CHAPTER 5

Building the Ultimate Ajax Developer's Toolbox

As an experienced Web application developer, you're likely adept at applying a particular server-side technology (or, perhaps, applying several server-side technologies) to build Web applications. The past few years have seen a large push to make sever-side software development easier and more robust, while the client side has been mostly ignored. The advent of Ajax techniques has changed that, as developers now have a larger client-side toolbox with which to work. You may not be used to working with large amounts of HTML, JavaScript, and CSS, but implementing Ajax techniques will force you to do so. This chapter introduces you to some tools and techniques that will help make developing Ajax applications a little bit easier. This chapter is not an in-depth tutorial but rather provides a jump start on a number of useful tools and techniques.

Documenting JavaScript Code with JSDoc

JavaScript, like many other programming languages, suffers from a basic flaw in the average software developer's psyche: it is often easier to write (or rewrite) a certain piece of functionality than it is to read some existing code and figure out how it works. Properly adding comments to code while writing the code can greatly reduce the amount of time and effort required by other developers to understand how the code works, especially when it comes time to modify the functionality of the code.

The Java language was introduced with a tool called `javadoc`. This tool produces API documentation in HTML format from documentation comments in the source code. Any Web browser can easily read the resulting HTML, and since it's rendered as HTML, it can be distributed online, which provides developers with easy access to it. Providing the API documentation in an easily browsable format often eliminates the need for developers to inspect source code to figure out how a certain class or method behaves and how it should be used.

JSDoc is a similar tool for JavaScript (`jsdoc.sourceforge.net`). JSDoc is an open-source tool that is licensed under the GNU Public License (GPL). JSDoc is written in Perl, meaning that Windows users will have to install a Perl runtime environment. (Perl is a standard part of most Linux and Unix operating systems.)

Installation

To use JSDoc, Windows users must install a Perl environment such as ActivePerl (`www.activeperl.com`). You must also install a nonstandard Perl module named `HTML::Template` (`www.cpan.org`). The JSDoc project page provides instructions and help for those who need further assistance.

JSDoc is distributed as a gzipped tarball. To install JSDoc, simply download the tarball from the JSDoc project page and unpack it to the desired directory. You can immediately test JSDoc by going to the JSDoc directory and entering the following command:

```
perl jsdoc.pl test.js
```

JSDoc sends the resulting HTML files to a directory named `js_docs_out`. Open the `index.html` file located in this folder to browse the documentation generated from the `test.js` file.

Usage

Now that you've gotten this far, you can investigate how to use JSDoc to document your JavaScript code. Table 5-1 outlines the special JSDoc tags that create the HTML documentation. The tags will seem familiar to anyone used to writing `javadoc` comments in Java code. Each comment block to be included in the generated documentation must start with `/**` and end with `*/`.

Table 5-1. *JSDoc Command Attributes*

Command Name	Description
`@param` `@argument`	Describes a function parameter by specifying the parameter name and description.
`@return` `@returns`	Describes the return value of the function.
`@author`	Indicates the author of the code.
`@deprecated`	Indicates that a function is deprecated and may be removed from future versions of the code. You should avoid using this particular piece of code.
`@see`	Creates an HTML link to the description of the specified class.
`@version`	Specifies the release version.
`@requires`	Creates an HTML link to the specified class that is required for this class.
`@throws` `@exception`	Describes the type of exception that a function may throw.
`{@link}`	Creates an HTML link to the specified class. This is similar to `@see` but can be embedded inside comment text.
`@author`	Indicates the author of the code.
`@fileoverview`	A special tag that when used in the first block of documentation in a file, specifies that the rest of the documentation block will be used to provide an overview of the file.
`@class`	Provides information about the class and is used in the constructor's documentation.
`@constructor`	Identifies a function as the constructor for a class.
`@type`	Indicates the return type of a function.

Command Name	Description
`@extends`	Indicates that a class subclasses another class. JSDoc can often detect this information on its own, but in some instances using this tag is required.
`@private`	Signifies that a class or function is private. Private classes and functions will not be available in the HTML documentation unless JSDoc is run with the `--private` command-line option.
`@final`	Indicates that a value is a constant value. Keep in mind that JavaScript can't actually enforce a value as being constant.
`@ignore`	JSDoc ignores functions that are labeled with this tag.

The JSDoc distribution includes a file named `test.js` that is a good reference example for how to use JSDoc. Recall that this is the documentation file that was created when you first tested your JSDoc installation. You can refer to this file if you have any questions regarding how to use JSDoc tags.

Listing 5-1 outlines a short example of JSDoc usage. The `jsDocExample.js` file defines two classes, `Person` and `Employee`. The `Person` class has one property, `name`, and one method, `getName`. The `Employee` class inherits from `Person` and adds the `title` and `salary` properties in addition to the `getDescription` method.

Listing 5-1. `jsDocExample.js`

```
/**
 * @fileoverview This file is an example of how JSDoc can be used to document
 * JavaScript.
 *
 * @author Ryan Asleson
 * @version 1.0
 */

/**
 * Construct a new Person class.
 * @class This class represents an instance of a Person.
 * @constructor
 * @param {String} name The name of the Person.
 * @return A new instance of a Person.
*/
function Person(name) {
    /**
     * The Person's name
     * @type String
    */
    this.name = name;
```

```
    /**
     * Return the Person's name. This function is assigned in the class
     * constructor rather than using the prototype keyword.
     * @returns The Person's name
     * @type String
    */
    this.getName = function() {
        return name;
    }
}

/**
 * Construct a new Employee class.
 * @extends Person
 * @class This class represents an instance of an Employee.
 * @constructor
 * @return A new instance of a Person.
*/
function Employee(name, title, salary) {
    this.name = name;

    /**
     * The Employee's title
     * @type String
    */
    this.title = title;

    /**
     * The Employee's salary
     * @type int
    */
    this.salary = salary;
}

/* Employee extends Person */
Employee.prototype = new Person();

/**
 * An example of function assignment using the prototype keyword.
 * This method returns a String representation of the Employee's data.
 * @returns The Employee's name, title, and salary
 * @type String
*/
Employee.prototype.getDescription = function() {
    return this.name + " - "
        + this.title + " - "
        + "$" + this.salary;
}
```

While not as complete an example as the `test.js` file included in the JSDoc distribution, this example shows the most common usages of JSDoc (see Figure 5-1). The `@fileoverview` tag gives an overview of the `jsDocExample.js` file. The `@class` tag describes the two classes, and the `@constructor` tag flags the appropriate functions as object constructors. The `@param` tag describes a function's input parameters, and the `@returns` and `@type` tags describe the function's return value. These are the tags you're likely to use most often and the ones that will prove most useful to other developers browsing the documentation.

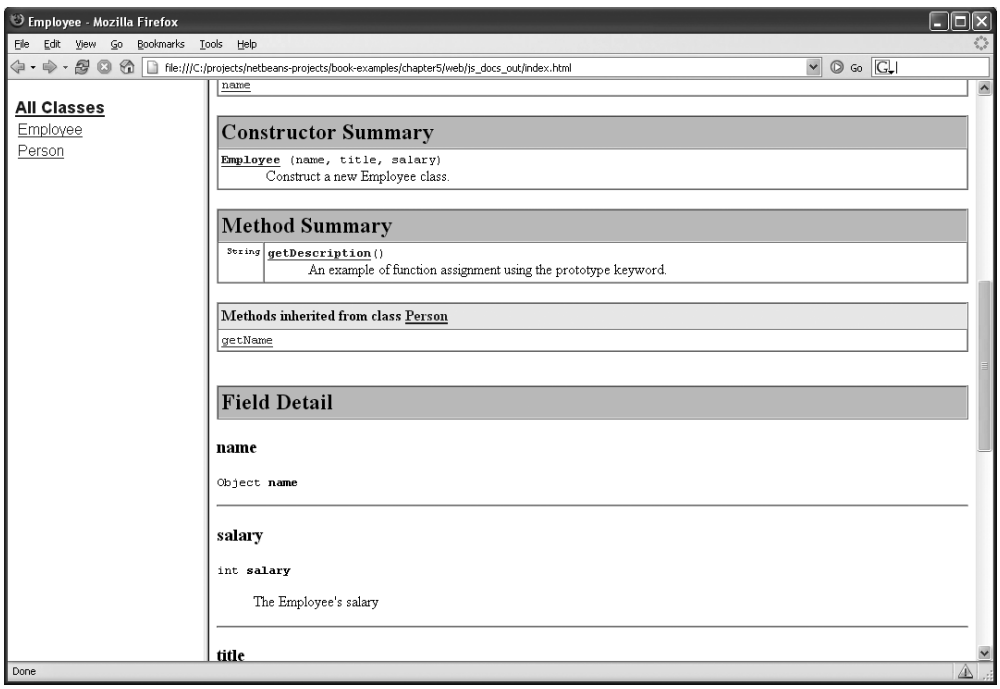

Figure 5-1. *The documentation produced from the* `jsDocExample.js` *file by JSDoc*

Validating HTML Content with Firefox Extensions

Today's modern browsers do a good job of implementing the standard W3C DOM. Authors can count on nearly universal browser support as long as they create content that follows standard HTML or XHTML.

