enviar os dados para um lugar seguro antes de matar os cookies.

Criamos uma função especialmente para eliminar todos os cookies: `killAll()`. Ela localiza os cookies um a um e os redefine com uma data de um ano atrás.

```
function killAll() {
    if (document.cookie) {
        cooks = document.cookie.split("; ");
        for (i=0; i < cooks.length; i++) {
            nome = cooks[i].split "=");
            setCookie(unescape(nome[0]), "aaa", -365); // mata
        }
    }
}
```

A chamada do botão “Esvaziar Carrinho”, além de matar todos os cookies, recarrega a página para que a mudança seja visível, usando `location.reload()`:

³ Veja o código desta função na solução em cap10/carrinhosol/carrinho.html

```
<input type=button value="Esvaziar Carrinho"
       onclick="killAll(); location.reload(true)">
```

Veja a aplicação completa do carrinho de compras explorado neste exercício no subdiretório cap10/carrinhosol/.

Exercícios

- 11.1 Escreva um conjunto de funções gerais para definir cookies com todos os parâmetros possíveis (a função utilizada nos nossos exemplos apenas admite três parâmetros e não define cookies fora do espaço de nomes *default*). A função deve poder definir cookies com apenas os parâmetros nome e valor, com os três parâmetros que usamos nos exemplos ou com os 6 parâmetros possíveis. Deve ser possível criar um cookie das formas:

```
setCookie("visitante","Fulano de Tal", 10, "/sub","abc.com", true)
```

ou

```
setCookie("visitante","Fulano de Tal")
```

- 11.2 Escreva uma aplicação que informe ao usuário quantas vezes ele já visitou a página, quando foi, e de onde ele tinha vindo antes.

12

JavaScript e Java

APPLETS JAVA OFERECEM RECURSOS QUE VÃO MUITO ALÉM do que se dispõe com JavaScript e HTML. Por outro lado, applets pouco interagem com a página. Não podem alterar propriedades da página nem utilizar formulários HTML. JavaScript oferece recursos de programação e integração total com o browser e a página, mas não pode ir além das limitações do HTML e do browser. Usando Java e JavaScript juntos, une-se a riqueza de recursos de Java à integração do JavaScript com o browser o que permite explorar o melhor de cada tecnologia em uma mesma aplicação.

Os browsers mais populares suportam o controle de applets a partir de JavaScript e vice-versa. Isto inclui os browsers Netscape Navigator a partir da versão 3.0 e o browser Microsoft Internet Explorer a partir da versão 4.0¹. Neste capítulo, mostraremos como manipular com os applets em uma página Web, e exploraremos, com exemplos e exercícios resolvidos, a comunicação entre applets e JavaScript.

Applets Java

Applets são pequenas aplicações geralmente escritas em Java que são executadas pelo browser. Diferentemente do que ocorre com JavaScript, o código Java não é interpretado pelo browser. Um applet também não tem código Java que o browser possa interpretar, já que foi compilado para uma linguagem de máquina. Browsers que suportam Java possuem uma plataforma virtual, a “Java Virtual Machine”, que é capaz de interpretar a linguagem de máquina Java, chamada de *bytecode*.

Applets podem ser usados para desenvolver aplicações que seriam impossíveis em JavaScript por causa das limitações do HTML, do protocolo HTTP e do próprio browser. Com um applet, é possível estender um browser fazendo-o suportar, por exemplo, novos

¹ Além da comunicação entre applets e scripts, o Netscape Navigator, permite ainda que o programador utilize diretamente classes da API Java, chame seus métodos e crie objetos Java a partir de instruções JavaScript. Não discutiremos este recurso aqui por ele não ter suporte além dos browsers Netscape.

protocolos de comunicação e segurança, novos formatos de mídia, etc. O preço dessa liberdade é sua fraca integração com o HTML da página. Aplicações Web baseadas em Java pouco ou nada aproveitam do HTML da página a não ser um espaço gráfico para sua exibição. Com JavaScript, é possível aumentar essa integração.

Applets são aplicações gráficas e precisam de uma página HTML para poderem executar. São exibidos na página de forma semelhante a imagens: carregam um arquivo externo, ocupam um espaço com determinada altura e largura, e podem ter seu alinhamento em relação ao texto definido pelos mesmos atributos presentes em . A sintaxe do elemento HTML <APPLET> está mostrada abaixo. Tudo o que não estiver em negrito é opcional:

```
<APPLET  
    CODE="nome_do_programa.class"  
    HEIGHT="altura em pixels"  
    WIDTH="largura em pixels"  
    NAME="nome_do_objeto"  
    CODEBASE="diretório base do arquivo de classe Java"  
    ALT="texto descritivo"  
    HSPACE="margens externas laterais em pixels"  
    VSPACE="margens externas verticais em pixels"  
    ALIGN="left" ou "right" ou "top" ou "middle" ou "bottom" ou  
          "texttop" ou "absmiddle" ou "absbottom" ou "baseline"  
    MAYSCRIPT>  
        <PARAM NAME="nome" VALUE="valor">  
        ...  
        <PARAM NAME="nome" VALUE="valor">  
</APPLET>
```

Diferentemente de , o elemento <APPLET> é um bloco e possui um descritor de fechamento </APPLET>. Entre <APPLET> e </APPLET> pode haver nenhum ou vários elementos <PARAM>, que contém parâmetros necessários ou não (depende do applet) para o funcionamento da aplicação. Cada elemento <PARAM> contém um par de atributos obrigatórios. O valor do atributo NAME é definido pelo programador do applet. Através dele, o programa pode recuperar um valor que o autor da página (que não precisa saber Java) definiu no atributo VALUE.

