Dive Into Greasemonkey

Table of Contents

Table of Contents

Table of Contents

Dive Into Greasemonkey

2005−05−09

This book lives at http://diveintogreasemonkey.org/. If you're reading it somewhere else, you may not have the latest version.

Chapter 1. Getting Started

1.1. What is Greasemonkey?

Greasemonkey is a Firefox extension that allows you to write scripts that alter the web pages you visit. You can use it to make a web site more readable or more usable. You can fix rendering bugs that the site owner can't be bothered to fix themselves. You can alter pages so they work better with assistive technologies that speak a web page out loud or convert it to Braille. You can even automatically retrieve data from other sites to make two sites more interconnected.

Greasemonkey by itself does none of these things. In fact, after you install it, you won't notice any change at all... until you start installing what are called "user scripts". A user script is just a chunk of Javascript code, with some additional information that tells Greasemonkey where and when it should be run. Each user script can target a specific page, a specific site, or a group of sites. A user script can do anything you can do in Javascript. In fact, it can do even more than that, because Greasemonkey provides special functions that are only available to user scripts.

There is a Greasemonkey script repository <http://dunck.us/collab/GreaseMonkeyUserScripts> that contains hundreds of user scripts that people have written to scratch their own personal itches. Once you write your own user script, you can add it to the repository if you think others might find it useful. Or you can keep it to yourself, content in the knowledge that you've made your own browsing experience is a little better.

There is also a Greasemonkey mailing list <http://greasemonkey.mozdev.org/list.html>, where you can ask questions, announce user scripts, and discuss ideas for new features. The Greasemonkey developers frequent the list; they may even answer your question!

Why this book?

Dive Into Greasemonkey grew out of discussions on the Greasemonkey mailing list, and out of my own experience in writing user scripts. After only a week on the list, I was already seeing questions from new users that had already been answered. With only a few user scripts under my belt, I was already seeing common patterns, chunks of reusable code that solved specific problems that cropped up again and again. I set out to document the most useful patterns, explain my own coding decisions, and learn as much as I could in the process.

This book would not be half as good as it is today without the help of the Greasemonkey developers, Aaron Boodman and Jeremy Dunck, and the rest of the people who provided invaluable feedback on my early drafts. Thank you all.

1.2. Installing Greasemonkey

To start writing user scripts, you first need to install the Greasemonkey browser extension, version 0.3 or later.

Procedure: Install the Greasemonkey extension

1. Visit the Greasemonkey home page <http://greasemonkey.mozdev.org/>.
2. Click the link titled "Install Greasemonkey".
3. Firefox will indicate (probably at the top of your browser window) that it prevented the site from installing software. Click ***Edit options...*** to bring up the "Allowed Sites" dialog, then click ***Allow*** to add the Greasemonkey site to your list of sites that are allowed to install software. Click ***OK*** to close the "Allowed Sites" dialog.
4. Once again, click the link titled "Install Greasemonkey".
5. Now an install dialog should pop up to confirm that you really want to install. Wait a few seconds for the install button to enable, then click ***Install now***.

6. Restart your browser.

Once you restart your browser, select the ***Tools*** menu. You should see three new menu items: ***Install User Script...***, ***Manage User Scripts...***, and ***User Script Commands***. Only ***Manage User Scripts...*** will be enabled, but that's OK. The others only become enabled under special circumstances.

By default, installing Greasemonkey does not add any functionality to your browser (other than those three menu items). All it does it enable you to install additional items, called "user scripts", which customize specific web pages.

1.3. Installing a user script

A Greasemonkey "user script" is a single file, written in Javascript, that customizes one or more web pages.

> **Tip:**
>
> Many user scripts are available at the Greasemonkey script repository <http://dunck.us/collab/GreaseMonkeyUserScripts>, although there is no requirement to list your scripts there. You may host (and users may install) your script from anywhere. You don't even need a web server; you can install a user script from a local file.

> **Note:**
>
> A user script's filename must end in `.user.js`.

The first user script I wrote was called "Butler". It adds functionality to Google search results.