Often that is easier said than done. Unlike compiled languages such as C++ or Java, HTML does not have a compiler that translates the human-readable code into machine-readable binary code. It's the role of the Web browser to interpret the human-readable HTML or XHTML code into an internal representation of the DOM and render the content appropriately on-screen.

The browser wars of the late 1990s saw browser makers such as Microsoft and Netscape adding proprietary HTML tags in an effort to gain market share. This, along with HTML's lack of a strict compiler, has led to massive amounts of nonstandard Web pages. Today's modern browsers, while supporting the latest in W3C standards, also attempt to be as forgiving as

possible to the poorly written HTML page. Most browsers work by having two rendering modes, based on the doctype of the HTML page (if it's available): strict and quirks. Web browsers use a *strict* mode when the doctype indicates that a Web page is written to follow a certain W3C recommendation, such as HTML 4.1 or XHTML 1.0. Web browsers use a *quirks* mode when a doctype is not available or when the page has a number of conflicts with the specified doctype.

As a developer, you should strive to create pages that adhere to a certain W3C standard. Doing so not only makes your Web pages accessible to all modern Web browsers but also makes your own life easier by ensuring that the browser can create an accurate representation of the DOM from the HTML code. The browser may not be able to create an accurate representation of the DOM if the page is poorly written, forcing the browser into rendering the page using a quirks mode. An incorrect representation of the DOM may make it difficult to access and modify the DOM via JavaScript, especially in a cross-browser way.

Since HTML does not have a strict compiler, how can you ensure that the HTML code you write adheres to W3C standards? Fortunately, a couple of extensions are available for the Firefox Web browser that make it easy to validate your Web pages.

HTML Validator

HTML Validator[1] is a Firefox extension that finds and flags errors on an HTML page. HTML Validator is based on Tidy, a tool originally developed by the W3C to validate HTML code. HTML Validator embeds the Tidy tool into Firefox and allows the source code of a page to be validated locally within the browser without sending the code off to a third party.

Tidy finds HTML errors and classifies them into three categories:

- *Errors*: Problems that Tidy cannot fix or understand
- *Warnings*: Errors that Tidy can fix automatically
- *Accessibility warnings*: HTML warnings for the three priority levels defined by the W3C Web Accessibility Initiative (WAI)

HTML Validator displays the status of the page and the number of errors in the lower-right corner of the browser, providing fast feedback during the development cycle (see Figure 5-2).

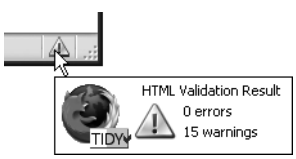

Figure 5-2. *HTML Validator summarizes the errors on a page using an icon on the status bar.*

HTML Validator provides even more help when you view the source of a Web page by accessing the View ➤ Page Source menu item. The Firefox view-source window opens as normal, but with HTML Validator enabled, the window includes two new panes (see Figure 5-3). The HTML Errors and Warnings pane lists all the errors found on the page. Clicking any of the items in the list jumps the main source window to the location of the problem in the HTML source. The Help pane fully describes the problem and offers suggestions on how you can fix the problem.

1. https://addons.mozilla.org/extensions/moreinfo.php?application=firefox&category=Developer%20Tools&numpg=10&id=249

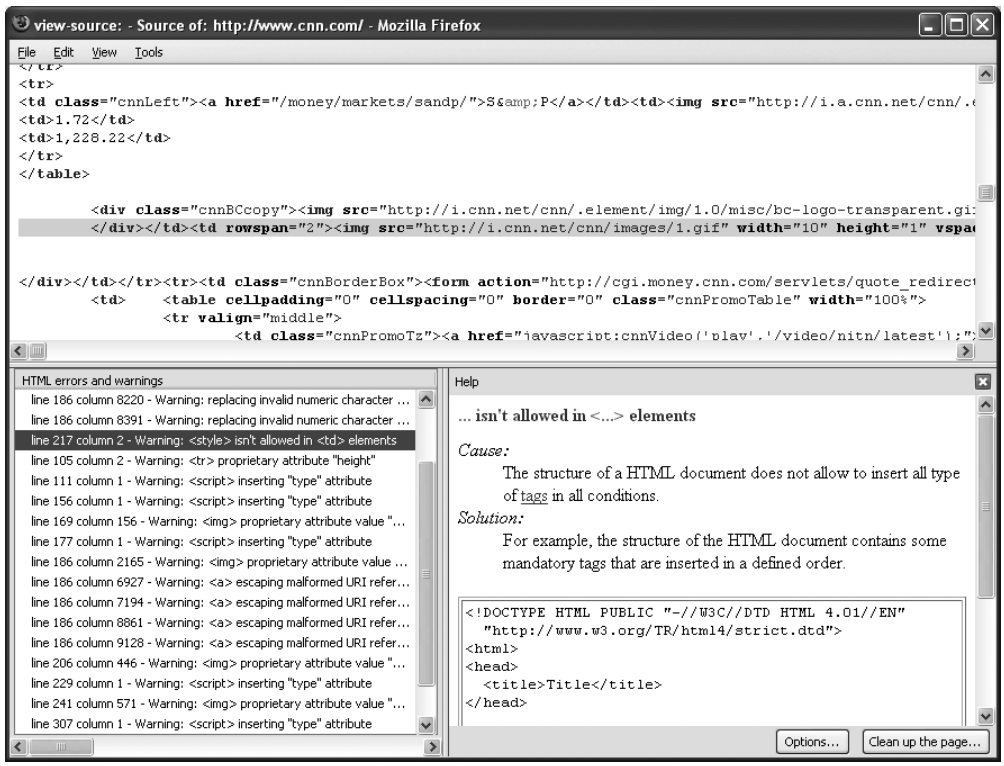

Figure 5-3. *HTML Validator lists the errors in the HTML source code and suggests ways to fix the problem when viewing a page's source.*

The bottom of the Firefox view-source window includes a Clean Up the Page button. This button launches a window that further helps you fix any errors on the page (see Figure 5-4). The Cleanup the Page window opens with four tabs along the top: Cleaned Html, Original Html, Cleaned Browser, and Original Browser.

The Cleaned Html tab is the most useful for Web developers. This tab lists the source code of the page after you put the page through HTML Validator for fixing. HTML Validator will do its best to automatically fix all the errors on the page, and the output is listed on this tab. The Original Html tab lists the source code of the page in its original form, before it was processed by HTML Validator.

Sometimes fixing the HTML errors on a page will change the way in which the browser renders the page, which may or may not be a desirable effect. The Cleaned Browser tab shows how the page will now look using the fixed source code provided by HTML Validator, while the Original Browser tab displays the page using the original source code.

In sum, HTML Validator is a powerful tool that can help you clean up your HTML and make it comply with W3C standards and recommendations. Unfortunately, HTML Validator is available only for Windows platforms. Luckily, another Firefox extension provides similar functionality to HTML Validator and is available on all platforms.

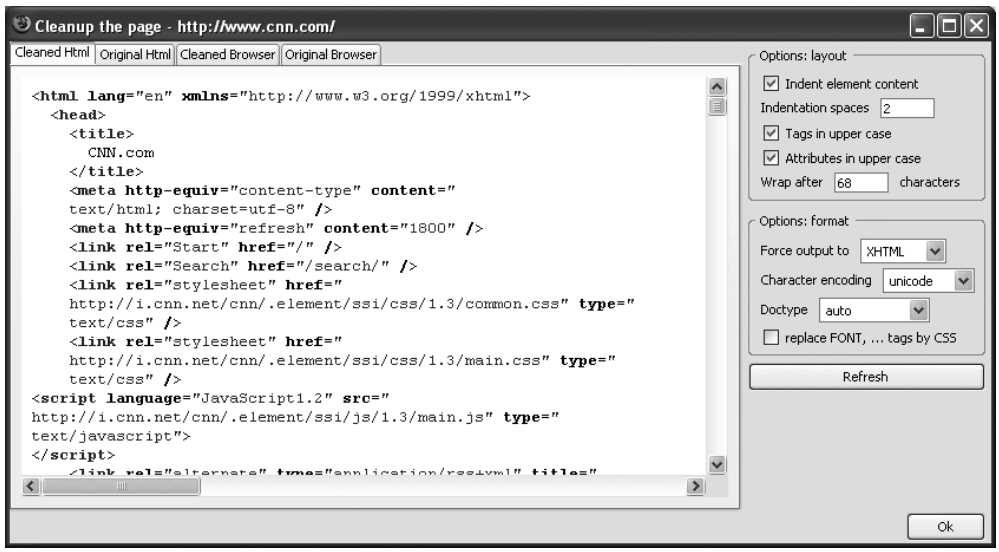

Figure 5-4. *The Cleanup the Page dialog box of HTML Validator will suggest new source code that fixes the errors found in the original source HTML.*

Checky

Checky[2] is another Firefox extension that helps developers write better HTML pages. Unlike HTML Validator, which validates the source code locally, Checky sends the page source off to various third-party sites to perform the HTML validation.

