Capítulo 15

Escrevendo script de documentos

JavaScript do lado do cliente existe para transformar documentos HTML estáticos em aplicativos Web interativos. Fazer scripts do conteúdo de páginas Web é o principal objetivo de JavaScript. Este capítulo – um dos mais importantes do livro – explica como fazer isso.

Os capítulos 13 e 14 explicaram que cada janela, guia e quadro do navegador Web é representado por um objeto Window. Todo objeto Window tem uma propriedade `document` que se refere a um objeto Document. O objeto Document representa o conteúdo da janela e esse é o tema deste capítulo. Contudo, o objeto Document não opera independentemente. Ele é o principal objeto de uma API maior, conhecida como *Document Object Model* (ou DOM), para representar e manipular conteúdo de documento.

Este capítulo começa explicando a arquitetura básica do DOM. Em seguida, passa a explicar:

- Como consultar ou *selecionar* elementos individuais de um documento.
- Como *percorrer* um documento como uma árvore de nós e como localizar os ascendentes, irmãos e descendentes de qualquer elemento do documento.
- Como consultar e configurar os atributos dos elementos do documento.
- Como consultar, configurar e modificar o conteúdo de um documento.
- Como modificar a estrutura de um documento, criando, inserindo e excluindo nós.
- Como trabalhar com formulários HTML.

A última seção do capítulo aborda diversos recursos de documento, incluindo a propriedade `referrer`, o método `write()` e técnicas para consultar o texto do documento selecionado.

15.1 Visão geral do DOM

Document Object Model, ou DOM, é a API fundamental para representar e manipular o conteúdo de documentos HTML e XML. A API não é especialmente complicada, mas existem vários detalhes de arquitetura que precisam ser entendidos.

Primeiramente, você deve entender que os elementos aninhados de um documento HTML ou XML são representados na DOM como uma árvore de objetos. A representação em árvore de um documento HTML contém nós representando marcações ou elementos HTML, como `<body>` e `<p>`, e nós representando strings de texto. Um documento HTML também pode conter nós representando comentários HTML. Considere o seguinte documento HTML simples:

```
<html>
  <head>
    <title>Sample Document</title>
  </head>
  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
</html>
```

A representação DOM desse documento é a árvore ilustrada na Figura 15-1.

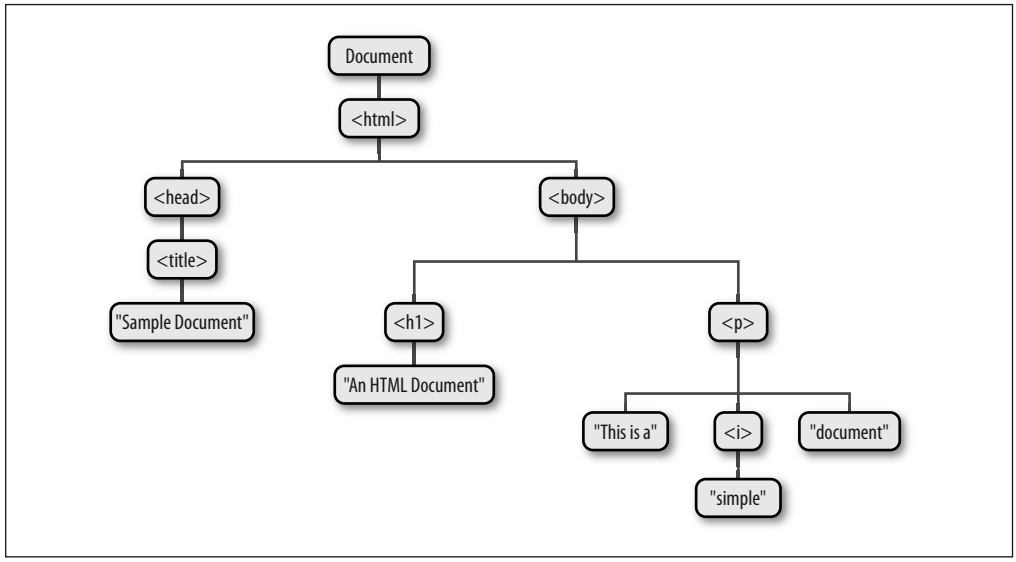

Figura 15-1 A representação em árvore de um documento HTML.

Se você ainda não conhece as estruturas em árvore da programação de computador, é útil saber que elas emprestam a terminologia das árvores genealógicas. O nó imediatamente acima de outro é o *pai* desse nó. Os nós um nível imediatamente abaixo de outro nó são os *filhos* desse nó. Os nós no mesmo nível e com o mesmo pai são *irmãos*. O conjunto de nós a qualquer número de níveis abaixo de outro nó são os *descendentes* desse nó. E o pai, avô e todos os outros nós acima de um nó são os *ascendentes* desse nó.

Cada caixa na Figura 15-1 é um nó do documento e é representado por um objeto Node. Vamos falar sobre as propriedades e métodos de Node em algumas das seções a seguir e você pode pesquisar

essas propriedades e métodos sob `Node` na Parte IV. Note que a figura contém três tipos diferentes de nós. Na raiz da árvore está o nó Document, que representa o documento inteiro. Os nós que representam elementos HTML são nós Element e os nós que representam texto são nós Text. Document, Element e Text são subclasses de Node (e têm suas próprias entradas na seção de referência). Document e Element são as duas classes DOM mais importantes e grande parte deste capítulo é dedicada às suas propriedades e métodos.

Node e seus subtipos formam a hierarquia de tipos ilustrada na Figura 15-2. Observe que há uma distinção formal entre os tipos genéricos Document e Element, e os tipos HTMLDocument e HTMLElement. O tipo Document representa um documento HTML ou XML e a classe Element representa um elemento desse documento. As subclasses HTMLDocument e HTMLElement são específicas de documentos e elementos HTML. Neste livro, usamos frequentemente os nomes de classe genéricos Document e Element, mesmo ao nos referirmos a documentos HTML. Isso também vale para a seção de referência: as propriedades e os métodos dos tipos HTMLDocument e HTMLElement estão documentados nas páginas de referência de Document e Element.

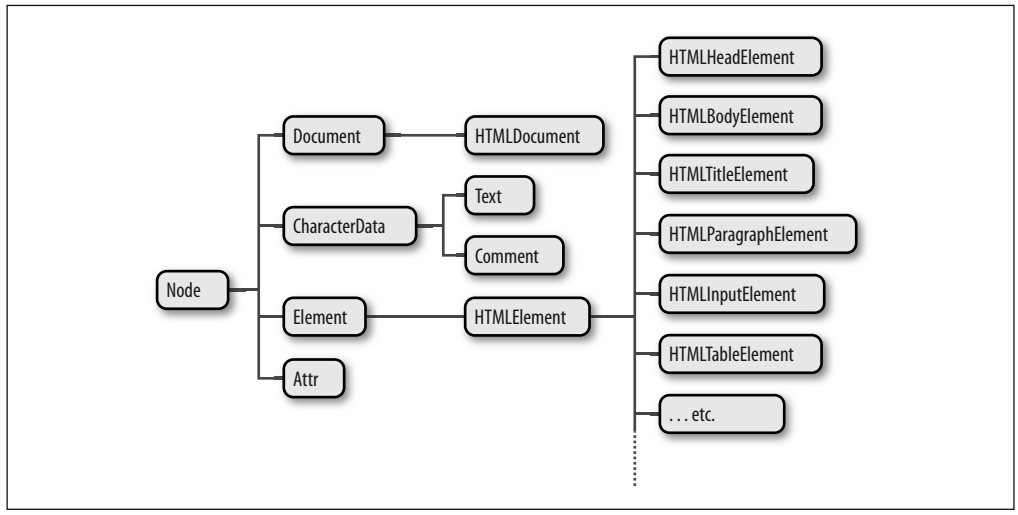

Figura 15-2 Uma hierarquia de classe parcial de nós de documento.

Também é interessante notar na Figura 15-2 que existem muitos subtipos de HTMLElement que representam tipos específicos de elementos HTML. Cada um define propriedades de JavaScript para espelhar os atributos HTML de um elemento específico ou de um grupo de elementos (consulte a Seção 15.4.1). Algumas dessas classes específicas do elemento definem mais propriedades ou métodos que vão além do simples espelhamento da sintaxe HTML. Essas classes e seus recursos adicionais são abordados na seção de referência.

Por fim, note que a Figura 15-2 mostra alguns tipos de nó que não foram mencionados até agora. Os nós Comment representam comentários HTML ou XML. Como os comentários são basicamente

strings de texto, esses nós são muito parecidos com os nós Text que representam o texto exibido de um documento. CharacterData, o ascendente comum de Text e de Comment, define métodos compartilhados pelos dois nós. O tipo de nó Attr representa um atributo XML ou HTML, mas quase nunca é usado, pois a classe Element define métodos para tratar atributos como pares nome/valor, em vez de nós de documento. A classe DocumentFragment (não mostrada) é um tipo de Node que nunca existe em um documento real: ela representa uma sequência de Nodes que não têm um pai comum. DocumentFragments são úteis para algumas manipulações de documento e são abordados na Seção 15.6.4. O DOM também define tipos pouco utilizados, para representar coisas como declarações doctype e instruções de processamento XML.