O atributo MAYSCRIPT é necessário se o applet pretende ter acesso ao código JavaScript da página. Sem este atributo, qualquer tentativa do applet de acessar variáveis ou executar funções ou métodos JavaScript causará em uma exceção de segurança no applet.

Existem muitos applets úteis disponíveis gratuitamente na Web que podem ser usados por autores de páginas Web e programadores JavaScript sem que precisem saber Java. Os mais populares implementam banners para rolagem de texto, ícones inteligentes, gráficos, planilhas de dados e interfaces para bancos de dados. A maioria são configuráveis através de

parâmetros que o autor da página define em elementos <PARAM>. No apêndice A há uma lista de sites onde se pode encontrar tais applets.

O exemplo a seguir mostra como incluir o applet `Clock2` em uma página Web. Este applet é distribuído pela *Sun* gratuitamente em <http://java.sun.com> e juntamente com o ambiente de desenvolvimento Java. O applet pode ser incluído na página da forma *default*, sem especificar parâmetros, ou definindo parâmetros que alteram a cor de fundo, dos ponteiros e do mostrador.

O applet é distribuído com uma página HTML que mostra como usá-lo. Ele deve ser incluído na página HTML usando o nome do arquivo executável `java`, que é `Clock2.class` e deve ocupar uma área de no mínimo 170x150 pixels:

```
<applet code="Clock2.class" height=150 width=170></applet>
```

Com o código acima, o relógio aparece na página como mostrado na figura, com ponteiros azuis, visor com letras pretas e fundo branco. O autor do applet permite, porém, que o autor da página altere esses parâmetros através de descritores <PARAM>. Os três parâmetros modificáveis são:

- `bgcolor` – cor de fundo (branco é *default*)
- `fgcolor1` – cor dos ponteiros e dial (azul é *default*)
- `fgcolor2` – cor dos números e ponteiro de segundos (preto é *default*)

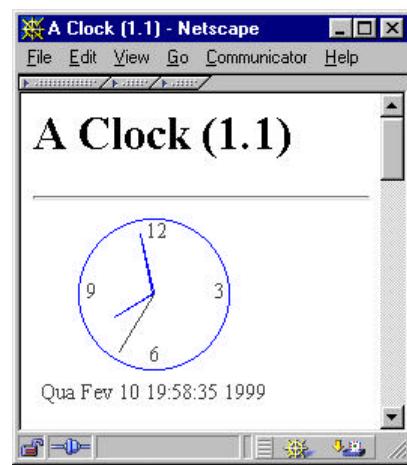
Todos os parâmetros devem receber como valor um número hexadecimal representando uma cor no formato RGB (mesmo formato usado em HTML): `ff0000` – vermelho, `ffffff` – branco, `0000ff` – azul, etc.

Portanto, para incluir o relógio acima em uma página, com um fundo cinza claro, ponteiros marrons e letras douradas, o código seria:

```
<applet code="Clock2.class" width=170 height=150>
    <param name=bgcolor value="#cccccc">
    <param name=fgcolor1 value="#800000">
    <param name=fgcolor2 value="#808000">
</applet>
```

Caso o applet esteja em um diretório diferente daquele onde está a página, será necessário usar o atributo `CODEBASE`, para informar a URL base. Por exemplo, se o arquivo `.class` que usamos acima estiver em <http://www.abc.com/clocks/>, precisamos usar:

```
<applet codebase="http://www.abc.com/clocks/" code="Clock2.class" ... >
    ...
</applet>
```



Objeto *Applet*

O tipo de objeto *Applet* representa, no modelo de objetos JavaScript, um applet Java embutido em uma página Web. Tendo uma referência para um objeto *Applet*, o programador pode controlar um applet Java usando JavaScript, *sem que precise ter acesso ao código* do applet. Precisará apenas saber os nomes dos métodos públicos do applet para que possa invocá-los via JavaScript. É possível também fazer o inverso: controlar JavaScript a partir de applets. Neste caso, *é preciso ter acesso ao código* do applet e conhecer a linguagem Java.

Não é possível criar objetos *Applet* usando JavaScript, apenas. Objetos *Applet* são fornecidos pelo código HTML da página. Se houver na página um bloco <APPLET> que tenha carregado um arquivo executável Java, existe um objeto *Applet* utilizável em JavaScript.

Uma página pode ter vários applets. Eles podem ser obtidos através da propriedade `document.applets` – um vetor que, como `document.images` e `document.forms`, contém referências para todos os applets presentes na página, na ordem em que aparecem no código. Por exemplo, em uma página com três applets, o primeiro e terceiro podem ser referenciados da forma:

```
appy1 = document.applets[0]; // primeiro applet da página atual  
appy3 = document.applets[2]; // terceiro applet da página atual
```

Os applets de uma página também são acessíveis através de um nome, especificado pelo atributo HTML opcional `NAME`. Acessar um applet pelo nome é mais prático e evita que a modificação da ordem dos applets na página afete o funcionamento da aplicação. Por exemplo, o applet:

```
<applet code="Clock2.class" name="reloj" width=170 height=150>  
</applet>
```

pode ser referenciado em qualquer atributo de eventos ou bloco <SCRIPT> da página da forma:

```
appy = document.reloj; // Applet!
```

O número de applets em uma página pode ser descoberto através da propriedade `applets.length`, de `document`:

```
numApplets = document.applets.length;
```

As propriedades dos objetos *Applet* são todas as variáveis públicas (definidas no código Java) do applet. As propriedades do HTML (`code`, `name`, `height`, `width`) podem ser lidas somente no browser Microsoft Internet Explorer. No Netscape, elas não existem.