Procedure: Install the Butler user script

1. Visit the Butler home page <http://diveintomark.org/projects/butler/> to see a brief description of the functionality that Butler offers. (Not all user scripts have home pages; the only thing Greasemonkey cares about is the script itself.)
2. Click the link titled "Download version..." (0.3 as of this writing), and you will see the script itself in your browser. It is several pages long.
3. In the ***Tools*** menu, the ***Install User Script...*** item should now be enabled. Select it.
4. A dialog will pop up titled "Install User Script", which displays the name of the script you are about to install, a brief description, and a list of included and excluded pages. All of this information is taken from the script itself; you'll learn how to define it in Describing your user script with metadata.
5. Click ***OK*** to install the user script.

If all went well, Greasemonkey will pop up an alert saying "Success! Refresh page to see changes."

Now, search for something in Google <http://www.google.com/>. In the search results page, there is a line at the top of the results that says "Try your search on: Yahoo, Ask Jeeves, AlltheWeb, ...". There is also a banner along the top that says "Enhanced by Butler". All of these were added by the Butler user script.

Further reading

- Greasemonkey script repository <http://dunck.us/collab/GreaseMonkeyUserScripts> contains hundreds of Greasemonkey scripts.

1.4. Managing your user scripts

You can install as many Greasemonkey scripts as you like. Greasemonkey has a graphical configuration dialog to manage your user scripts: disable them temporarily, change their configuration, or uninstall them completely.

Procedure: Disable Butler temporarily

1. From the menu, select ***Tools***−>***Manage User Scripts...***. Greasemonkey will pop up a dialog titled "Manage User Scripts".
2. In the left−hand pane of the dialog is a list of all the user scripts you have installed. (If you've been following along from the beginning, then this will be just one script: Butler.)
3. Select Butler in the list if it is not already selected, and deselect the ***Enabled*** checkbox. The color of "Butler" in the left−hand pane should change subtly from black to gray. (This is difficult to see while it is still selected, but more useful once you have dozens of scripts installed.)
4. Click ***OK*** to exit the "Manage User Scripts" dialog.

Now Butler is installed, but inactive. You can verify this by searching for something on Google. It should no longer say "Enhanced by Butler" along the top. You can re−enable Butler by repeating the procedure and re−selecting the ***Enabled*** checkbox in the "Manage User Scripts" dialog.

> **Note:**
>
> Although I refer to disabling a user script as "temporary", it will remain disabled until you explicitly re−enable it. The only thing that's really temporary about it is that you can easily re−enable it without having to find the original script on my web site and re−install it.

You can also use the "Manage User Scripts" dialog to uninstall scripts entirely.

Procedure: Uninstall Butler

1. From the menu, select ***Tools***−>***Manage User Scripts...***. Greasemonkey will pop up the "Manage User Scripts" dialog.
2. In the left−hand pane, select Butler and click ***Uninstall***. There is no confirmation; the user script is immediately uninstalled.
3. Step 3... There is no step 3! (With apologies to Jeff Goldblum.)

But wait, there's more! You can also change the configuration of user scripts you have previously installed. Do you remember that dialog you got when you first installed Butler, the one that had the two lists of sites to include and exclude? Well, you can change those parameters yourself, either at install time, or at any time in the "Manage User Scripts" dialog.

Let's say, for example, that you like Butler, but you have no use for it on Froogle <http://froogle.google.com/>, Google's product comparison site. You can modify the user script configuration to exclude that site, but still let it work on other Google sites.

Procedure: Reconfigure Butler to leave Froogle alone

1. From the menu, select ***Tools***−>***Manage User Scripts...***. Greasemonkey will pop up the "Manage User Scripts" dialog.