You can access Checky by right-clicking any page in Firefox and selecting the Checky menu item (see Figure 5-5). The Checky menu item contains several subitems that perform various tasks. The HTML/XHTML menu item lists several sites that offer HTML validation services. Clicking any of the sites in the list opens a new tab in Firefox that points to the validation site. Checky automatically fills in the address of the page to validate and starts the validation process.

As you can see in Figure 5-6, the code to be validated needs to be publicly available on the Internet so the validation site can access the HTML.

Checky also provides access to sites that validate more than just HTML. The Links menu lists sites that will validate all the links on a page, ensuring that all the links connect to existing URLs. The CSS menu lists sites that will validate any of the CSS files used on the page to ensure that they follow standard CSS rules.

Take the time to test some of the validation sites provided by Checky. Using these validation tools will make your code more standards compliant and will reduce the time spent manually attempting to track down problems or issues.

2. https://addons.mozilla.org/extensions/moreinfo.php?application=firefox&category=Developer%20Tools&numpg=10&id=165

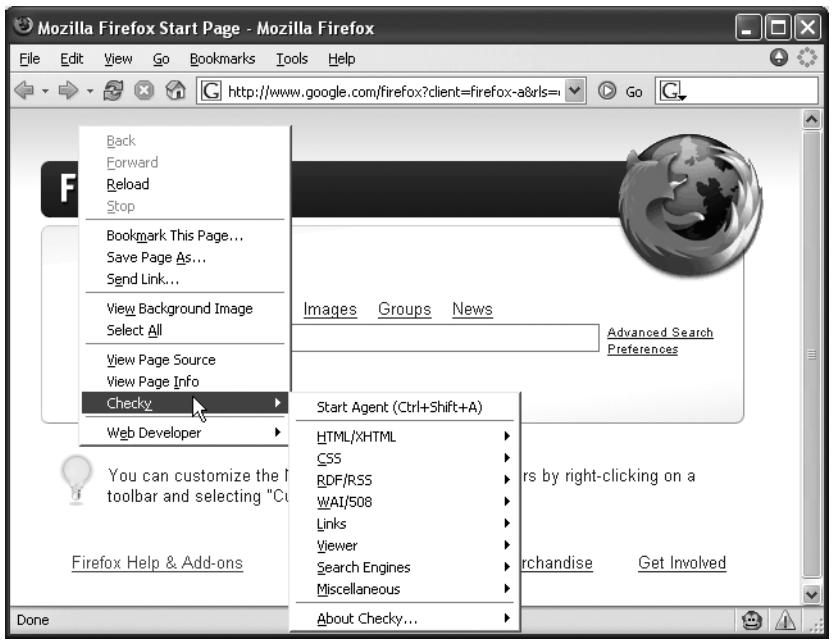

Figure 5-5. *You can access Checky via a context menu in Firefox.*

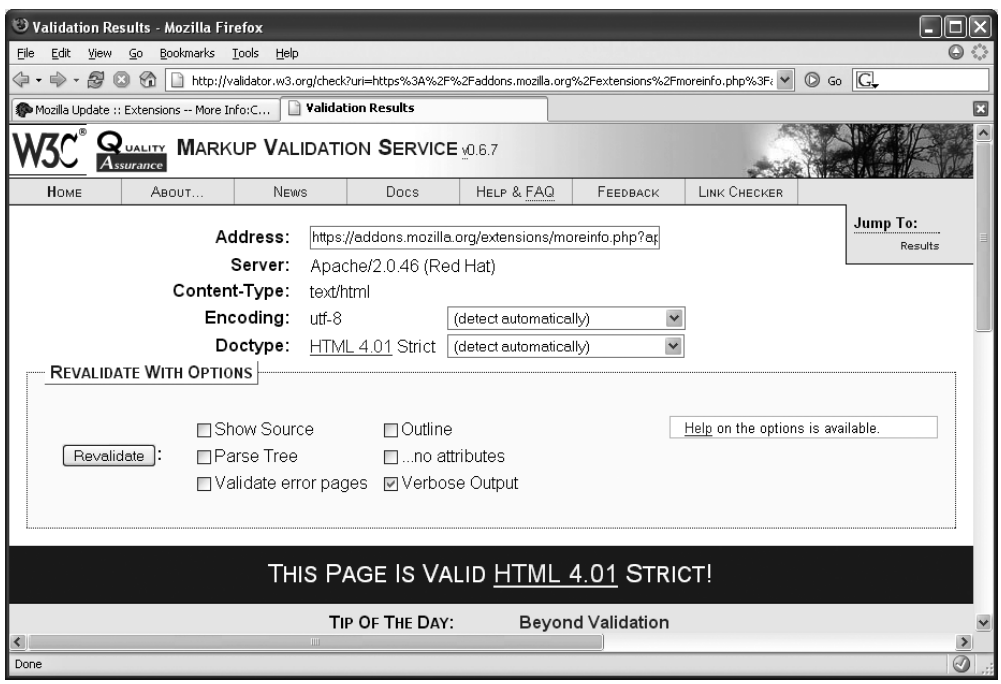

Figure 5-6. *The results of HTML validation using the W3C's online validator, accessed using Checky*

Searching for Nodes Using DOM Inspector

DOM Inspector is a tool packaged with the Mozilla Suite and Firefox browser. DOM Inspector allows you to view a structured representation of a Web page and even gives you the ability to search for specific nodes and dynamically update nodes in the DOM. In Firefox, you can access DOM Inspector via the Tools menu item. To inspect a Web page using DOM Inspector, enter the desired URL into the textbox, and click the Inspect box; alternatively, select a window from the File ➤ Inspect a Window menu, which lists the Web pages currently open within the browser (see Figure 5-7).

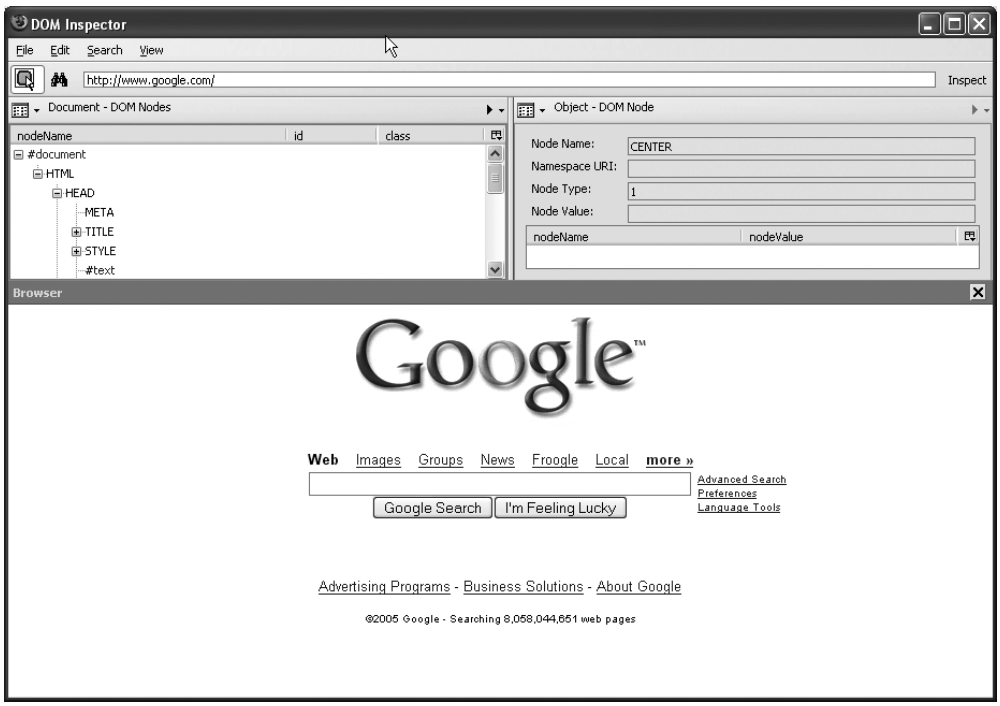

Figure 5-7. *DOM Inspector main window*

The DOM Inspector main window has three panes. The upper-left pane is a hierarchical view of the Web page's DOM. The root element is always the document itself, and from there every node in the Web page is listed. For most Web pages the root node will almost always be HTML. The upper-right pane gives detailed information regarding the node that is selected in the structured view pane. If a browser window is not open in the bottom part of the window, you can open it by selecting the View ➤ Browser menu item.