Os métodos dos objetos *Applet* consistem de todos os métodos públicos (definidos no código Java) do applet. Não há eventos JavaScript associados ao objeto *Applet*.

Controle de Applets via JavaScript

O controle de applets a partir do código JavaScript é bastante simples, pois em muitas aplicações não exige conhecimentos de Java nem dos detalhes internos do applet. Conhecendo-se os métodos e variáveis públicas de um applet, pode-se acessá-los diretamente pelo nome através do objeto, que é referência ao applet em JavaScript².

No ambiente de desenvolvimento Java (JDK – Java Development Kit), há uma ferramenta chamada **javap** que imprime uma lista das *assinaturas* de todos os métodos e variáveis públicas de um programa Java compilado (arquivo com extensão **.class**). A assinatura consiste do nome do método, seu tipo de retorno e os tipos de seus argumentos. Por exemplo, suponha que você possua um arquivo **Carta.class**, que é um applet Java e está incluído em uma página HTML através do bloco:

```
<applet code="Carta.class" height=100 width=200 name="mens">
</applet>
```

Você não conhece o formato e nome dos métodos de **Carta.class** mas possui o JDK, que tem a ferramenta **javap**. Rodando a ferramenta **javap** sobre o arquivo **Carta.class**, obtém-se o seguinte:

```
c:\> javap Carta          (é preciso omitir a extensão .class)
public class Carta extends java.applet.Applet {
    public int numero;
    public void mudarMensagem(String);
    public String lerMensagem();
}
c:\>_
```

A primeira linha, identifica a *classe.java* (**Carta**), que é um applet. Todo programa em Java é considerado uma *classe*. A segunda linha contém a declaração de uma variável chamada **numero**. A palavra **public** indica que se trata de uma variável pública (pode ser usada em JavaScript) e a palavra **int** indica que é um número inteiro. Se formos atribuir um valor à variável **numero**, através de JavaScript, precisaremos ter o cuidado de passar um número inteiro e não um *String* ou outro tipo de dados. Java, diferente de JavaScript, só permite que uma variável receba um valor, se for de um tipo previamente declarado para a variável.

As duas últimas linhas contém as assinaturas dos métodos **mudarMensagem()** e **lerMensagem()**. A palavra **public** indica que ambos são públicos e portanto podem ser usados em JavaScript. O método **mudarMensagem()** é declarado **void**, o que significa que ele não retorna valor. Ele recebe como argumento uma variável que deve necessariamente

² Variáveis e métodos em Java que são declarados ‘static’ não são acessíveis através da referência da applet mas através do tipo *Applet*, da forma *Applet.variavel* ou *Applet.metodo()*.

ser um objeto do tipo *String*. O método `lerMensagem()` não tem argumentos, mas retorna um valor do tipo *String*.

Com estas informações, temos condições de manipular as variáveis e métodos do applet definido pela classe `Carta.class`, através de propriedades e métodos de um objeto *Applet*:

```
appy = document.applets[0];      // se for o primeiro applet
appy.numero = 6;
document.write("A mensagem atual é " + appy.lerMensagem());
appy.mudarMensagem("Esta é a nova mensagem!");
document.write("A mensagem agora é " + appy.lerMensagem());
document.write("O número é " + appy.numero);
```

No exercício resolvido a seguir, mostraremos um exemplo prático do controle de applets através de JavaScript.

Exercício Resolvido

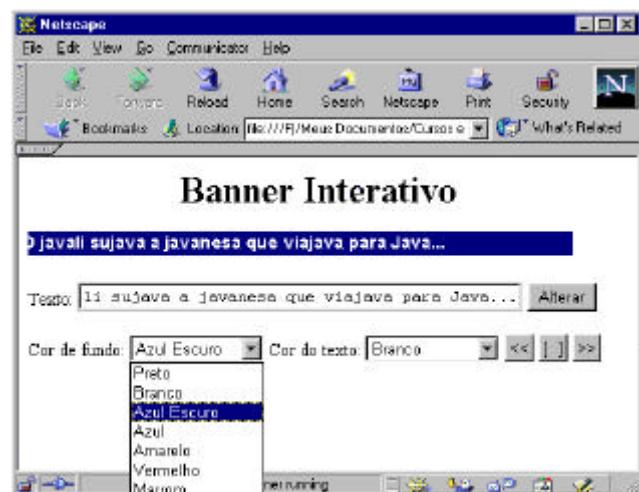
Para este exercício, utilize o arquivo `Banner.class`, que é um applet que faz uma mensagem de texto rolar horizontalmente na tela. A mensagem do `Banner` pode ser definida na página HTML dentro de um atributo `<PARAM>` com o nome (atributo `NAME`) “`MSG`”. Se o `<PARAM>` não estiver presente, o applet ainda funcionará, porém, apresentará uma mensagem *default*.

`Banner` possui vários métodos públicos que permitem mudar a mensagem atual, mudar as cores de fundo e do texto, parar a rolagem e aumentar a velocidade de rolagem para a direita ou para a esquerda. Os métodos públicos de `Banner` são os seguintes:

```
public void corDeFundo(int r, int g, int b) {
public void corDoTexto(int r, int g, int b) {
public void mensagem(String msg) {
public void esquerda() {
public void direita() {
public void para() {
```

Use a página esqueleto `Banner.html` (figura ao lado) para:

- acrescentar o applet no lugar indicado com o texto inicial “Bom Dia!”.
- programar em JavaScript o campo de texto para que ele mude a mensagem do applet.
- programar os campos `<SELECT>` para



que mudem as cores de fundo e do texto.

