

Aprendendo

JavaScript

Shelley Powers

Novatec

CAPÍTULO 1

Olá, JavaScript!

Um motivo pelo qual JavaScript é tão popular é ser relativamente fácil de ser acrescentado a uma página web. Tudo que você precisa fazer é incluir pelo menos um elemento HTML `script` na página, especificar `"text/javascript"` para o atributo `type` e adicionar qualquer JavaScript que quiser:

```
<script type="text/javascript">  
... algum código-fonte JavaScript  
</script>
```

Não é necessária instalação e você também não precisa ter que lidar com configurações estranhas de caminhos (paths) de alguma biblioteca. JavaScript funciona imediatamente e na maioria dos navegadores web, incluindo os quatro mais importantes: Firefox, Internet Explorer, Opera e Safari. Tudo o que você tem que fazer é adicionar um bloco de scripting e já está em ação.

Tradicionalmente, você adiciona blocos escritos em JavaScript ao elemento `head` (cabeçalho) do documento (delimitado por tags `head` de abertura e fechamento do cabeçalho), mas você também pode incluí-los no elemento `body` – ou até em ambas as seções. Todavia, adicionar um `script` ao corpo geralmente não é considerada uma boa técnica, já que dificulta encontrar esse `script` quando você for modificá-lo no futuro. A exceção a essa regra é quando o desempenho for uma preocupação, o que será visto no capítulo 6. Todos os exemplos neste livro adicionam blocos de scripting apenas à seção `head` (cabeçalho) da página web.

Hello World!

Também é comum que o primeiro exemplo ao se aprender uma nova linguagem de programação seja conhecido como “Hello World” – uma aplicação muito simples que imprima essa mensagem na interface do usuário, seja qual esta for. No caso de JavaScript, a interface do usuário é a página web. O exemplo 1.1 mostra uma página web com um bloco em JavaScript que, usando apenas uma linha escrita nessa linguagem, abre uma pequena janela normalmente chamada de caixa de alerta com as palavras “Hello World!”

⇒ Exemplo 1.1 – A menor aplicação em JavaScript: “Hello World!”

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>Hello, World!</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">

alert("Hello, World!");

</script>
</head>
<body>
</body>
</html>

```

Copiar o exemplo 1.1 para um arquivo e abri-lo em navegadores web que suportem JavaScript deve resultar em uma caixa de alerta onde se lê “Hello World!”. Caso isso não aconteça, você deve se assegurar de que JavaScript esteja habilitado.



Versões mais antigas do Internet Explorer também desabilitam scripts se você abrir a página pelo menu File Open em vez de usar um endereço de página web como `http://<algum_domínio.com>/index.html`.

Essa aplicação, embora muito limitada em funcionalidade, demonstra um pouco os componentes mínimos de uma aplicação JavaScript: você tem uma página web, um elemento script e uma linha em JavaScript. Experimente você mesmo, exceto editar a string substituindo a palavra “World” por seu nome.

É claro que, se você quiser fazer mais do que simplesmente imprimir uma mensagem no navegador, precisará estender de alguma forma esse exemplo.

Hello World! novamente

Outra variação da aplicação “Hello World!” escreve a mensagem realmente na página web, em vez de em uma caixa de alerta. Para fazer isso, usa quatro componentes importantes de JavaScript: o objeto interno `document` do navegador, variáveis JavaScript, uma função JavaScript e um manipulador de eventos. Por incrível que pareça, você ainda pode codificar a aplicação em sete linhas de JavaScript, conforme mostra o exemplo 1.2.

⇒ Exemplo 1.2 – “Hello World!” exibido na página web

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

