CHAPTER 2

Development Tools

It's no secret: developing software can be rather difficult. Unlike other disciplines, software development doesn't produce a concrete, three-dimensional product that we can hold, touch, and inspect. Regardless of how much time we may spend drawing class diagrams or designing flowcharts, ultimately we're expected to write code. Reams and reams of source code are necessary to produce today's software systems. Even though we may have designed the software ourselves, and in most cases even written the source code ourselves, it doesn't take long before the amount of source code simply becomes overwhelming. Let's face it: we need help.

Fortunately, improving development tools are greatly easing the life of the software developer. Source code editors, debugging tools, and others are making it easier to build and debug today's large-scale software systems.

In this chapter we'll investigate some of the tools that can make Ajax development easier. Fortunately we'll be able to leverage a lot of the tools and techniques that you may already use in your normal day-to-day Java development.

JavaScript Source Code Editor

Many of us use some type of integrated development environment (IDE) to write Java. Yes, it is possible to write Java using a simple text editor and the command-line compiler. However, most of us benefit from the productivity gains afforded by today's modern IDEs.

If you're reading this book, you obviously write web applications using Java. Java-based web applications are made up of more than just Java source code files. Web applications built using the Java EE platform are made up of Java source code files, HTML and XHTML files, XML files, and even plain text files. Modern IDEs are capable of managing all of these files, which reduces the number of applications we need to use (and learn!) to build Java EE applications.

How many of us like having to read reams and reams of source code? Worse yet, what if it's source code that we didn't write, and thus we're not familiar with it? One handy feature of an intelligent source code editor is reserved word highlighting. It's a feature that you probably find handy but rarely think about. Reading source code in an editor that highlights the language's reserved words is much more pleasant to the eye.

Object-oriented development would be nearly impossible without today's modern IDEs. Imagine the scenario where you're writing a class that exists within an object hierarchy. You want to write a method, but you're not sure whether any of the parent classes have already defined such a method. How do you find out? The most time-consuming option is to open the source file for the parent class. You may find that the method you have in mind doesn't exist in the parent class. But does it exist in the parent class's parent? You could continually open the source files for each class in the object hierarchy. This is tedious and time-consuming.

Worse yet, what if you don't have the source code to all the classes in the object hierarchy? Suddenly you can't just browse the source code files. How about checking the Javadocs? Browsing Javadocs can be slow, too, and that's assuming that Javadocs are available.

The single greatest productivity improvement in the history of software development is arguably the intelligent code completion that is standard on nearly every Java IDE available today. If you don't know what methods or properties are available on a class, simply invoke the IDE's code-completion feature, and a list of the available methods and properties will pop up, allowing you to choose the desired one. Better yet, many editors even provide a list of all available objects and classes as part of the code completion list. Not only does this intelligent code completion reduce the amount of time spent searching through source code, but it also reduces the amount of typing we do, which in turn reduces the number of typing mistakes you're likely to make, which in turns reduces the number of failed compilations.

So why all this talk about Java IDEs? The truth is that you have likely experienced large gains in productivity by using an intelligent IDE for editing Java source code. Now that Ajax is becoming a part of your developer's toolbox, you'll likely be writing more JavaScript than you have in the past. This, of course, begs the question: are there intelligent code editors available for JavaScript?

In short, the answer is yes. JavaScript editors are not as advanced as Java editors, although they will continue to get better as Ajax and JavaScript development matures. Many editors already offer simple features like reserved word highlighting.

Today's most popular Java IDEs—Eclipse, NetBeans, and IntelliJ IDEA—all offer support for editing JavaScript source code files. At the time of this writing, IDEA supports JavaScript source files right out of the box, although JavaScript support is available as plug-ins for Eclipse and NetBeans. In this section you'll see how to install and configure the JavaScript plug-ins for Eclipse and NetBeans.

JSEclipse

One of the most advanced JavaScript editors available is JSEclipse offered by InterAKT. JSEclipse is a free plug-in for the Eclipse development environment. It offers features such as code completion, an outline window, error reporting, and code wrapping.

Installing JSEclipse is easy thanks to Eclipse's plug-in architecture. Open Eclipse's plug-in installation wizard by selecting Help ➤ Software Updates ➤ Find and Install. Select "Search for new features to install" and click Next.

The Install window should now be showing. Click the New Remote Site button and enter "InterAKT Online" into the Name field and "http://www.interaktonline.com/" into the URL field, then click OK. At this point the Install window should look something like Figure 2-1, with InterAKT Online appearing in the list.

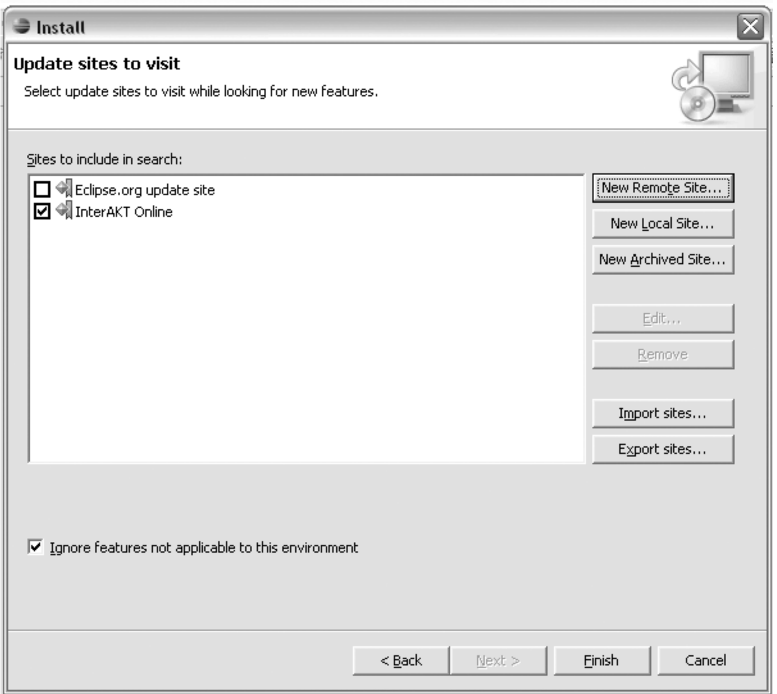


Figure 2-1. *Eclipse's Install window after adding InterAKT Online as a remote site*

Note that JSEclipse requires Java 1.5 or later. Click through the rest of the installation to finish installing JSEclipse. Restart Eclipse to ensure that the plug-in is installed completely.

With JSEclipse installed, you can now start writing JavaScript. JSEclipse identifies all files with the `.js` extension as JavaScript source files. Create a new JavaScript source file by choosing File ➤ New ➤ File and enter a file name that ends with the `.js` extension. With a new JavaScript source file now created, you're ready to start writing JavaScript.

The first thing you'll notice when editing JavaScript in JSEclipse is that it provides syntax highlighting of JavaScript keywords. Also notice that the Outline view includes an outline of the JavaScript source, showing the methods that exist on the objects defined within the source file.

Figure 2-2 shows a JavaScript source file open in JSEclipse. There are two JavaScript files in the project named `Department.js` and `Employee.js`. The `Employee.js` file is opened in the source editor. The JavaScript files have their own icons in the Navigator panel on the left side of the window, and the outline of the currently opened source file is displayed in the Outline panel on the right side of the window.

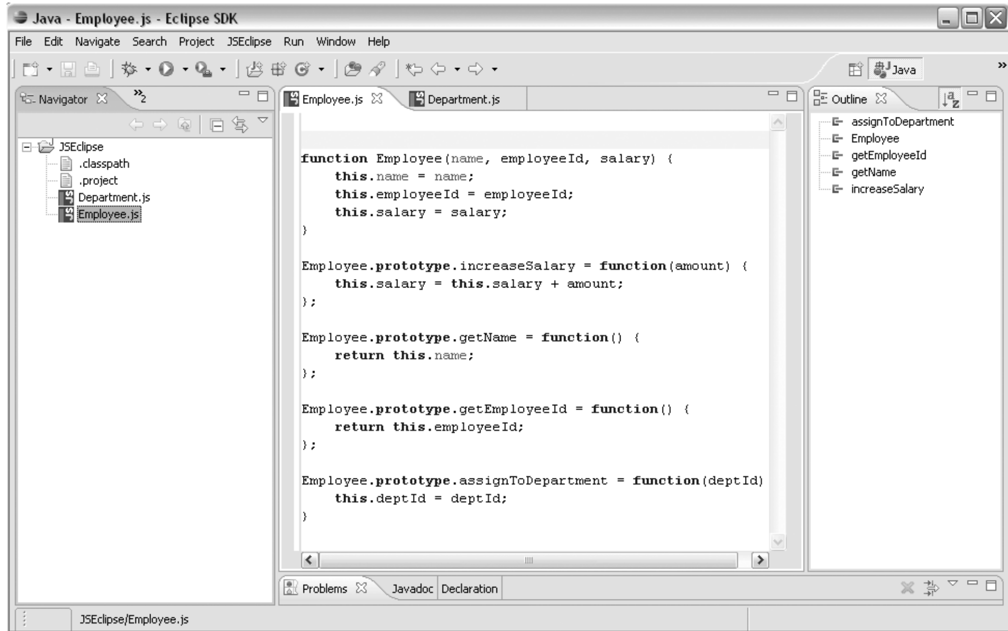


Figure 2-2. *The JSEclipse source code editor*

That's not all that JSEclipse will do. As discussed earlier, one of the greatest productivity tools of all time is intelligent code completion. JSEclipse provides intelligent code completion for user-defined JavaScript objects. Figure 2-3 demonstrates the code-completion functionality. A simple function in the `Department.js` file creates an instance of an `Employee` object. Then, the object's methods are accessed via dot notation. JSEclipse automatically shows the code completion window when the dot is typed.

JSEclipse also provides code completion for built-in JavaScript objects such as `Date`, `String`, `document`, and `window`.

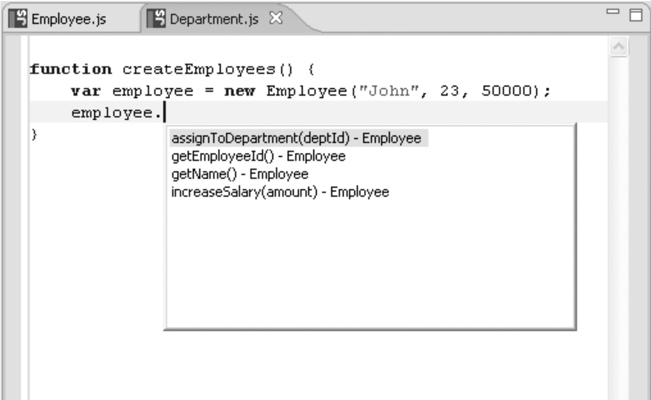


Figure 2-3. *JSEclipse provides code completion for JavaScript objects.*

JSEclipse also provides a way to extend the native code-completion functionality. You can write a simple XML file that defines the code-completion list that should appear for objects of a certain type. In fact, the default JSEclipse distribution defines the code completion for built-in JavaScript objects using XML files. Listing 2-1 lists the contents of the `object.xml` file, which defines the code completion for a base JavaScript object.

Listing 2-1. `object.xml` *Defines the Code Completion for a Basic JavaScript Object*

```
<?xml version="1.0" encoding="UTF-8"?>
<completion prefix="Object">
    <item repl="constructor" display="constructor Function" />
    <item repl="prototype" display="prototype" />
    <item repl="eval()" display="eval(String expresion)" />
    <item repl="toSource()" display="toSource() String" />
    <item repl="toString()" display="toString() String" />
    <item repl="unwatch()" display="unwatch(String prop)" />
    <item repl="valueOf()" display="valueOf() #primitive value#" />
    <item repl="watch()" display="watch(String prop, Function handler)" />
</completion>
```

The `object.xml` file is located in the `library` directory of JSEclipse's installation directory. In this directory you'll find other XML files that define the code completion for various JavaScript objects. You can modify these files to fit your needs. Better yet, you can create your own XML files that describe JavaScript files you've already built. You could build a set of XML files that provide code completion functionality for JavaScript files that are shared across your organization. The JSEclipse Help system provides a nice tutorial for creating your own code-completion libraries. You can access the tutorial by opening Eclipse's

Help menu and searching for the "JSEclipse extending the code completion" section and selecting the appropriate result.