- d) programar os botões para que invoquem métodos que façam o texto parar (botão “[”), andar mais rápido para a esquerda (botão “<<”), e para a direita (botão “>>”).

Solução

A primeira tarefa é colocar o applet na página. Definimos o parâmetro **MSG** com o valor “Bom Dia!”, como foi pedido no requisito (a):

```
<!–Coloque o applet aqui -->
<body>
<h1>Applets controlados por JavaScript</h1>
<applet code="Banner.class" height=20 width=450 hspace=10>
    <param name="msg" value="Bom Dia!">
</applet>
<form>
    (...)
```

Com o bloco de código acima, o applet já deve aparecer na página e começar a rolar para a esquerda. Para permitir a mudança do texto durante a execução do applet, chamamos o método **mensagem()**, que muda o texto para o string recebido. O string é obtido do campo de textos **novotexto**:

```
<p>Texto:
<input type=text name=novotexto size=45>
<input type=button value="Alterar"
       onclick="document.applets[0].mensagem(this.form.novotexto.value)">
```

A mudança das cores exige mais trabalho já que os métodos **corDeFundo()** e **corDoTexto()** recebem três parâmetros inteiros, e a lista **<SELECT>** fornece apenas um valor *String*, com os valores RGB separados por vírgulas:

```
<option value="255,255,0">Amarelo</option>
```

Criamos então, uma função **cor()**, que converte o valor da opção selecionada em três números inteiros. A função, que recebe um objeto *Select* como argumento, também identifica qual das duas listas foi selecionada, para invocar o método correto:

```
<head>
<script>
function cor(selObj) {
    corStr = selObj.options[selObj.selectedIndex].value;
    rgb = corStr.split(",");
    r = parseInt(rgb[0]);
    g = parseInt(rgb[1]);
    b = parseInt(rgb[2]);
    if (selObj.name == "bg") {
        document.applets[0].corDeFundo(r, g, b);
```

```
        } else if (selObj.name == "fg") {
            document.applets[0].corDoTexto(r, g, b);
        }
    }
</script>
</head>
```

A única alteração necessária nos blocos <SELECT> a programação do atributo ONCHANGE, para chamar a função, passando o próprio *Select* como argumento:

```
<p>Cor de fundo:
<select name=bg onchange="cor(this)">
    <option value="0,0,0">Preto</option>
    <option value="255,255,255">Branco</option>
    <option value="0,0,128">Azul Escuro</option>
    <option value="0,0,255" selected>Azul</option>
    <option value="255,255,0">Amarelo</option>
    <option value="255,0,0">Vermelho</option>
    <option value="128,0,0">Marrom</option>
    <option value="0,128,0">Verde Escuro</option>
</select>
(...)
```

A programação dos botões é simples. Eles simplesmente chamam o método correspondente à sua função:

```
<input type=button value="&lt;&lt;" 
       onclick="document.applets[0].esquerda()">
<input type=button value="[" 
       onclick="document.applets[0].para()">
<input type=button value="&gt;&gt;" 
       onclick="document.applets[0].direita()">
</form>
</body>
```

A listagem completa desta solução está no arquivo **Bannersol.html**.

Exercícios

- 12.1 Se o applet **Desenho.class** (disponível no diretório **cap12/**) for instalado em uma página HTML, a página passará a ter uma área onde o usuário poderá fazer desenhos em preto-e-branco. Há, porém, dois métodos públicos no applet:
mudaCor() e **clear()** que permitem respectivamente mudar a cor e limpar a tela. Inclua o applet **Desenho** na página **Desenho.html** (figura ao lado) e programe: a) o



botão, para que limpe a tela de desenho ao ser apertado, e b) a lista de opções, para que mude a cor do lápis de acordo com a seleção do usuário.

Controle de JavaScript através de Applets

Para fazer applets controlarem páginas HTML e código JavaScript, é preciso programar os applets em Java. Esta seção, portanto, assume que o leitor conhece e está familiarizado com a linguagem Java. Para poder testar os exemplos apresentados e resolver os exercícios, é necessário ter um ambiente de desenvolvimento Java instalado, como o JDK da Sun.

Um applet Java só poderá ter acesso a uma página HTML se o autor da página permitir. A permissão é dada colocando o atributo **MAYSCRIPT** em cada applet que tem permissão para acessar a página. **MAYSCRIPT** só é necessário para que applets acessem scripts e não o contrário, como nos exemplos que vimos até aqui.

O suporte à JavaScript em aplicações escritas em Java é obtido através do pacote **netscape.javascript**, fornecido pela Netscape. Este pacote é suportado não só nos browsers da Netscape mas também nos browsers Microsoft Internet Explorer (a partir da versão 4).

O pacote contém a classe **JSObject** e a exceção **JSEException**. **JSObject** é usado para representar qualquer objeto JavaScript em Java. Toda aplicação Java que pretenda se comunicar com uma página HTML via JavaScript precisa usá-lo. A melhor forma de ter acesso aos recursos disponíveis em todo o pacote é importá-lo no código da aplicação:

```
import netscape.javascript.*;
```

A maior parte dos métodos públicos usados na comunicação Java com JavaScript estão na classe **JSObject**, listados nas tabelas abaixo (*Atenção*: estes métodos são métodos Java. Não os confunda com métodos JavaScript). **JSObject** contém um único método estático, que retorna uma referência à janela do browser:

Assinatura do método de classe (public static)	Descrição
JSObject getWindow(Applet applet)	Obtém um JSObject representando a janela do browser onde está o applet passado como argumento.