DOM Inspector is a powerful tool that allows you to quickly traverse the structure of a given Web page and view and modify the individual nodes that make up the Web page's DOM. You can normally find nodes manually by drilling down through the items in the structured view. You can also find individual nodes using the Search ➤ Find Nodes menu item. This search functionality allows you to find nodes based on the `id` attribute, tag name, or attribute name and value (see Figure 5-8).

The easiest way to find nodes in DOM Inspector is by using the mouse. To find a node in the structured view, select the Search ➤ Select Element by Click menu item, and click the item in the browser window. The selected item should blink momentarily with a red border, and its node will be selected in the structured view pane.

Figure 5-8. *DOM Inspector's Find Nodes dialog box*

Once you have selected a particular node in the structured view pane, you can start inspecting and modifying its properties. For example, you can right-click a node, select Cut from the context menu, select another node in the structured view, right-click, and then select Paste from the context menu. Doing so effectively moves the selected node from one spot in the DOM to another. Figure 5-9 displays how the main image on the Google search page was moved to another part of the page using this method.

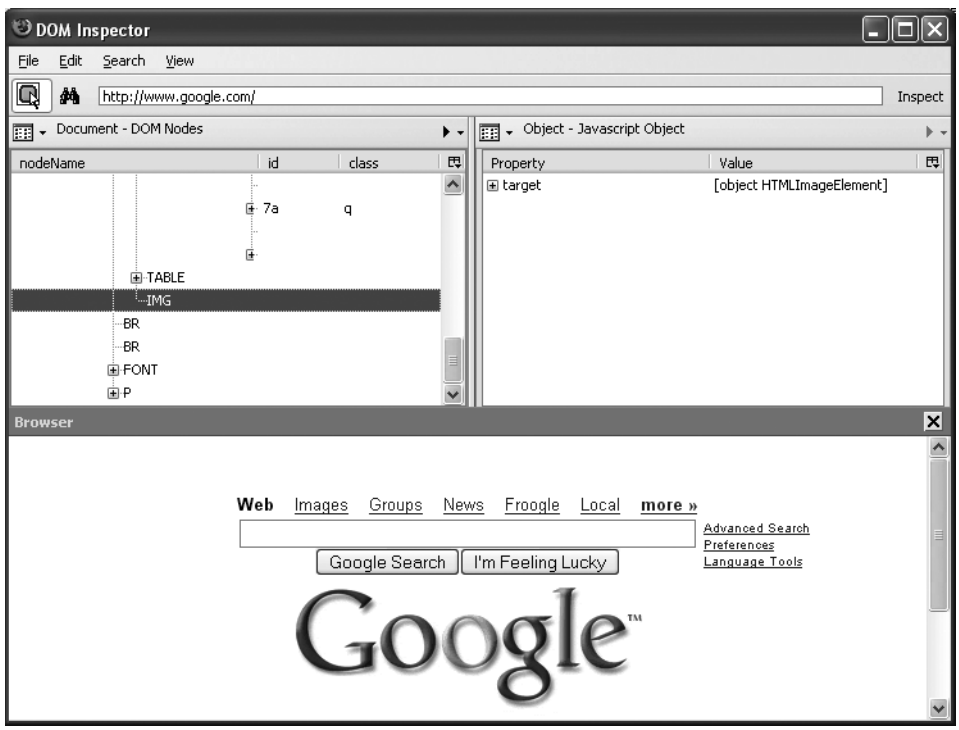

Figure 5-9. *The results of moving the main image on the Google search page using DOM Inspector*

You can find more functionality in the upper-right information pane. This window displays various types of information about the node selected in the structured view pane. You can toggle the types of information available using the drop-down list icon in the top header

area. The available types of information are DOM Node, Box Model, XBL Bindings, CSS Style Rules, Computed Style, and JavaScript Object. The Box Model and XBL Bindings information types are more useful when developing applications using Mozilla's XML User Interface Language (XUL) toolkit.

The DOM Node information type shows basic information about the node such as its tag name, its node value, and the node's attributes. Right-clicking a node displays a context menu with an Edit menu item that allows you to modify the value of a node's attribute. For instance, try it on a font node by changing the `size` attribute. Figure 5-10 shows how the size of the fonts above the input box on the Google search page were increased using this technique.

Figure 5-10. *The size of the fonts above the input box modified dynamically using DOM Inspector*

The JavaScript Object information type lists the DOM properties and methods available for the selected node. This can be a powerful feature when trying to determine what properties and methods are available for a specific DOM node. For example, in addition to a normal method such as `appendChild`, methods such as `insertRow` and `deleteRow` are listed for a table node.

Right-clicking in the information pane when it's set to the JavaScript Object information type displays a context menu with an Evaluate JavaScript menu item. Selecting this menu item launches a pop-up window that allows you to evaluate a JavaScript expression against the selected node. Figure 5-11 shows the JavaScript evaluation menu opened for the `body` node of the Google search page and shows that executing the JavaScript expression as shown in the evaluation window will append the specified text to the end of the page. Note that the target is used as the variable name and refers to the selected node, in this case the `body` element.

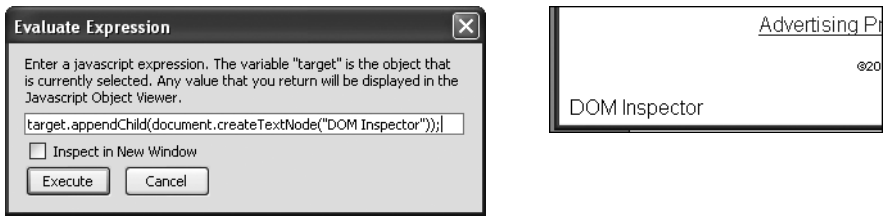

Figure 5-11. *Using the JavaScript evaluation window to dynamically add a text node to the body of the page (left) and the result in the browser pane (right)*

The CSS Style Rules and Computed Style information types display information about the selected node's style rules. The Computed Style information type lists all the style-related attributes as the browser's rendering engine sees them, including styles set explicitly using the `style` attribute, styles specified in external CSS files, or styles inherited from parent nodes.

Now that you've seen a brief overview of DOM Inspector's features, you can begin to imagine the scenarios in which it may prove to be a useful tool in your development environment. You can use DOM Inspector to inspect DOM nodes created dynamically via the `document.createElement` methods to ensure their property values are as you expect them to be. You could also use DOM Inspector to troubleshoot why a particular node doesn't have the style rules applied to it that you expect. As you become more familiar with DOM Inspector's capabilities, you'll undoubtedly find scenarios in which it will be an invaluable tool during your Web development process.

Performing JavaScript Syntax Checking with JSLint

JSLint is a JavaScript verifier (`www.jslint.com`) that scans JavaScript source code looking for problems. If JSLint finds a problem, the tool displays a message describing the problem and the approximate location of the error in the source code. JSLint flags structural problems in addition to coding-style conventions that often lead to unintended behavior or errors. JSLint does not guarantee that the logic is correct, but it does help find errors that may cause the browser's JavaScript engine to throw an error.

JSLint defines a set of coding conventions that is stricter than the language defined by the ECMA. These coding conventions are harvested from years of experience and follow this age-old rule of programming: just because you can do it doesn't mean you should. JSLint attempts to promote good JavaScript coding habits by flagging what it determines to be risky coding practices, in addition to flagging instances that are outright errors (see Figure 5-12).

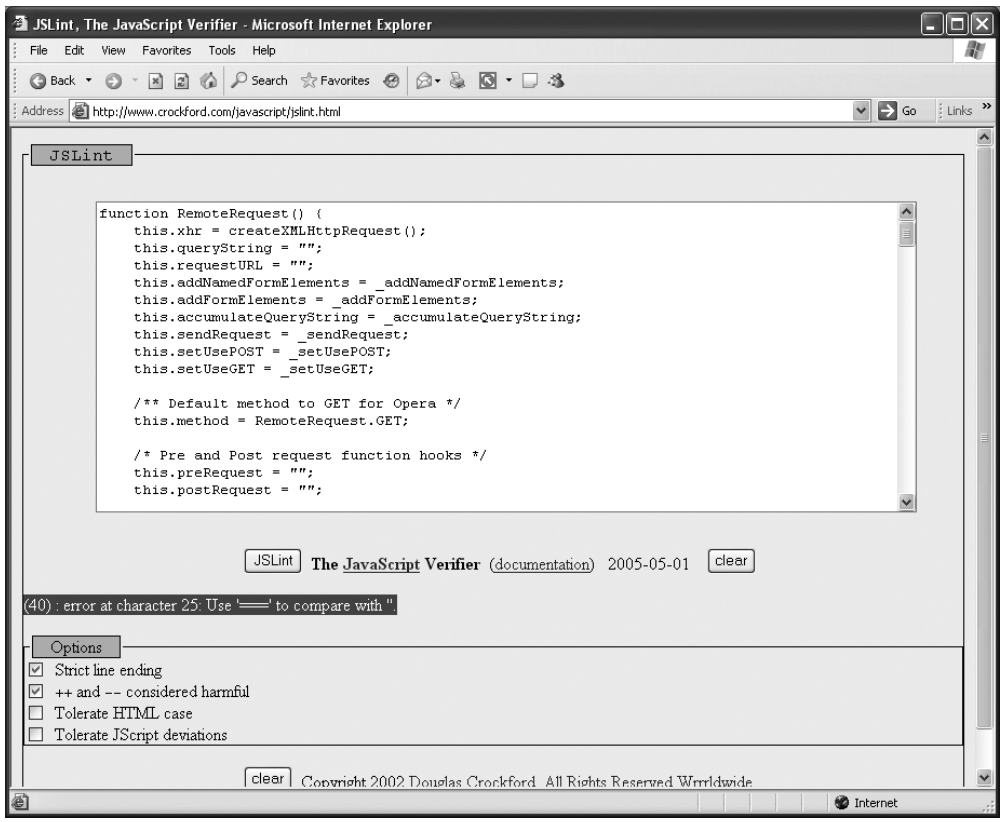

Figure 5-12. *JSLint offers JavaScript verification by checking for errors and poor coding styles.*

The following is a partial list of structural errors that JSLint will flag as being suspect coding practices. (You can find the complete list in JSLint's documentation.)