JSEclipse is a powerful JavaScript editor that is sure to simplify your JavaScript development tasks. Development tools will continue to improve as Ajax and JavaScript usage become more mainstream, but for now, JSEclipse is a fine choice for your JavaScript editor.

NetBeans JavaScript Plug-in

At the time of this writing the current version of NetBeans, version 5.0, does not provide native support for JavaScript source files. However, NetBeans user and contributor Nicolas Désy built a plug-in for NetBeans that supports editing of JavaScript source files.

The plug-in's home page is `www.liguorien.com/jseditor`. From here you can read more about the plug-in's features and download the plug-in itself.

Nicolas warns that this plug-in is only meant to be a stopgap remedy until NetBeans natively supports JavaScript source files and that there will be no further development on this plug-in. Nevertheless, this plug-in provides a nice set of features for those who work in the NetBeans environment.

Installation

There are two ways to install the NetBeans JavaScript plug-in. The first way is to register the plug-in home page as a NetBeans update center and install the plug-in from there. The second way is to manually download and install the plug-in module. The two methods are similar, and here we'll demonstrate the installation using the manually downloaded files.

Visit the plug-in home page and click on the link that points to the NetBeans JavaScript Editor binaries. This will initiate the download of the plug-in module, which has a `.nbm` file extension.

After you download the plug-in module, open NetBeans and select the Update Center menu option from the Tools menu. This opens the Update Center Wizard that will guide you through the process of installing the plug-in. Select the Install Manually Downloaded Modules radio button and click Next. In the following window, click the Add button and navigate to and select the `.nbm` file you downloaded. Click the Next button.

The next window, shown in Figure 2-4, shows the modules that will be installed. The JavaScript editor should already be selected as shown in Figure 2-4.

Click the Next button and accept the license agreement, then click Next on the download screen. On the final screen, check the Include check box and click the Finish button to complete the installation. The JavaScript editor becomes fully operational after restarting NetBeans.

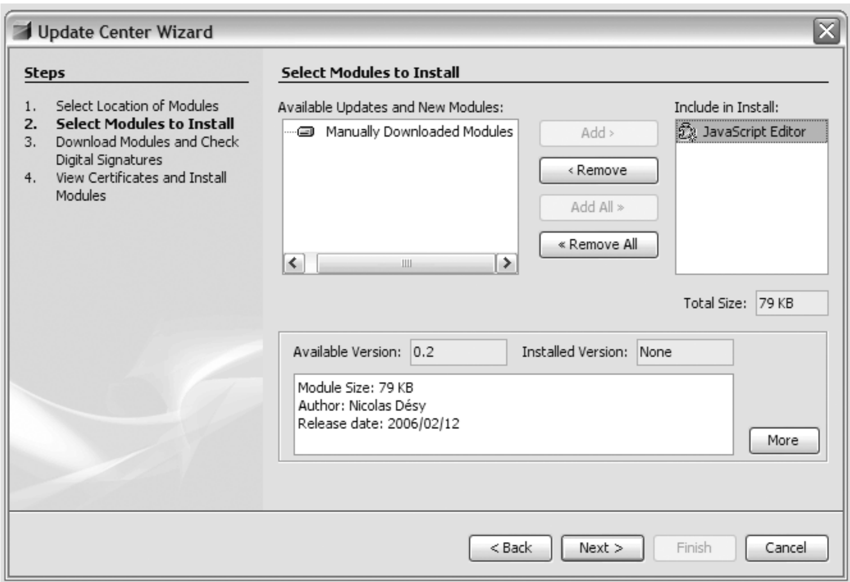


Figure 2-4. *Installing the NetBeans JavaScript Edtior plug-in*

Features and Usage

The NetBeans JavaScript Editor recognizes all files with a `.js` file extension as JavaScript source code files. You can create a new JavaScript source file by selecting New File item from the File menu and then selecting the JavaScript category. The editor provides syntax highlighting for all JavaScript keywords, highlighting of matching opening and closing curly braces and parentheses, and basic code completion for the standard DOM objects such as `document` and `window`. Code completion can be invoked with the Ctrl+Space key combination.

One of the nicest features of the JavaScript editor is that it will format JavaScript source files. How many times have you inherited a lengthy JavaScript source code file that was difficult to read due to incredibly poor formatting and indentation? With the NetBeans JavaScript Editor you can reformat an entire JavaScript source file using the Ctrl+Shift+F key combination.

How well does it work? Figure 2-5 shows a simple JavaScript file that suffers from poor form and indentation. Even with its small size the file is difficult to read and prone to errors should it ever need to be updated. Figure 2-6 shows the same file after the JavaScript editor's formatting function was invoked using the Ctrl+Shift+F key combination.

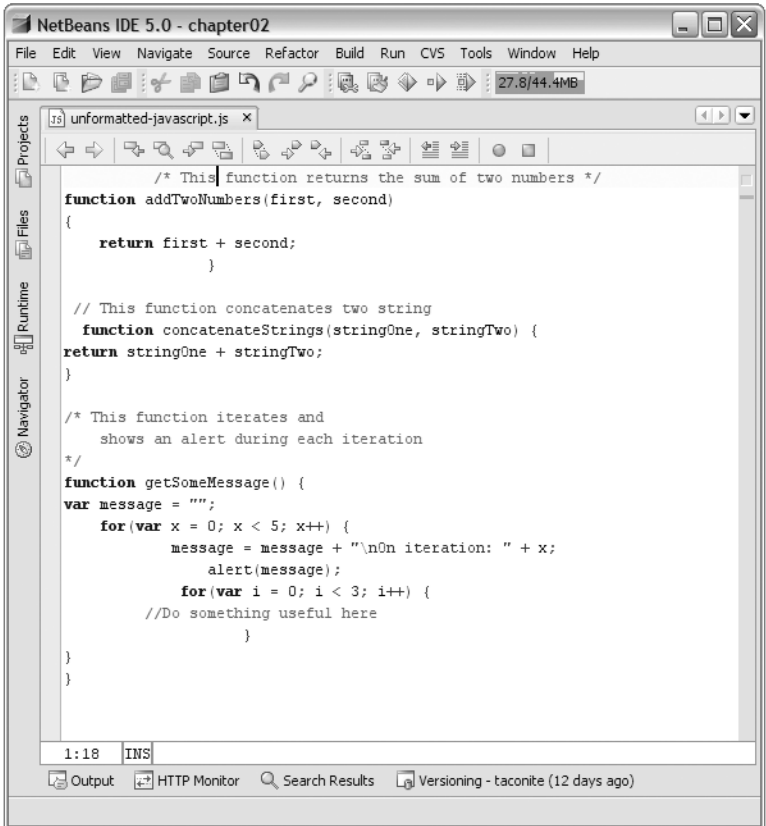


Figure 2-5. *Poorly formatted JavaScript is difficult to read and maintain.*

The NetBeans JavaScript Editor plug-in is a handy tool for those who use NetBeans for Java development. The plug-in is easy to install and possesses a nice set of features. NetBeans support for JavaScript development is sure to improve once JavaScript editing support becomes part of the standard NetBeans distribution.

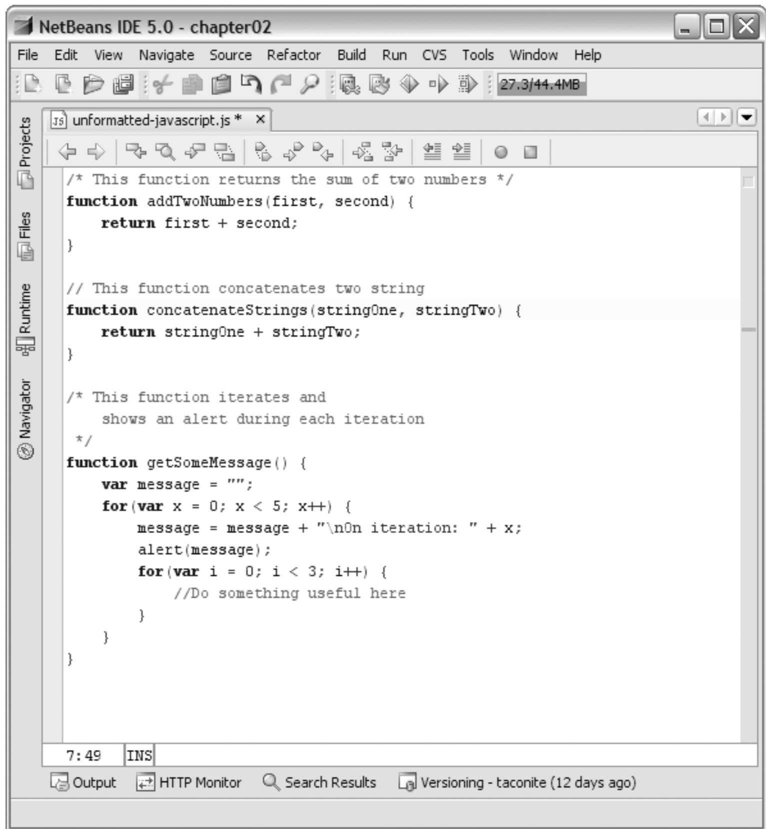


Figure 2-6. *Nicely formatted JavaScript, thanks to the NetBeans JavaScript Editor*

JavaScript Compression and Obfuscation

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

JavaScript source code can always be read by the user 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, it can still be downloaded and read by the user. Because the JavaScript source is always available to anybody viewing the page, proprietary or sensitive logic algorithms should not be placed 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 as 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 heavily commented.

We can see that there are two problems with JavaScript's lack of a binary executable package: 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. The removal of 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 the JavaScript source must correctly terminate all statements with a semicolon before it can be compressed with such tools. Failure to do so will result in errors or unintended behavior.

Remember that although JavaScript statements should always end with a semicolon, most browsers are very forgiving and allow the trailing semicolon to be absent as long as there is a line break where the semicolon should be. Many compressors eliminate line breaks, which can cause unintended results on JavaScript source files where the statements aren't properly terminated with a semicolon. JavaScript does not have a compiler to catch these sorts of issues, so instead, you should use a JavaScript verifier that ensures that all lines end with semicolons and that all statement blocks use curly braces. One fine JavaScript verifier is JSLint, found at `www.jslint.com`. Always use a JavaScript verifier such as JSLint before compressing JavaScript as it will help prevent errors caused by compressing and obfuscating noncompliant JavaScript.

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 like C++ that are compiled to machine-level binary instructions. Even modern languages like Java and C#, which are compiled to intermediate bytecodes rather than binary instructions, require obfuscation tools for maximum security.

Building a JavaScript obfuscation tool is challenging because of JavaScript's interpreted nature. JavaScript obfuscation tools can have trouble ascertaining the scope and visibility of function names, function variables, and global variables, and as a result, the compressed source code often doesn't function correctly. The best rule of thumb is to simply avoid exposing sensitive or secret algorithms in JavaScript, instead keeping them on the server where they are safely under wraps. If you insist on using a JavaScript obfuscator, be sure to thoroughly test the obfuscated JavaScript to ensure that it still functions as expected.

The Dojo Toolkit's JavaScript Compressor

One of the most impressive JavaScript compressor and obfuscator tools available today is provided by the Dojo toolkit available at `www.dojotoolkit.org/docs/compressor_system.html`. Like many JavaScript compressors, Dojo's compressor shrinks JavaScript source files by removing comments, removing spaces around operators, and replacing variable names with shorter names. But wait—replacing function and variable names sounds a lot like obfuscation. Didn't we just say that obfuscation should be avoided due to the problems it presents?