Para manipular com os objetos do browser, é preciso obter primeiro uma referência para a janela do browser. Para isto, utilizamos o método **getWindow()** passando o applet atual como argumento, no código Java, da forma:

```
JSObject janela = JSObject.getWindow(this); // código Java!
```

Depois de obtida uma referência a um objeto JavaScript, podemos obter referências a todas as suas propriedades e métodos usando os *métodos de instância* da classe **JSObject**, listados na tabela abaixo. Esses métodos precisam ser invocados sobre objetos **JSObject**. Todos os métodos que retornam valor, retornam **Object** (exceto **toString()**). Se houver

necessidade de usar tipos primitivos em parâmetros eles precisam ser encapsulados em objetos como `Integer`, `Boolean`, etc.:

Assinatura do método de instância (public)	Descrição
<code>Object getMember(String nome)</code>	Recupera uma propriedade de um objeto JavaScript pelo <i>nome</i> .
<code>Object getSlot(int indice)</code>	Recupera uma propriedade de um objeto JavaScript pelo <i>índice</i> .
<code>Object eval(String expressao)</code>	Executa expressões JavaScript.
<code>Object call(String nomeMetodo, Object[] args)</code>	Chama um método ou função JavaScript. Os argumentos da função devem ser passados no vetor <i>args</i> .
<code>void setMember(String nome, Object valor)</code>	Define um novo <i>valor</i> para uma propriedade de um objeto JavaScript pelo <i>nome</i> .
<code>public void setSlot(int indice, Object valor)</code>	Define um novo <i>valor</i> para uma propriedade de um objeto JavaScript pelo <i>índice</i> .
<code>void removeMember(String nome)</code>	Remove uma propriedade de um objeto JavaScript pelo <i>nome</i> .
<code>String toString()</code>	Devolve uma String com uma descrição do objeto.

Qualquer elemento HTML referenciado na hierarquia de objetos JavaScript pode ser obtido a partir da referência à janela do browser. O método `getMember()`, invocado em qualquer `JSObject`, retorna uma referência Java para a o nome da propriedade passada como parâmetro ou `null` se a propriedade não existir. Se a propriedade for um vetor em JavaScript, cada elemento pode ser recuperado com o método `getSlot()`, passando o índice correspondente como argumento. Como exemplo do uso desses métodos, considere o seguinte trecho HTML:

```
<body>
  <form>
    <input type=text name=dados>
  </form>
</body>
```

As seguintes operações, dentro do código do applet (Java), permitem que ele tenha acesso ao conteúdo de um campo de textos na página HTML:

```
JSObject janela = JSObject.getWindow(this);           // código Java!
JSObject docmt = (JSObject)janela.getMember("document");
JSObject frmArray = (JSObject)docmt.getMember("forms");
```

```

JSObject form1 = (JSObject) frmArray.getSlot(0);
JSObject campoTxt = (JSObject)form1.getMember("dados");
String valor = (String)campoTxt.getMember("value");

```

Expressões em JavaScript podem ser executadas a partir de applets Java usando o método `eval()`. É semelhante à função `eval()` do JavaScript só que neste caso, é um método Java, que opera sobre um `JSObject`. A obtenção da janela do browser é o suficiente para usar `eval()`. Com `eval()`, applets poderão ter diálogos modais: applets não possuem janelas de diálogo pré-definidas como as janelas de alerta e confirmação do JavaScript. Essas janelas, disponíveis em JavaScript, podem ser usadas em Java com `eval()`:

```
JSObject.getWindow().eval("alert(\"Saudações Javanesas!\")");
```

O método `eval()` também pode ser usado para obter referências a propriedades de `Window` e objetos de sua hierarquia de forma mais direta que usando sucessivas chamadas a `getMember()` e `getSlot()`. O código abaixo tem o mesmo efeito que o listado anteriormente para acessar o valor de um campo de texto:

```

JSObject janela = JSObject.getWindow(this);
String valor = (String)janela.eval("document.form1.campo.value");

```

Outra forma de chamar métodos ou funções JavaScript a partir de Java é usando o método `call()`. Este método recebe dois argumentos: o primeiro é o nome da função ou método, o segundo, os argumentos que esta função ou método recebem, dentro de um vetor. O vetor pode ser de qualquer descendente de `java.lang.Object`. Para passar argumentos de tipos primitivos, é necessário empacotá-los em objetos como `Integer`, `Boolean`, etc.

O exemplo abaixo é semelhante ao mostrado anteriormente com `eval()`. Desta vez usamos `call()`:

```

JSObject win = JSObject.getWindow();
String[] args = {"Saudações Javanesas!"};
win.call("alert", args);

```

Neste outro trecho de código, chamamos uma função `soma()`, disponível na página JavaScript, que recebe dois inteiros (os números 7 e 9) e retorna a soma. Precisamos colocar valores do tipo `int` dentro de objetos `Integer`:

```

JSObject win = JSObject.getWindow();
Object[] args = {new Integer(7), new Integer(9)};
Integer intObj = (Integer)win.call("soma", args);
int resultado = intObj.intValue();

```

Os métodos `setMember()` e `setSlot()` são usados para definir novos valores ou novas propriedades em JavaScript a partir de Java. Por exemplo, para definir a propriedade `value` de um campo de textos, pode-se usar:

```
JSObject win = JSObject.getWindow();
JSObject campoTxt = (JSObject)win.eval("document.form1.dados");
campoTxt.setMember("value", "Contatos Imediatos de JavaScript!");
```

Na seção seguinte, resolveremos um exercício que utiliza comunicação Java-JavaScript nos dois sentidos.