15.2 Selecionando elementos do documento

A maioria dos programas JavaScript do lado do cliente funciona manipulando de alguma forma um ou mais elementos de documento. Quando esses programas começam, podem utilizar a variável global `document` para se referirem ao objeto Document. Contudo, para manipular elementos do documento, eles precisam de algum modo obter ou *selecionar* os objetos Element que se referem a esses elementos de documento. O DOM define várias maneiras de selecionar elementos. Você pode consultar um documento quanto a um elemento (ou elementos):

- com um atributo `id` especificado;
- com um atributo `name` especificado;
- com o nome de marcação especificado;
- com a classe ou classes CSS especificadas; ou
- correspondente ao seletor CSS especificado

As subseções a seguir explicam cada uma dessas técnicas de seleção de elemento.

15.2.1 Selecionando elementos pela identificação

Qualquer elemento HTML pode ter um atributo `id`. O valor desse atributo deve ser único dentro do documento – dois elementos no mesmo documento não podem ter a mesma identificação. Você pode selecionar um elemento com base nessa identificação exclusiva com o método `getElementById()` do objeto Document. Já usamos esse método no Capítulo 13 e no Capítulo 14:

```
var section1 = document.getElementById("section1");
```

Essa é a maneira mais simples e normalmente usada de selecionar elementos. Se seu script vai manipular determinado conjunto de elementos de documento, dê a esses elementos atributos `id` e pesquise os objetos Element usando essa identificação. Se precisar pesquisar mais de um elemento pela identificação, talvez ache útil a função `getElements()` do Exemplo 15-1.

Exemplo 15-1 Pesquisando vários elementos pela identificação

```
/**
 * Esta função espera qualquer número de argumentos string. Ela trata cada
 * argumento como uma identificação de elemento e chama document.getElementById() para
 * cada um.
```

```
 * Retorna um objeto que mapeia identificações no objeto Element correspondente.
 * Lança um objeto Error se qualquer uma das identificações for indefinida.
 */
function getElements(/*ids...*/) {
    var elements = {};                                  // Começa com um mapa vazio
    for(var i = 0; i < arguments.length; i++) {         // Para cada argumento
        var id = arguments[i];                          // O argumento é uma
                                                        // identificação de elemento
        var elt = document.getElementById(id);          // Pesquisa Element
        if (elt == null)                                // Se não estiver definido,
            throw new Error("No element with id: " + id);  // lança um erro
        elements[id] = elt;                             // Mapeia a identificação no
                                                        // elemento
    }
    return elements;                                    // Retorna a identificação
                                                        // para o mapa de elementos
}
```

Nas versões do Internet Explorer anteriores ao IE8, `getElementById()` faz uma correspondência que não diferencia letras maiúsculas e minúsculas nas identificações de elemento e também retorna elementos que tenham um atributo `name` coincidente.

15.2.2 Selecionando elementos pelo nome

O atributo HTML `name` se destinava originalmente a atribuir nomes a elementos de formulário e o valor desse atributo é usado quando dados de formulário são enviados para um servidor. Assim como o atributo `id`, `name` atribui um nome a um elemento. Ao contrário de `id`, contudo, o valor de um atributo `name` não precisa ser único: vários elementos podem ter o mesmo nome e isso é comum no caso de botões de seleção e caixas de seleção em formulários. Além disso, ao contrário de `id`, o atributo `name` é válido somente em alguns elementos HTML, incluindo formulários, elementos de formulário, `<iframe>` e elementos `<img>`.

Para selecionar elementos HTML com base no valor de seus atributos `name`, você pode usar o método `getElementsByName()` do objeto Document:

```
var radiobuttons = document.getElementsByName("favorite_color");
```

`getElementsByName()` é definido pela classe HTMLDocument (e não pela classe Document) e, assim, só está disponível para documentos HTML e não para documentos XML. Ele retorna um objeto NodeList que se comporta como um array somente para leitura de objetos Element. No IE, `getElementsByName()` também retorna elementos que têm um atributo `id` com o valor especificado. Por compatibilidade, você deve tomar o cuidado de não usar a mesma string como nome e como identificação.

Vimos, na Seção 14.7, que configurar o atributo `name` de certos elementos HTML criava propriedades com esses nomes automaticamente no objeto Window. Algo semelhante acontece para o objeto Document. Configurar o atributo `name` de um elemento `<form>`, `<img>`, `<iframe>`, `<applet>`, `<embed>` ou `<object>` (mas somente elementos `<object>` que não contenham objetos de fallback) cria uma propriedade do objeto Document cujo nome é o valor do atributo (supondo, é claro, que o documento ainda não tenha uma propriedade com esse nome).

Se existe apenas um elemento com determinado nome, o valor da propriedade do documento criada automaticamente é o próprio elemento. Se existe mais de um elemento, então o valor da propriedade é um objeto NodeList que atua como um array de elementos. Conforme vimos na Seção 14.7, as

propriedades de documento criadas para elementos <iframe> nomeados são especiais: em vez de se referirem ao objeto Element, se referem ao objeto Window do quadro.

Isso significa que alguns elementos podem ser selecionados pelo nome simplesmente usando-se o nome como uma propriedade de Document:

```
// Obtém o objeto Element para o elemento <form name="shipping_address">
var form = document.shipping_address;
```

Os motivos dados na Seção 14.7 para não se usar as propriedades de janela criadas automaticamente se aplicam igualmente a essas propriedades de documento criadas automaticamente. Se você precisa pesquisar elementos nomeados, é melhor pesquisá-los explicitamente com uma chamada para `getElementsByName()`.

15.2.3 Selecionando elementos pelo tipo

Todos os elementos HTML ou XML de um tipo (ou nome de marcação) especificado podem ser selecionados usando-se o método `getElementsByTagName()` do objeto Document. Para obter um objeto semelhante a um array somente para leitura, contendo os objetos Element de todos os elementos em um documento, por exemplo, você poderia escrever:

```
var spans = document.getElementsByTagName("span");
```

Assim como `getElementsByName()`, `getElementsByTagName()` retorna um objeto NodeList. (Consulte o quadro desta seção para obter mais informações sobre a classe NodeList.) Os elementos do objeto NodeList retornado estão na ordem do documento; portanto, o primeiro elemento <p> de um documento pode ser selecionado como segue:

```
var firstpara = document.getElementsByTagName("p")[0];
```

As marcações HTML não diferenciam letras maiúsculas e minúsculas, e quando `getElementsByTagName()` é usado em um documento HTML, faz uma comparação de nomes de marcações que não diferencia letras maiúsculas e minúsculas. A variável `spans` anterior, por exemplo, vai incluir todos os elementos que foram escritos como <SPAN>.

Um objeto NodeList representando todos os elementos de um documento pode ser obtido passando-se o argumento curinga "*" para `getElementsByTagName()`.

A classe Element também define um método `getElementsByTagName()`. Ele funciona da mesma maneira que a versão de Document, mas seleciona apenas os elementos descendentes do elemento no qual é chamado. Assim, para encontrar todos os elementos dentro do primeiro elemento <p> de um documento, você poderia escrever:

```
var firstpara = document.getElementsByTagName("p")[0];
var firstParaSpans = firstpara.getElementsByTagName("span");
```

Por motivos históricos, a classe HTMLDocument define propriedades de atalho para acessar certos tipos de nós. As propriedades `images`, `forms` e `links`, por exemplo, se referem aos objetos que se comportam como arrays somente para leitura de elementos <img>, <form> e <a> (mas somente marcações <a> que tenham um atributo `href`). Essas propriedades se referem a objetos HTMLCollection, os quais são muito mais parecidos com objetos NodeList, mas podem também ser indexados pela

identificação ou pelo nome do elemento. Anteriormente, vimos como se pode referir a um elemento `<form>` nomeado, com uma expressão como a seguinte:

```
document.shipping_address
```

Com a propriedade `document.forms`, também é possível se referir mais especificamente ao formulário nomeado (ou identificado) como segue:

```
document.forms.shipping_address;
```

O objeto HTMLDocument também define propriedades sinônimas `embeds` e `plugins` que são HTMLCollections de elementos `<embed>`. A propriedade `anchors` não é padronizada, mas se refere a elementos `<a>` que têm um atributo `name`, em vez de um atributo `href`. A propriedade `<scripts>` é padronizada pela HTML5 para ser uma HTMLCollection de elementos `<script>`, mas quando este livro estava sendo produzido ainda não estava implementada universalmente.

HTMLDocument define ainda duas propriedades que se referem a elementos únicos especiais, em vez de a coleções de elementos. `document.body` é o elemento `<body>` de um documento HTML e `document.head` é o elemento `<head>`. Essas propriedades são sempre definidas: se a origem do documento não inclui elementos `<head>` e `<body>` explicitamente, o navegador os cria implicitamente. A propriedade `documentElement` da classe Document se refere ao elemento-raiz do documento. Em documentos HTML isso é sempre um elemento `<html>`.

NodeLists e HTMLCollections

`getElementsByName()` e `getElementsByTagName()` retornam objetos NodeList, e propriedades como `document.images` e `document.forms` são objetos HTMLCollection.