What sets the Dojo compressor apart from its peers is *how* it goes about compressing and obfuscating the JavaScript source code. Many JavaScript compressors use regular expressions to remove spaces and comment lines. Regular expressions can easily break, and they also have no context by which to determine the scope of a variable name. The Dojo compressor uses the Rhino JavaScript engine provided by the Mozilla Foundation.

Using Rhino gives the Dojo compressor the ability to determine the context and scope of a variable name. Since Rhino is a real live JavaScript interpreter, the Dojo compressor can determine the scope of a variable name and safely shorten the variable name without worry.

The main goal of the Dojo compressor is to maintain public API compatibility and ensure that the compressed script functions identically to the uncompressed script. Even though the Dojo compressor shortens variable names, it does so for the sake of size reduction, not obfuscation. The Dojo compressor's creators believe that JavaScript obfuscation is not a useful goal, but size reduction definitely is. As such, the Dojo compressor attempts to strike a nice balance between size reduction while still maintaining some readability and debuggability.

Using the Dojo Compressor

The Dojo compressor is packaged as a single JAR file named `custom_rhino.jar` that can be downloaded from the Dojo website. The Dojo compressor can be easily invoked from the command line by specifying the name of the JavaScript file to be compressed and the name of the compressed file:

```
java -jar custom_rhino.jar -c uncompressed.js > compressed.js
```

In that line, `uncompressed.js` is the name of the JavaScript file to be compressed and `compressed.js` is the name of the compressed file.

As a Java developer you probably use Ant to automate your build process, and you're probably not too keen on having to manually compress your JavaScript files each time you perform a build. Fortunately the Dojo compressor can also be used within an Ant task as shown in Listing 2-2.

Listing 2-2. *The Dojo JavaScript Compressor Used Within an Ant Task*

```
<target name="dojo-compress-javascript">
    <java jar="${basedir}/lib/custom_rhino.jar"
                        fork="true" output="compressed-javascript.js">
        <arg value="-c" />
        <arg value="uncompressed-javascript" />
    </java>
</target>
```

When using the JavaScript compressor within an Ant task, be sure to provide the correct path to `custom_rhino.jar`. You must also ensure that the Rhino `.jar` file is included in Ant's classpath.

By how much will the Dojo compressor reduce the size of a JavaScript source file? Using the Dojo compressor, a JavaScript source file from a leading Ajax framework was compressed. Figure 2-7 shows the size of the file before compression (left) and after compression (right).

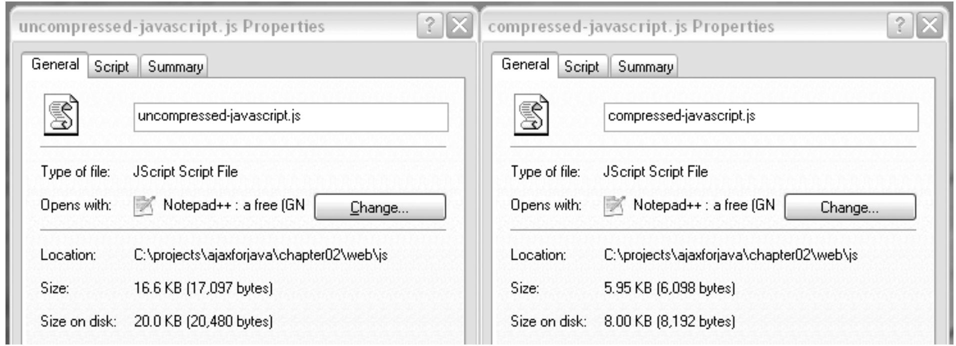


Figure 2-7. *The size of the uncompressed script (left) and the compressed script (right)*

As you can see in Figure 2-7, the uncompressed JavaScript source was 16.6KB in size. The compressed file, created using the Dojo compressor, is 5.95KB in size. That's a 64 percent reduction in size!

Inspecting a DOM Structure

Implementing Ajax in your web applications means that you'll likely have to write some JavaScript to update the DOM after making an Ajax request. You'll have to use DOM methods and properties such as `getElementById` and `childNodes` to traverse the DOM. In today's development environment it's unlikely that a single web page is built from a single HTML or JSP page. Instead, behind the scenes, there are likely several JSPs and other include

files that are brought together at runtime to generate a single web page. Although this practice increases maintainability and reusability, it makes it more difficult for the developer to conceptualize the DOM structure of the resulting web page.

You can always use your browser's View Source feature to view the resulting page's DOM structure, but this lists the page's entire source, which may not be well formatted for easy reading. There's got to be a better way to inspect the DOM structure of a web page.

Mouseover DOM Inspector

The Mouseover DOM Inspector solves this problem by providing a convenient way to inspect the DOM structure of a web page. The Inspector is implemented as a favelet (or "bookmarklet") that allows the user to inspect and modify a web page's DOM by simply mousing over the document. The Inspector can be installed by visiting `www.slayeroffice.com/tools/modi/v2.0/modi_help.html`. The Mouseover DOM Inspector supports Firefox, Mozilla, Netscape 8, Opera 7.5 and above, and Internet Explorer 6 and above.

You activate the Inspector by clicking on the Inspector's bookmark. Once activated, moving the mouse over the web page will open the Inspector window, as shown in Figure 2-8.

Figure 2-8. *Inspecting the DOM using the Mouseover DOM Inspector*

Figure 2-8 shows the Mouseover DOM Inspector in action. In this example, the mouse is hovering over a paragraph on the page, indicated by the solid red outline. The Inspector

window is shown in the upper-left corner of the screen. The window indicates that you're looking at a `<p>` element by displaying the tag name at the top of the window.

The first section lists all of the node's attributes; in this case, the `style` attribute is listed. In this particular instance, the XHTML source for this node doesn't explicitly define a `style` attribute. Instead, the style is applied from an external style sheet. Knowing the styles that are being applied to a node from an external style sheet is an extremely powerful tool that can aid in debugging or otherwise understanding the DOM structure. The second section lists the dimensions of the current node.

The third and fourth sections are key to understanding the structure of the DOM relative to the currently selected node. These sections, respectively, list the parent nodes and child nodes of the currently selected node. The Inspector lists all of the parent nodes, starting with the root `<html>` node and working down to the current node's direct parent. In this example, the only child of the current node is the paragraph text.

The Mouseover DOM Inspector provides basic DOM manipulation functionality through a combination of the mouse and keyboard. Pressing the A key while hovering over a node clones the node, then hovering over a different node and pressing S appends it to the node. The H key hides the current node, and J shows all elements hidden using the H command. Nodes are removed using the R key. Finally, the T key starts at the top node of the document and steps through each node in the DOM, including nondisplay elements such as `<meta>`. Use the Esc key to close the Mouseover DOM Inspector window. Other commands can be found on the Inspector's home page.

What good are these DOM manipulation methods, other than for having some fun? One potential use of these methods is rapid prototyping. Consider, for instance, that you're halfway through a development iteration and you're showing your latest work to your business customer. Your customer sees a page you've built and decides that a certain section of text is no longer necessary and that another section should be moved to the bottom of the page. Instead of recoding the JSP (or JSPs) and redeploying the application, you can simply modify the page using the Mouseover DOM Inspector and receive immediate feedback from your customer.

Debugging Ajax Requests

Ajax is, by definition, an asynchronous request sent from the browser to the server. This means that as soon as the Ajax request is initiated, the browser continues evaluating the script that initiated the Ajax request. The script does not block and wait for the server's response before continuing to execute. Doing so prevents the browser from locking up and allows it to continue responding to user input while the browser waits for the server to respond. The upshot of all this is that the application feels much faster and more responsive to the user.

The downside of the asynchronous nature of Ajax is that it makes debugging more difficult. Consider the scenario where you're building a form that takes user input, such as a name and telephone number. The user clicks a button to submit the form elements to the server via Ajax, and the response is displayed at the bottom of the page.

You've built the page, written the JavaScript, built a servlet to handle the Ajax request, and deployed the application for testing. You fill in the form, click the button, and ... nothing happens. What went wrong? Did the servlet fail to process the request properly? Did the JavaScript fail somewhere along the way? Was the request even sent in the first place?

As Java EE developers we're used to having powerful tools and IDEs at our disposal. We can use our favorite IDE to start the application in debug mode and step through the servlet to see if any errors occur there. However, that works only if we know that the request was sent in the first place. Furthermore, how do we know that the correct query parameters were sent? Once an error condition is found, wouldn't it be nice to be able to replay the scenario to verify that the error has been fixed?

A convenient tool would be to use some sort of HTTP sniffer. Such a tool would monitor all of the HTTP traffic to and from a specific port. A sniffer would allow us to inspect the communication between the browser and the server and determine whether there are any errors.

NetBeans HTTP Monitor

A handy tool for debugging Ajax requests is the NetBeans HTTP Monitor. NetBeans IDE is a full-featured Java IDE available from `www.netbeans.org`. An HTTP sniffer called the HTTP Monitor comes standard with every installation of NetBeans. The HTTP Monitor is integrated into NetBeans and the Apache Tomcat servlet container that ships with NetBeans. HTTP Monitor can be installed on any servlet container such as JBoss or Jetty, and the instructions to do so can be found in NetBean's built-in help system.

HTTP Monitor, when enabled, automatically records all of the requests that are handled by the servlet container. It records information such as request parameters, cookies, session attributes, and HTTP headers. The information is saved until explicitly deleted by the user or until the IDE closes. A powerful feature is that saved requests can be reopened, modified, and resubmitted to the servlet container. Figure 2-9 shows the HTTP Monitor main panel.

As you can see in Figure 2-9, the left side of the HTTP Monitor main window displays the recorded requests. You can right-click on a record and select Save to permanently store a record. You can select a record to display its information in a series of tabs on the right side of the panel. The tabs are self-explanatory; for example, the Request tab displays data about the request and any query string parameters, and the Cookies tab lists data about any cookies that were sent as part of the request or returned along with the response.

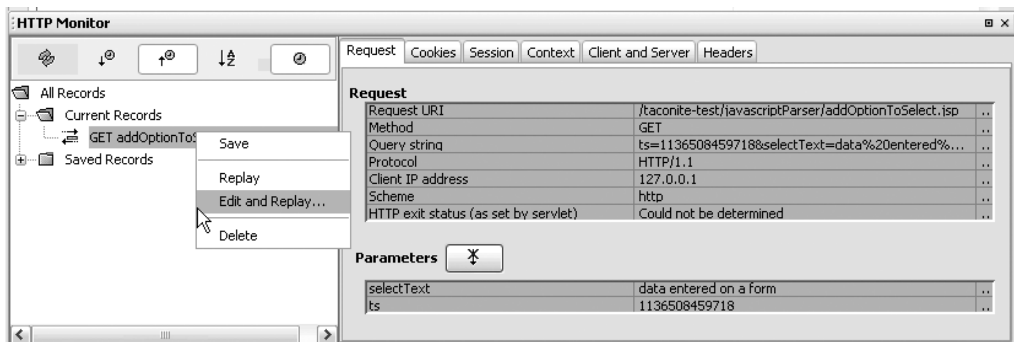


Figure 2-9. *HTTP Monitor main panel*

As you inspect the information on the tabs, you'll see how useful this tool could potentially be when debugging Ajax requests. The Parameters section on the Request tab contains the most useful information. Here you can verify that the correct parameters were sent from the browser to the server without having to debug through the servlet.

The most powerful and unique feature of the HTTP Monitor is the ability to modify a stored request and resubmit the request to the server. Right-click on the desired record and select Edit and Replay. This will open the Edit and Replay window, shown in Figure 2-10.

Figure 2-10. *Edit and Replay window*

The Edit and Replay window allows you to modify the information making up a stored HTTP request and resubmit the data to the web container. As shown in Figure 2-10, you can add, modify, and delete query parameters. You can also add or remove cookies to be sent along with the request and modify the HTTP headers that are part of the request. You can then click the Send HTTP Request button to submit the request to the server. NetBeans will open your system's web browser to display the server's response.