Exercício Resolvido

O objetivo deste exercício é estender a aplicação proposta no exercício 12.1 para que um evento ocorrido no applet Java provoque uma alteração na página HTML. Faz parte deste exercício a alteração de um programa em Java, portanto é necessário que esteja disponível um ambiente de desenvolvimento Java como o JDK da Sun. A figura abaixo mostra a aplicação rodando no Microsoft Internet Explorer.

Utilize os seguintes arquivos, localizados no diretório cap12/:

- Desenha2.java
- Desenha2.html

As etapas do exercício são as seguintes:

a) Primeiro altere o arquivo

Desenha2.html, criando uma função que receba dois argumentos do tipo *String* e preencha os campos de texto coordx e coordy com os valores recebidos.

b) Depois altere o programa

Desenha2.java para que na ocorrência do evento de movimento do mouse (método `mouseMoved()`) as coordenadas x e y do ponteiro do mouse sejam recuperadas e que a função JavaScript definida no item anterior seja chamada com os valores.

c) Compile o programa, teste e carregue no browser.



Solução

A primeira parte é simples. Consiste apenas em definir a função que irá alterar os valores dos campos de texto. Esta função será chamada a partir do applet Java.

```
function setCoords(x, y) {
    document.forms[0].coordx.value = x;
    document.forms[0].coordy.value = y;
}
```

e irá alterar os valores dos campos de texto **coordx** e **coordy**, definidos no formulário:

```
(...)
x <input type=text name=coordx size=4><br>
y <input type=text name=coordy size=4>
</form>
(...)
```

Agora precisamos alterar o programa em Java. Uma listagem parcial do programa está mostrada abaixo. Os trechos que foram adicionados ao programa original estão em negrito:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import netscape.javascript.*; // suporte a JavaScript pelo Applet

public class Desenha2 extends Applet
    implements MouseMotionListener,
               MouseListener {
    private Dimension dim;
    private int x, y, oldx, oldy;
    private boolean clearAll = false;
    private Color cor = Color.black;
    private JSObject win;           // janela do browser
    private String[] args;         // argumentos da função setCoords()

    public void init() {
        dim = getSize();
        this.addMouseMotionListener(this);
        this.addMouseListener(this);
        win = JSObject.getWindow(this);
        args = new String[2];        // são 2 os argumentos da função
    }

    public void mouseDragged(MouseEvent e) {
        if ((x == 0) && (y == 0)) {
            x = e.getX();
            y = e.getY();
        }
        oldx = x;
        oldy = y;
        x = e.getX();
        y = e.getY();
        args[0] = "" + x;
        args[1] = "" + y;
        win.call("setCoords", args);
        repaint();
    }

    public void mouseMoved(MouseEvent e) {
```

```
    args[0] = "" + e.getX();           // String com o valor de x
    args[1] = "" + e.getY();           // String com o valor de y
    win.call("setCoords", args);      // Chama função setCoords(x, y)
}

// (... restante do código inalterado: métodos não
// mostrados aqui não foram modificados ... )
}
```

Para compilar o programa³, precisamos definir o caminho onde o compilador poderá procurar pelo pacote `netscape.javascript`. Isto é feito através da variável de ambiente `CLASSPATH` e só é necessário para a compilação, portanto, podemos defini-la na linha de comando.

A localização do pacote `netscape.javascript` depende do browser instalado e da plataforma. Se o seu browser é Microsoft Internet Explorer em Windows95/98, ele está em `C:\Windows\Java\Classes\classes.zip` (se o seu Windows estiver instalado no drive `C:\`), portanto defina o `CLASSPATH` da seguinte forma:

```
set CLASSPATH=%CLASSPATH%;C:\Windows\Java\Classes\classes.zip
```

Se seu browser é Netscape 4, nas plataformas Windows95/98 o pacote está em no caminho `{diretório_de_instalação}\Program\Java\Classes\java40.jar`, e nos browsers Netscape 3, no caminho `{diretório_de_instalação}\Program\java\classes\java_30`, que devem ser usados para definir o `CLASSPATH` em cada caso.

Tendo-se definido o `CLASSPATH`, pode-se compilar, usando o compilador Java do JDK da Sun (ou outro compatível):