Esses objetos são objetos semelhantes a um array somente para leitura (consulte a Seção 7.11). Eles têm propriedades `length` e podem ser indexados (para leitura, mas não para gravação) como arrays verdadeiros. Você pode iterar através do conteúdo de um NodeList ou HTMLCollection com um laço padrão, como segue:

```
for(var i = 0; i < document.images.length; i++)  // Itera por todas as imagens
document.images[i].style.display = "none";       // ...e as oculta.
```

Você não pode chamar métodos Array em NodeLists e HTMLCollections diretamente, mas pode fazer isso indiretamente:

```
var content = Array.prototype.map.call(document.getElementsByTagName("p"),
                                       function(e) { return e.innerHTML; });
```

Os objetos HTMLCollection podem ter propriedades nomeadas adicionais e podem ser indexados com strings e com números.

Por motivos históricos, tanto objetos NodeList como HTMLCollection também podem ser tratados como funções: chamá-los com um argumento numérico ou de string é o mesmo que indexá-los com um número ou com uma string. O uso dessa peculiaridade é desestimulado.

As interfaces NodeList e HTMLCollection foram projetadas tendo em mente linguagens menos dinâmicas do que JavaScript em mente. Ambas definem um método `item()`. Ele espera um inteiro e retorna o

elemento que está nesse índice. Em JavaScript nunca há necessidade de chamar esse método, pois pode-se simplesmente utilizar indexação de array em seu lugar. Da mesma forma, HTMLCollection define um método `namedItem()` que retorna o valor de uma propriedade nomeada, mas os programas JavaScript podem usar indexação de array ou acesso à propriedade normal em seu lugar.

Uma das características mais importantes e surpreendentes de NodeList e HTMLCollection é não serem instantâneos estáticos de um estado histórico do documento, mas geralmente são *dinâmicos* e a lista de elementos que contêm pode variar à medida que o documento muda. Suponha que você chame `getElementsByTagName('div')` em um documento sem nenhum elemento `<div>`. O valor de retorno é um NodeList com `length` igual a 0. Se, então, você insere um novo elemento `<div>` no documento, esse elemento se torna automaticamente um membro do NodeList e a propriedade `length` muda para 1.

Normalmente, o caráter dinâmico de NodeLists e HTMLCollections é muito útil. Contudo, se você vai adicionar ou remover elementos do documento enquanto itera por um NodeList, talvez queira primeiro fazer uma cópia estática do NodeList:

```
var snapshot = Array.prototype.slice.call(nodelist, 0);
```

15.2.4 Selecionando elementos por classe CSS

O atributo `class` de uma HTML é uma lista separada de zero ou mais identificadores por espaços. Ele descreve uma maneira de definir conjuntos de elementos relacionados do documento: todos os elementos que têm o mesmo identificador em seu atributo `class` fazem parte do mesmo conjunto. `class` é uma palavra reservada de JavaScript, de modo que JavaScript do lado do cliente utiliza a propriedade `className` para conter o valor do atributo HTML `class`. O atributo `class` normalmente é usado em conjunto com uma folha de estilos CSS para aplicar os mesmos estilos de apresentação em todos os membros de um conjunto. Vamos vê-lo outra vez, no Capítulo 16. Além disso, contudo, a HTML5 define um método `getElementsByClassName()` que nos permite selecionar conjuntos de elementos de documento com base nos identificadores que estão em seu atributo `class`.

Assim como `getElementsByTagName()`, `getElementsByClassName()` pode ser chamado em documentos HTML e em elementos HTML, retornando um NodeList dinâmico, contendo todos os descendentes coincidentes do documento ou elemento. `getElementsByClassName()` recebe um único argumento de string, mas a string pode especificar vários identificadores separados por espaços. Somente os elementos que incluem todos os identificadores especificados em seus atributos `class` são coincidentes. A ordem dos identificadores não importa. Note que tanto o atributo `class` como os métodos `getElementsByClassName()` separam identificadores de classe com espaços e não com vírgulas. Aqui estão alguns exemplos de `getElementsByClassName()`:

```
// Localiza todo os elementos que têm "warning" em seus atributos class
var warnings = document.getElementsByClassName("warning");
// Localiza todos os descendentes do elemento chamado "log" que têm a classe
// "error" e a classe "fatal"
var log = document.getElementById("log");
var fatal = log.getElementsByClassName("fatal error");
```

Os navegadores Web atuais exibem documentos HTML no "modo Quirks" ou no "modo Standards", dependendo do quanto a declaração `<!DOCTYPE>` no início do documento é restrita. O modo Quirks existe por compatibilidade com versões anteriores e uma de suas peculiaridades é que os

identificadores de classe no atributo `class` e nas folhas de estilos CSS não diferenciam letras maiúsculas e minúsculas. `getElementsByClassName()` segue o algoritmo de correspondência usado pelas folhas de estilo. Se o documento é renderizado no modo Quirks, o método faz uma comparação de string que não diferencia letras maiúsculas e minúsculas. Caso contrário, a comparação diferencia letras maiúsculas e minúsculas.

Quando este livro estava sendo escrito, `getElementsByClassName()` era implementada por todos os navegadores atuais, exceto o IE8 e anteriores. O IE8 suporta `querySelectorAll()`, descrito na próxima seção, e `getElementsByClassName()` pode ser implementado em cima desse método.

15.2.5 Selecionando elementos com seletores CSS

As folhas de estilos CSS têm uma sintaxe muito poderosa, conhecida como *seletores*, para descrever elementos ou conjuntos de elementos dentro de um documento. Os detalhes completos sobre a sintaxe de seletor CSS estão fora dos objetivos deste livro[1], mas alguns exemplos demonstrarão os fundamentos. Os elementos podem ser descritos pela identificação, nome de tag ou classe:

```
#nav                        // Um elemento com id="nav"
div                         // Qualquer elemento <div>
.warning                    // Qualquer elemento com "warning" em seu atributo class
```

De forma mais geral, os elementos podem ser selecionados com base em valores de atributo:

```
p[lang="fr"]                // Um parágrafo escrito em francês: <p lang="fr">
*[name="x"]                 // Qualquer elemento com um atributo name="x"
```

Esses seletores básicos podem ser combinados:

```
span.fatal.error            // Qualquer <span> com "fatal" e "error" em sua classe
span[lang="fr"].warning     // Qualquer aviso em francês
```

Os seletores também podem especificar estrutura de documento:

```
#log span                   // Qualquer descendente <span> do elemento com id="log"
#log>span                   // Qualquer filho <span> do elemento com id="log"
body>h1:first-child         // O primeiro filho <h1> de <body>
```

Os seletores podem ser combinados para selecionar vários elementos ou vários conjuntos de elementos:

```
div, #log    // Todos os elementos <div>, mais o elemento com id="log"
```

Como você pode ver, os seletores CSS permitem que elementos sejam selecionados de todas as maneiras descritas anteriormente: pela identificação, pelo nome, pelo nome de tag e pelo nome da classe. Junto com a padronização de seletores CSS3, outro padrão da W3C, conhecido como "API de Seletores" define métodos JavaScript para obter os elementos que coincidem com determinado seletor[2]. O segredo dessa API é o método `querySelectorAll()` de Document. Ele recebe um argumento de string contendo um seletor CSS e retorna um objeto NodeList representando todos

[1] Os seletores CSS3 estão especificados em *http://www.w3.org/TR/css3-selectors/*.

[2] O padrão API de Seletores não faz parte de HTML5, mas é intimamente relacionado a ela. Consulte *http://www.w3.org/TR/selectors-api/*.

os elementos do documento que correspondem ao seletor. Ao contrário dos métodos de seleção de elemento descritos anteriormente, o objeto NodeList retornado por `querySelectorAll()` não é dinâmico: ele contém os elementos que correspondiam ao seletor no momento em que o método foi chamado, mas não é atualizado quando o documento muda. Se nenhum elemento coincide, `querySelectorAll()` retorna um objeto NodeList vazio. Se a string do seletor é inválida, `querySelectorAll()` lança uma exceção.

Além de `querySelectorAll()`, o objeto documento também define `querySelector()`, que é como `querySelectorAll()` mas retorna somente o primeiro (na ordem do documento) elemento coincidente ou `null`, caso não haja elemento coincidente.

Esses dois métodos também são definidos em Elements (e também em nós DocumentFragment; consulte a Seção 15.6.4). Quando chamados em um elemento, o seletor especificado é comparado no documento inteiro e, então, o conjunto resultante é filtrado para que inclua somente os descendentes do elemento especificado. Isso pode parecer absurdo, pois significa que a string do seletor pode incluir ascendentes do elemento em relação ao qual é comparado.

Note que CSS define pseudoelementos `:first-line` e `:first-letter`. Em CSS, isso corresponde a partes de nós de texto, em vez de elementos reais. Eles não vão corresponder se usados com `querySelectorAll()` ou `querySelector()`. Além disso, muitos navegadores vão se recusar a retornar correspondências para as pseudoclasses `:link` e `:visited`, pois isso poderia expor informações sobre o histórico de navegação do usuário.