The NetBeans HTTP Monitor is a powerful tool that can aid in debugging Ajax requests (or any kind of HTTP request, for that matter). It is integrated into the NetBeans IDE and can be installed on any servlet container. With it, you can record the requests sent to the container, inspect the properties and data of the request, and even edit the request and resubmit it to the servlet container. Consider adding it to your developer toolbox.

Firefox FireBug Extension

FireBug is an extension for Firefox described by its developer as a combination of the JavaScript console, DOM Inspector, and a command-line JavaScript interpreter. Better yet, FireBug includes a feature called XMLHttpRequest Spy that allows you to see the requests and responses used by the `XMLHttpRequest` object. In this section you'll get a quick introduction to FireBug's most useful features. FireBug's home page is located at `https://addons.mozilla.org/extensions/moreinfo.php?id=1843&application=firefox`.

After installing FireBug and restarting Firefox, the first thing you'll notice is that FireBug opens as a window pane at the bottom of the Firefox window. The FireBug pane's visibility can be toggled using the F12 key.

The top of the FireBug window is the menu bar from which you can access FireBug's functionality. The first button on the menu bar is the Inspect Element button. Like the Mouseover DOM Inspector, this tool displays various DOM information about the selected element. To use it, just click the Inspect Element button and then click on the desired element in the browser window. Once selected, you'll be able to view the XML markup that defines the element, the CSS rules that apply to the element, the CSS box model attributes of the element, and the JavaScript properties and methods that apply to the element. Figure 2-11 shows the CSS box model information supplied by FireBug for the Firefox icon that appears in the upper-right corner of the page.

The standard Firefox error console is a useful web development tool. The error console shows all JavaScript and CSS errors that occur on a page, and most of the JavaScript error messages are descriptive and helpful. The downside is that the error console logs *all* JavaScript and CSS errors in one place, which can cause the console window to become cluttered and difficult to read.

Fortunately FireBug solves this problem by extending the standard error console's functionality to include the ability to filter errors by type and by the originating domain. The third button on FireBug's menu bar is the Errors button. Clicking the Errors button shows a drop-down list from which you can toggle the various filters. The available filters are shown in Figure 2-12.

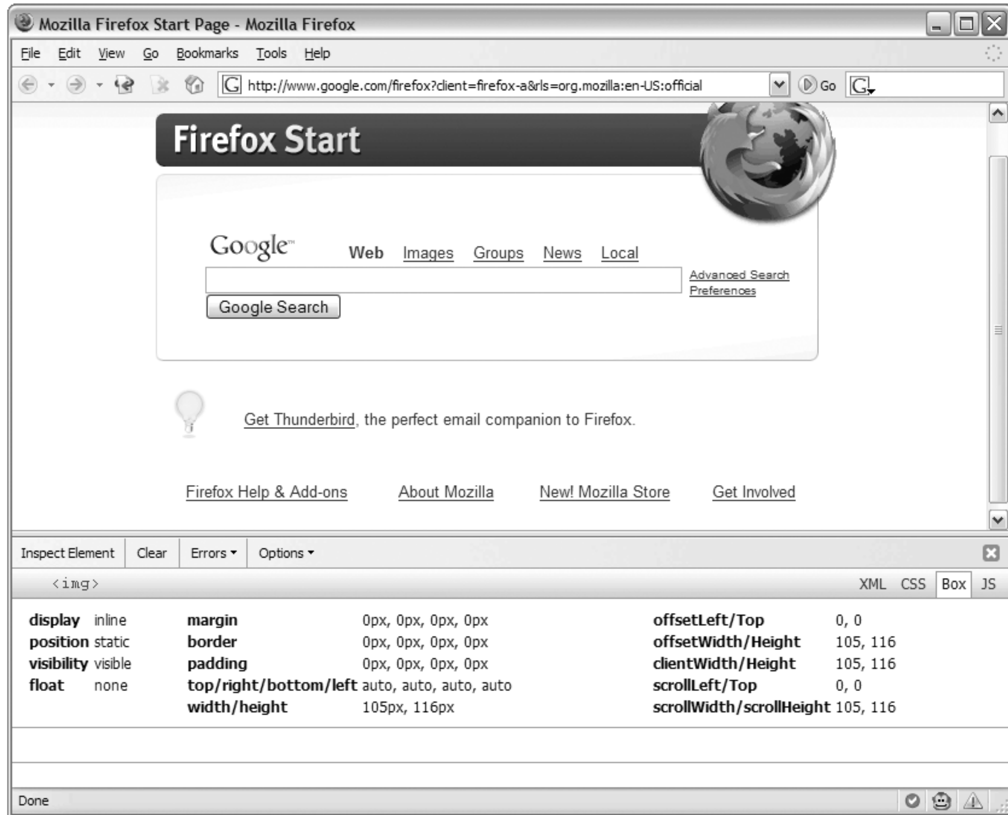


Figure 2-11. *The Inspect Element feature provides a host of information about any element on the page.*

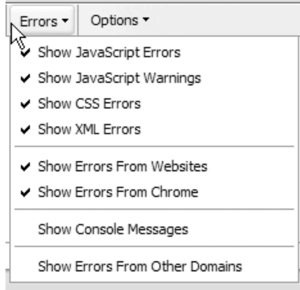


Figure 2-12. *FireBug lets you filter the errors by type and originating domain.*

The most useful feature of FireBug for Ajax developers is the XMLHttpRequest Spy feature. This feature inspects all Ajax requests and the associated server responses. `XMLHttpRequest` is

disabled by default and must be enabled by clicking the Options button and toggling the Show XMLHttpRequests item.

Once you've enabled XMLHttpRequest Spy, every Ajax request made using `XMLHttpRequest` is logged by FireBug. You can click the Post button to view any POST parameters that were sent as part of the Ajax request. This can be useful when you need to ensure that the correct information was sent to the server as part of the Ajax request.

Clicking the Response button will display the Ajax response that the server sent. FireBug will show the text of the response regardless of whether it's plain text, XML, or even JavaScript Object Notation (JSON). An example of the server response captured by FireBug is shown in Figure 2-13.

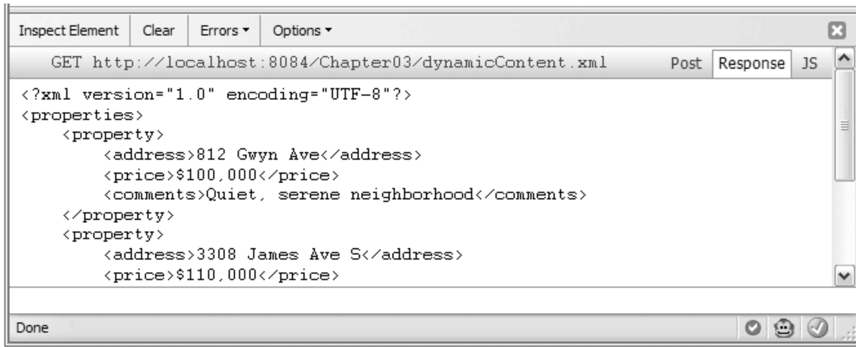


Figure 2-13. *FireBug captures the server response from an Ajax request.*

The last option provided by XMLHttpRequest Spy is the JavaScript option, which you can access by clicking the JS button. Here you will see the JavaScript method and property values of the `XMLHttpRequest` object that sent the Ajax request.

FireBug is a powerful addition to your web development toolbox that can help you rapidly diagnose problems or better understand the structure and CSS rules of a web page. The best feature of FireBug is its ability to capture the request, response, and JavaScript information pertaining to an Ajax request made using the `XMLHttpRequest` object.

JavaScript Logging

As a Java developer you're almost surely familiar with a tool called log4j. Log4j is a logging utility available from the Apache Foundation at `http://logging.apache.org/log4j/docs`. Log4j's goal is to be a powerful, easy-to-use, and unobtrusive logging facility that allows developers to instrument their code with logging statements. The logging is configurable via an external XML file allowing changes to the logging configuration without changing the application binary. Logging provides detailed context information for application failures,

which can aid in debugging, as it is sometimes difficult or impossible to debug distributed, remote, or multithreaded applications. Logging can also be used for auditing purposes.

JavaScript development has long been hampered by its lack of quality development tools and libraries. How many of us have debugged JavaScript by peppering the code with `alert` statements? Certainly this gets the job done, but the problem with `alert` statements is that they must be removed before the application is released to the general public. Adding a host of `alert` statements only to have to remove them later can be tedious and error-prone.

New tools are becoming available that bring log4j-like capabilities to JavaScript. The next few sections review these tools and give a brief overview of their use.

Log4JS

Log4JS is, as described on its website, a JavaScript logging class similar in spirit to Apache's log4j. Log4JS writes logging output to a customizable logging class. Log4JS can be found at `http://log4js.sourceforge.net`.

Four logging classes are available: `alert`, `write`, `popup`, and `console`. The `alert` class pops up logging messages in an alert box. The `write` logger writes to a new page in the browser. The `popup` logger writes messages to a separate browser window, which is convenient for keeping track of the log messages. Finally, the `console` logger writes to Safari's console or to a pop-up window if the browser is not Safari.

Log4JS is similar to log4j in that it defines different logging thresholds. The logging thresholds for Log4JS are, in increasing order, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`, and `NONE`. A logger will only log messages that are of equal or greater value to the logger's threshold. For example, if a logger is created with a threshold of `WARN`, then logging a message with the `info()` method will produce no output, but logging a message with the `warn()`, `error()`, or `fatal()` method will output the message.

If you've ever used log4j then using Log4JS will be rather easy. Figure 2-14 shows the simple test used to demonstrate Log4JS. It's a simple page that uses JavaScript to calculate the sum of two integers. Log4JS is used to log messages during the execution of the script. The page's source code is shown in Listing 2-3.

Listing 2-3. *log4jExample.js*

```
<!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>Log4js Example</title>
        <script type="text/javascript" src="js/log4js.js"></script>
        <script type="text/javascript" src="js/log4jsExample.js"></script>
    </head>
    <body>
        <p>
            Add Two Integers:
            <input type="text" id="addOne" size="4"/>
            +
            <input type="text" id="addTwo" size="4"/>
            =
            <input type="text" id="result" size="10" disabled="disabled"/>
            <button value="Calculate" onclick="calculateSum();">
                Calculate
            </button>
        </p>
    </body>
</html>
```

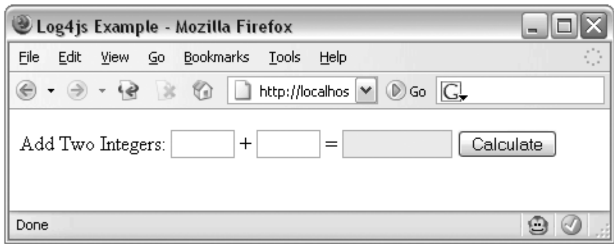


Figure 2-14. *Simple page for calculating the sum of two numbers*

The JavaScript that calculates the sum of the two integers is shown in Listing 2-4. The script starts by creating two global loggers, one with a logging threshold of `info` and the other with a threshold of `error`. The `calculateSum` function starts by retrieving the input values from the input boxes. Log4JS is used to log the input values to the pop-up logger. The script continues by attempting to convert the input values to integers. If one or both of the values is not a valid integer, then errors stating so are written to the pop-up logger and the function exits. Finally, the two integers are summed and the result is displayed on the page.

Listing 2-4. *log4jsExample.js*

```
/* Create Log objects */
var logger = new Log(Log.INFO, Log.popupLogger);
var errorLogger = new Log(Log.ERROR, Log.popupLogger);
function calculateSum() {
    /* Retrieve the user's input from the text boxes */
    var inputOne = document.getElementById("addOne").value;
    var inputTwo = document.getElementById("addTwo").value;
    /* Log the user's input */
    logger.info("first input: " + inputOne
                                        + "\nsecond input: " + inputTwo);
    /* Attempt to convert the user's input values to integers */
    var firstNumber = parseInt(inputOne);
    var secondNumber = parseInt(inputTwo);
    /* Log an error if either of the values is not an integer */
    if(isNaN(firstNumber)) {
        errorLogger.error("firstNumber is not a number: " + inputOne);
        clearResult();
        return;
    }
    if(isNaN(secondNumber)) {
        errorLogger.error("secondNumber is not a number: " + inputTwo);
        clearResult();
        return;
    }
    /* Calculate the sum and display on the page */
    var sum = firstNumber + secondNumber;
    document.getElementById("result").value = sum;
}
function clearResult() {
    document.getElementById("result").value = "";
}
```

The logging messages from this example were written to the popup logger, shown in Figure 2-15. The messages are appended to the list, so as long as you don't close the messages window you'll be able see the entire history of logged messages. This can be useful if you've made some changes to the script and want to see how the behavior of the script has changed as a result of the edits.