```

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Hello, World!</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
function hello() {
    // diz alô para o mundo
    var msg = "Hello, World!";
    document.open();
    document.write(msg);
    document.close();
}
</script>
</head>
<body onload="hello()">
<p>Hi</p>
</body>
</html>
```

Embora o exemplo 1.2 seja uma aplicação muito pequena, expõe diversos componentes básicos da maioria das aplicações em JavaScript em uso atualmente, cada um dos quais merecendo um exame mais detalhado. No resto deste capítulo, faremos esse exame mais de perto, um componente de cada vez.



Ainda não coberto neste capítulo é a Document Type Declaration (DOC-TYPE) usada nos exemplos 1.1 e 1.2, que pode influenciar como diferentes navegadores processam JavaScript. Abordarei o impacto de um DOCTYPE no capítulo 6.

A tag script

JavaScript é frequentemente usado dentro do contexto de outra linguagem, como as linguagens de marcação HTML e XHTML. Entretanto, você não pode simplesmente colocar JavaScript na marcação onde e como quiser.

No exemplo 1.2, o elemento `script` contém o código JavaScript. Isso informa o navegador o que, quando encontrar a tag de abertura do elemento de `script`, não deve processar seu conteúdo como HTML ou XHTML. Nesse momento, o controle sobre o conteúdo é passado para o mecanismo de scripting do navegador.

Nem todos os scripts existentes dentro de páginas web são JavaScript, e a tag de abertura do elemento `script` contém um atributo definindo o tipo do script. Nesse exemplo, este é apresentado como `text/javascript`. Entre outros valores permitidos para o atributo `type` estão:

- text/ecmascript
- text/jscript
- text/vbscript
- text/vbs

O primeiro valor `type` listado especifica que o script é interpretado como ECMAScript, baseado no padrão de scripting ECMA-262. O valor a seguir faz com que o script seja interpretado como JScript, uma variação de ECMAScript que a Microsoft implementa no Internet Explorer. Os dois últimos valores são para o VBScript da Microsoft, uma linguagem de scripting completamente diferente.

Todos esses valores de `type` descrevem o tipo MIME do conteúdo. MIME, ou Multipurpose Internet Mail Extension, é uma forma de identificar como o conteúdo está codificado (p. ex., `text`) e seu formato específico (`javascript`). Fornecendo um tipo MIME, os navegadores capazes de processar o tipo o fazem, enquanto outros navegadores pulam a seção. Isso garante que apenas aplicações que possam processar o script realmente o acessem.

Versões anteriores da tag `script` recebiam um atributo `language`, que era usado para designar a versão da linguagem, assim como o tipo: `javascript1.2` em comparação a `javascript 1.1`. Entretanto, o uso de `language` ficou obsoleto em HTML 4.01, embora ainda apareça em muitos exemplos de JavaScript. Aí está uma das primeiras técnicas multinavegadores.



Uso o termo multinavegadores para denotar JavaScript que funciona em todos os navegadores-alvo, ou que usa funcionalidades para gerenciar quaisquer diferenças de modo que a aplicação funcione em “múltiplos navegadores”.

Anos atrás, ao trabalhar com problemas relacionados a compatibilidade de múltiplos navegadores, não era incomum que se criasse um script específico para cada navegador em uma seção ou arquivo separado e depois se usasse o atributo `language` para assegurar que apenas um navegador compatível pudesse acessar o código. Examinando alguns de meus exemplos antigos (por volta de 1997), encontrei o seguinte:

```
<script src="ns4_obj.js" language="javascript1.2">
</script>
<script src="ie4_obj.js" language="jscript">
</script>
```

A filosofia dessa abordagem era que apenas um navegador capaz de processar JavaScript 1.2 poderia pegar o primeiro arquivo (principalmente o Netscape Navigator 4.x na época) e apenas um navegador capaz de processar JScript poderia pegar o

segundo arquivo (Internet Explorer 4). Pouco prático? Claro, mas também funcionou nos primeiros anos de tentativas de se lidar com efeitos de páginas dinâmicas frequentemente falhados em múltiplos navegadores.

Outros atributos de scripts válidos são `src`, `defer` e `charset`. O atributo `charset` define a codificação de caracteres usada com o script. Geralmente não é estabelecido a menos que você precise de uma codificação diferente de caracteres da que estiver definida para o documento.

Um atributo que pode ser bastante útil é `defer`. Se você configurar `defer` com um valor de "defer", isso indica ao navegador que o script não irá gerar qualquer conteúdo de documento e que o navegador pode continuar processando o resto do conteúdo da página, retornando para o script quando a página tiver sido processada e exibida:

```
<script type="text/javascript" defer="defer">  
... nenhum conteúdo sendo gerado  
</script>
```

O atributo `defer` pode ajudar a acelerar o carregamento da página quando você tiver um bloco maior de JavaScript ou incluir uma biblioteca maior de JavaScript.

O último atributo, `src`, tem a ver com o carregamento de arquivos JavaScript externos, o que você verá mais adiante. Primeiro, porém, examinaremos mais de perto o atributo de tipo `text/javascript` e o que isso significa para cada navegador.

Adição de script ao corpo do documento

Mencionei anteriormente que o elemento `script` geralmente é adicionado ao elemento `head` de uma página web porque é mais fácil realizar a manutenção de páginas web quando os elementos `script` estão organizados em um só local. Todavia, há um motivo válido para incluir scripts dentro do elemento `body`: o desempenho.

Quando scripts são adicionados ao elemento `head`, o resto do documento não precisa ser trazido até que o script tenha terminado de ser carregado porque os navegadores carregam muitos recursos em paralelo do mesmo domínio. Além disso, o navegador pode ter que esperar para exibir o resto da página porque pode haver um elemento `document.write` dentro do script. Se os arquivos JavaScript forem grandes, as imagens da página web e outras informações importantes podem demorar, talvez além do viável.

Mesmo o uso do atributo `defer` no elemento `script` não terá impacto nos problemas de carregamento paralelo ou renderização da página.

No livro *High Performance Web Sites* (O'Reilly), Steve Souders recomenda a colocação dos elementos `script` na parte de baixo de um documento, para permitir que o resto da página web seja carregada primeiro, antes do script. Os desenvolvedores de aplicações web mais complexas priorizam essa abordagem. A desvantagem de se colocar o script no final da página é que o script fica mais difícil de ser encontrado, e a manutenção das páginas, mais difícil.

Qual é a melhor abordagem? Descobri que a maioria dos websites não incorpora bibliotecas JavaScript que sejam tão grandes que a colocação de scripts se torne um problema, não em comparação à importância de ser capaz de garantir que as páginas sejam mais fáceis de manter. Ainda assim, se você desenvolver bibliotecas JavaScript mais complexas, talvez deva considerar uma mudança para scripts baseados em rodapés.

Independentemente da abordagem que você usar, seja consistente: coloque seus scripts sempre no elemento `head` ou sempre na parte de baixo do elemento `body`.

JavaScript versus ECMAScript versus JScript

O exemplo 1.2 usava o tipo `text/javascript` com o elemento `script`, e a aplicação funciona no Firefox, IE, Opera e Safari. Entretanto, nem todos os navegadores implementam JavaScript.

Embora o nome “JavaScript” tenha se tornado onipresente para scripting baseados em navegadores no lado cliente, apenas Mozilla e seu popular navegador, o Firefox, implementam JavaScript, que é o nome real de uma instância de uma especificação de scripting mais ampla, a ECMAScript. ECMAScript é, na verdade, uma especificação de scripting no lado cliente que abrange todo o mercado. A versão mais recente de ECMAScript é ECMA-262, Edição 3.

Contudo, a maioria dos navegadores reverencia o tipo `text/javascript`, além do mais apropriado (embora muito menos comum) `text/ecmascript`, embora possam existir diferenças, até mesmo significativas, em exatamente o que cada navegador ou outra aplicação suporta.



ECMAScript não se restringe apenas a navegadores: o suporte ActionScript da Adobe em Flash é baseado em ECMA-262, Edição 3.

Todos os navegadores usados para testar as aplicações deste livro – Firefox 3.x, Safari 3.x, Opera 9.x e IE8 – suportam a maioria, senão toda, ECMA-262, Edição 3, e até uma parte da próxima geração de ECMAScript, ECMAScript 3.1 (e posteriores). Neste livro, farei observações sempre que houver diferenças nos navegadores ou fornecerei formas para contornar problemas relacionados ao uso em mais de um navegador. Também usarei o `text/javascript` mais familiar para o atributo de tipo do elemento `script`, conforme mostrado no exemplo 1.2.

Definição de funções em JavaScript

No exemplo 1.2, a parte de JavaScript que realmente cria a mensagem “Hello, World!” existe dentro de uma função chamada `hello`. Funções são formas de modularizar uma ou mais linhas de script de forma que possam ser executadas uma ou mais vezes. Você também pode usar funções para controlar quando o script modularizado é executado. Por exemplo, no exemplo 1.2, a função é chamada apenas após a página web ser carregada.

Aqui está a sintaxe típica para a criação de uma função:

```
function nomeFunção(parâmetros) {  
    ...  
}
```

A palavra-chave `function` é seguida pelo nome da função e por parênteses contendo zero ou mais parâmetros (argumentos da função). No exemplo 1.2, não há parâmetros, mas veremos muitos exemplos com parâmetros por todo o livro. O script que constitui a função fica entre as chaves.

Eu digo “típica” quando forneço a sintaxe da função porque essa não é a única sintaxe que você pode usar para criar uma função. Entretanto, veremos outras variações começando pelo capítulo 5, que cobre funções JavaScript em detalhes.

É claro que, assim que você tiver uma função, tem que chamá-la para executar o script que ela contém, o que nos leva aos manipuladores de eventos.

Manipuladores de eventos

Na tag `body` de abertura do exemplo 1.2, um atributo HTML chamado `onload` é atribuído à função `hello`. O atributo `onload` é o que é conhecido como manipulador de eventos. Esse manipulador de eventos, e outros, é parte do modelo de objetos associado que cada navegador fornece.

Você usa manipuladores de eventos para mapear uma função para um evento específico de modo que, quando o evento ocorrer, o script da função seja processado. Um dos manipuladores de eventos mais usados é o recém-demonstrado, o evento `onload` anexado ao elemento `body`. Quando a página web tiver sido carregada, o evento é disparado, e o manipulador chama a função mapeada.

Veja alguns mapeadores de eventos normalmente usados:

`onclick`

Disparado quando o elemento recebe um clique do mouse.

`onmouseover`

Disparado quando o cursor do mouse está sobre o elemento.

`onmouseout`

Disparado quando o cursor do mouse não está mais sobre o elemento.

`onfocus`

Disparado quando o elemento recebe o foco (por meio do mouse ou do teclado).

`onblur`

Disparado quando o elemento não tem mais o foco.

Estes são apenas alguns dos manipuladores de eventos, e nem todos os elementos suportam todos os manipuladores. O manipulador de eventos `onload` é suportado por apenas alguns elementos, como os elementos `body` e `img` – o que não é surpresa, já que o evento está associado ao carregamento de algo.

Acrescentar um manipulador de eventos diretamente à tag de abertura é uma forma de anexá-lo. Uma segunda técnica ocorre diretamente dentro de JavaScript usando uma sintaxe como a seguinte:

```
<script type="text/javascript">
window.onload=hello;

function hello(??) {
    // diz alô para o mundo
    var msg = "Hello, World!";
    document.open();
    document.writeln(msg);
    document.close();
}
</script>
```

O manipulador de eventos `onload` é uma propriedade de outro objeto interno de navegador, o `window`. A primeira linha do script atribui a função `hello` diretamente ao manipulador de eventos `onload` da janela.



Funções em JavaScript também são objetos em JavaScript, de modo que você pode atribuir uma função, por nome ou diretamente, a uma variável ou outra propriedade do objeto.

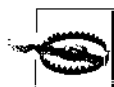
Usando a abordagem da propriedade do objeto, você não tem que adicionar manipuladores de eventos como atributos em tags de elementos, mas pode em vez disso adicioná-los ao próprio JavaScript. Entraremos em maiores detalhes sobre manipuladores de eventos e formas mais avançadas de manipulação de eventos no início do capítulo 7. Enquanto isso, examinaremos mais de perto o objeto `document`.

Objeto `document` de navegador

O exemplo 1.2, embora pequeno, usou um dos objetos mais poderosos disponíveis em seu navegador: o objeto `document`. Esse objeto é, para todas as intenções e propósitos, uma representação da página, incluindo todos os elementos dentro dela. É por meio do `document` que podemos acessar o conteúdo da página e, como você acabou de ver, é por meio dele que também podemos modificar os conteúdos da mesma.

O `document` contém coleções mapeadas a elementos da página, como todas as imagens ou elementos de formulário da página. Também contém métodos que você pode usar tanto para acessar quanto para alterar a página web, incluindo os métodos `open`, `writeIn` e `close` usados no exemplo 1.2.

O método `open` abre o documento para gravação. No exemplo 1.2, o documento aberto era o mesmo no qual o script estava. O método `writeIn` é uma variação do método `write`, que imprime uma string de texto no documento. A única diferença entre eles é que `writeIn` também insere um caractere de nova linha após o texto. O método `close` fecha o documento e também força a exibição imediata do conteúdo do mesmo.



Uma consequência negativa de se gravar sobre documentos existentes é que, com o IE, pelo menos com a versão beta do IE8, você perderá a funcionalidade de seu botão de retorno.

Os métodos `open` e `close` não são necessários para o exemplo 1.2, já que os navegadores abrirão e fecharão automaticamente o documento quando o método `writeIn` for chamado depois de ele já ter sido carregado. Se você usasse o script no corpo da página, precisaria chamar explicitamente o método `open`.

O `document`, assim como o método `window` mencionado anteriormente, faz parte de uma hierarquia de objetos conhecida como Browser Object Model (BOM). Esse modelo é um conjunto básico de objetos implementados na maioria dos navegadores modernos. Abordarei o `document` e outros objetos BOM no capítulo 9.



BOM é a versão mais antiga do Document Object Model (DOM), mais formal, e às vezes é chamado de DOM Nível 0.

Operador `property`

No exemplo 1.2, você acessou os métodos do objeto `document` por meio de um dos muitos operadores suportados em JavaScript: o operador `property`, representado por um operador de ponto simples (`.`).

Diversos operadores estão disponíveis em JavaScript: os aritméticos (`+`, `-`), os de expressões condicionais (`>`, `<`) e outros que detalharei mais adiante neste livro. Um dos

mais importantes, todavia, é o operador `property`. Elementos de dados, manipuladores de eventos e métodos de objetos são todos considerados propriedades de objetos dentro de JavaScript, e é possível acessar todos eles por meio do operador `property`.

Você também usa esse operador em um processo chamado de encadeamento de métodos, ou às vezes apenas encadeamento, por meio do qual você pode aplicar chamadas a múltiplos métodos, uma após a outra, sendo todas dentro da mesma instrução. Veremos o exemplo a seguir no livro:

```
var tstValue = document.getElementById("test").style.backgroundColor="#ffffff";
```

Neste exemplo, um elemento `page` é acessado usando o método `getElementById` do documento, e seu objeto `style` é acessado para configurar a cor de fundo desse elemento. A propriedade `backgroundColor` pertence ao objeto `style`, que é uma propriedade do objeto `document`.

Cobrirei todos esses métodos e objetos em capítulos posteriores, mas queria apresentar o encadeamento de métodos agora, já que você o verá com bastante frequência. Você não pode encadear todas as propriedades de todos os objetos – apenas aquelas que retornam um objeto.



Uma das bibliotecas Ajax mais populares, JQuery, faz uso intenso de encadeamento de métodos. Cobrirei a JQuery rapidamente no capítulo 14.

Palavra-chave `var` e o escopo

A string “Hello World!” que usei no exemplo 1.2 é atribuída a um objeto chamado `msg`, que é um exemplo de uma variável JavaScript. Uma variável nada mais é que uma referência nomeada a determinados dados. Os dados podem ser uma string, como no exemplo 1.2, um número, ou o valor booleano `true` ou `false`. Eles também podem ser uma referência a uma função, a um array ou a outro objeto.

No exemplo, defini a variável com a palavra-chave `var`. Quando você usa `var` com uma variável, está definindo essa variável com escopo local, o que significa que você pode acessá-la apenas dentro da função na qual a definiu. Se eu não usasse `var`, a variável `msg` seria global e teria escopo dentro e fora da função. Usar uma variável global em um contexto local não é uma coisa ruim – e pode ser necessário às vezes – mas não é uma boa prática e deve ser evitada, se possível.

O motivo pelo qual você deve querer evitar variáveis globais é porque, se a aplicação fizer parte de uma aplicação JavaScript maior, `msg` pode estar em uso em outra parte do código em outro arquivo e você terá sobrescrito os dados que ela continha originalmente. Ou, se você criar uma variável global chamada `msg`, o script de alguma

outra biblioteca poderia sobrescrevê-la não usando de forma correta a palavra-chave `var`, e os dados que você estava registrando serão perdidos.

Estabelecer o escopo de uma variável é importante se você tiver variáveis globais e locais com o mesmo nome. O exemplo 1.2 não tem variáveis globais com algum nome, mas é importante que se desenvolvam boas práticas de codificação JavaScript desde o início.

Aqui estão as regras quanto ao escopo:

- Se você declarar uma variável com a palavra-chave `var` em uma função ou em um bloco de código, seu uso é local àquela função.
- Se você usar uma variável sem declará-la com a palavra-chave `var`, e existir uma variável global com o mesmo nome, a variável local é assumida como sendo a global já existente.
- Se você declarar uma variável localmente com uma palavra-chave `var`, mas não a inicializar (i.e., atribuir um valor a ela), ela será local e ficará acessível, mas não definida.
- Se você declarar uma variável localmente sem a palavra-chave `var`, ou se declará-la explicitamente como global mas não a inicializar, ela será acessível globalmente, mas também não estará definida.

Usando `var` dentro de uma função, você pode evitar problemas ao usar variáveis globais e locais com o mesmo nome. Isso é particularmente importante ao se usar bibliotecas JavaScript – como Dojo, jQuery e Prototype – porque você não saberá que nomes de variáveis o outro código JavaScript está usando.

Instruções (statements)

JavaScript também suporta diferentes tipos de instruções de processamento, conhecidos como instruções. O exemplo 1.2 demonstrou um tipo básico de instrução em JavaScript: a atribuição, na qual um valor é atribuído em uma variável. Outros tipos de instrução são laços `for`, que processam um bloco de script um determinado número de iterações; a instrução condicional `if...else`, que verifica uma condição para ver se um bloco de script é executado; a instrução `switch`, que verifica um valor dentro de um determinado conjunto e a seguir executa o bloco de script associado a esse valor; e assim por diante.

Cada tipo de instrução contém determinados requisitos sintáticos. No exemplo 1.2, a instrução de atribuição terminava com um ponto-e-vírgula. Usar um ponto-e-vírgula para terminar uma instrução não é um requisito em JavaScript, a menos que você queira digitar muitas instruções na mesma linha. Se `for` fazê-lo, terá que inserir um ponto-e-vírgula para separar as instruções individuais.

Quando você digita uma instrução completa em uma linha, usa uma quebra de linha para terminá-la. Todavia, da mesma forma que com o uso de `var`, é uma boa prática usar ponto-e-vírgulas para terminar todas as instruções, senão para tornar o código mais fácil de ler. Veja mais sobre o ponto-e-vírgula, outros operadores e instruções no capítulo 3.

Comentários

Como espero que este capítulo demonstre, há muita coisa em JavaScript, mesmo em uma aplicação pequena como a do exemplo 1.2. Espere um pouco, porém, já que ainda não terminamos. Por fim, mas certamente não menos importante, vamos ver um pouco mais sobre comentários em JavaScript.

Os comentários fornecem um resumo ou explicação do código que se segue. Comentários em JavaScript são uma forma extremamente útil de fazer observações rapidamente sobre o que um bloco de código faz ou quais dependências ele tem. Eles tornam o código mais legível e mais fácil de manter.

Você pode usar dois tipos diferentes de comentários em suas aplicações. O primeiro, usando as barras duplas (`//`), comenta o que estiver a seguir na mesma linha:

```
// Esta é uma linha comentada
var i = 1; // este é um comentário de linha
```

O segundo faz uso dos delimitadores de abertura e fechamento de comentários em JavaScript, `/*` e `*/`, para marcar um bloco de comentários que pode se estender por uma ou mais linhas:

```
/* Este é um comentário multilinhas
   que se estende por três linhas. Comentários multilinhas são especialmente úteis para
   comentar uma função*/
```

Comentários de apenas uma linha são relativamente seguros de usar, mas os de várias linhas podem gerar problemas se o caractere inicial ou final for apagado acidentalmente.

Em geral, usamos comentários de linha única antes de um bloco de script executando um determinado processo ou criando um objeto específico; usamos comentários de blocos de várias linhas no início de um arquivo JavaScript. Uma boa prática a adotar com JavaScript é começar cada bloco, função ou definição de objeto em JavaScript com pelo menos uma linha de comentário. Além disso, forneça um bloco de comentários mais detalhados no início de todos os arquivos de bibliotecas em JavaScript; inclua informações sobre autor, data e dependências, assim como um objetivo detalhado do script.

Já exploramos o que você viu no exemplo 1.2. Agora examinaremos o que você não viu.

O que você não viu: seções CDATA e comentários em HTML

Há dez anos, quando a maioria dos navegadores estava em sua primeira ou segunda versão, o suporte a JavaScript era vago, com cada navegador implementando uma versão diferente. Quando os navegadores, como o Lynx baseado em texto, encontravam a tag `script`, geralmente apenas imprimiam a saída na página.

Para evitar isso, o conteúdo do `script` ficava dentro de comentários HTML: `<!-- e -->`. Quando comentários em HTML eram usados, navegadores que não suportavam JavaScript ignoravam o `script` dentro desses comentários, mas os navegadores mais novos sabiam executá-lo.

Era algo não muito sofisticado, porém, muito usado. A maioria das páginas web com JavaScript atualmente apresenta comentários HTML acrescentados porque é mais comum que o `script` seja copiado do que não. Infelizmente, alguns navegadores novos hoje podem processar a página web como XHTML, e estritamente como XML, o que significa que o código comentado é descartado. Como consequência, o uso de comentários em HTML para “esconder” o `script` é desencorajado.

Outra forma de “esconder” `script`, porém, é encorajada: por meio do uso da seção XML CDATA, especialmente se o `script` for ser usado em XHTML. O exemplo 1.3 é uma modificação do exemplo 1.2 com a adição de uma seção CDATA, mostrada em negrito.

⇒ Exemplo 1.3 – Modificação do exemplo 1.2 para acrescentar uma seção CDATA para “esconder” o `script`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Hello, World!</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//<![CDATA[
function hello() {
    // diz alô para o mundo
    var msg = "Hello, <em>World!</em>";
    document.open();
    document.write(msg);
    document.close();
}
//]]>
</script>
</head>
<body onload="hello()">
<p>Hi</p>
</body>
</html>
```

O motivo para a seção `CDATA` é que processadores XHTML interpretam marcação, como as tags de abertura e fechamento `em` nesse novo exemplo, mesmo quando eles estão dentro de strings JavaScript. Embora o script possa ser processado corretamente e também exibir a página de forma correta, se você tentar validá-lo sem a seção `CDATA` obterá erros de validação, conforme mostra a figura 1.1.

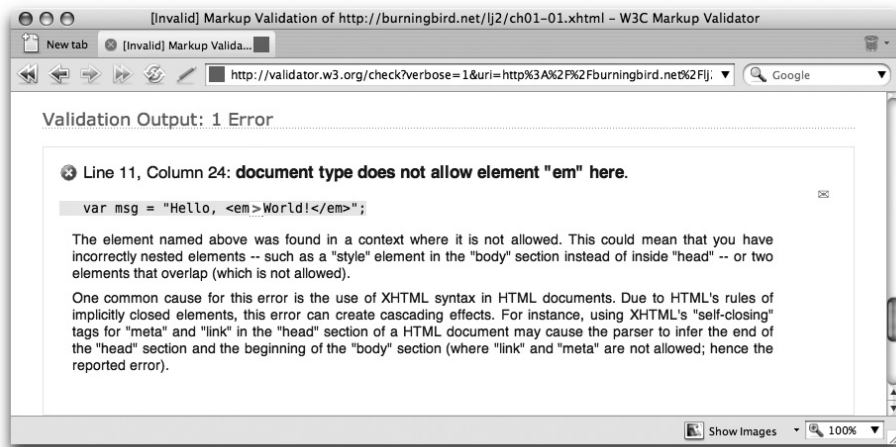


Figura 1.1 – Erro de validação sem o uso da seção `CDATA`.

Supõe-se que o JavaScript que é importado para a página usando o atributo `src` do elemento `script` seja compatível com XHTML e não requeira a seção `CDATA`. Você deve, contudo, delimitar JavaScript in-line ou embutido com `CDATA`, especialmente se estiver dentro do elemento `body`. Para a maioria dos navegadores, você também precisará esconder as tags de abertura e fechamento da seção `CDATA` com comentários (`//`), conforme mostrado anteriormente no exemplo 1.3, ou obterá um erro JavaScript.

É claro que a melhor forma de manter suas páginas web organizadas é remover inteiramente JavaScript das mesmas, por meio do uso de arquivos JavaScript.

A maioria dos exemplos deste livro é inserida nas páginas principalmente para torná-las mais fáceis de serem lidas e seguidas. Todavia, a Mozilla Foundation recomenda (e eu concordo) que todo JavaScript in-line ou embutido seja removido de uma página e colocado em arquivos JavaScript separados. Usar um arquivo separado, o que é coberto na próxima seção, evita problemas com validação e interpretação incorreta de texto, independentemente de a página ser processada como HTML ou XHTML.



Arquivos JavaScript também são mais eficientes, já que o navegador os coloca em cache na primeira vez em que são carregados. Referências adicionais ao mesmo arquivo são trazidas da cache.

Arquivos JavaScript

O uso de JavaScript está se tornando mais orientado a objetos e complexo. Para simplificar seu trabalho, assim como para compartilhá-lo, desenvolvedores JavaScript estão criando objetos JavaScript reutilizáveis que podem ser incorporados em muitas aplicações, incluindo aquelas criadas por outros desenvolvedores. A única forma eficiente de se compartilhar esses objetos é criá-los em arquivos separados e fornecer um link para cada arquivo na página web. Agora com o código em arquivos, tudo o que o desenvolvedor precisa fazer é conectar o código às páginas web. Se o código precisar ser alterado posteriormente, ele o é em apenas um lugar.