2. In the left−hand pane, select "Butler". In the right−hand pane, it should show you two lists, one of included pages ("http://*.google.*/*") and one of excluded pages (blank).
3. Next to the "Excluded pages" list, click ***Add...***.
4. Greasemonkey will pop up a secondary dialog titled "Add Page" and prompt you to enter a new URL. Enter ***`http://froogle.google.com/*`*** and click ***OK***.
5. Back in the "Manage User Scripts" dialog, the excluded pages list should now include your new URL wildcard, ***`http://froogle.google.com/*`***, meaning that Butler will *not* be executed on any page of the `froogle.google.com` site. The asterisk serves as a simple wildcard, and you may use it within any part of the URL: domain name, path, or even within the URL scheme (`http://`).
6. Click ***OK*** to exit the "Manage User Scripts" dialog and search for a product on Froogle to verify that Butler is no longer being executed. It should still be executed on normal web search results, image search results, and other Google sites.

Chapter 2. Your First User Script

2.1. Hello World

Our hundred−mile−long journey into the wonderful world of Greasemonkey scripting begins with a single step, and it is a step familiar to all readers of technical manuals: getting your computer to print out "Hello world".

Example: `helloworld.user.js` <http://diveintogreasemonkey.org/download/helloworld.user.js>

```
// Hello World! example user script
// version 0.1 BETA!
// 2005−04−22
// Copyright (c) 2005, Mark Pilgrim
// Released under the GPL license
// http://www.gnu.org/copyleft/gpl.html
//
// −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
//
// This is a Greasemonkey user script.
//
// To install, you need Greasemonkey: http://greasemonkey.mozdev.org/
// Then restart Firefox and revisit this script.
// Under Tools, there will be a new menu item to "Install User Script".
// Accept the default configuration and install.
//
// To uninstall, go to Tools/Manage User Scripts,
// select "Hello World", and click Uninstall.
//
// −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
//
// ==UserScript==
// @name          Hello World
// @namespace     http://diveintogreasemonkey.org/download/
// @description   example script to alert "Hello world!" on every page
// @include       *
// @exclude       http://diveintogreasemonkey.org/*
// @exclude       http://www.diveintogreasemonkey.org/*
// ==/UserScript==

alert('Hello world!');
```

As you can see, most of the lines in the `Hello World` user script are comments. Some of these comments, like the instructions on how to install it, have no special meaning; they are simply there for the benefit of clueless end users. However, there is a section in the comments that *does* have special meaning, as you'll see in the next section.

To see the user script in action, you should install it in the usual way, then visit a page outside the `diveintogreasemonkey.org` domain (for example, Google <http://www.google.com/>). The page should display as normal, but an alert will pop up saying "Hello world!"

Download

- `helloworld.user.js` <http://diveintogreasemonkey.org/download/helloworld.user.js>

2.2. Describing your user script with metadata

Every user script has a section of metadata that tells Greasemonkey about the script itself, where it came from, and when to run it.

Example: Hello World metadata

```
// ==UserScript==
// @name          Hello World
// @namespace     http://diveintogreasemonkey.org/download/
// @description   example script to alert "Hello world!" on every page
// @include       *
// @exclude       http://diveintogreasemonkey.org/*
// @exclude       http://www.diveintogreasemonkey.org/*
// ==/UserScript==
```

There are six separate pieces of metadata here, wrapped in a set of Greasemonkey−specific comments. Let's take them in order, starting with the wrapper.

```
// ==UserScript==
//
// ==/UserScript==
```

These comments are significant, and must match exactly. Greasemonkey uses them to signal the start and end of your user script metadata. This section may be defined anywhere in your script, but is usually near the top.

Within the Greasemonkey metadata section, the first item is the name.

```
// @name          Hello World
```

This is the name of your user script. It is displayed in the install dialog when you first install the script, and later in the "Manage User Scripts" dialog. It should be short and to the point.

`@name` is optional. If present, it may appear only once. If not present, it defaults to the filename of the user script, minus the `.user.js` extension.

Next comes the namespace.

```
// @namespace     http://diveintogreasemonkey.org/download/
```

This is a URL, and Greasemonkey uses it to distinguish user scripts that have the same name but are written by different authors. If you have a domain name, you can use it (or a subdirectory) as your namespace. Otherwise you can use a `tag:` URI <http://taguri.org/>.