Figure 2-15. *Log messages written to the pop-up logger*

Log4JS also includes a facility for logging the properties of objects, which can be useful in debugging. The Log class exposes a static function called dumpObject that writes a string representation of an object to the logger:

```
logger.debug(Log.dumpObject(new Array('red','white','yellow','blue')));
```

Remember that logging can be disabled by setting a logger's logging threshold to NONE. One way to deploy Log4JS in your production application might be to define all of the application's loggers in a separate JavaScript file that is used across all pages. When deploying the application to the production environment, you could simply update that simple JavaScript file to disable all logging. Taking it one step further, you might also employ Ant's replace task to set the logging thresholds depending on the type of build being performed: development, testing, or production.

Lumberjack

Lumberjack is another JavaScript-based logging utility found at http://gleepglop.com/javascripts/logger. Like Log4JS, Lumberjack draws inspiration from Apache's log4j.

Lumberjack distinguishes itself from other JavaScript logging frameworks by the way that it displays the logged information. Lumberjack writes all the logging information to a special window whose display can be toggled by pressing Alt+D (or Cmd+D on Mac OS X). The output window is a floating window that appears at the bottom of the web page, even when the page is scrolled up and down. The output window even includes a regular expression-based filter so that only errors of the desired type are displayed. It also includes a JavaScript command-line area into which JavaScript commands can be entered.

IMPORTANT—LUMBERJACK REQUIRES THE PROTOTYPE LIBRARY

Lumberjack is dependent on the Prototype JavaScript library, which can be found at `http://prototype.conio.net`. Because of this dependency, Prototype must be listed before Lumberjack when including their respective JavaScript files onto an HTML page using the `<script>` tag. Browsers read and evaluate JavaScript in these files in the order in which they are listed on the HTML page. If Lumberjack is listed before Prototype, then errors will occur when the browser attempts to evaluate the Lumberjack script, because values from the yet unevaluated Prototype script are not found.

Instead of creating separate logging classes as you do for Log4JS, Lumberjack exposes logging methods as static methods on a `Logger` class. The `Logger` class exposes four logging methods: `info`, `debug`, `warning`, and `error`. Each method takes a single string parameter representing the message to be logged. The `Logger` class also exposes a `log` method that takes two parameters: a string representing the message to be logged and string representing a custom logging level.

Let's reuse the previous example from Log4JS to demonstrate the usage of Lumberjack. As you can see from Listing 2-5, the JavaScript is nearly identical to the script from the Log4JS example. In this instance there are no longer two globally defined loggers at the top of the script. In addition, each logging statement now uses static methods on the `Logger` class to write log messages to the output. Note how static methods of the `Logger` class are used to write log messages.

Listing 2-5. *lumberjackExample.js*

```
function calculateSum() {
    /* Retrieve the user's input from the text boxes */
    var inputOne = document.getElementById("addOne").value;
    var inputTwo = document.getElementById("addTwo").value;
    /* Log the user's input */
    Logger.info("first input: " + inputOne
                                        + "\nsecond input: " + inputTwo);
    /* Attempt to convert the user's input values to integers */
    var firstNumber = parseInt(inputOne);
    var secondNumber = parseInt(inputTwo);
    /* Log an error if either of the values is not an integer */
    if(isNaN(firstNumber)) {
        Logger.error("firstNumber is not a number: " + inputOne);
        clearResult();
        return;
    }
```

```
    if(isNaN(secondNumber)) {
        Logger.error("secondNumber is not a number: " + inputTwo);
        clearResult();
        return;
    }
    /* Calculate the sum and display on the page */
    var sum = firstNumber + secondNumber;
    document.getElementById("result").value = sum;
}
function clearResult() {
    document.getElementById("result").value = "";
}
```

Figure 2-16 shows the Lumberjack output window. Remember that the output window's view is toggled using Alt+D (or Cmd+D for Mac OS X). The same log messages written to the output window and the messages are color-coded by type, with errors being red and informational messages being white. At the top of the output window is the input box for filtering the messages. You can type "error" into the box to see only the error messages, or type "error|warning" to see only errors and warnings.

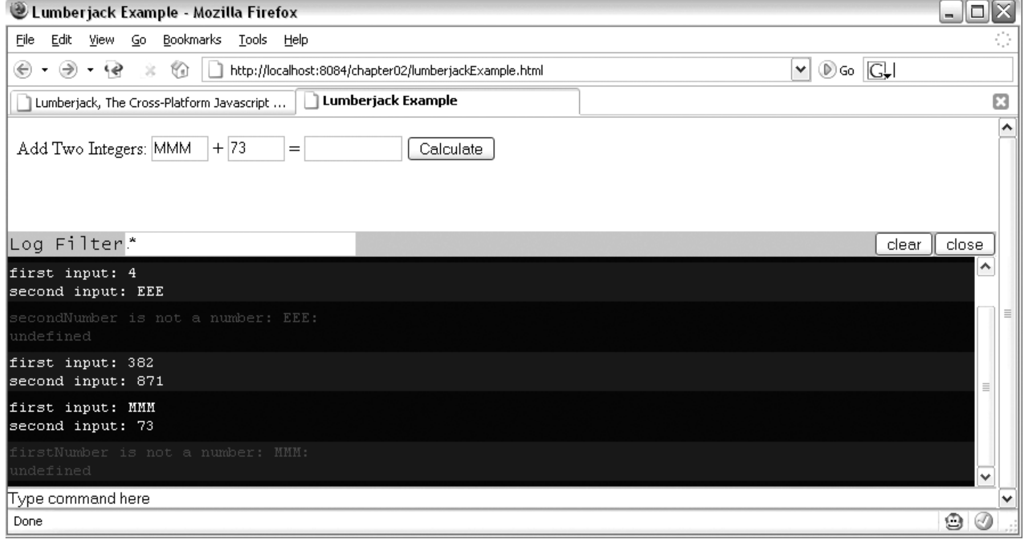


Figure 2-16. *Lumberjack writes logging messages to a floating window at the bottom of the page.*

Lumberjack provides a convenient method for inspecting JavaScript objects. The object's properties are written to the output window. To inspect a JavaScript object, open the Lumberjack output window, enter something like the following, and press Enter:

```
inspect(document.body)
```

Alternatively, you could pass a string to the inspect method that represents the ID of a DOM object on the page. For example, if you wanted to view the properties of the first input box that appears on the page in Figure 2-16, you would enter the following into Lumberjack's command line interface and press Enter:

```
inspect("addOne")
```

This will print all of the input box's properties to the output window.

JavaScript Debugging Tools

Java development has long been enhanced by a debugging architecture that makes it easy to debug through application source code step by step. Many of today's Java IDEs have fabulous debugging environments that allow you to debug standard Java SE applications, applications deployed on a local Java EE application server, and even applications deployed on a remote Java EE application server.

Traditionally JavaScript has been more difficult to debug than Java because JavaScript lacked a high-quality debugging environment. This has now changed. The Venkman JavaScript debugger is an extension for Mozilla-based browsers such as Firefox that provides a full-featured JavaScript debugging environment.

The following is meant to be a quick introduction to Venkman and its features. A more complete tutorial can be found online at `www.svendtofte.com/code/learning_venkman`.

Using Venkman

Venkman is available from `www.mozilla.org/projects/venkman/`. Venkman development started in April 2001 by Robert Ginda. Venkman is based on the Mozilla JavaScript debugging API known as js/jsd. The js/jsd API formed the basis of the Netscape JavaScript Debugger 1.1 that was available for the 4.x series of Netscape browsers.

Once you've installed it, you can start Venkman from the Tools ➤ JavaScript Debugger menu item from within Firefox. Figure 2-17 shows the default layout for Venkman.

Venkman offers a plethora of information, divided into eight panes. The default layout consists of a large pane showing the selected source code. Smaller panes are arranged vertically on the left side of the window. Venkman's command-line interface resides on the bottom of the window under the Source Code pane.

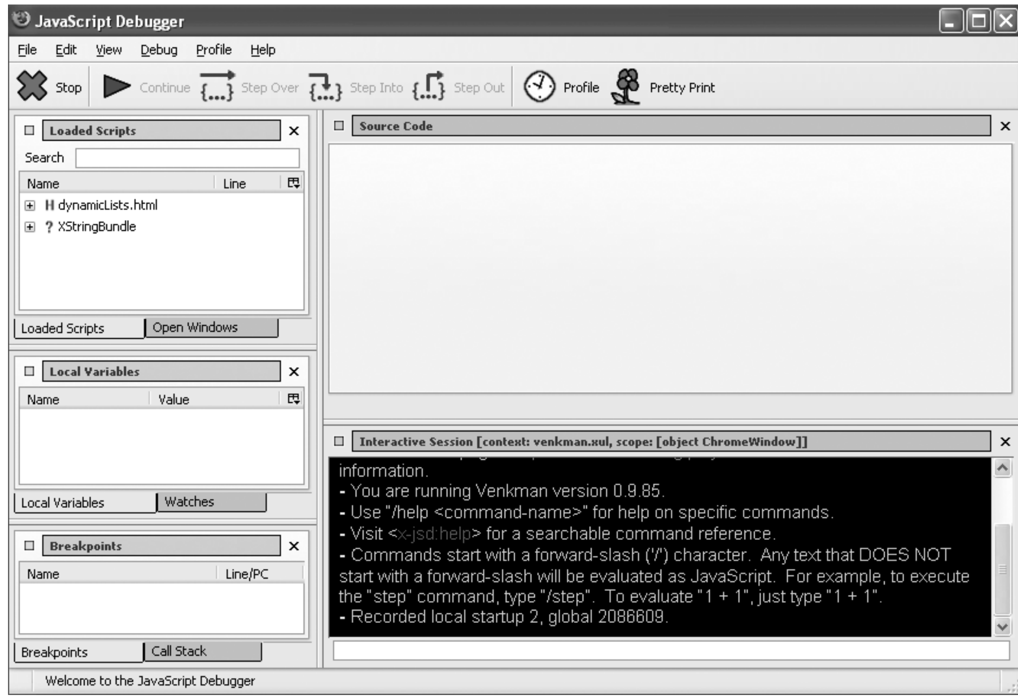


Figure 2-17. *Default window layout of Venkman*

You can drag each pane with the mouse and dock them at other locations within the main window. You can also add each pane as a separate tab to an existing pane. For example, referring to Figure 2-17, to make the Loaded Scripts tab exist within the Local Variables pane, drag and drop the Loaded Scripts tab to the Local Variables tab. You can also undock the panes from the main window by clicking the docking button located on the left side of the pane's title bar. Docking the pane back to the main window is as easy as clicking the docking button again.

As you work with Venkman, you'll get a feel for the panes that you use most often. Close the panes you don't often use by clicking the X button on the right side of the pane's title bar. You can reopen panes by selecting View ➤ Show/Hide. If at any time you want to return the pane layout to the default setting, enter "/restore-layout factory" at the command-line interface in the Interactive Session pane.

Viewing Loaded Scripts

When Venkman is open, it recognizes all the JavaScript that is available to the page currently open in the browser window. Venkman recognizes JavaScript that is embedded within the HTML page using <script> tags and also recognizes external JavaScript files that are included in the HTML page using <script src="js_file.js"> tags.