Atualmente, mesmo o JavaScript mais simples é criado em arquivos de script separados. Qualquer overhead que seja criado pelo uso de vários arquivos é compensado pelos benefícios gerados. Para incluir uma biblioteca JavaScript ou arquivo de script em sua página web, use esta sintaxe:

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

O elemento `script` não contém conteúdo, mas a tag de fechamento ainda assim é necessária.

O navegador carrega os arquivos de script para a página na ordem na qual eles aparecem na mesma e os processa em ordem a menos que `defer` seja usado. Um arquivo de script deve ser tratado como se o código estivesse realmente inserido na página: o comportamento não é diferente entre arquivos de script e blocos de JavaScript embutido.

O exemplo 1.4 é outra modificação de nossa aplicação “Hello, World!”, exceto que, dessa vez, o script é movido para um arquivo separado, chamado *helloworld.js*. A extensão *.js* é necessária, a menos que você direcione o servidor web para usar alguma outra extensão para representar o tipo MIME JavaScript. Entretanto, pelo fato de a *.js* ser usada como padrão há anos, é melhor não inventar moda.



Toda regra sempre tem exceções, e o uso de *.js* é uma delas. Se o JavaScript for gerado dinamicamente usando uma aplicação no lado servidor construída em uma linguagem como PHP, o arquivo terá uma extensão diferente.

O exemplo 1.4 contém o script, e o exemplo 1.5 mostra a página web agora alterada.

⇒ Exemplo 1.4 – O script Hello World em um arquivo separado

```
/*  
  função: hello  
  autoria: Shelley  
  hello imprime a mensagem "Hello, World!"  
*/
```

```
function hello() {
    // diz alô para o mundo
    var msg = "Hello, <em>World!</em>";
    document.open();
    document.write(msg);
    document.close();
}
```