- JSLint expects that all lines end with a semicolon. While JavaScript does allow a line-feed character to act as a line termination character, this is considered ambiguous and poor coding style.
- Statements using `if` and `for` must use curly braces for the statement blocks.
- In JavaScript, unlike other programming languages, a block does not introduce variable scope. JavaScript supports only function-level scope. Thus, JSLint accepts only blocks that are part of `function`, `if`, `switch`, `while`, `for`, `do`, and `try` statements and will flag any other blocks as errors.
- A `var` may be declared only once, and it must be declared before it is used.
- JSLint flags code that comes immediately after a `return`, `break`, `continue`, or `throw` statement as unreachable code. These statements must be immediately followed by a closing curly brace.

JSLint is an especially good tool for beginning JavaScript programmers because it helps teach good JavaScript coding practices. It can reduce debugging time by flagging areas that could potentially be causing logic errors or other unintended behavior. Be sure to try JSLint if you are having trouble debugging a particular piece of JavaScript code.

Performing JavaScript Compression and Obfuscation

We all know that JavaScript is an interpreted language that executes within the client's browser. In other words, JavaScript is downloaded as plain text to the browser, which then executes the JavaScript code as needed.

The user can always read JavaScript source code by using the browser's view source functionality, which displays the complete HTML markup of the page, including any JavaScript blocks. Even if the JavaScript source is placed in an external file and referenced with the `script` tag's `src` attribute, the user can still download and read it. Because the JavaScript source is always available to anybody viewing the page, you should not place proprietary or sensitive logic algorithms in JavaScript. Such logic is best left on the server where it is more secure.

The size of JavaScript files can become an issue as JavaScript usage grows in Ajax-based applications. Because it is an interpreted language, JavaScript is never compiled to machine-level binary code, which is a more efficient storage format for executable code. A large number of JavaScript files can slow an application down because it needs to download the source from the server to the browser before it can be executed. In addition, a large set of JavaScript code will only become larger if the code is commented with a tool such as JSDoc, as described earlier.

You can probably see that JavaScript's lack of a binary executable package has two problems: poor security and large source code downloads. Is there a way around these problems?

JavaScript's increasing popularity has produced a number of tools that can help solve these problems. The simplest compression tools simply strip JavaScript source of all comments and line-feed characters, which helps reduce the size of the source code download. Removing comment lines and line-feed characters can reduce the size of a JavaScript file by 30 percent or more, depending on the situation. Note that all statements in the JavaScript source must correctly terminate with a semicolon before the source can be compressed with such tools. If this isn't the case, you'll receive errors or unintended behavior. So, before compressing your JavaScript source, be sure to use JSLint to ensure that all statements end with a semicolon!

Other tools go one step further by offering obfuscation services. *Obfuscation* is the process of scanning through source code and changing field and function names from their original names to coded, nonsensical names in an effort to prevent others from learning the intent and inner workings of the source code. Obfuscation is typically not needed for languages such as C++ that are compiled to machine-level binary instructions. Even modern languages such as Java and C#, which are compiled to intermediate bytecodes rather than binary instructions, require obfuscation tools for maximum security. JavaScript, being a completely interpreted language, is another example.

One freeware tool that offers both compression and obfuscation services is MemTronic's HTML/JavaScript Cruncher-Compressor (`hometown.aol.de/_ht_a/memtronic/`). This tool supports multiple levels of JavaScript compression. The lowest level of compression, called *crunching* in the tool's nomenclature, simply removes all comments and line-feed characters. According to

the tool's documentation, this can lead to bandwidth savings of 20–50 percent. Using the "crunch" mode, we saw the size of one JavaScript file reduced by 30 percent.

The highest level of compression, named *compressing* in the tool's vernacular, actually compresses the JavaScript source with a real compression scheme, with autodecompression added to the file. The tool claims bandwidth savings of 40–90 percent when using this mode and claims that the compressed output has been successfully tested on modern versions of Internet Explorer, Netscape, Mozilla, and Opera. We used the "compress" mode on the same JavaScript file used in the "crunch" test, and this time we saw the size of the file reduced by more than 65 percent (see Figure 5-13).

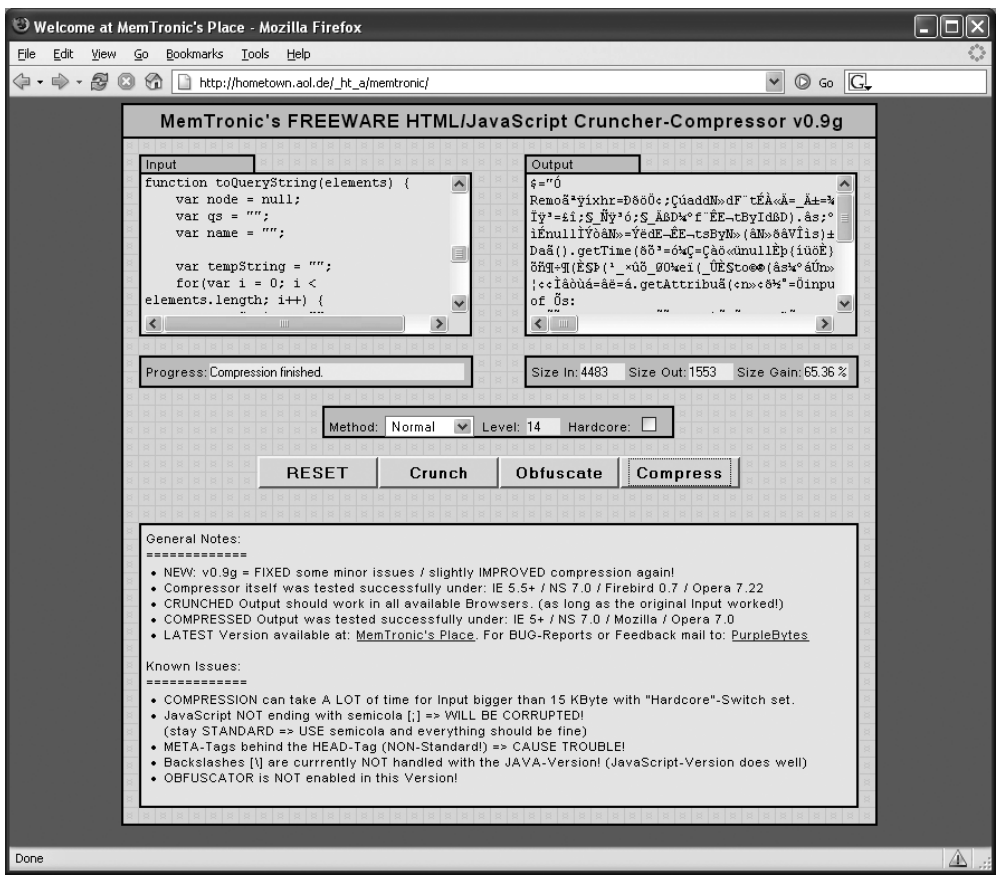

Figure 5-13. *MemTronic's HTML/JavaScript Cruncher-Compressor significantly reduces the size of JavaScript source while also making it difficult to read.*

At the time of this writing, the MemTronic tool's documentation claims that the JavaScript obfuscator is not yet complete. However, look at the output window in Figure 5-13. This is the result of a "compress" operation on a JavaScript file. The output is hardly readable and in fact contains numerous odd characters. While it may not be true obfuscation, it certainly is enough to prevent the casual user from inspecting (and possibly stealing) your JavaScript source.

Using the Web Developer Extension for Firefox

The Web Developer extension for Firefox adds a multitude of useful Web developer tools to the browser. You access the tools via a toolbar that is added to your browser once you install the extension (see Figure 5-14). The extension is available for all platforms for which Firefox is currently available, meaning it includes Windows, OS X, and Linux. The Web Developer extension for Firefox is available at `chrispederick.com/work/firefox/webdeveloper/`.