Todos os navegadores atuais suportam `querySelector()` e `querySelectorAll()`. Note, entretanto, que a especificação desses métodos não exige suporte para seletores CSS3: os navegadores são estimulados a suportar o mesmo conjunto de seletores que suportam em folhas de estilo. Os navegadores atuais, fora o IE, suportam seletores CSS3. O IE7 e 8 suportam seletores CSS2. (É esperado que o IE9 tenha suporte para CSS3.)

`querySelectorAll()` é o método definitivo de seleção de elemento: trata-se de uma técnica muito poderosa por meio da qual os programas JavaScript do lado do cliente podem selecionar os elementos do documento que vão manipular. Felizmente, esse uso de seletores CSS está disponível mesmo em navegadores sem suporte nativo para `querySelectorAll()`. A biblioteca jQuery (consulte o Capítulo 19) usa esse tipo de consulta baseada em seletor CSS como principal paradigma de programação. Os aplicativos Web baseados na jQuery utilizam um equivalente de `querySelectorAll()` portável e independente de navegador, chamado `$()`.

O código de correspondência de seletor CSS da jQuery foi decomposto e lançado como uma biblioteca independente, chamada Sizzle, que foi adotada pela Dojo e por outras bibliotecas do lado do cliente[3]. A vantagem de usar uma biblioteca como a Sizzle (ou uma biblioteca que utiliza Sizzle) é que as seleções funcionam até em navegadores mais antigos, e existe um conjunto básico de seletores que garantidamente funcionam em todos os navegadores.

15.2.6 document.all[]

Antes do DOM ser padronizado, o IE4 introduziu a coleção `document.all[]` que representava todos os elementos (mas não nós Text) do documento. `document.all[]` foi substituída por métodos

[3] Uma versão independente da Sizzle está disponível no endereço *http://sizzlejs.com*.

padrão, como `getElementById()` e `getElementsByTagName()`, e agora está obsoleta, não devendo ser usada. Contudo, quando foi apresentada, era revolucionária, sendo que ainda se pode ver código utilizando-a de uma destas maneiras:

```
document.all[0]           // O primeiro elemento no documento
document.all["navbar"]    // O elemento (ou elementos) com identificação ou nome "navbar"
document.all.navbar       // Idem
document.all.tags("div")  // Todos os elementos <div> no documento
document.all.tags("p")[0] // O primeiro <p> no documento
```


15.3 Estrutura de documentos e como percorrê-los

Após ter selecionado um Element de um Document, às vezes você precisa encontrar partes estruturalmente relacionadas (pai, irmãos, filhos) do documento. Um Document pode ser conceituado como uma árvore de objetos Node, como ilustrado na Figura 15-1. O tipo Node define propriedades para percorrer essa árvore, o que vamos abordar na Seção 15.3.1. Outra API permite percorrer documentos como árvores de objetos Element. A Seção 15.3.2 aborda essa API mais recente (e frequentemente mais fácil de usar).

15.3.1 Documentos como árvores de Nodes

O objeto Document, seus objetos Element e os objetos Text que representam texto no documento, são todos objetos Node. Node define as seguintes propriedades importantes:

`parentNode`

O objeto Node que é o pai desse nó, ou `null` para nós como o objeto Document, que não têm pai.

`childNodes`

Um objeto semelhante a um array somente para leitura (um NodeList) que é uma representação dinâmica dos nós filhos de um Node.

`firstChild, lastChild`

O primeiro e o último nós filhos de um nó, ou `null` se o nó não tem filhos.

`nextSibling, previousSibling`

O nó irmão próximo e anterior de um nó. Dois nós com o mesmo pai são irmãos. Sua ordem reflete a ordem na qual aparecem no documento. Essas propriedades conectam nós em uma lista duplamente encadeada.

`nodeType`

O tipo do nó. Os nós Document têm o valor 9. Os nós Element têm o valor 1. Os nós Text têm o valor 3. Os nós Comments são 8 e os nós DocumentFragment são 11.

`nodeValue`

O conteúdo textual de um nó Text ou Comment.

`nodeName`

O nome da marca de um Element, convertido em letras maiúsculas.

Usando-se as propriedades Node, o segundo nó filho do primeiro filho do Document pode ser referido com expressões como as seguintes:

```
document.childNodes[0].childNodes[1]
document.firstChild.firstChild.nextSibling
```

Suponha que o documento em questão seja o seguinte:

```
<html><head><title>Test</title></head><body>Hello World!</body></html>
```

Então, o segundo filho do primeiro filho é o elemento `<body>`. Ele tem `nodeType` 1 e `nodeName` "BODY".

Note, entretanto, que essa API é extremamente sensível a variações no texto do documento. Se o documento é modificado pela inserção de uma nova linha entre a marcação `<html>` e a marcação `<head>`, por exemplo, o nó Text que representa essa nova linha se torna o primeiro filho do primeiro filho e o segundo filho é o elemento `<head>`, em vez do corpo de `<body>`.

15.3.2 Documentos como árvores de Elements

Quando estamos interessados principalmente nos objetos Element de um documento, em vez do texto dentro deles (e o espaço em branco entre eles), é útil usar uma API que nos permita tratar um documento como uma árvore de objetos Element, ignorando os nós Text e Comment que também fazem parte do documento.

A primeira parte dessa API é a propriedade `children` de objetos Element. Assim como `childNodes`, isso é um NodeList. Ao contrário de `childNodes`, contudo, a lista de `children` contém apenas objetos Element. A propriedade `children` não é padronizada, mas funciona em todos os navegadores atuais. O IE a implementou por um longo tempo e a maioria dos outros navegadores fez o mesmo. O último navegador importante a adotá-la foi o Firefox 3.5.

Note que os nós Text e Comment não podem ter filhos, ou seja, a propriedade `Node.parentNode`, descrita anteriormente, nunca retorna um nó Text ou Comment. O `parentNode` de qualquer Element vai ser sempre outro Element ou, na raiz da árvore, um Document ou DocumentFragment.

A segunda parte de uma API para percorrer documentos baseada em elemento são propriedades Element análogas às propriedades filho e irmão do objeto Node:

`firstElementChild, lastElementChild`

Parecidas com `firstChild` e `lastChild`, mas apenas para filhos de Element.

`nextElementSibling, previousElementSibling`

Parecidas com `nextSibling` e `previousSibling`, mas apenas para irmãos de Element.

`childElementCount`

O número de filhos do elemento. Retorna o mesmo valor que `children.length`.

Essas propriedades filho e irmão são padronizadas e implementadas em todos os navegadores atuais, exceto o IE[4].

Como a API para percorrer documentos elemento por elemento ainda não é completamente universal, talvez você queira definir funções portáveis para percorrê-los, como as do Exemplo 15-2.

Exemplo 15-2 Funções portáveis para percorrer documentos

```
/**
 * Retorna o n-ésimo ascendente de e, ou null se não existe tal ascendente
 * ou, se esse ascendente não é um Element (um Document ou DocumentFragment, por
 * exemplo).
 * Se n é 0, retorna o próprio e. Se n é 1 (ou
 * é omitido), retorna o pai. Se n é 2, retorna o avô, etc.
 */
function parent(e, n) {
    if (n === undefined) n = 1;
    while(n-- && e) e = e.parentNode;
    if (!e || e.nodeType !== 1) return null;
    return e;
}

/**
 * Retorna o n-ésimo elemento irmão do Element e.
 * Se n é positivo, retorna o n-ésimo próximo elemento irmão.
 * Se n é negativo, retorna o n-ésimo elemento irmão anterior.
 * Se n é zero, retorna o próprio e.
 */
function sibling(e,n) {
    while(e && n !== 0) {       // Se e não está definido, apenas o retornamos
        if (n > 0) {            // Localiza o próximo irmão do elemento
            if (e.nextElementSibling) e = e.nextElementSibling;
            else {
                for(e=e.nextSibling; e && e.nodeType !== 1; e=e.nextSibling)
                    /* laço vazio */ ;
            }
            n--;
        }
        else {  // Localiza o irmão anterior do elemento
            if (e.previousElementSibing) e = e.previousElementSibling;
            else {
                for(e=e.previousSibling; e&&e.nodeType!==1; e=e.previousSibling)
                    /* laço vazio */ ;
            }
            n++;
        }
    }
    return e;
}
```

[4] *http://www.w3.org/TR/ElementTraversal.*

```
/**
 * Retorna o n-ésimo elemento filho de e, ou null se ele não tem um.
 * Valores negativos de n contam a partir do final. 0 significa o primeiro filho, mas
 * -1 significa o último filho, -2 significa o penúltimo e assim por diante.
 */
function child(e, n) {
    if (e.children) {                       // Se o array children existe
        if (n < 0) n += e.children.length;  // Converte n negativo no índice do array
        if (n < 0) return null;             // Se ainda é negativo, nenhum filho
        return e.children[n];               // Retorna o filho especificado
    }

    // Se e não tem um array de filhos, localiza o primeiro filho e conta
    // para frente ou localiza o último filho e conta para trás a partir de lá.
    if (n >= 0) {     // n é não negativo: conta para frente a partir do primeiro filho
        // Localiza o primeiro elemento filho de e
        if (e.firstElementChild) e = e.firstElementChild;
        else {
            for(e = e.firstChild; e && e.nodeType !== 1; e = e.nextSibling)
                /* vazio */;
        }
        return sibling(e, n);   // Retorna o n-ésimo irmão do primeiro filho
    }
    else {                      // n é negativo; portanto, conta para trás a partir do fim
        if (e.lastElementChild) e = e.lastElementChild;
        else {
            for(e = e.lastChild; e && e.nodeType !== 1; e=e.previousSibling)
                /* vazio */;
        }
        return sibling(e, n+1); // +1 para converter filho -1 para irmão 0 do último
    }
}
```

Definindo métodos de Element personalizados

Todos os navegadores atuais (incluindo o IE8, mas não o IE7 e anteriores) implementam o DOM, de modo que tipos como Element e HTMLDocument[5] são classes como String e Array. Elas não são construtoras (vamos ver como se cria novos objetos Element posteriormente no capítulo), mas têm protótipos de objetos e você pode estendê-las com métodos personalizados:

```
Element.prototype.next = function() {
    if (this.nextElementSibling) return this.nextElementSibling;
    var sib = this.nextSibling;
    while(sib && sib.nodeType !== 1) sib = sib.nextSibling;
    return sib;
};
```

[5] O IE8 suporta protótipos que podem ser estendidos para Element, HTMLDocument e Text, mas não para Node, Document, HTMLElement ou qualquer um dos subtipos mais específicos de HTMLElement.