Venkman displays the currently available JavaScript in the Loaded Scripts pane. Clicking the plus sign next to each file opens a list under the file that details all the available JavaScript functions within that file along with the line number at which the function occurs within the file. Optionally, it displays the number of lines of code the function encompasses. Double-clicking a file within the Loaded Scripts pane opens the file in the Source Code pane and also scrolls directly to the function within the Source Code pane.

Right-clicking a file in the Loaded Scripts pane displays a number of options for both the file itself and the JavaScript functions contained within the file. For the file, this pop-up menu allows you to perform tasks such as disabling the debugging of eval and timeout statements, disabling the debugging of contained functions, and disabling the performance monitoring of contained functions. For individual functions, the context menu provides facilities for disabling the debugging and performance profiling.

Source Code

The Source Code pane lists the source code for the currently open file. The file could be an HTML file, XHTML file, or JavaScript file. The Source Code pane implements a tabbed theme so multiple files can be opened at once with each file residing in its own tab. The code has some simple colorization that improves readability. JavaScript keywords such as `function` and `var` have bold formatting, and string literals have a different colored font. On the left side of the pane is the line numbering for the file. The far-left side of the Source Code pane is the gutter on which debugging breakpoints can be set.

Breakpoints

Venkman supports two kinds of breakpoints, a *hard* breakpoint and a *future* breakpoint. This is a departure from most debugging environments, so we'll discuss the difference between the two.

A hard breakpoint is the type of breakpoint you're used to seeing in modern programming languages such as Java. It instructs Venkman to suspend processing at the breakpoint. Execution cannot continue until the user instructs it to do so. In Venkman, a hard breakpoint always exists *within the body of a function.*

A future breakpoint is similar to a hard breakpoint in that it instructs Venkman to suspend execution of the JavaScript at the breakpoint. The difference between the two is that a future breakpoint is set on lines that *do not exist within the body of a function.* These lines of code are executed as soon as they are loaded by the browser. By contrast, code that resides within the body of a function is not executed until the function is executed in response to some action or event.

For the most part you can simply ignore the differences between hard and future breakpoints. You'll likely use hard breakpoints in most of your work, and they should function identically to breakpoints in other debugging environments.

Venkman provides a window that lists all the currently set breakpoints. This can come in handy when you're debugging a page that has multiple breakpoints set in multiple files.

Each file in which a breakpoint is set is listed in the Breakpoints pane, and listed under each file are all that file's breakpoints.

Stepping Through Code

With breakpoints set, you're now ready to start actually debugging code. Venkman will automatically suspend execution once a breakpoint is encountered. At that point you're in control of the script's execution. You can inspect variable values, modify variable values, and continue the script execution, either by looking at one step at a time or by restarting execution and letting it run to completion.

Venkman offers developers a few options for stepping through code once a breakpoint has been encountered. Once a breakpoint has been encountered, you can choose Continue, Step Over, Step Into, or Step Out.

The Continue option restarts script execution. Execution will not end until either another breakpoint is encountered or the script completes. The Continue option is useful when you need to track down the location of a problem. You can set breakpoints at points along the execution chain and, each time a breakpoint is encountered, inspect variable values to see whether the problem has cropped up yet. Once the problem appears, you know the error occurred somewhere between the current breakpoint and the previous breakpoint, and you can narrow it down further from there. The Continue option is also useful when debugging an iteration. You can set a breakpoint at one point within the iteration and use the Continue option to speed through the iterated code, checking each time execution suspends to see whether any problems have occurred.

The Step Over function is useful when you want to avoid stepping through a function that is called by the current function. The called function may be a function that has been extensively debugged and you *just know* the problem isn't there, or you may just want to avoid stepping through its code because you're concerned about only the current function. Keep in mind that stepping over a function does not prevent it from being executed; it merely means you're not going to step through it line by line.

The Step Into option is the opposite of the Step Over function. Step Into will step into a called function so you can debug the called function. Step Over and Step Into work well together when you're trying to track down the exact location of an error.

Local Variables List

The Local Variables pane allows you to inspect and even modify variable values during script execution. The Local Variables pane always displays all the variables within scope whenever a breakpoint is encountered and execution of the script is suspended.

The Local Variables pane always has two top-level items, Scope and This. Scope refers to all the variables within the nearest current scope of execution. Because most JavaScript is written as a function, the nearest scope is usually *function* scope. For example, if a breakpoint within a function is encountered, then the Scope item within the Local Variables window will refer to all variables that are within that function's scope—namely, any variable

defined with the keyword `var` within that function. Variables defined in the global scope (those defined outside any function body) are technically accessible within functions, but they are not shown within the current variable scope.

The second top-level item displayed in the Local Variables pane is the This item. The This item refers to whichever object the keyword `this` refers to. If the breakpoint occurs within a function that is part of an object, then `this` refers to the current object instance. The normal reference for `this` is the browser's `window` object. Note that any variables defined within the global scope will appear under the `this` item.

The Local Variables list also allows you to change the value of variables during runtime. This can be extremely powerful when you want to test the effects of different variable values on the script's output. It's also useful when you *think* you've found where a problem is occurring and want to see whether changing a variable value fixes the problem.

Right-click the variable value you want to change, and select Change Value from the context menu. This opens a small prompt window in which you can modify the variable's value. You can enter any valid JavaScript expression into the prompt, including expressions such as `new Object()`. Be sure that any string literals are enclosed in either double or single quotes. Remember that in the prompt window you can also reference other variables by using the variable name.

Testing Tools

As a Java developer you're almost certainly familiar with development methodologies like test-driven development (TDD) and unit testing tools like JUnit, and you may even use these techniques and tools in your daily work routine. If you use JUnit, there has probably been at least one instance where a unit test has saved your hide by uncovering a previously unknown bug, or it's allowed you to comfortably refactor a class knowing that as long as all the tests pass, you've managed to refactor without breaking any functionality.

Wouldn't it be great to be able to apply TDD and other automated testing techniques to JavaScript and, more specifically, Ajax development? Now you can. New tools are popping up every day that allow you to develop JavaScript and Ajax-enabled applications that have complete test coverage. In this section we'll explore a couple of testing tools that allow you create automated tests for your JavaScript code and Ajax-enabled web applications.

JsUnit

In 2001, Edward Hieatt began work on a "port" of JUnit, the popular Java unit testing framework (www.junit.org), for use in testing JavaScript in the browser—JsUnit. Since then, JsUnit (found at www.jsunit.net) has had nearly 10,000 downloads and counts almost 300 people as members of its newsgroup. JsUnit supports the common xUnit functions and is written entirely in JavaScript. If you're comfortable with JUnit (or similar xUnit frameworks), you'll find JsUnit pretty intuitive.

The usual suspects are present: `setUp()` and `tearDown()` are there, though as functions instead of methods; test functions (instead of test methods) are grouped into test pages (instead of test cases); and JsUnit comes with its own HTML-based test runners. Table 2-1 compares the two frameworks.

Table 2-1. *JUnit vs. JsUnit*

JUnit	JsUnit
`Test` class extends `TestCase`	Test page includes `jsUnitCore.js`
`Test` methods	Test functions
`Test` classes	HTML-based test pages
`TestSuite` classes	HTML-based test suites
Various test runners	HTML/JavaScript-based test runner
`setUp()` and `tearDown()` methods	`setUp()` and `tearDown()` functions
Runs in the virtual machine	Runs in a browser
Written in Java	Written in JavaScript

JsUnit tests are written much like JUnit tests. A test case is an HTML page that includes some JavaScript you want to test. You then write test methods much like you do in JUnit and use various assert methods to verify expected outcomes. Like JUnit, test cases can be grouped together as test suites. JsUnit's Test Runner (Figure 2-18) is an HTML page through which test cases and test suites can be executed within a browser, allowing fast code-test-repeat cycles.

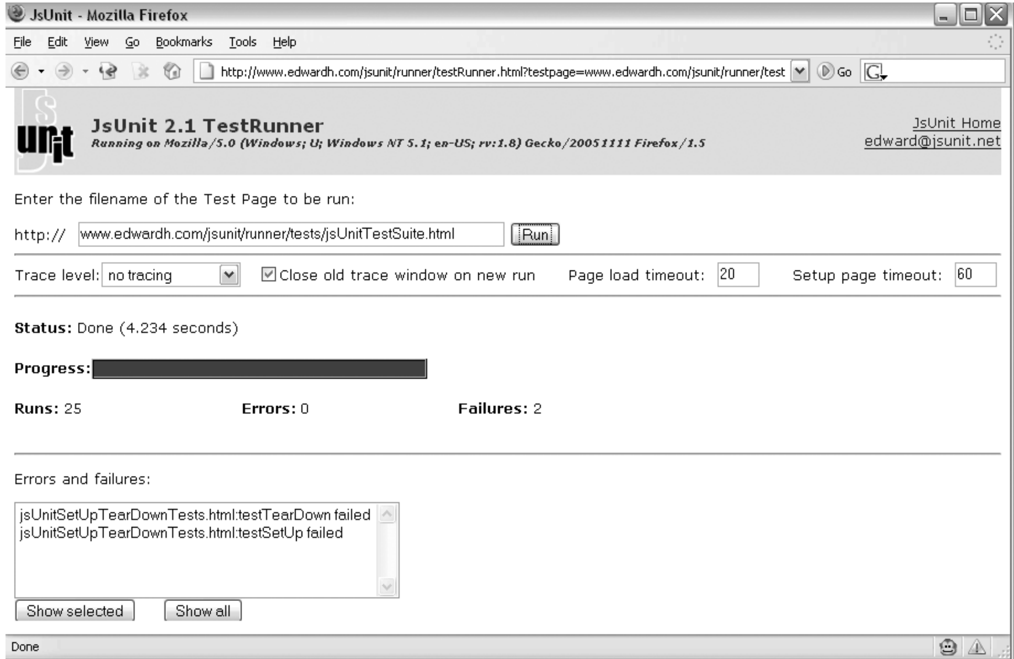


Figure 2-18. *Example Test Runner from JsUnit's home page*

This text won't discuss the nitty-gritty details of installing and using JsUnit. The JsUnit home page at `jsunit.net` has plenty of examples demonstrating how to use JsUnit.

Selenium

Selenium, found at `www.openqa.org/selenium`, is a test tool for web applications. Selenium runs completely within the browser, just like real users do. It runs on all major browsers on all major platforms and serves two major purposes: browser compatibility testing and system functional testing. Since Selenium runs on all major browsers on all major platforms, you can use it to verify that your application works correctly across browsers and platforms. System functional testing is achieved by the creation of Selenium regression tests that verify application functionality and user acceptance.

Like JsUnit, Selenium can be used to test JavaScript and Ajax requests. The difference between the two is that JsUnit is more of a unit testing framework, and Selenium is more of an acceptance testing framework.[1] Because of this, JsUnit and Selenium complement each other. JsUnit can only be used to unit test JavaScript code, but Selenium can test

1. Unit tests are written by developers and test a single class or a small group of classes. Acceptance tests are written by the application's user community and test the functionality of the entire application.

JavaScript code at the application level, in addition to testing non-JavaScript aspects of a web application.

A Selenium test case is written as a simple HTML file containing a table with three columns. The three columns represent a *command*, *target*, and *value*. Not all commands require a value, in which case the table cell can be left blank. Since test cases are written as simple HTML files, not JavaScript functions as in JsUnit, it's possible that nontechnical personnel could create the test cases.

A command simply tells Selenium to do something. Commands come in two varieties: actions and assertions. Actions are generally tasks that change the state of the application, such as "click this button" or "enter text into a text field." Assertions verify the state of the application. Examples include "verify this text appears on a page" or "the current URL is this." All assertions can be one of two modes, *assert* or *verify*. They are identical except that when an assert fails, the test stops. Table 2-2 lists some of the most common commands. For a complete list, visit Selenium's home page.