```
javac Desenha2.java
```

Corrija quaisquer erros e depois teste o applet, carregando-o no browser. Para ver possíveis mensagens de erro, abra o “Java Console” do seu browser. Após a correção de erros, e recompilação, pode ser necessário fechar todas as janelas do browser e abri-lo novamente para que a versão atualizada do applet seja carregada corretamente.

Conversão de tipos

Java é uma linguagem rigorosa em relação a tipos de dados. Na comunicação entre Java e JavaScript ocorrem diversas conversões de tipos. Quando valores são passados de Java para JavaScript, eles obedecem às conversões mostradas na tabela abaixo:

Tipo de dados Java	Tipo de dados (objeto) JavaScript
<code>byte, char, short, int, long, float, double</code>	<code>Number</code>
<code>boolean</code>	<code>Boolean</code>
<code>java.lang.String</code>	<code>String</code>

³ O procedimento descrito é referente ao JDK da Sun, que roda em linha de comando.

<code>netscape.javascript.JSObject</code>	<code>Object</code>
<code>java.lang.Object</code>	<code>JavaObject</code>
<code>java.lang.Class</code>	<code>JavaClass</code>
<code>tipo[]</code>	<code>JavaArray</code>

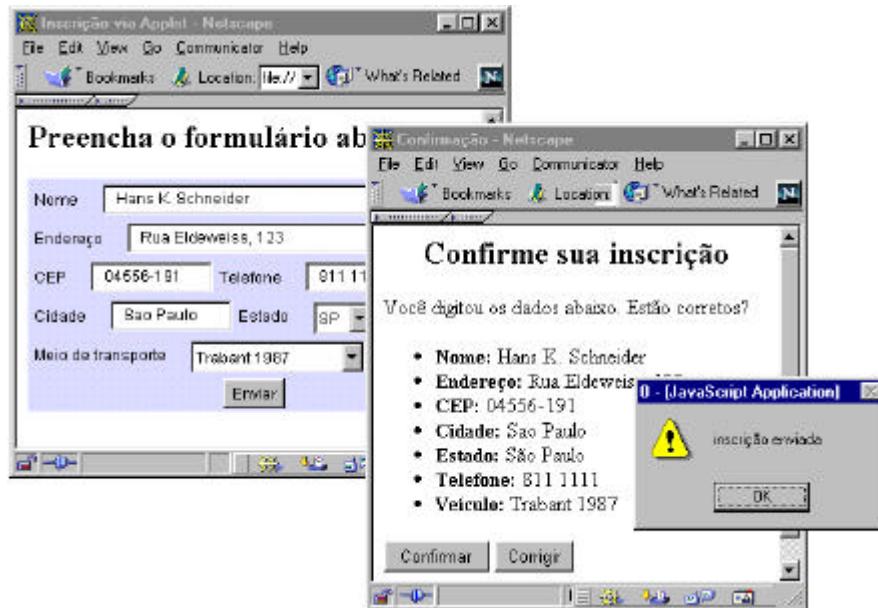
Os tipos de objeto `JavaObject` e `JavaArray` são representações JavaScript de objetos genéricos e vetores de qualquer tipo em Java. O tipo `JavaClass` refere-se à classes Java. É utilizado, por exemplo, para obter acesso a métodos estáticos.

Quando valores são passados de JavaScript para Java, a conversão obedece à tabela abaixo:

Tipo de dados (objeto) JavaScript	Tipo de dados Java
<code>String</code>	<code>java.lang.String</code>
<code>Number</code>	<code>java.lang.Float</code>
<code>Boolean</code>	<code>java.lang.Boolean</code>
<code>JavaObject</code>	<code>java.lang.Object</code>
<code>Object</code>	<code>netscape.javascript.JSObject</code>

Exercícios

- 12.2 Este exercício parece o exercício resolvido do capítulo 8. A aplicação, do ponto de vista do usuário, é idêntica. A diferença é que o formulário que deve ser preenchido desta vez é um applet! Ainda assim uma nova página HTML deve ser gerada em uma nova janela (como no capítulo 8). O que é necessário fazer aqui é realizar a comunicação Java-JavaScript para que o applet consiga criar a nova página. Use os arquivos `Formulario.html`, que contém a página HTML, e `Formulario.java`, código-fonte Java incompleto (falta implementar um método apenas).



- 12.3 Realize a validação dos campos do formulário do exercício anterior. Informe que os dados estão incorretos usando uma janela de alerta JavaScript (`alert()`).

A

Bibliografia

Referências

Documentos consultados na elaboração deste livro.

- [1] Tim Berners-Lee. *Information Management: A Proposal*. CERN – European Laboratory for Particle Physics, March 1989, May 1990. URL:
<http://www.w3.org/History/1989/proposal.html>.
- [2] CERN – European Laboratory for Particle Physics. *An Overview of the World Wide Web - History and Growth*. 1997. URL:
<http://www.cern.ch/Public/ACHIEVEMENTS/WEB/history.html>.
- [3] Netscape Corporation. *JavaScript (1.1) Guide for Netscape 3.0*. 1996. URL:
<http://home.netscape.com/eng.mozilla/3.0/handbook/javascript/>.
- [4] Microsoft Corporation. *Jscript 4.0/5.0beta Reference and Tutorial*. 1998. URL:
<http://www.microsoft.com/scripting/>
- [5] ECMA General Assembly. *ECMA-262: ECMAScript Language Specification (ISO/IEC 16262)*. 1998. URL: <http://www.ecma.ch/stand/ecma-262.htm>.
- [6] World Wide Web Consortium – W3C. *Document Object Model Specification*. 1998. URL:
<http://www.w3.org/DOM/>.
- [7] Microsoft Corporation. *DHTML Document Object Model*. 1998. URL:
<http://www.microsoft.com/workshop/author/dhtml/reference/objects.asp>.
- [8] Netscape Corporation. *JavaScript Guide (1.3)*. 1998. URL:
<http://developer.netscape.com/library/documentation/communicator/jsguide4/>.

- [9] Netscape Corporation. *JavaScript Reference (1.3)*. 1998. URL:
<http://developer.netscape.com/docs/manuals/communicator/jsref/>.
- [10] William R. Stanek. *Netscape ONE (Open Network Environment) Developer's Guide*. SamsNet/Macmillian Computer Publishing 1997.
- [11] Stephen Spainhour & Valerie Quercia. *WebMaster in a NutShell – A Desktop Quick Reference*. O'Reilly and Associates, 1996.
- [12] Netscape Corporation. *JavaScript 1.0 Authoring Guide*. 1995. URL:
<http://home.netscape.com/eng/mozilla/2.0/handbook/javascript/>.
- [13] Netscape Corporation. *JavaScript Sample Code*. 1998. URL:
<http://developer.netscape.com/library/examples/javascript.html>.