As funções do Exemplo 15-2 não são definidas como métodos de Element porque essa técnica não é suportada pelo IE7.

No entanto, essa capacidade de estender tipos DOM ainda é útil se você quer implementar recursos específicos do IE em outros navegadores. Conforme mencionado anteriormente, a propriedade não padronizada `children` de Element foi introduzida pelo IE e adotada por outros navegadores. Você pode usar código como o seguinte para simulá-la em navegadores que não a suportam, como o Firefox 3.0:

```
// Simula a propriedade Element.children em navegadores que não o IE que não a tem
// Note que isso retorna um array estático, em vez de um NodeList dinâmico
if (!document.documentElement.children) {
    Element.prototype.__defineGetter__("children", function() {
        var kids = [];
        for(var c = this.firstChild; c != null; c = c.nextSibling)
            if (c.nodeType === 1) kids.push(c);
        return kids;
    });
}
```

O método `__defineGetter__` (abordado na Seção 6.7.1) é completamente não padrão, mas é perfeito para código de portabilidade como esse.

15.4 Atributos

Os elementos HTML consistem em um nome de tag e um conjunto de pares nome/valor conhecidos como *atributos*. O elemento `<a>` que define um hiperlink, por exemplo, utiliza o valor de seu atributo `href` como destino do link. Os valores de atributo dos elementos HTML estão disponíveis como propriedades dos objetos HTMLElement que representam esses elementos. O DOM também define outras APIs para obter e configurar os valores de atributos XML e atributos HTML não padronizados. As subseções a seguir têm detalhes.

15.4.1 Atributos HTML como propriedades de Element

Os objetos HTMLElement que representam os elementos de um documento HTML definem propriedades de leitura/gravação que espelham os atributos HTML dos elementos. HTMLElement define propriedades para os atributos HTTP universais, como `id`, `title` `lang` e `dir`, e propriedades de rotina de tratamento de evento, como `onclick`. Os subtipos específicos dos elementos definem atributos específicos para esses elementos. Para consultar o URL de uma imagem, por exemplo, você pode usar a propriedade `src` do objeto HTMLElement que representa o elemento `<img>`:

```
var image = document.getElementById("myimage");
var imgurl = image.src;          // O atributo src é o URL da imagem
image.id === "myimage"           // Visto que pesquisamos a imagem pela identificação
```

Da mesma forma, você poderia configurar os atributos de envio de formulário de um elemento `<form>` com código como o seguinte:

```
var f = document.forms[0];                      // Primeiro <form> no documento
f.action = "http://www.example.com/submit.php"; // Configura o URL para envio.
f.method = "POST";                              // Tipo de pedido HTTP
```

Os atributos HTML não diferenciam letras maiúsculas e minúsculas, mas os nomes de propriedade de JavaScript, sim. Para converter um nome de atributo em propriedade JavaScript, escreva-o em letras minúsculas. No entanto, se o atributo utiliza mais de uma palavra, coloque a primeira letra de cada palavra após a primeira delas em maiúscula: `defaultChecked` e `tabIndex`, por exemplo.

Alguns nomes de atributo HTML são palavras reservadas em JavaScript. Para esses, a regra geral é prefixar o nome de propriedade com "html". O atributo HTML `for` (do elemento `<label>`), por exemplo, se torna a propriedade JavaScript `htmlFor`. "class" é uma palavra reservada (mas não utilizada) em JavaScript e o importante atributo HTML `class` é uma exceção à regra anterior: ele se torna `className` em código JavaScript. Vamos ver a propriedade `className` novamente, no Capítulo 16.

As propriedades que representam atributos HTML normalmente têm um valor de string. Quando o atributo é um valor booleano ou numérico (os atributos `defaultChecked` e `maxLength` de um elemento `<input>`, por exemplo), os valores das propriedades são booleanos ou números, em vez de strings. Os atributos de rotina de tratamento de evento sempre têm objetos Function (ou `null`) como valores. A especificação HTML5 define alguns atributos (como o atributo `form` de `<input>` e elementos relacionados) que convertem identificações de elemento em objetos Element reais. Por fim, o valor da propriedade `style` de qualquer elemento HTML é um objeto CSSStyleDeclaration, em vez de uma string. Vamos ver muito mais sobre essa importante propriedade, no Capítulo 16.

Note que essa API baseada em propriedades para obter e configurar valores de atributo não define nenhuma maneira de remover um atributo de um elemento. Em especial, o operador `delete` não pode ser usado para esse propósito. A seção a seguir descreve um método que pode ser usado para isso.

15.4.2 Obtendo e configurando atributos que não são HTML

Conforme descrito anteriormente, HTMLElement e seus subtipos definem propriedades que correspondem aos atributos padrão de elementos HTML. O tipo Element também define métodos `getAttribute()` e `setAttribute()` que podem ser usados para consultar e configurar atributos HTML não padronizados e para consultar e configurar atributos nos elementos de um documento XML:

```
var image = document.images[0];
var width = parseInt(image.getAttribute("WIDTH"));
image.setAttribute("class", "thumbnail");
```

O código anterior destaca duas importantes diferenças entre esses métodos e a API baseada em propriedades descritas. Primeiramente, todos os valores de atributo são tratados como strings. `getAttribute()` nunca retorna um número, booleano ou objeto. Segundo, esses métodos utilizam nomes de atributo padrão, mesmo quando esses nomes são palavras reservadas em JavaScript. Para elementos HTML, os nomes de atributo não diferenciam letras maiúsculas e minúsculas.

Element também define dois métodos relacionados, `hasAttribute()` e `removeAttribute()`, o primeiro dos quais verifica a presença de um atributo nomeado e o outro remove um atributo inteiramente. Esses métodos são especialmente úteis com atributos booleanos: esses são atributos (como o atributo `disabled` de elementos de formulário HTML) cuja presença ou ausência em um elemento importa, mas cujo valor não é relevante.

Se estiver trabalhando com documentos XML que incluem atributos de outros espaço de nomes, pode usar as respectivas variantes desses quatro métodos: `getAttributeNS()`, `setAttributeNS()`, `hasAttributeNS()` e `removeAttributeNS()`. Em vez de receberem uma única string como nome de atributo, esses métodos recebem duas. A primeira é o URI que identifica o espaço de nomes. O segundo normalmente é o nome local não qualificado do atributo dentro do espaço de nomes. Contudo, apenas para `setAttributeNS()`, o segundo argumento é o nome qualificado do atributo e inclui o prefixo do espaço de nomes. Você pode ler mais sobre esses métodos com atributo com consciência de espaço de nomes na Parte IV.

15.4.3 Atributos de conjuntos de dados

Às vezes é útil anexar informações nos elementos HTML, normalmente quando o código JavaScript vai selecioná-los e manipulá-los de algum modo. Às vezes isso pode ser feito pela adição de identificadores especiais no atributo `class`. Outras vezes, para dados mais complexos, os programadores do lado do cliente recorrem a atributos não padronizados. Conforme mencionado, você pode usar os métodos `getAttribute()` e `setAttribute()` para ler e gravar valores de atributos não padronizados. O preço a ser pago, no entanto, é que seu documento não vai ser um HTML válido.

HTML5 oferece uma solução. Em um documento HTML5, qualquer atributo cujo nome apareça em letras minúsculas e comece com o prefixo "data-" é considerado válido. Esses "atributos de conjunto de dados" não vão afetar a apresentação dos elementos nos quais aparecem e definem uma maneira padronizada de anexar mais dados sem comprometer a validade do documento.

HTML5 também define uma propriedade `dataset` em objetos Element. Essa propriedade se refere a um objeto, o qual tem propriedades que correspondem aos atributos `data-` com o prefixo removido. Assim, `dataset.x` conteria o valor do atributo `data-x`. Os atributos hifenizados são mapeados em nomes de propriedade com maiúsculas no meio: o atributo `data-jquery-test` se torna a propriedade `dataset.jqueryTest`.