⇒ Exemplo 1.5 – A página web, agora chamando um arquivo de script externo

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Hello, World!</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript" src="helloworld.js">
</script>
</head>
<body onload="hello()">
<p>Hi</p>
</body>
</html>
```

Como você pode ver, a página fica muito mais limpa, e a aplicação é mais eficiente a partir de uma perspectiva de manutenção. Além disso, outras aplicações agora podem reutilizar o código. Embora seja improvável que você reutilizasse algo tão simples quanto o script “Hello, World!”, criará exemplos mais adiante neste livro nos quais a reutilização se torna mais importante.

Temos uma última seção de material a cobrir neste capítulo antes de passar para variáveis e tipos de dados no capítulo 2.

Acessibilidade e melhores práticas em JavaScript

Em um mundo ideal, todos aqueles que visitam seu website usariam o mesmo tipo de sistema operacional e navegador e teriam JavaScript habilitado. Seu site nunca seria acessado por meio de um telefone celular ou algum outro dispositivo diferente, pessoas com deficiências visuais não precisariam de leitores de telas e as com problemas de paralisia não precisariam de navegação ativada por voz.

Este não é um mundo ideal, mas muitos desenvolvedores JavaScript codificam como se fosse. Ficamos tão envolvidos com as maravilhas que podem criar que esquecemos que nem todo mundo pode compartilhá-las.

Muitas das melhores práticas estão associadas a JavaScript mas, se houver uma para se usar deste livro, é a seguinte: seja qual for a funcionalidade JavaScript que você crie, ela não deve atrapalhar o uso de seu site pelos visitantes.

O que quero dizer por “atrapalhar o uso do seu site pelos visitantes”? Quero dizer que você deve evitar o uso de JavaScript de uma forma em que aqueles que não podem, ou não irão, habilitar JavaScript, fiquem impedidos de acessar recursos essenciais do site. Se você criar um menu drop-down usando JavaScript, também precisa fornecer uma alternativa que não use script. Se seus visitantes tiverem deficiências visuais, JavaScript não deve interferir em navegadores de áudio, o que acontece quando instruções são adicionadas a uma página dinamicamente.

Muitos desenvolvedores não seguem esses princípios porque supõem que as práticas requerem trabalho adicional e, em sua maior parte, requerem. Entretanto, o trabalho não precisa ser um fardo – não quando os resultados podem aumentar a acessibilidade de seu site. Além disso, muitas empresas agora exigem que seus websites satisfaçam a um determinado nível de acessibilidade. É melhor desenvolver o hábito de criar páginas acessíveis no início do que tentar consertá-las, ou seus hábitos, posteriormente.

Diretrizes de acessibilidade

O site WebAIM (<http://www.webaim.org>) contém um ótimo tutorial sobre a criação de JavaScript acessível (disponível em <http://www.webaim.org/techniques/javascript/>). Ele cobre as maneiras como você não deve usar JavaScript, como o uso de JavaScript para menus e outros tipos de navegação. Todavia, o site fornece formas para você usar JavaScript para tornar um site mais acessível.

Uma sugestão é basear a resposta em eventos que possam ser disparados quer você use um mouse ou não. Por exemplo, em vez de capturar cliques do mouse, você deve capturar eventos que sejam disparados se você usar um teclado ou um mouse, como `onfocus` e `onblur`. Se você tiver um menu drop-down, adicione um link para uma página separada e forneça um menu estático nessa segunda página.

Depois de revisar o tutorial na WebAIM, você talvez queira dedicar algum tempo à Web Accessibility Initiative do World Wide Consortium (W3C) (em <http://www.w3.org/WAI/>). A partir daí, você também pode acessar o website da Seção 508 do governo norte-americano (<http://www.section508.gov/>), que discute o que é conhecido como “compatibilidade 508”. Sites que sejam compatíveis com a Seção 508 são acessíveis independentemente de restrições físicas. Nesse website, você pode acessar diversas ferramentas que avaliam seu site quanto à acessibilidade, como Cynthia Says (em <http://www.cynthiasays.com/>).

Esteja seu site localizado dentro dos Estados Unidos ou não, você desejará que ele seja acessível. Assim, uma visita à Seção 508 é útil independentemente de sua localização. É claro que nem todos os problemas de acessibilidade estão relacionados àqueles navegadores nos quais JavaScript está limitado ou desabilitado como padrão, como com leitores de tela. Muitas pessoas não confiam em JavaScript, ou não ligam para a linguagem e preferem desabilitá-la. Para ambos os grupos de pessoas – aquelas que preferem não usar JavaScript e as que não têm escolha – é importante que se forneçam alternativas quando nenhum script estiver presente. Uma alternativa é `noscript`.

noscript

Alguns navegadores ou outras aplicações não estão preparados para processar JavaScript ou estão limitados no modo de interpretação do script. Se JavaScript não for essencial para a navegação ou interação, e o navegador ignorar o script, não há prejuízo. Entretanto, se JavaScript for essencial para o acesso de recursos do site e você não fornecer alternativas, está basicamente mandando essas pessoas embora.

Anos atrás, quando JavaScript era razoavelmente uma novidade, uma abordagem popular era fornecer uma página simples ou apenas de texto acessível por meio de um link, geralmente colocado no topo da página. Contudo, a quantidade de trabalho para realizar a manutenção dos dois sites poderia ser proibitiva e os desenvolvedores tinham que se preocupar constantemente em manter os sites sincronizados.

Uma técnica melhor é fornecer alternativas estáticas ao conteúdo dinâmico gerado por script. Quando você usa JavaScript para criar um menu drop-down, também fornece um menu com links hierárquicos padrão; quando você usa script para mostrar elementos de formulário para edição baseados em interação com o usuário, fornece os links mais tradicionais para uma segunda página fazer o mesmo.

O elemento que permite tudo isso é `noscript`. Onde quer que você precise de conteúdo estático, adicione um elegante `noscript` com o conteúdo dentro das tags de abertura e fechamento. Assim, se um navegador ou outra aplicação não puder processar o script (devido a JavaScript não estar habilitado por algum motivo), o conteúdo `noscript` é processado; caso contrário, ele é ignorado.

O exemplo 1.6 é uma última variação de “Hello, World!” mostrando o exemplo protegido por `CDATA` modificado com a adição de `noscript`. Acessar a página com um navegador habilitado com JavaScript deve exibir uma página com “Hello, World!” impresso. Entretanto, se você acessar essa página com um navegador com scripts desabilitados, uma mensagem diferente será exibida.

⇒ Exemplo 1.6 – O uso de noscript para navegadores sem JavaScript habilitado

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<title>Hello, World!</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<script type="text/javascript">
//
function hello() {
    // diz alô para o mundo
    var msg = "Hello, &lt;em&gt;World!&lt;/em&gt;";
    document.open();
    document.write(msg);
    document.close();
}
//]]&gt;
&lt;/script&gt;
&lt;/head&gt;
&lt;body onload="hello()"&gt;

&lt;noscript&gt;
&lt;p&gt;I'm still here, World!&lt;/p&gt;
&lt;/noscript&gt;
&lt;/body&gt;
&lt;/html&gt;</pre></div><div data-bbox="112 581 884 645" data-label="Text"><p>É claro que o exemplo 1.6 é apenas um uso simplificado de noscript; você verá usos mais sofisticados mais adiante no livro, assim como métodos alternativos com proteção para scripts.</p></div><div data-bbox="112 654 885 799" data-label="Text"><p>Para testar o exemplo 1.6, usei uma extensão Firefox chamada de Web Developer Toolbar. Nessa barra fica uma opção para desabilitar o suporte a JavaScript. Quando JavaScript estiver ativo, a mensagem original “Hello, World!” é exibida. Contudo, quando uso a ferramenta para desativar o suporte a JavaScript, outra mensagem exibe: I'm still here, world! Embora você possa desativar o scripting diretamente no navegador, descobri que ferramentas de desenvolvimento como a Web Developer Toolbar tornam o teste muito mais fácil.</p></div><div data-bbox="112 810 885 914" data-label="Text"><p>Quais ferramentas você usa depende do navegador com o qual você prefere desenvolver. Prefiro desenvolver com o Firefox e fazer uso extensivo da Web Developer Toolbar e do Firebug, uma ferramenta sofisticada de depuração. Mais adiante, no capítulo 6, que cobre a detecção e solução de problemas e a depuração, examinaremos mais de perto esses tópicos, assim como outras ferramentas e opções disponíveis para outros navegadores.</p></div>
```