- [14] Matthew J. Rechs, Angelo Sirigos, Nik Williams. *DevEdge Newsgroup FAQ: JavaScript*. Netscape Corporation. 1998. URL:
<http://developer.netscape.com/support/faqs/champions/javascript.html>.
- [15] World Wide Web Consortium – W3C. *HTML 3.2/4.0 Specifications*. 1998. URL:
<http://www.w3.org/pub/WWW/MarkUp/Wilbur/>.
- [16] Danny Goodman. *JavaScript Object Roadmap and Compatibility Guide*. 1997. URL:
<http://www.dannyg.com/update.html>.
- [17] Lisa Rein and Jennifer Spelman. *ECMAScript in a nutshell: Our guide to the new specification*. NetscapeWorld Magazine, July 1997. URL:
<http://www.netscapeworld.com/nw-07-1997/nw-07-javascript.html>.
- [18] Danny Goodman, *Cookie Recipes – Client Side Persistent Data*.
http://developer.netscape.com/viewsource/goodman_cookies.html.
- [19] Gordon McComb. *Beginner's JavaScript*.
JavaWorld Magazine. March 1996. URL:
<http://www.javaworld.com/javaworld/jw-03-1996/jw-03-javascript.intro.html>.
- [20] Gordon McComb. *New JavaScript Features in Navigator 3.0*.
JavaWorld Magazine, October 1996. URL:
<http://www.javaworld.com/javaworld/jw-10-1996/jw-10-javascript.html>.
- [21] Gordon McComb. *Frequently Sought Solutions in JavaScript*.
JavaWorld Magazine, November 1996. URL:
<http://www.javaworld.com/javaworld/jw-11-1996/jw-11-javascript.html>.
- [22] Reaz Hoque. *Getting Started with JavaScript's 13 Event Handlers*. NetscapeWorld Magazine. December 1996. URL:

<http://www.netscapeworld.com/netscapeworld/nw-12-1996/nw-12-javascript1.html>.

[23]William R. Stanek. *FrontPage97 Unleashed*. SamsNet / Macmillian Computer Publishing 1997.

[24]JavaSoft / Sun Microsystems. *The Java 2 Platform Documentation*. 1998. URL:
<http://java.sun.com/>.

Websites

Alguns websites dedicados a JavaScript, HTML e Java.

Netscape Developer's Corner: Site da Netscape dedicado à suas tecnologias Web: JavaScript, LiveWire, LiveConnect, etc. Contém detalhada documentação, artigos, exemplos de código e software. URL: <http://developer.netscape.com/>.

Microsoft Scripting Technologies: Site Microsoft dedicado a tecnologias de Scripting como JScript, VBScript e DHTML. Contém, tutoriais, referências, FAQs e exemplos de código. URL: <http://msdn.microsoft.com/scripting/default.htm>.

NetscapeWorld e *Netscape Enterprise Developer*: Revistas mensais dedicadas a tecnologias Netscape como JavaScript, Server-Side JavaScript, LiveWire, etc. Foram descontinuadas em Maio/98. URL do índice de artigos publicados entre Julho/96 e Maio/98.

<http://www.netscapeworld.com/ned-ti/ned-ti-index.html> URL.

JavaWorld: Revista mensal dedicada à Java, com artigos, exemplos de código, análise de livros, etc. URL: <http://www.javaworld.com/>.

WebDeveloper.Com: Revista interativa dedicada à tecnologias Web. Possui uma seção sobre JavaScript, com artigos, tutoriais e exemplos de código. URL:
<http://www.webdeveloper.com/>.

Web Developer's Virtual Library: Coleção de tutoriais sobre tecnologias Web, incluindo JavaScript, Java e DHTML. URL: <http://www.wdvl.com/>.

WebReference.Com: Revista e referência sobre tecnologias Web. Tutoriais detalhados, repositório de aplicações, etc. URL da seção dedicada a JavaScript:
<http://www.webreference.com/javascript/>.

DevHead – Web Development for the Next Millennium: Revista interativa da ZDNet com seções detalhadas sobre diversas tecnologias Web. Tutoriais, artigos, análise de produtos e livros. URL: <http://www.zdnet.com/devhead/>. Seção dedicada a JavaScript:
<http://www.zdnet.com/devhead/filters/javascript/>.

The JavaScript Source. Repositório da Internet.Com com vários exemplos de código JavaScript gratuitos. URL: <http://javascript.internet.com/>.

EarthWeb Gamelan: Grande repositório dedicado a Java. Muitos applets, aplicações, beans, componentes e documentação. URL: <http://www.gamelan.com/>.

EarthWeb JavaScripts.Com: Um grande repositório de programas JavaScript com mais de 2000 aplicações gratuitas (Dez/1998). URL: <http://www.javascripts.com/>.

ECMA: Organização européia que desenvolve as especificações abertas do JavaScript/JScript (ECMA-262 – ECMAScript). URL: <http://www.ecma.ch/>.

W3C – World Wide Web Consortium: Site do consórcio que define os padrões da World Wide Web. Contém especificações e software experimental. URL: <http://www.w3.org/>.

Opera. Um browser alternativo que suporta HTML4, Java (através de plug-in), JavaScript (parcialmente - baseado em JavaScript 1.1) e CSS (folhas de estilo). URL: <http://www.operasoftware.com/>.