Como um exemplo mais concreto, suponha que um documento contém a seguinte marcação:

```
<span class="sparkline" data-ymin="0" data-ymax="10">
1 1 1 2 2 3 4 5 5 4 3 5 6 7 7 4 2 1
</span>
```

Sparkline é um pequeno gráfico – frequentemente um gráfico de linhas – destinado a exibição dentro do fluxo de texto. Para gerar um gráfico de linhas, você poderia extrair o valor dos atributos do conjunto de dados anterior com código como o seguinte:

```
// Supõe que o método ES5 Array.map() (ou um de funcionamento igual) esteja definido
var sparklines = document.getElementsByClassName("sparkline");
for(var i = 0; i < sparklines.length; i++) {
    var dataset = sparklines[i].dataset;
    var ymin = parseFloat(dataset.ymin);
    var ymax = parseFloat(dataset.ymax);
    var data = sparklines[i].textContent.split(" ").map(parseFloat);
    drawSparkline(sparklines[i], ymin, ymax, data);      // Ainda não implementado
}
```

Quando este livro estava sendo escrito, a propriedade `dataset` não estava implementada nos navegadores e o código anterior teria de ser escrito como segue:

```
var sparklines = document.getElementsByClassName("sparkline");
for(var i = 0; i < sparklines.length; i++) {
    var elt = sparklines[i];
    var ymin = parseFloat(elt.getAttribute("data-ymin"));
    var ymin = parseFloat(elt.getAttribute("data-ymax"));
    var points = elt.getAttribute("data-points");
    var data = elt.textContent.split(" ").map(parseFloat);
    drawSparkline(elt, ymin, ymax, data);       // Ainda não implementado
}
```

Note que a propriedade `dataset` é (ou será, quando for implementada) uma interface bidirecional dinâmica para os atributos `data-` de um elemento. Configurar ou excluir uma propriedade de `dataset` configura ou remove o atributo `data-` correspondente do elemento.

A função `drawSparkline()` nos exemplos anteriores é fictícia, mas o Exemplo 21-13 traça gráficos de linha com marcação como esse, usando o elemento `<canvas>`.

15.4.4 Atributos como nós Attr

Há mais uma maneira de trabalhar com os atributos de um objeto Element. O tipo Node define uma propriedade `attributes`. Essa propriedade é `null` para todos os nós que não são objetos Element. Para objetos Element, `attributes` é um objeto semelhante a um array somente para leitura que representa todos os atributos do elemento. O objeto `attributes` é dinâmico como os NodeLists. Ele pode ser indexado numericamente, ou seja, é possível enumerar todos os atributos de um elemento. E também pode ser indexado por nome de atributo:

```
document.body.attributes[0]          // O primeiro atributo do elemento <body>
document.body.attributes.bgcolor     // O atributo bgcolor do elemento <body>
document.body.attributes["ONLOAD"]   // O atributo onload do elemento <body>
```

Os valores obtidos ao se indexar o objeto `attributes` são objetos Attr. Os objetos Attr são um tipo especializado de Node, mas nunca são utilizados dessa forma. As propriedades `name` e `value` de um Attr retornam o nome e o valor do atributo.

15.5 Conteúdo de elemento

Veja novamente a Figura 15-1 e pergunte-se qual é o "conteúdo" do elemento `<p>`. Existem três maneiras de respondermos a essa questão:

- O conteúdo é a string HTML "This is a simple document".
- O conteúdo é a string de texto puro "This is a simple document".
- O conteúdo é um nó Text, um nó Element que tem um nó filho Text e outro nó Text.

Todas essas são respostas válidas e cada resposta é útil à sua própria maneira. As seções a seguir explicam como trabalhar com a representação HTML, com a representação em texto puro e com a representação em árvore de conteúdo de elemento.

15.5.1 Conteúdo de elemento como HTML

A leitura da propriedade `innerHTML` de um Element retorna o conteúdo desse elemento como uma string de marcação. Configurar essa propriedade em um elemento invoca o parser do navegador Web e substitui o conteúdo atual do elemento por uma representação analisada da nova string. (Apesar de seu nome, `innerHTML` pode ser usada com elementos XML e com elementos HTML.)

Os navegadores Web são muito bons na análise de HTML e a configuração de `innerHTML` normalmente é muito eficiente, mesmo que o valor especificado precise ser analisado. Note, entretanto, que anexar trechos de texto repetidamente na propriedade `innerHTML` com o operador `+=` normalmente não é eficiente, pois isso exige uma etapa de serialização e uma etapa de análise.

A propriedade `innerHTML` foi introduzida no IE4. Embora seja suportada há tempos por todos os navegadores, somente com o advento de HTML5 foi padronizada. HTML5 diz que `innerHTML` deve funcionar em nós Document e em nós Element, mas isso ainda não é suportado universalmente.

HTML5 também padroniza uma propriedade chamada `outerHTML`. Quando se consulta `outerHTML`, a string de marcação HTML ou XML retornada inclui as tags de abertura e fechamento do elemento no qual ela foi consultada. Quando se configura `outerHTML` em um elemento, o novo conteúdo substitui o elemento em si. `outerHTML` só é definida para nós Element, não para Documents. Quando este livro estava sendo escrito, `outerHTML` era suportada por todos os navegadores vigentes, exceto o Firefox. (Veja o Exemplo 15-5, posteriormente neste capítulo, para uma implementação de `outerHTML` baseada em `innerHTML`.)

Outro recurso introduzido pelo IE e padronizado em HTML5 é o método `insertAdjacentHTML()`, o qual permite inserir uma string de marcação HTML arbitrária "adjacente" ao elemento especificado. A marcação é passada como segundo argumento para esse método e o significado preciso de "adjacente" depende do valor do primeiro argumento. Esse primeiro argumento deve ser uma string com um dos valores "beforebegin", "afterbegin", "beforeend" ou "afterend". Esses valores correspondem aos pontos de inserção ilustrados na Figura 15-3.

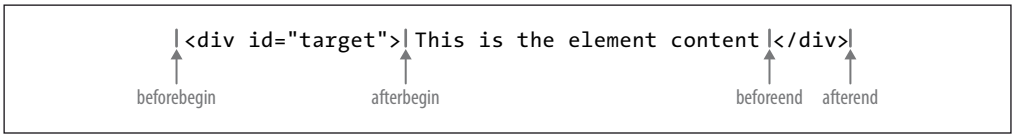

Figura 15-3 Pontos de inserção para insertAdjacentHTML().

`insertAdjacentHTML()` não é suportado pelas versões atuais do Firefox. Posteriormente neste capítulo, o Exemplo 15-6 mostra como implementar `insertAdjacentHTML()` usando a propriedade `innerHTML`

e também demonstra como escrever métodos de inserção de HTML que não exigem especificar a posição de inserção com um argumento de string.

15.5.2 Conteúdo de elemento como texto puro

Às vezes você quer consultar o conteúdo de um elemento como texto puro ou inserir texto puro em um documento (sem fazer o escape dos sinais de menor e maior e dos E comerciais utilizados na marcação HTML). O modo padrão de fazer isso é com a propriedade `textContent` de Node:

```
var para = document.getElementsByTagName("p")[0];         // Primeiro <p> no documento
var text = para.textContent;         // O texto é "This is a simple document."
para.textContent = "Hello World!";   // Altera o conteúdo do parágrafo
```

A propriedade `textContent` é suportada por todos os navegadores atuais, exceto o IE. No IE, você pode usar a propriedade Element `innerText`. A Microsoft introduziu `innerText` no IE4 e ela é suportada por todos os navegadores atuais, exceto o Firefox.

As propriedades `textContent` e `innerText` são semelhantes o bastante para que em geral possam ser utilizadas indistintamente. Tome o cuidado, contudo, para diferenciar elementos vazios (a string "" em JavaScript é falsa) das propriedades indefinidas:

```
/**
 * Com um argumento, retorna textContent ou innerText do elemento.
 * Com dois argumentos, configura textContent ou innerText do elemento com value.
 */
function textContent(element, value) {
    var content = element.textContent;  // Verifica se textContent está definida
    if (value === undefined) {  // Nenhum valor passado; portanto, retorna o texto atual
        if (content !== undefined) return content;
        else return element.innerText;
    }
    else {   // Um valor foi passado; portanto, configura o texto
        if (content !== undefined) element.textContent = value;
        else element.innerText = value;
    }
}
```

A propriedade `textContent` é uma concatenação simples de todos os descendentes de nó Text do elemento especificado. `innerText` não tem um comportamento claramente especificado, mas difere de `textContent` de várias maneiras. `innerText` não retorna o conteúdo de elementos `<script>`, omite espaço em branco irrelevante e tenta preservar formatação de tabela. Além disso, `innerText` é tratada como uma propriedade somente para leitura em certos elementos de tabela, como `<table>`, `<tbody>` e `<tr>`.