`@namespace` is optional. If present, it may appear only once. If not present, it defaults to the domain from which the user downloaded the user script.

> **Tip:**
>
> You can specify the items of your user script metadata in any order. I like `@name`, `@namespace`, `@description`, `@include`, and finally `@exclude`, but there is nothing special about this order.

Next comes the description.

```
// @description   example script to alert "Hello world!" on every page
```

This is a human−readable description of what the user script does. It is displayed in the install dialog when you first install the script, and later in the "Manage User Scripts" dialog. It should be no more than two sentences.

`@description` is optional. If present, it may appear only once. If not present, it defaults to an empty string.

> **Important:**
>
> Don't forget the `@description`. Even if you are only writing user scripts for yourself, you will eventually end up with dozens of them, and administering them all in the "Manage User Scripts" dialog will be much more difficult if you don't include a description.

The next three lines are the most important items (from Greasemonkey's perspective): the `@include` and `@exclude` URLs.

```
// @include       *
// @exclude       http://diveintogreasemonkey.org/*
// @exclude       http://www.diveintogreasemonkey.org/*
```

These lines tell Greasemonkey on which sites you want your user script to execute. Both specify a URL, with the `*` character as a simple wildcard for part of the domain name or path. In this case, we are telling Greasemonkey to execute the Hello World script on all sites except `http://diveintogreasemonkey.org/` and `http://www.diveintogreasemonkey.org/`. Excludes take precedence over includes, so even though `http://diveintogreasemonkey.org/download/` matches `*` (all sites), it will be excluded because it also matches `http://diveintogreasemonkey.org/*`.

`@include` and `@exclude` are optional. You may specify as many included and excluded URLs as you need, but you must specify each on its own line. If neither is specified, Greasemonkey will execute your user script on all sites (as if you had specified `@include *`).

> **Note:**
>
> You need to be very specific with your `@include` and `@exclude` metadata. Greasemonkey makes no assumptions about URLs that an end user might consider equivalent. If a site responds on both `http://example.com/` and `http://www.example.com/`, you need to declare both variations.

Further reading

- `tag:` URIs <http://taguri.org/>

2.3. Coding your user script

Our first user script simply displays an alert saying "Hello world!" when it is executed.

Example: Display the "Hello world!" alert

```
alert('Hello world!');
```

Although this code looks obvious enough, and does exactly what you would expect, Greasemonkey is actually doing a number of things behind the scenes to ensure that user scripts do not interact badly with other scripts defined by the original page. Specifically, it automatically wraps your user script in an anonymous function wrapper. Ordinarily you can ignore this, but it will eventually creep up and bite you in the ass, so you may as well learn about it now.

One of the most common ways this can bite you is that variables and functions that you define in a user script are *not* available to other scripts. In fact, they are not available at all once the user script has finished running. This means that you will run into problems if you are expecting to be able to call your own functions later by using the `window.setTimeout` function, or by setting string−based `onclick` attributes on links and expecting Javascript to evaluate your function names later.

For example, this user script defines a function `helloworld`, then attempts to set a timer to call it one second later.

Example: Bad way to delay calling a function

```
function helloworld() {
    alert('Hello world!');
}

window.setTimeout("helloworld()", 60);
```

This will not work; no alert will be displayed. If you open JavaScript Console, you will see an exception displayed: `Error: helloworld is not defined.` This is because, by the time the timeout expires and the call to `helloworld()` is evaluated, the `helloworld` function no longer exists.

If you need to reference your user script's variables or functions later, you will need to explicitly define them as properties of the `window` object, which is always available.

Example: Better way to delay calling a function

```
window.helloworld = function() {
    alert('Hello world!');
}

window.setTimeout("helloworld()", 60);
```

This works as expected: one second after the page loads, an alert pops up proudly displaying "Hello world!"

However, setting properties on `window` is still not ideal; it's a bit like using a global variable when a local one will do. (Actually, it's exactly like that, since `window` is global and available to all scripts on the page.) More practically, you could end up interfering with other scripts that were defined on the page, or even other user scripts.