Figure 5-14. *The Web Developer extension's toolbar added to Firefox*

The Web Developer extension provides more than 80 individual tools that do everything from converting `GET` requests to `POST` requests (and vice versa) to allowing live editing of a page's CSS rules. Too many tools exist to list them individually, but the following are the general tool categories:

- The Disable menu provides the ability to disable browser functionality such as JavaScript, CSS, cookies, and animated images.
- The CSS menu contains tools related to CSS rules and style sheets.
- You can use the Forms menu to convert `GET` requests to `POST` requests (and vice versa), automatically populate form values, and remove maximum lengths from input elements.
- You can outline, hide, and outline images using functionality located in the Images menu.
- You can inspect various information relating to the page such as cookie information, link information, and response headers from the Information menu.
- The Miscellaneous menu provides tools for clearing the browser's cache, history, and session cookies, as well as zooming in or out on the page.
- You can outline tables, table cells, frames, block-level elements, and more using the Outline menu.
- The Resize menu displays the current window size in the title bar along with other tools for resizing the current window.
- You can find quick links to third-party sites for validating CSS, HTML, and download speed under the Tools menu.
- The View Source button provides easy access to viewing the page's source.
- The Options menu provides custom editing of the Web Developer extension's colors, shortcut keys, and behavior.

Some Web developers have described the tools and functionality provided by the Web Developer extension as "indispensable," "the best," and "essential." Install it to experiment with its various tools and determine whether it aids your development and debugging process.

Implementing Advanced JavaScript Techniques

We're assuming the audience of this book has at least a basic working knowledge of JavaScript. A complete tutorial on JavaScript fills a complete book itself, so we'll avoid trying to teach the language here. Instead, this section discusses some of the advanced and possibly little-known features of JavaScript and how you can incorporate them into your Ajax development.

We'll first cover a little history about JavaScript so you know where it has been and how it got here. Brendan Eich of Netscape developed JavaScript in 1995. His task was to develop a way to make Java applets more accessible to the nontechnical Web designers who created and maintained Web sites. Eich decided that a loosely typed language devoid of compilers was the appropriate choice.

Various names were attached originally to Eich's creation, but it was finally renamed to JavaScript in an effort to capitalize on Java's newfound marketing success. JavaScript swiftly became the most popular scripting language on the Web, thanks to a low barrier of entry and an ability to be copied and pasted from one page into another. Early revisions of JavaScript and the Navigator DOM gave rise to the DOM Level 0 standard, which defined form elements and images as children of their elements.

Not to be outdone, Microsoft created its own scripting language called VBScript. VBScript was functionally similar to JavaScript but had a Visual Basic–like syntax and worked only in Internet Explorer. Microsoft also supported an implementation of JavaScript (which by now had been and standardized and named ECMAScript by ECMA) as JScript. While the syntax of the various flavors of JavaScript were nearly identical, the vast differences in the implementations of the DOM among browsers made cross-browser scripts almost impossible to create. Using a lowest common denominator approach usually led to scripts that could do no more than the most trivial of tasks.

By 1998 Netscape had opened the source code for its browser and decided to rewrite the browser from scratch with a focus on closely following W3C standards. At that same time, version 5 of Internet Explorer had by far the best implementation of the W3C DOM and ECMAScript. The first complete release of the open-source Netscape code under the *Mozilla* banner came in 2002. This started a trend in the browser space: more and more browsers worked to comply with Web standards maintained by the W3C and ECMA. Today, modern browsers such as Firefox, Mozilla, Opera, Konqueror, and Safari all adhere closely to Web standards, greatly simplifying the task of writing cross-browser HTML and JavaScript. Internet Explorer 6, not much changed from the version 5 browser of 1998, exhibits the most nonstandard behavior.

Object-Oriented JavaScript via the prototype Property

JavaScript supports a form of inheritance through a linking mechanism rather than through the classical inheritance model supported by fully object-oriented languages such as Java. Each JavaScript object has a built-in property named `prototype`. The `prototype` property holds a reference to another JavaScript object that acts as the current object's parent.

An object's `prototype` property is used only whenever a function or property of the object is referenced via the dot notation method but is not found on the object. When this situation occurs, the object's `prototype` object is inspected for the requested property or function. If it doesn't exist on the object's `prototype` property, then the `prototype`'s `prototype` is examined and so forth up the chain until the requested function or property is found or the end of the

chain is reached, in which case the value `undefined` is returned. In this sense, the inheritance structure is more of a "has a" relationship than an "is a" relationship.

The `prototype` mechanism takes some getting used to for those who are accustomed to a more classically based inheritance scheme. The `prototype` mechanism is dynamic and can be configured at runtime at will, without the need for a recompile. You can add properties and functions to an object only when necessary, and you can join disparate functions together dynamically to create dynamic, highly versatile objects. The highly dynamic nature of the prototype mechanism can be both a curse and a blessing, as it can be difficult to learn and apply but powerful and robust when applied correctly.

The dynamic nature is akin to the concept of polymorphism found in classical inheritance schemes. Two objects may share the same properties and functions, but the function methods may be completely different, and the properties may hold different data types. This polymorphism allows JavaScript objects to be generically handled by other scripts and functions.

Figure 5-15 shows the prototype inheritance scheme at work. The script defines three classes of objects: `Vehicle`, `SportsCar`, and `CementTruck`. `Vehicle` is considered the base class from which the other two classes inherit. `Vehicle` defines two properties, `wheelCount` and `curbWeightInPounds`, that represent the `Vehicle`'s number of wheels and total weight, respectively. JavaScript does not support the concept of an abstract class (a class that cannot be instantiated and must be extended by other classes), so for the base `Vehicle` class the `wheelCount` defaults to 4 and `curbWeightInPounds` defaults to 3,000.

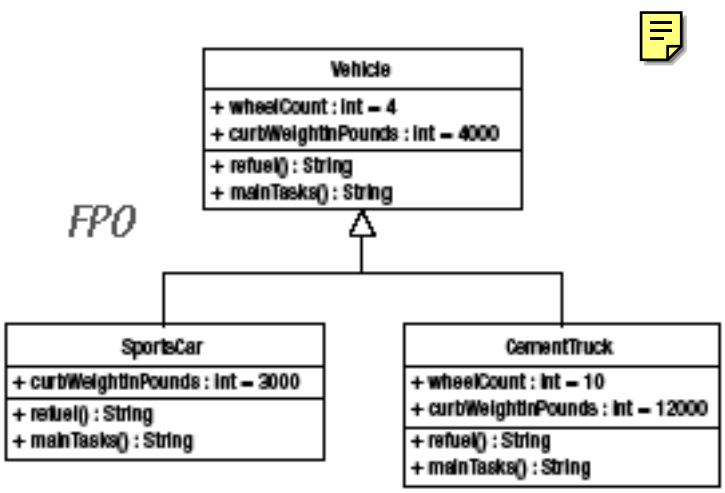

Figure 5-15. The relationships between the `Vehicle`, `SportsCar`, *and* `CementTruck` *objects*

Note how the UML diagram depicts that the `SportsCar` and `CementTruck` objects override the `refuel` and `mainTasks` functions of vehicle, because a normal `Vehicle`, a `SportsCar`, and a `CementTruck` all perform these tasks differently. A `SportsCar` has the same number of wheels as a `Vehicle`, so there is no need for `SportsCar` to override the `Vehicle`'s `wheelCount` property. A `CementTruck` has more wheels and weighs more than a `Vehicle`, so it overrides both the `wheelCount` and `curbWeightInPounds` properties.

Listing 5-2 contains the JavaScript code that defines the three classes. Pay special attention to how the `prototype` keyword attaches properties and functions to the object definition.

Also note that each object is defined by a constructor function that has the same name as the object type.