Table 2-2. *Common Selenium Action and Assertion Commands*

Command	Description
`open(url)`	Opens the specified URL; accepts both relative and absolute URLs
`click(element)`	Clicks a button, link, radio button, or check box
`clickAndWait(element)`	Same as click, except waits for the new page that is loaded in response to the click; typically used on links and submit buttons
`type(element, value)`	Sets the value of an input field as if it was typed in
`pause(interval)`	Waits the specified number of milliseconds; useful for Ajax requests
`assertValue(element, pattern)`	Asserts that the value of an input field matches the specified pattern
`verifyTextPresent(text)`	Verifies that the specified text appears somewhere on the page

A target is the DOM object on which the specified command is to operate. All commands must have a specified target. If the `clickAndWait` command is specified then a target object for the click must be specified—in this case, probably a button or a link. Selenium supports several strategies for specifying the desired target. The most common and likely the easiest to use is to specify the DOM object's `id` attribute. Selenium can also locate DOM objects by searching on the object's `name` attribute, by evaluating a given JavaScript or XPath expression, or by specifying a link's text.

Like JUnit, Selenium test cases are grouped together as test suites. Like test cases, test suites are written as simple HTML files with tables. The test suite's table consists of multiple table rows that have a single table column. Each table cell contains an anchor tag whose `href` attribute points to the test case, and the anchor tag's text is text describing the test case.

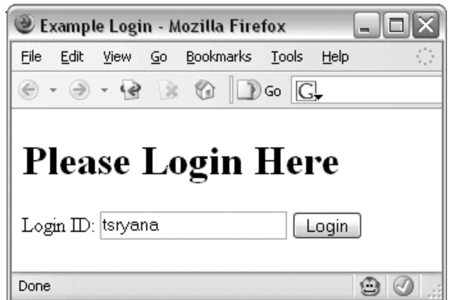
Test suites are executed via Selenium's Test Runner page. The Test Runner is a web page from which tests are executed and test results displayed. By default, Selenium looks for test suites in a default directory, although this can be overridden.

Example Tests

Selenium is easy to use once you've seen it in action, but installing it and writing the first tests can be a bit daunting. In this section we'll go through installing Selenium into a Java EE web app and writing a couple of simple tests.

Start by downloading the latest release of Selenium. The Selenium download is a ZIP file that includes the Selenium runtime engine and documentation. To install Selenium into your Java EE application, copy the `selenium` folder from the Selenium distribution to the root of your application's WAR file. That's all—Selenium is now installed. Next, you need to write some tests.

Before tackling an Ajax test we'll first investigate testing a "normal" HTTP request. This example simulates the login process for a web application. The user enters a login ID and clicks the Login button. The next page echoes the login ID. Figure 2-19 shows the example login page and the next page that echoes the login ID.



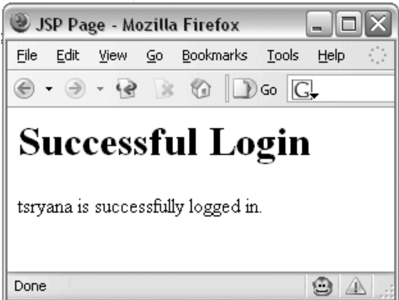


Figure 2-19. *The login page (left) and the next page echoing the login ID*

Listing 2-6 shows the source code for the login page, and Listing 2-7 lists the source code for the following page.

Listing 2-6. *login.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Example Login</title>
    </head>
    <body>
        <h1>Please Login Here</h1>
        <form id="loginForm" action="loggedIn.jsp" method="get">
            Login ID:
            <input type="text" id="loginId" name="loginId" value=""/>
            <input type="submit" value="Login"
                                            id="loginButton" name="loginButton"/>
        </form>
    </body>
</html>
```

Listing 2-7. *loggedIn.jsp*

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
    <h1>Successful Login</h1>
    <%= request.getParameter("loginId")%> is successfully logged in.
    </body>
</html>
```

This is a very simple example, but it illustrates how to write a simple Selenium test. The test should simulate the user entering a user ID into the input field and then clicking the Login button. The next screen should echo the user ID with text saying that the login was successful.

Listing 2-8 lists the HTML page that executes this test scenario. As explained earlier, the test is written as a three-column table within an HTML page. This test consists of four tasks: opening the login page, entering text into the input box, clicking the Login button, and verifying that the next page contains the entered login ID. Each task gets its own row on the table, but note that the table has five rows because Selenium ignores the first row so it can be used for the test case's name.

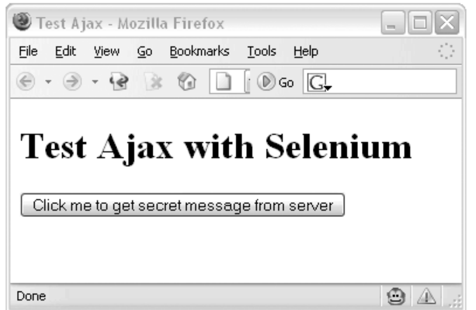
Listing 2-8. *testLogin.html*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <title></title>
    </head>
    <body>
        <table border="1">
            <tr>
            <td colspan="3">Test Echoing Login ID</td>
            <tr>
                <td>open</td>
                <td>/chapter02/login.jsp</td>
                <td> </td>
            </tr>
            <tr>
                <td>type</td>
                <td>loginId</td>
                <td>tsryana</td>
            </tr>
            <tr>
                <td>clickAndWait</td>
                <td>loginButton</td>
                <td> </td>
            </tr>
            <tr>
                <td>verifyTextPresent</td>
                <td>tsryana</td>
                <td> </td>
            </tr>
        </table>
    </body>
</html>
```

The commands are rather self-explanatory. The `open` command tells Selenium to point the browser to the login page. The `type` command tells Selenium to enter the text “tsryana” into the DOM element whose `id` attribute value is `loginButton`. The following command, `clickAndWait`, clicks the DOM element whose `id` attribute value is `loginButton` and waits for the next page to appear. Finally, the `verifyTextPresent` assertion verifies that the application correctly echoes the user ID entered into the text box on the login page.

That’s all! By writing a simple HTML file we’ve written a repeatable test case that will run in multiple browsers on multiple platforms. Before we actually execute the test in Selenium’s Test Runner we’ll write a second test, this time testing a simple Ajax function.

The Ajax example retrieves a single line of text from the server and displays the text on the page. Selenium will verify that the text retrieved from the server appears on the page, verifying that the Ajax request finished successfully. Figure 2-20 illustrates the page before and after the Ajax request.



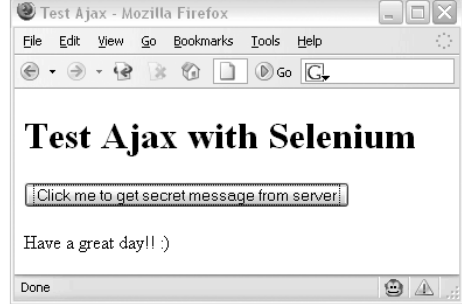


Figure 2-20. *Retrieving simple text via Ajax*

Listing 2-9 lists the JSP page source code, including the JavaScript that takes care of the Ajax request, which by now should be quite familiar to you. Listing 2-10 is the JSP that responds to the Ajax request.

Listing 2-9. *`testAjax.jsp`*

```
<%@page contentType="text/html"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Test Ajax</title>
```

```
        <script type="text/javascript">
            var xmlHttp;
            function createXMLHttpRequest() {
                if (window.ActiveXObject) {
                    xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
                }
                else if (window.XMLHttpRequest) {
                    xmlHttp = new XMLHttpRequest();
                }
            }
            function getSecretMessage() {
                createXMLHttpRequest();
                xmlHttp.onreadystatechange = handleStateChange;
                xmlHttp.open("GET", "ajaxResponse.jsp", true);
                xmlHttp.send(null);
            }
            function handleStateChange() {
                if(xmlHttp.readyState == 4) {
                    if(xmlHttp.status == 200) {
                        document.getElementById("secretMessage").innerHTML =
                                            xmlHttp.responseText;
                    }
                }
            }
        </script>
    </head>
    <body>
        <h1>Test Ajax with Selenium</h1>
        <input type="button" id="button" name="button"
        value="Click me to get secret message from server"
        onclick="getSecretMessage();"/>
        <br/><br/>
        <div id="secretMessage">
        </div>
    </body>
</html>
```

Listing 2-10. *ajaxResponse.jsp*

```
<%@page contentType="text/plain"%>
Have a great day!! :)
```

As in the first example, the test case is written as an HTML page containing a three-column table, which is shown in Listing 2-11. This test case consists of four tasks. The first task is the `open` task, which points the browser to the desired page. The next task, `click`, clicks the DOM element whose `id` attribute is `button`. Note that this differs from the `clickAndWait` task used in the previous example; in this case, the `click` task does not wait for a new page to load.

Unlike a normal HTTP request, the asynchronous nature of the Ajax request means that a new page is not loaded. Instead, the JavaScript that launched the Ajax request continues on without waiting for the Ajax request to finish. So Selenium can't use the `clickAndWait` task because a new page is not being loaded. After initiating the `click` task, which clicks the web page's button, you can use Selenium's `pause` task to wait a specified amount of time, giving the Ajax request time to complete. In this case the pause time is set to 2,000 milliseconds.

Finally, the `verifyTextPresent` assertion checks that the text returned by the server in response to the Ajax request appears on the page. Listing 2-11 is the complete HTML source for this test case.

Listing 2-11. *testAjax.html*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <title></title>
    </head>
    <body>
        <table border="1">
            <tr>
            <td colspan="3">Test Echoing Login ID</td>
            <tr>
                <td>open</td>
                <td>/chapter02/testAjax.jsp</td>
                <td> </td>
            </tr>
            <tr>
                <td>click</td>
                <td>button</td>
                <td> </td>
            </tr>
```

```
            <tr>
                <td>pause</td>
                <td>2000</td>
                <td> </td>
            </tr>
            <tr>
                <td>verifyTextPresent</td>
                <td>Have a great day!! :)</td>
                <td> </td>
            </tr>
        </table>
    </body>
</html>
```

With both test cases written we can now turn our focus to actually running the tests. The first small task we must do is create a test suite that contains these two test cases. As mentioned earlier, a test suite is an HMTL file containing a table that lists the individual test cases. Listing 2-12 lists the source code for the test suite.