The best solution is to define an anonymous function yourself and pass it as the first argument to `window.setTimeout`.

Example: Best way to delay calling a function

```
window.setTimeout(function() { alert('Hello world!') }, 60);
```

What I'm doing here is creating a function without a name (an "anonymous function"), then immediately passing the function itself to `window.setTimeout`. This accomplishes the same thing as the previous example, but it leaves no trace, i.e. it's undetectable to other scripts.

I find that I use anonymous functions regularly while writing user scripts. They are ideal for creating "one−off" functions and passing them as arguments to things like `window.setTimeout`, `document.addEventListener`, or assigning to event handlers like `click` or `submit`.

Further reading

- Anonymous functions in Javascript <http://novemberborn.net/sifr/explained/terminology>
- Block Scope in Javascript <http://youngpup.net/2004/jsblockscope> and associated discussion thread <http://youngpup.net/2004/jsblockscope/comments>

2.4. Editing your user script

For script authors, the "Manage User Scripts" dialog has a feature that is especially useful: an ***Edit*** button to edit an installed script "live".

Procedure: Edit Hello World source code and see the results

1. From the menu, select ***Tools***−>***Manage User Scripts...***. Greasemonkey will pop up the "Manage User Scripts" dialog.
2. In the left−hand pane, select Hello World and click ***Edit***. This should open the installed version of Hello World in your favorite text editor. (If it doesn't, verify that `.js` files are associated with your favorite text editor.)
3. Change the `alert` statement to display "Live editing!" instead.
4. Save your changes in your editor, then go back to your browser and test the change by refreshing any page. You should see the results of your change immediately; there is no need to re−install or "refresh" your user script in any way. You are editing "live".

   Tip:

   When you click ***Edit*** in the "Manage User Scripts" dialog, you are "live" editing a copy of your script that is buried deep inside your Firefox profile directory. I've gotten into the habit, once I've finished a "live" editing session, of going back to my text editor one last time and selecting ***File***−>***Save as...***, and saving the user script in another directory. While it's not necessary (Greasemonkey only pays attention to the copy in your profile directory), I prefer to keep the "master copy" of my scripts in another directory with the rest of my work.

Chapter 3. Debugging User Scripts

3.1. Tracking crashes with JavaScript Console

If your user script doesn't appear to be running, the first place to check is JavaScript Console, which lists all script−related errors, including user scripts.

Procedure: Open JavaScript Console

1. From the Firefox menu, select ***Tools***−>***Javascript Console***.
2. The console lists all script errors on all pages since you opened Firefox, which can be quite a lot. (You'd be surprised how many high−profile sites have scripts that crash regularly.) Click ***Clear*** to clear the list before starting to debug your own user script.

Now refresh the page you were working on to test the user script that doesn't appear to be doing anything. If it is indeed crashing, an exception will be displayed in JavaScript Console.

Note:

If your user script is crashing, JavaScript Console will display an exception and a line number. Due to the way Greasemonkey injects user scripts into a page, this line number is not actually useful, and you should ignore it. It is *not* the line number within your user script where the exception occurred.

3.2. Logging with `GM_log`

Greasemonkey provides a logging function, `GM_log`, that allows you to write messages to JavaScript Console. Such messages should be taken out before release, but they are enormously helpful in debugging. Plus, watching the console pile up with log messages is much more satisfying than sprinkling alerts throughout your code and clicking ***OK*** over and over.

`GM_log` takes one argument, the string to be logged. After logging to JavaScript Console, the user script will continue executing normally.