Texto em elementos <script>

Os elementos `<script>` em linha (isto é, aqueles que não têm um atributo `src`) têm uma propriedade `text` que pode ser usada para recuperar seu texto. O conteúdo de um elemento `<script>` nunca é exibido pelo navegador e o parser de HTML ignora sinais de menor e maior e sinais de E comercial dentro de um script. Isso torna o elemento `<script>` um lugar ideal para incorporar dados textuais arbitrários para uso por seu aplicativo. Basta configurar o atributo `type` do elemento com algum valor (como "text/x-custom-

-data") que torne claro que o script não é código JavaScript executável. Se você fizer isso, o interpretador JavaScript vai ignorar o script, mas o elemento vai existir na árvore de documentos e sua propriedade `text` vai retornar os dados.

15.5.3 Conteúdo de elemento como nós Text

Outra maneira de trabalhar com o conteúdo de um elemento é como uma lista de nós filhos, cada um dos quais podendo ter seu próprio conjunto de filhos. Quando se pensa em conteúdo de elemento, normalmente são os nós Text que têm interesse. Em documentos XML, você também deve estar preparado para tratar de nós CDATASection – eles são um subtipo de Text e representam o conteúdo de seções CDATA.

O Exemplo 15-3 mostra uma função `textContent()` que percorre os filhos de um elemento recursivamente e concatena o texto de todos os descendentes do nó Text. Para entender o código, lembre-se de que a propriedade `nodeValue` (definida pelo tipo Node) possui o conteúdo de um nó Text.

Exemplo 15-3 Localizando todos os descendentes do nó Text de um elemento

```
// Retorna o conteúdo de texto puro do elemento e, usando recursividade para os elementos
// filhos.
// Este método funciona como a propriedade textContent
function textContent(e) {
    var child, type, s = "";            // s contém o texto de todos os filhos
    for(child = e.firstChild; child != null; child = child.nextSibling) {
        type = child.nodeType;
        if (type === 3 || type === 4)   // Nós Text e CDATASection
            s += child.nodeValue;
        else if (type === 1)            // Recursividade para nós Element
            s += textContent(child);
    }
    return s;
}
```

A propriedade `nodeValue` é de leitura/gravação e você pode configurá-la de modo a alterar o conteúdo exibido por um nó Text ou CDATASection. Tanto Text como CDATASection são subtipos de CharacterData, sobre o qual você pode pesquisar na Parte IV. CharacterData define uma propriedade `data`, a qual é o mesmo texto de `nodeValue`. A função a seguir converte o conteúdo de nós Text para maiúsculas, configurando a propriedade `data`:

```
// Converte recursivamente todos os descendentes do nó Text de n para maiúsculas.
function upcase(n) {
    if (n.nodeType == 3 || n.nodeTyep == 4)      // Se n é Text ou CDATA
        n.data = n.data.toUpperCase();           // ...converte o conteúdo para
                                                 // maiúsculas.
    else                                         // Caso contrário, usa recursividade nos
                                                 // nós filhos
        for(var i = 0; i < n.childNodes.length; i++)
            upcase(n.childNodes[i]);
}
```

CharacterData também define métodos pouco usados para anexar, excluir, inserir e substituir texto dentro de um nó Text ou CDATASection. Em vez de alterar o conteúdo de nós Text existentes, também é possível inserir novos nós Text em um Element ou substituir nós existentes por novos nós Text. A criação, inserção e exclusão de nós são o tema da próxima seção.

15.6 Criando, inserindo e excluindo nós

Vimos como consultar e alterar conteúdo de documento usando strings HTML e de texto puro. E também vimos que podemos percorrer um objeto Document para examinar os nós Element e Text individuais de que é constituído. Também é possível alterar um documento no nível dos nós individuais. O tipo Document define métodos para criar objetos Element e Text e o tipo Node define métodos para inserir, excluir e substituir nós na árvore. O Exemplo 13-4 demonstrou a criação e a inserção de nós e aquele breve exemplo está duplicado aqui:

```
// Carrega e executa um script de forma assíncrona a partir de um URL especificado
function loadasync(url) {
    var head = document.getElementsByTagName("head")[0];  // Localiza <head> do documento
    var s = document.createElement("script");             // Cria um elemento <script>
    s.src = url;                                          // Configura seu atributo src
    head.appendChild(s);                                  // Insere o <script> no cabeçalho
}
```

As subseções a seguir contêm mais detalhes e exemplos de criação de nó, de inserção e exclusão de nós e também do uso de DocumentFragment como atalho ao se trabalhar com vários nós.

15.6.1 Criando nós

Como mostrado no código anterior, é possível criar novos nós Element com o método `createElement()` do objeto Document. Passe o nome da tag do elemento como argumento do método: esse nome não diferencia letras maiúsculas e minúsculas para documentos HTML e diferencia para documentos XML.

Os nós Text são criados com um método semelhante:

```
var newnode = document.createTextNode("text node content");
```

Document também define outros métodos de fábrica, como o pouco usado `createComment()`. Vamos usar o método `createDocumentFragment()` na Seção 15.6.4. Ao trabalhar com documentos que usam espaço de nomes XML, você pode utilizar `createElementNS()` para especificar o URI do espaço de nomes e o nome de tag do objeto Element a ser criado.

Outra maneira de criar novos nós de documento é fazer cópias dos já existentes. Todo nó tem um método `cloneNode()` que retorna uma nova cópia do nó. Passe `true` para também copiar todos os descendentes recursivamente, ou `false` para fazer apenas uma cópia rasa. Nos navegadores (menos o IE), o objeto Document também define um método semelhante, chamado `importNode()`. Se você passa para ele um nó de outro documento, ele retorna uma cópia conveniente para inserção nesse documento. Passe `true` como segundo argumento para importar recursivamente todos os descendentes.

15.6.2 Inserindo nós

Uma vez que você tenha um novo nó, pode inseri-lo no documento com os métodos `appendChild()` ou `insertBefore()` de Node. `appendChild()` é chamado no nó Element em que você deseja inserir, sendo que ele insere o nó especificado de modo a se tornar o `last Child` desse nó.

`insertBefore()` é como `appendChild()`, mas recebe dois argumentos. O primeiro é o nó a ser inserido. O segundo argumento é o nó antes do qual esse nó vai ser inserido. Esse método é chamado no nó que vai ser o pai do novo nó e o segundo argumento deve ser filho desse nó pai. Se você passa `null` como segundo argumento, `insertBefore()` se comporta como `appendChild()` e insere no final.

Aqui está uma função simples para inserir um nó em um índice numérico. Ela demonstra `appendChild()` e `insertBefore()`:

```
// Insere o nó filho no pai de modo a se tornar o nó filho n
function insertAt(parent, child, n) {
    if (n < 0 || n > parent.childNodes.length) throw new Error("invalid index");
    else if (n == parent.childNodes.length) parent.appendChild(child);
    else parent.insertBefore(child, parent.childNodes[n]);
}
```

Se você chamar `appendChild()` ou `insertBefore()` para inserir um nó que já está no documento, esse nó será automaticamente removido de sua posição atual e reinserido em sua nova posição: não há necessidade de remover o nó explicitamente. O Exemplo 15-4 mostra uma função para classificar as linhas de uma tabela com base nos valores das células em uma coluna especificada. Ela não cria novos nós, mas utiliza `appendChild()` para mudar a ordem dos nós existentes.

Exemplo 15-4 Classificando as linhas de uma tabela

```
// Classifica as linhas no primeiro <tbody> da tabela especificada, de acordo com
// o valor da n-ésima célula dentro de cada linha. Usa a função comparator
// se uma estiver especificada. Caso contrário, compara os valores alfabeticamente.
function sortrows(table, n, comparator) {
    var tbody = table.tBodies[0];   // Primeiro <tbody>; pode ser criado implicitamente
    var rows = tbody.getElementsByTagName("tr"); // Todas as linhas no tbody
    rows = Array.prototype.slice.call(rows,0);  // Instantâneo em um array real

    // Agora classifica as linhas com base no texto do n-ésimo elemento <td>
    rows.sort(function(row1,row2) {
        var cell1 = row1.getElementsByTagName("td")[n];  // Obtém a n-ésima célula
        var cell2 = row2.getElementsByTagName("td")[n];  // das duas linhas
        var val1 = cell1.textContent || cell1.innerText; // Obtém conteúdo do texto
        var val2 = cell2.textContent || cell2.innerText; // das duas células
        if (comparator) return comparator(val1, val2);   // Compara-os!
        if (val1 < val2) return -1;
        else if (val1 > val2) return 1;
        else return 0;
    });
```

```
        // Agora anexa as linhas no tbody, em sua ordem classificada.
        // Isso as move automaticamente de sua posição atual; portanto, não há
        // necessidade de removê-las primeiro. Se o <tbody> contiver quaisquer
        // nós que não sejam elementos <tr>, esses nós vão flutuar para o topo.
        for(var i = 0; i < rows.length; i++) tbody.appendChild(rows[i]);
    }

    // Localiza os elementos <th> da tabela (supondo que exista apenas uma linha deles)
    // e os torna clicáveis para que um clique em um cabeçalho de coluna classifique
    // por essa coluna.
    function makeSortable(table) {
        var headers = table.getElementsByTagName("th");
        for(var i = 0; i < headers.length; i++) {
            (function(n) {  // Função aninhada para criar um escopo local
                headers[i].onclick = function() { sortrows(table, n); };
            }(i));          // Atribui o valor de i à variável local n
        }
    }
```