Listing 2-12. *TestSuite.html*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <title></title>
    </head>
    <body>
        <table>
            <tbody>
                <tr><td><b>Example Test Suite</b></td></tr>
                <tr><td><a href="testLogin.html">Echo Login ID</a></td></tr>
                <tr><td><a href="testAjax.html">Test Ajax</a></td></tr>
            </tbody>
        </table>
    </body>
</html>
```

You are now ready to run the tests. Remember that the Selenium runtime files should reside at the root of your application's WAR file. Deploy the application and point your browser to `$application-root/selenium/TestRunner.html`. By default, Selenium looks for test suites in the `selenium/tests` directory. You can point Selenium to your test suite by appending the query parameter `?test=path-to-test-suite` to the URL. The path to your test suite is listed relative to the `selenium` directory.

The Test Runner page will open and list all of the individual tests that exist in the test suite. You can choose to run all of the tests collectively or singularly. Selenium indicates the results of the test using familiar JUnit colors: green is success and red is a failure. Figure 2-21 shows the results of executing the two tests.

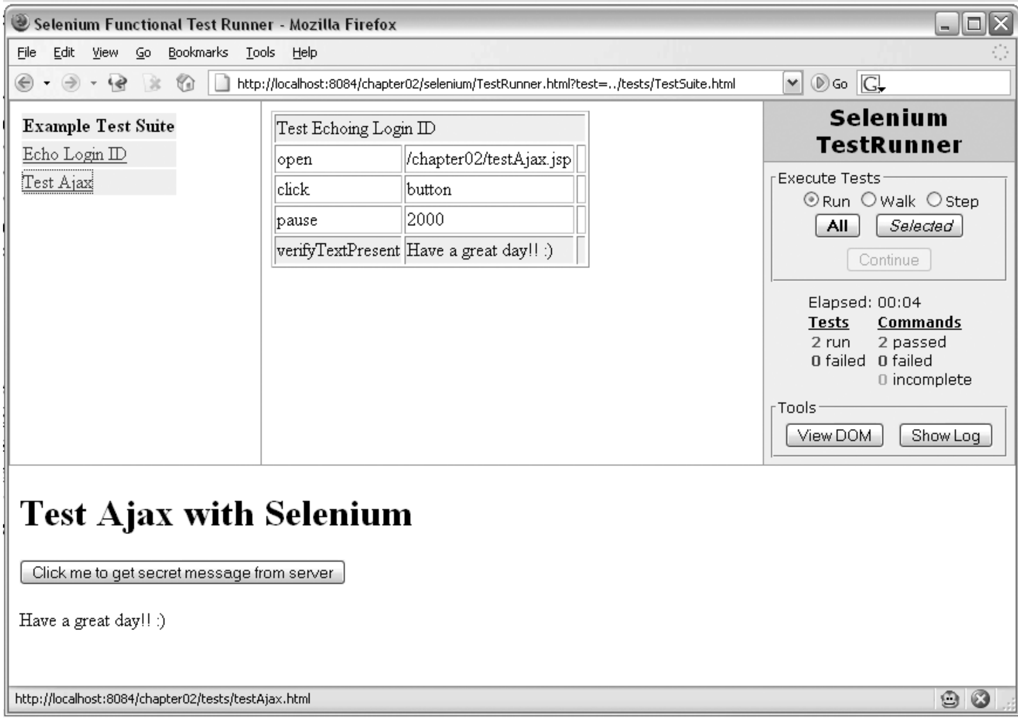


Figure 2-21. *The Selenium Test Runner after running two successful tests*

Figure 2-21 indicates that both test cases passed, including the Ajax test case. If you write a test case for an Ajax request that fails, be sure to check that the amount of wait time set by the `pause` command is sufficient. Sometimes Ajax requests can be slow the first few times they are executed, especially if there are JSPs that need to be compiled. Also note the browser's address bar in Figure 2-21. A query string is used to point the browser to the specific test suite shown in Listing 2-12. The test suite resides in the `tests` directory which is at the application root, and since Selenium searches relative to the `selenium` directory, the query string used is `?test=../tests/TestSuite.html`.

Selenium IDE

You've now seen how powerful Selenium is yet also how easy it is to use. Fortunately there's an even easier way to write Selenium tests than by writing simple HTML files: the Selenium IDE.

The Selenium IDE, as described on its home page at `www.openqa.org/selenium-ide`, is an integrated development environment for authoring and executing Selenium tests. Selenium IDE is implemented as a Firefox extension and includes the complete Selenium Core, so you can record and play back scripts directly from the IDE. Selenium IDE not only automatically records scripts but also allows you to create and edit scripts by hand. It even supports autocomplete for the common Selenium commands.

Installing Selenium IDE is as simple as pointing your browser to the installation page at `www.openqa.org/selenium-ide/download.action` and clicking on the link to initiate the installation process. Firefox might first warn you that the page is attempting to install an extension, and if so, you'll need to grant installation rights to the website. Click the download link again and Firefox will open a window confirming your intent to install the extension. Click Install to start the downloading the extension. You must restart Firefox in order to complete the installation of Selenium IDE.

After restarting Firefox you can open Selenium IDE by selecting the Selenium IDE item under the Tools menu. Figure 2-22 shows the main Selenium IDE window.

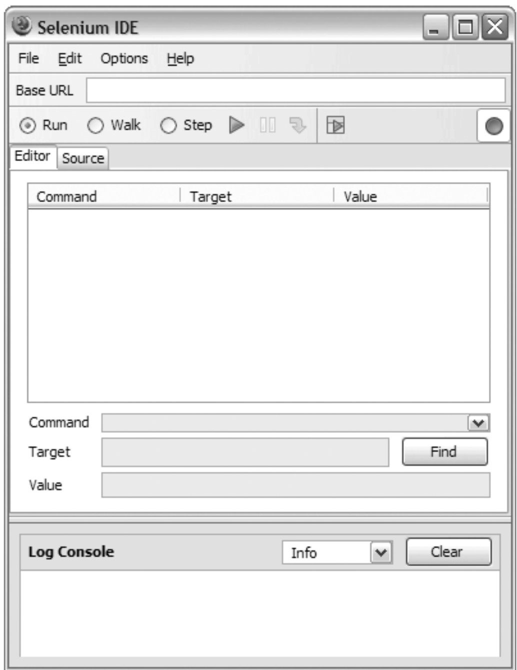


Figure 2-22. *Selenium IDE main window*

The main part of the Selenium IDE window is the Editor tab. The Editor lists all of the commands that appear in the currently opened Selenium test. Here you can add and edit commands by filling in the Command, Target, and Value fields. If you prefer, you can edit the HTML source directly by selecting the Source tab.

You can open an existing test by selecting the Open command from the File menu and navigating to the HTML file representing the Selenium test. Selenium IDE loads the test and lists all of the test's commands in the Editor pane. All of the commands can be edited, moved, copied, or deleted, and new commands can be inserted at any point in the test. Right-click on a command in the Editor pane to see the list of available tasks.

Selenium IDE automatically begins recording browser activity as soon as the Selenium IDE window is opened. You can toggle the recording behavior by clicking on the red circle at the right-hand side of the toolbar just above the Editor pane.

Enough with the basics—let's get busy! We'll do a short example to demonstrate how easy it is to write tests with Selenium IDE. This example will replicate the Ajax test shown earlier and whose Selenium test file is `testAjax.html`, shown in Listing 2-11. The only difference is that instead of writing the HTML file by hand, we'll use the Selenium IDE to help us write the test.

First, be sure that the web server is running so that the pages will be served correctly. Open Firefox and point it to the `testAjax.jsp` page. Then, open Selenium IDE.

With the blank Selenium IDE window open, go back to the browser window and click the button. Selenium IDE will automatically detect the button click and fill in some commands, as shown in Figure 2-23.

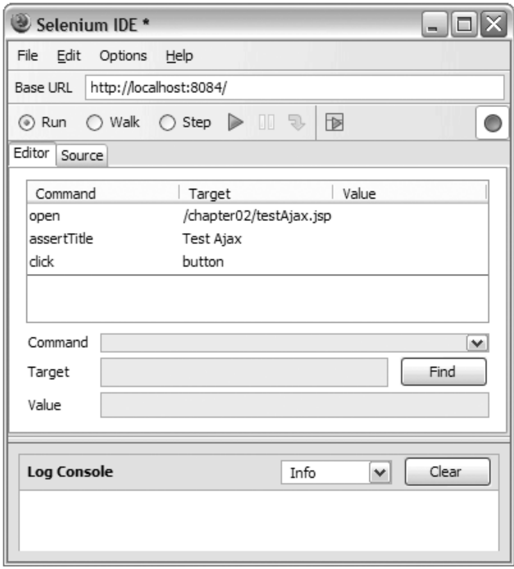


Figure 2-23. *Selenium IDE automatically detects the button click and adds commands to the test.*

You can see that Selenium has automatically added the `open` command and pointed it to the `testAjax.jsp` file. Also note that at the top of the window Selenium lists the base URL, which is kept separate from the application context. This is useful in case you ever

want to test the application in different domains, such as in a development and preproduction domain.

Since Selenium IDE already created the `open` and `click` commands for us, all we need to do to make the test identical to the test shown in Listing 2-11 is add the `verifyTextPresent` command to verify that the Ajax call was successful and the response text was added to the page. To do so, click in the area immediately under the "click" command. The row should become highlighted, and you can now enter the `verifyTextPresent` command and the text value into their respective input boxes, being sure to enter the text value into the Value input box. Note that as you begin typing `verifyTextPresent` into the input box, a list of matching commands will appear in the drop-down list, allowing you to select the command while minimizing typing and reducing the chances of errors.

You can click on the Source tab to verify that the command was added correctly to the underlying HTML file. At this point the contents of the Source tab should be nearly identical to Listing 2-11.

With the test complete, all you need to do is run the test. Selenium IDE provides two ways to run the test. The first is to run the test from the browser window as you saw earlier in this chapter. You can do this by clicking the green arrow on the right side of the window next to the red Record button. This will open a browser window in Firefox pointed to the Selenium Test Runner, and you can start the test from here.

The other option is to run the test directly from within Selenium IDE. To do this, click the leftmost green arrow. The test will execute and commands will be highlighted in green if they pass or red if they don't. The successful test is shown in Figure 2-24.

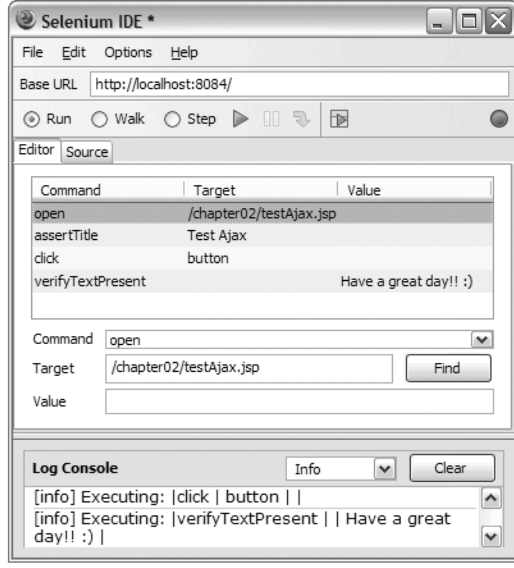


Figure 2-24. *A successful test executed from within Selenium IDE*

Using Selenium IDE, we were able create and execute a Selenium test without writing any HTML. In fact, the only typing required was to add the `verifyTextPresent` command, and even then autocomplete helped with that task. The best part of Selenium IDE is that it's easy enough to be used by nontechnical staff, such as by project managers or business owners.

You should consider using Selenium IDE if you're using Selenium, because it offers so many benefits over writing the tests manually. More information on using Selenium IDE can be found on its home page. A video demonstrating how to record a test can be found at `http://wiki.openqa.org/display/SIDE/Recording+a+Test`.

Which Testing Tool Is Right for Me?

JsUnit and Selenium are powerful tools in their own right. JsUnit is probably the most natural choice for those who are very familiar with JUnit. Selenium sets itself apart from JsUnit by the fact that Selenium tests the functional aspects of an application (instead of testing discrete JavaScript functions) and the way in which test cases are written. Instead of writing JavaScript, Selenium test cases are written as HTML tables, and even then, the Selenium IDE can be used to automatically record and edit tests.

The real answer is that the tools are more complementary to each other than anything else. Selenium is more of a user acceptance or integration testing tool, and the way in which its test cases are written means that nontechnical personnel can potentially write test cases. Selenium could be paired with a more "traditional" unit testing tool like JsUnit to provide a very powerful testing architecture.

Summary

In this chapter you've been exposed to several tools that can make Ajax development easier. JavaScript editors like JSEclipse are making it easier to write nontrivial amounts of JavaScript with features like syntax highlighting and code completion, and formatting tools help ease working with poorly formatted legacy JavaScript files.

The Dojo JavaScript compressor compresses your JavaScript files, reducing the amount of network bandwidth they consume and providing some obfuscation features in the process, and it can even be integrated into your Ant build process. The Mouseover DOM Inspector is a fast, easy-to-use tool that helps you understand the structure of an HTML document, which can aid in learning and debugging.

NetBean's HTTP Monitor can help track down bugs by showing you exactly how a request was made by the browser and what (if any) parameters were included in the request. Venkman is a powerful JavaScript debugging environment that gives you the ability to step through JavaScript code one line at a time. JavaScript logging frameworks like Log4JS and Lumberjack bring log4j-like logging capabilities to your JavaScript code.

Finally, all of your test-first and unit testing development techniques can be brought to JavaScript development using tools like JsUnit and Selenium. Both are great tools that bring the concepts of repeatable test cases to JavaScript and web applications in general.

These tools and the tools that are sure to continue popping up will help you develop higher-quality Ajax applications in less time and with less effort from you, the developer. Try them out and see how you can integrate them into your development process.