Listing 5-2. inheritanceViaPrototype.js

```
/* Constructor function for the Vehicle object */
function Vehicle() { }

/* Standard properties of a Vehicle */
Vehicle.prototype.wheelCount = 4;
Vehicle.prototype.curbWeightInPounds = 4000;

/* Function for refueling a Vehicle */
Vehicle.prototype.refuel = function() {
    return "Refueling Vehicle with regular 87 octane gasoline";
}

/* Function for performing the main tasks of a Vehicle */
Vehicle.prototype.mainTasks = function() {
    return "Driving to work, school, and the grocery store";
}


/* Constructor function for the SportsCar object */
function SportsCar() { }

/* SportsCar extends Vehicle */
SportsCar.prototype = new Vehicle();

/* SportsCar is lighter than Vehicle */
SportsCar.prototype.curbWeightInPounds = 3000;

/* SportsCar requires premium fuel */
SportsCar.prototype.refuel = function() {
    return "Refueling SportsCar with premium 94 octane gasoline";
}

/* Function for performing the main tasks of a SportsCar */
SportsCar.prototype.mainTasks = function() {
    return "Spirited driving, looking good, driving to the beach";
}


/* Constructor function for the CementTruck object */
function CementTruck() { }

/* CementTruck extends Vehicle */
CementTruck.prototype = new Vehicle();
```

```
/* CementTruck has 10 wheels and weighs 12,000 pounds*/
CementTruck.prototype.wheelCount = 10;
CementTruck.prototype.curbWeightInPounds = 12000;

/* CementTruck refuels with diesel fuel */
CementTruck.prototype.refuel = function() {
    return "Refueling CementTruck with diesel fuel";
}

/* Function for performing the main tasks of a SportsCar */
CementTruck.prototype.mainTasks = function() {
    return "Arrive at construction site, extend boom, deliver cement";
}
```

Listing 5-3 details a small Web page that demonstrates the inheritance mechanism of the three objects. The page simply contains three buttons, with each button creating one type of object (`Vehicle`, `SportsCar`, or `CementTruck`) and passing the object to the `describe` function. The `describe` function is responsible for displaying the values of each object's properties and the return values of the object's functions. Notice how the `describe` method doesn't know whether the object it's describing is a `Vehicle`, `SportsCar`, or `CementTruck`—it just assumes that the object has the appropriate properties and functions and lets the object return its own values.

Listing 5-3. `inheritanceViaPrototype.html`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>JavaScript Inheritance via Prototype</title>

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

<script type="text/javaScript">

function describe(vehicle) {
    var description = "";
    description = description + "Number of wheels: " + vehicle.wheelCount;
    description = description + "\n\nCurb Weight: " + vehicle.curbWeightInPounds;
    description = description + "\n\nRefueling Method: " + vehicle.refuel();
    description = description + "\n\nMain Tasks: " + vehicle.mainTasks();
    alert(description);
}

function createVehicle() {
    var vehicle = new Vehicle();
    describe(vehicle);
}
```

```
function createSportsCar() {
    var sportsCar = new SportsCar();
    describe(sportsCar);
}

function createCementTruck() {
    var cementTruck = new CementTruck();
    describe(cementTruck);
}
</script>
</head>

<body>
  <h1>Examples of JavaScript Inheritance via the Prototype Method</h1>

  <br/><br/>
  <button onclick="createVehicle();">Create an instance of Vehicle</button>

  <br/><br/>
  <button onclick="createSportsCar();">Create an instance of SportsCar</button>

  <br/><br/>
  <button onclick="createCementTruck();">Create an instance of CementTruck</button>

</body>
</html>
```

Figure 5-16 depicts the results when each of the three objects is created and described using the `describe` function.

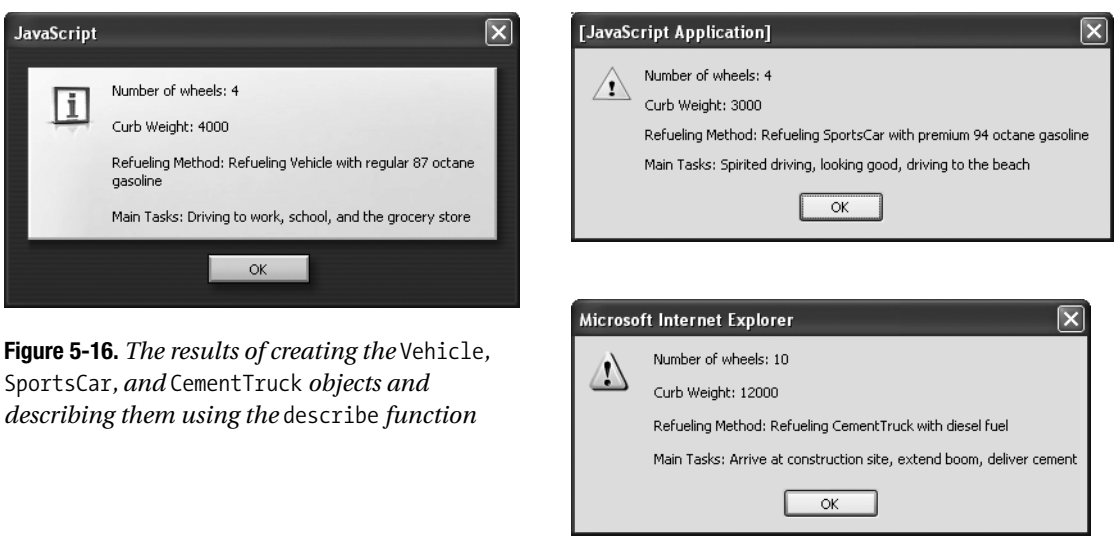

Figure 5-16. *The results of creating the* `Vehicle`, `SportsCar`, *and* `CementTruck` *objects and describing them using the* `describe` *function*

Private Properties and Information Hiding with JavaScript

The true die-hard fans of object-oriented design will notice that when using the prototype method for adding properties and functions to a JavaScript object, the added properties and functions are public and accessible to all other objects. For functions this typically isn't a problem, as most functions should be exposed to external clients anyway. But in the case of properties, the fans of object-oriented design will point out that public properties break the concept of information hiding. Object-oriented design dictates that an object's properties should be private and therefore not accessible to external clients. External clients should be able to access an object's private properties only through publicly available functions.

A little known fact about JavaScript is that it is possible to create private properties that are not accessible to external clients and instead accessible only via the object's methods. Douglas Crockford[3] has demonstrated a method of creating private properties in JavaScript. It's rather simple, as summarized here:

- Private properties are defined in the constructor function using the `var` keyword.
- Private properties can be publicly accessed only by *privileged functions*. A privileged function is a function that has been defined in the constructor using the `this` keyword. Privileged functions are accessible to external clients but also have access to the object's private properties.

Let's consider the `Vehicle` class from the earlier example. Say you want to make the `wheelCount` and `curbWeightInPounds` properties private and accessible only via publicly available methods. The new `Vehicle` object now looks like Listing 5-4.

Listing 5-4. *The Rewritten* `Vehicle` *Object*

```
function Vehicle() {
    var wheelCount = 4;
    var curbWeightInPounds = 4000;

    this.getWheelCount = function() {
        return wheelCount;
    }

    this.setWheelCount = function(count) {
        wheelCount = count;
    }

    this.getCurbWeightInPounds = function() {
        return curbWeightInPounds;
    }

    this.setCurbWeightInPounds = function(weight) {
        curbWeightInPounds = weight;
    }
```

3. http://www.crockford.com/

```
    this.refuel = function() {
        return "Refueling Vehicle with regular 87 octane gasoline";
    }

    this.mainTasks = function() {
        return "Driving to work, school, and the grocery store";
    }
}
```

Note how the `wheelCount` and `curbWeightInPounds` properties are defined within the constructor using the `var` keyword, making the properties private. The properties are no longer public, and attempting to access the value of the `wheelCount` property via dot notation, like so:

```
var numberOfWheels = vehicle.wheelCount;
```

will return `undefined` instead of the actual value of `wheelCount`.

Since the properties are now private, you need to provide publicly available functions that can access these properties. The `getWheelCount`, `setWheelCount`, `getCurbWeightInPounds`, and `setCurbWeightInPounds` functions do just that. The `Vehicle` object now satisfies the concept of information hiding by allowing access to private properties only via publicly available functions.

Classical Inheritance in JavaScript

The prototype-based inheritance scheme of JavaScript works well enough, but it's not a natural way of programming for those used to the class-based inheritance schemes in languages such as C++ and Java. For those who would rather eschew the prototype-based method of inheritance and use a more classically based approach, read on.

Bob Clary[4] of Netscape proposed a method by which an object could inherit the properties and functions from another object using a single, generic script. The script simply copies the properties and functions of the "parent" object to the "child" object. For this purpose, we'll show how to modify the script slightly so that only the properties and functions that *don't exist* on the child object are copied to the child object; doing so allows functions on the child object to override functions on the parent. The generic function for creating an inheritance relationship between two objects looks like this:

```
function createInheritance(parent, child) {
    var property;
    for(property in parent) {
        if(!child[property]) {
            child[property] = parent[property];
        }
    }
}
```

The `createInheritance` function takes two arguments, the parent object and the child. The function simply iterates through all the members of the parent object (a member being

4. http://devedge-temp.mozilla.org/toolbox/examples/2003/inheritFrom/index_en.html

either a property or a function), and if the member does not exist on the child object, it is copied to the child object.

Using the `createInheritance` function is rather trivial: first create an instance of the child object, and then use the `createInheritance` function, passing to it the child object and an instance of the parent object, like so:

```
var child = new Child();
createInheritance(new Parent(), child);
```

All the properties and methods on the parent object that don't exist on the child object will be copied to the child object.

Putting It All Together

You've now seen how private properties are possible in JavaScript and how JavaScript can support a more class-based approach to inheritance like C++ and Java. To demonstrate how it all works, we'll show how to convert the earlier example that used the `Vehicle`, `SportsCar`, and `CementTruck` objects to use the new pattern of information hiding and inheritance. Listing 5-5 lists the new object definitions.

Listing 5-5. `classicalInheritance.js`