15.6.3 Removendo e substituindo nós

O método `removeChild()` remove um nó da árvore de documentos. Contudo, tome cuidado: esse método não é chamado no nó a ser removido, mas (conforme implica a parte "child" – filho – de seu nome) no pai desse nó. Chame o método a partir do nó pai e passe como argumento o nó filho que deve ser removido. Para remover o nó `n` do documento, você escreveria:

```
n.parentNode.removeChild(n);
```

`replaceChild()` remove um nó filho e o substitui por um novo nó. Chame esse método no nó pai, passando o novo nó como primeiro argumento e o nó a ser substituído como segundo argumento. Para substituir o nó `n` por uma string de texto, por exemplo, você poderia escrever:

```
n.parentNode.replaceChild(document.createTextNode("[ REDACTED ]"), n);
```

A função a seguir demonstra outro uso de `replaceChild()`:

```
// Substitui o nó n por um novo elemento <b> e torna n um filho desse elemento.
function embolden(n) {
    // Se passamos uma string em vez de um nó, trata-a como uma identificação de elemento
    if (typeof n == "string") n = document.getElementById(n);
    var parent = n.parentNode;                  // Obtém o pai de n
    var b = document.createElement("b");        // Cria um elemento <b>
    parent.replaceChild(b, n);                  // Substitui pelo elemento <b>
    b.appendChild(n);                           // Torna n um filho do elemento <b>
}
```

A Seção 15.5.1 apresentou a propriedade `outerHTML` de um elemento e explicou que ela não foi implementada nas versões atuais do Firefox. O Exemplo 15-5 mostra como implementar essa propriedade no Firefox (e em qualquer outro navegador que suporte `innerHTML`, tenha um objeto `Element.prototype` extensível e tenha métodos para definir getters e setters de propriedade). O código também demonstra um uso prático para os métodos `removeChild()` e `cloneNode()`.

Exemplo 15-5 Implementando a propriedade outerHTML usando innerHTML

```
// Implementa a propriedade outerHTML para navegadores que não a suportam.
// Presume que o navegador suporta innerHTML, tem um
// Element.prototype extensível e permite a definição de getters e setters.
(function() {
    // Se já temos outerHTML retorna sem fazer nada
    if (document.createElement("div").outerHTML) return;

    // Retorna a HTML externa do elemento referido por this
    function outerHTMLGetter() {
        var container = document.createElement("div");    // Elemento fictício
        container.appendChild(this.cloneNode(true));      // Copia this no fictício
        return container.innerHTML;                       // Retorna conteúdo fictício
    }

    // Configura a HTML externa desse elemento com o valor especificado
    function outerHTMLSetter(value) {
        // Cria um elemento fictício e configura seu conteúdo com o valor especificado
        var container = document.createElement("div");
        container.innerHTML = value;
        // Move cada um dos nós do fictício para o documento
        while(container.firstChild) // Itera até que container não tenha mais filhos
            this.parentNode.insertBefore(container.firstChild, this);
        // E remove o nó que foi substituído
        this.parentNode.removeChild(this);
    }

    // Agora usa estas duas funções como getters e setters para a
    // propriedade outerHTML de todos os objetos Element. Usa Object.defineProperty da ES5
    // se existe; caso contrário, recorre a __defineGetter__ e Setter__.
    if (Object.defineProperty) {
        Object.defineProperty(Element.prototype, "outerHTML", {
                                    get: outerHTMLGetter,
                                    set: outerHTMLSetter,
                                    enumerable: false, configurable: true
                                });
    }
    else {
        Element.prototype.__defineGetter__("outerHTML", outerHTMLGetter);
        Element.prototype.__defineSetter__("outerHTML", outerHTMLSetter);
    }
}());
```

15.6.4 Usando DocumentFragments

DocumentFragment é um tipo especial de Node que serve como contêiner temporário para outros nós. Crie um DocumentFragment como segue:

```
var frag = document.createDocumentFragment();
```

Assim como um nó Document, um DocumentFragment é independente e não faz parte de nenhum outro documento. Seu `parentNode` é sempre `null`. Contudo, assim como um Element, um

DocumentFragment pode ter qualquer número de filhos, os quais podem ser manipulados com `appendChild()`, `insertBefore()`, etc.

O detalhe especial sobre DocumentFragment é que permite a um conjunto de nós ser tratado como um único nó: se você passa um DocumentFragment para `appendChild()`, `insertBefore()` ou `replaceChild()`, são os filhos do fragmento que são inseridos no documento e não o próprio fragmento. (Os filhos são movidos do fragmento para o documento e o fragmento se torna vazio e pronto para reutilização.) A função a seguir usa um DocumentFragment para inverter a ordem dos filhos de um nó:

```
// Inverte a ordem dos filhos do Node n
function reverse(n) {
    // Cria um DocumentFragment vazio como contêiner temporário
    var f = document.createDocumentFragment();
    // Agora itera para trás através dos filhos, movendo cada um para o fragmento.
    // O último filho de n se torna o primeiro filho de f e vice-versa.
    // Note que anexar um filho em f o remove automaticamente de n.
    while(n.lastChild) f.appendChild(n.lastChild);

    // Por fim, move os filhos de f, todos de uma vez, de volta para n, tudo de uma só vez.
    n.appendChild(f);
}
```

O Exemplo 15-6 implementa o método `insertAdjacentHTML()` (consulte a Seção 15.5.1) usando a propriedade `innerHTML` e um DocumentFragment. Ele também define funções de inserção HTML com nomes lógicos, como uma alternativa à confusa API `insertAdjacentHTML()`. A função utilitária interna `fragment()` possivelmente é a parte mais útil desse código: ela retorna um DocumentFragment que contém a representação analisada de uma string de texto HTML especificada.

Exemplo 15-6 Implementando insertAdjacentHTML() usando innerHTML

```
// Este módulo define Element.insertAdjacentHTML para navegadores que não
// a suportam e também define funções portáveis de inserção de HTML que têm
// nomes mais lógicos do que insertAdjacentHTML:
//      Insert.before(), Insert.after(), Insert.atStart(), Insert.atEnd()
var Insert = (function() {
    // Se os elementos têm uma insertAdjacentHTML nativa, a utiliza em quatro
    // funções de inserção HTML com nomes mais sensatos.
    if (document.createElement("div").insertAdjacentHTML) {
        return {
            before: function(e,h) {e.insertAdjacentHTML("beforebegin",h);},
            after: function(e,h) {e.insertAdjacentHTML("afterend",h);},
            atStart: function(e,h) {e.insertAdjacentHTML("afterbegin",h);},
            atEnd: function(e,h) {e.insertAdjacentHTML("beforeend",h);}
        };
    }

    // Caso contrário, não temos nenhuma insertAdjacentHTML nativa. Implementa as mesmas
    // quatro funções de inserção e, então, as utiliza para definir insertAdjacentHTML.

    // Primeiramente, define um método utilitário que recebe uma string HTML e retorna
    // um DocumentFragment contendo a representação analisada desse HTML.
```

```
    function fragment(html) {
        var elt = document.createElement("div");          // Cria elemento vazio
        var frag = document.createDocumentFragment();     // Cria fragmento vazio
        elt.innerHTML = html;                    // Configura o conteúdo do elemento
        while(elt.firstChild)                    // Move todos os nós
            frag.appendChild(elt.firstChild);    // de elt para frag
        return frag;                             // E retorna o frag
    }

    var Insert = {
        before: function(elt, html) {
            elt.parentNode.insertBefore(fragment(html), elt);
        },
        after: function(elt, html) {
            elt.parentNode.insertBefore(fragment(html),elt.nextSibling);
        },
        atStart: function(elt, html) {
            elt.insertBefore(fragment(html), elt.firstChild);
        },
        atEnd: function(elt, html) { elt.appendChild(fragment(html)); }
    };

    // Agora implementa insertAdjacentHTML com base nas funções anteriores
    Element.prototype.insertAdjacentHTML = function(pos, html) {
        switch(pos.toLowerCase()) {
        case "beforebegin": return Insert.before(this, html);
        case "afterend": return Insert.after(this, html);
        case "afterbegin": return Insert.atStart(this, html);
        case "beforeend": return Insert.atEnd(this, html);
        }
    };
    return Insert;   // Finalmente, retorna as quatro funções de inserção
}());
```

15.7 Exemplo: gerando um sumário

O Exemplo 15-7 mostra como criar um sumário para um documento dinamicamente. Ele demonstra muitos dos conceitos de script de documento descritos nas seções anteriores: seleção de elementos, como percorrer documentos, configuração de atributos do elemento, configuração da propriedade `innerHTML` e criação de novos nós e sua inserção no documento. O exemplo tem muitos comentários e você não deverá ter problemas para acompanhar o código.

Exemplo 15-7 Um sumário gerado automaticamente

```
/**
 * TOC.js: cria um sumário para um documento.
 *
 * Este módulo registra uma função anônima que é executada automaticamente
 * quando o documento termina de carregar. Ao ser executada, a função primeiramente
 * procura um elemento no documento com a identificação "TOC". Se esse
 * elemento não existir, cria um no início do documento.
 *
```