Example: Write to JavaScript Console and continue (`gmlog.user.js` <http://diveintogreasemonkey.org/download/gmlog.user.js>)

```
if (/^http:\/\/diveintogreasemonkey\.org\//.test(window.location.href)) {
    GM_log('running on Dive Into Greasemonkey site w/o www prefix');
} else {
    GM_log('running elsewhere');
}
GM_log('this line is always printed');
```

If you install this user script and open http://diveintogreasemonkey.org/, these two lines will appear in JavaScript Console:

```
Greasemonkey: http://diveintomark.org/projects/greasemonkey//Test Log:
running on Dive Into Greasemonkey site w/o www prefix
Greasemonkey: http://diveintomark.org/projects/greasemonkey//Test Log:
this line is always printed
```

As you can see, Greasemonkey includes the namespace and script name, taken from the user script's metadata section, then the message that was passed as an argument to `GM_log`.

If you visit somewhere other than http://diveintogreasemonkey.org/, these two lines will appear in JavaScript Console:

```
Greasemonkey: http://diveintomark.org/projects/greasemonkey//Test Log:
running elsewhere
Greasemonkey: http://diveintomark.org/projects/greasemonkey//Test Log:
this line is always printed
```

I tried and failed to find a length limit for logged messages. It is larger than 255 characters. Plus, lines in JavaScript Console wrap properly, so you can always scroll down to see the rest of your log message. Go nuts with logging!

Tip:

In JavaScript Console, you can right−click (Mac users control−click) on any line and select ***Copy*** to copy it to the clipboard.

See also

- GM_log

3.3. Inspecting elements with DOM Inspector

DOM Inspector allows you to explore the parsed document object model (DOM) of any page. You can get details on each HTML element, attribute, and text node. You can see all the CSS rules from each of the page's stylesheets. You can explore all the scriptable properties of an object. It's extremely powerful.

DOM Inspector is included with the Firefox installation program, but depending on your platform, it may not installed by default. If you don't see ***DOM Inspector*** in the ***Tools*** menu, you will need to re−install Firefox to get DOM Inspector. This will not affect your existing bookmarks, preferences, extensions, or user scripts.

Procedure: Install DOM Inspector

1. Run the Firefox installation program.
2. After accepting the licensing agreement, select ***Custom*** install.
3. After selecting the destination directory, the installation wizard will ask you to choose additional components you want to install. Select ***Developer Tools***.
4. After the installation is finished, run Firefox. You should see a new menu item, ***Tools***−>***DOM Inspector***.

To get a sense of how powerful this tool really is, let's take a tour of the DOM of Dive Into Greasemonkey's home page.

Procedure: Inspect and edit the Dive Into Greasemonkey home page

1. Visit http://diveintogreasemonkey.org/.
2. From the menu, select ***Tools***−>***DOM Inspector*** to open DOM Inspector.
3. In the DOM Inspector window, you should see a list of DOM nodes in the left−hand pane. If you don't, open the dropdown menu in the upper−left corner and select ***DOM Nodes***.
4. The DOM node tree always starts with the document element, labeled `#document`. Expand this to reveal the `HTML` element.
5. Expand the `HTML` element to reveal 3 nodes: `HEAD`, `#text`, and `BODY`. Note that `BODY` has an `id` of `diveintogreasemonkey−org`. Adjust the column widths if you don't see this.
6. Expand `BODY` to reveal 5 nodes: `#text`, `DIV id="intro"`, `#text`, `DIV id="main"`, and `#text`.
7. Expand `DIV id="intro"` to reveal 2 nodes: `#text` and `DIV class="sectionInner"`.
8. Expand `DIV class="sectionInner"` to reveal 2 nodes: `#text` and `DIV class="sectionInner2"`.
9. Expand `DIV class="sectionInner2"` to reveal 5 nodes: `#text`, `DIV class="s"`, `#text`, `DIV class="s"`, and `#text`.
10. Expand the first `DIV class="s"` to reveal 5 nodes: `#text`, `H1`, `#text`, `P`, and `#text`.
11. Select the `H1` node. On the original page (behind DOM Inspector), the `H1` element should briefly flash a red border. In the right−hand pane, you should see the node name ("H1"), namespace URI (blank, since HTML does not have a namespace −− this is only used on pages served as `application/xhtml+xml`, or pages displaying XML in some other namespace), node type (`1` is an element), and node value (blank, since headers do not have a value −− the text within the header has its own node).
12. In the dropdown menu at the top of the right−hand pane, you will see a number of choices: ***DOM Node***, ***Box Model***, ***XBL Bindings***, ***CSS Style Rules***, ***Computed Style***, and ***Javascript Object***. These provide different information about the currently selected node. Some of them are editable, and changes are immediately reflected on the original page. Select ***Javascript Object*** to see all the scriptable properties and methods of the selected `H1` element.
13. Select ***CSS Style Rules***. The right−hand pane will split into two panes, the top showing a list of all rules that affect the element (including default rules built into the browser itself), the bottom showing individual