```
function Vehicle() {
    var wheelCount = 4;
    var curbWeightInPounds = 4000;

    this.getWheelCount = function() {
        return wheelCount;
    }

    this.setWheelCount = function(count) {
        wheelCount = count;
    }

    this.getCurbWeightInPounds = function() {
        return curbWeightInPounds;
    }

    this.setCurbWeightInPounds = function(weight) {
        curbWeightInPounds = weight;
    }

    this.refuel = function() {
        return "Refueling Vehicle with regular 87 octane gasoline";
    }

    this.mainTasks = function() {
        return "Driving to work, school, and the grocery store";
    }
}
```

```
function SportsCar() {
    this.refuel = function() {
        return "Refueling SportsCar with premium 94 octane gasoline";
    }

    this.mainTasks = function() {
        return "Spirited driving, looking good, driving to the beach";
    }
}

function CementTruck() {
    this.refuel = function() {
        return "Refueling CementTruck with diesel fuel";
    }

    this.mainTasks = function() {
        return "Arrive at construction site, extend boom, deliver cement";
    }
}
```

Note how the `SportsCar` and `CementTruck` objects do not define their own `wheelCount` and `curbWeightInPounds` properties and the associated accessor `functions`, as these will be inherited from the `Vehicle` object.

As before, you need a simple HTML page to test the new objects. Listing 5-6 lists the HTML page that will test these new objects. Pay special attention to the `createInheritance` function and how it's used to create the inheritance relationships between the `Vehicle` and `SportsCar` objects and the `Vehicle` and `CementTruck` objects. Also note that the `describe` function has been modified to attempt to access the `wheelCount` and `curbWeightInPounds` properties directly. Doing so should result in a value of `undefined` being returned.

Listing 5-6. `classicalInheritance.html`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Classical Inheritance in JavaScript</title>

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

<script type="text/javaScript">
function createInheritance(parent, child) {
    var property;
    for(property in parent) {
        if(!child[property]) {
            child[property] = parent[property];
        }
    }
}
```

```
function describe(vehicle) {
    var description = "";
    description = description + "Number of wheels (via property): "
                                                          + vehicle.wheelCount;
    description = description + "\n\nNumber of wheels (via accessor): "
                                  + vehicle.getWheelCount();
    description = description + "\n\nCurb Weight (via property): "
                                  + vehicle.curbWeightInPounds;
    description = description + "\n\nCurb Weight (via accessor): "
                                 + vehicle.getCurbWeightInPounds();
    description = description + "\n\nRefueling Method: " + vehicle.refuel();
    description = description + "\n\nMain Tasks: " + vehicle.mainTasks();
    alert(description);
}

function createVehicle() {
    var vehicle = new Vehicle();
    describe(vehicle);
}

function createSportsCar() {
    var sportsCar = new SportsCar();
    createInheritance(new Vehicle(), sportsCar);
    sportsCar.setCurbWeightInPounds(3000);
    describe(sportsCar);
}

function createCementTruck() {
    var cementTruck = new CementTruck();
    createInheritance(new Vehicle(), cementTruck);
    cementTruck.setWheelCount(10);
    cementTruck.setCurbWeightInPounds(10000);
    describe(cementTruck);
}
</script>
</head>

<body>
  <h1>Examples of Classical Inheritance in JavaScript</h1>

  <br/><br/>
  <button onclick="createVehicle();">Create an instance of Vehicle</button>

  <br/><br/>
  <button onclick="createSportsCar();">Create an instance of SportsCar</button>

  <br/><br/>
  <button onclick="createCementTruck();">Create an instance of CementTruck</button>

</body>
</html>
```

Clicking each of the buttons on the page produces the results shown in Figure 5-17. Note that as expected, attempting to access the private properties directly simply returns `undefined`.

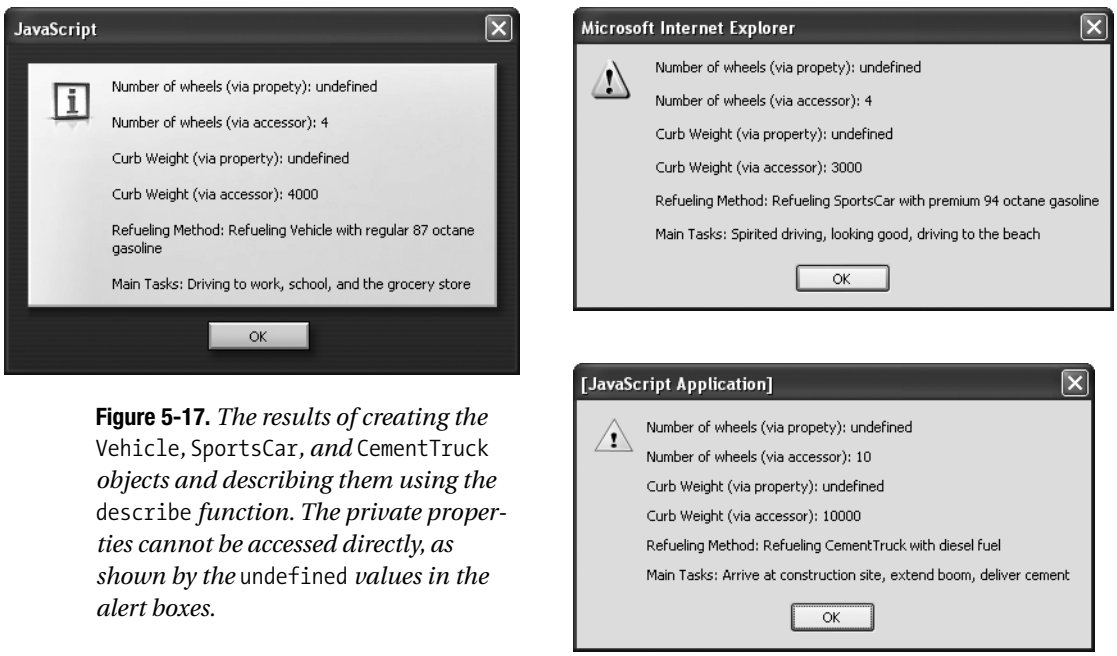

Figure 5-17. *The results of creating the* `Vehicle`, `SportsCar`, *and* `CementTruck` *objects and describing them using the* `describe` *function. The private properties cannot be accessed directly, as shown by the* `undefined` *values in the alert boxes.*

Summary

In this chapter, we introduced a number of tools and techniques that can make your development process a much more enjoyable experience. JSDoc helps you document your JavaScript code so it can be more easily understood and used by other developers. You'll surely write some of your own reusable JavaScript libraries if you start using Ajax techniques frequently, and you'll want to document the code with JSDoc to make it easy for others to use those libraries.

Tools such as HTML Validator and Checky help you ensure that the HTML code you write is valid HTML. Invalid HTML can cause undesirable behavior, so using valid HTML or XHTML eliminates one potential cause of errors. Valid XHTML or HTML is also more likely to be rendered identically across multiple browser platforms.

The DOM Inspector tool ships with Firefox and other Mozilla browsers and lets you inspect the nodes of an HTML document as a structured tree. DOM Inspector allows you to view each node, inspect its attribute values, and even change them on the fly. You can move nodes dynamically from one spot in the page to another, without rewriting the HTML. DOM Inspector is useful for inspecting nodes that have been dynamically created via JavaScript.

JSLint is a JavaScript verifier tool. While it can't determine that the logic of JavaScript is correct, it can help identify errors in the language syntax or areas that could be prone to error because of poor coding styles.

Removing comment lines and carriage return characters in JavaScript can greatly reduce the size of the JavaScript file and the time it takes to download to the client browser. MemTronic's HTML/JavaScript Cruncher-Compressor not only removes comment lines and carriage returns

but actually compresses the JavaScript code to make it download even faster. A nice side effect of the compression is that it makes the JavaScript rather difficult to read, which helps secure the inner workings of your JavaScript code from prying eyes.

The Web Developer extension for Firefox provides a number of useful tools and utilities for the active Web developer. The tools allow you to resize images, dynamically edit CSS style rules, and change form methods from `GET` to `POST` (and vice versa), just to name a few.

Finally, you saw some advanced JavaScript techniques such as object-oriented programming. First you saw how JavaScript uses a prototype-based scheme to mimic inheritance. Then you saw how JavaScript supports the concept of information hiding using private properties that are accessible only via publicly available methods. Finally, you saw a technique by which JavaScript can mimic classically based inheritance schemes, like those used by C++ and Java. This technique will likely prove to be a more natural coding style for those accustomed to fully object-oriented languages.

These tools and techniques can make your life as an Ajax developer much easier and more enjoyable. Try them all so you can pick the ones you like; also, you may come across others on the Web that you'll find useful.

Taken from the forthcoming title 'Foundations of Ajax' to be published by Apress in October 2005. ISBN 1590595823