properties defined by those rules.

14. Select the second rule from the top−right pane, the style rules defined by the stylesheet at http://diveintogreasemonkey.org/css/dig.css.
15. Double−click the `font−variant` property from the bottom−right pane and enter a new value of ***`normal`***. In the original page (behind DOM Inspector), the "Dive Into Greasemonkey" logo text should immediately change from small−caps to normal uppercase and lowercase letters.
16. Right−click (Mac users control−click) anywhere in the bottom−right pane and select ***New Property***. A dialog will pop up titled "New Style Rule". Enter a property name of ***`background−color`*** and click ***OK***, then a property value of ***`red`*** and click ***OK*** to apply the new property. The new property and value should show up in the bottom−right pane along with the existing properties, and the logo text in the original page should immediately change to a red background.

If clicking through each level of the DOM node tree is not your idea of a good time, I highly recommend the Inspect Element extension, which allows you to drill directly to a specific element in DOM Inspector.

Warning:

You *must* install DOM Inspector before installing the Inspect Element extension, otherwise Firefox will crash on startup. If this has already happened to you, open a command line window, navigate to the directory where you installed Firefox, and type ***`firefox −safe−mode`***. Firefox will start up without loading extensions, then select ***Tools***−>***Extensions*** and uninstall Inspect Element.

Procedure: Directly inspect an element with Inspect Element

1. Visit the Inspect Element download page and click ***Install Now***.
2. Restart Firefox.
3. Revisit http://diveintogreasemonkey.org/.
4. Right−click (Mac users control−click) on the logo text, `Dive Into Greasemonkey`.
5. From the context menu, select ***Inspect element***. DOM Inspector should open with the `H1` element already selected, and you can immediately start inspecting and/or editing its properties.

Warning:

DOM Inspector does not "follow" you as you browse. If you open DOM Inspector and then navigate somewhere else in the original window, DOM Inspector will get very confused. It's best to go where you want to go, inspect what you want to inspect, then close DOM Inspector before doing anything else.

Further reading

- Introduction to the DOM Inspector <http://www.brownhen.com/DI.html>
- Inspect Element extension
- Inspector Widget extension <http://www.projectit.com/freestuff.html>, an alternative to the Inspect Element extension that adds a toolbar button instead of a context menu item.

3.4. Evaluating expressions with Javascript Shell

Javascript Shell is a bookmarklet that allows you to evaluate arbitrary Javascript expressions in the context of the current page.

Procedure: Install Javascript Shell

1. Visit Jesse's Web Development Bookmarklets <http://www.squarefree.com/bookmarklets/webdevel.html>.
2. Drag the ***Shell*** bookmarklet to your links toolbar.
3. Visit a page (for example, Dive Into Greasemonkey <http://diveintogreasemonkey.org/> home page), and click the ***Shell*** bookmarklet in your links toolbar. The Javascript Shell window will open in the background.

Javascript Shell offers you the same power as DOM Inspector, but in a free−form environment. Think of it as a command line for the DOM. Let's play.

Example: Introduction to Javascript Shell

```
document.title
Dive Into Greasemonkey
document.title = 'Hello World'
Hello World
var paragraphs = document.getElementsByTagName('p')
paragraphs
[object HTMLCollection]
paragraphs.length
5
paragraphs[0]
[object HTMLParagraphElement]
paragraphs[0].innerHTML
Teaching an old web new tricks
paragraphs[0].innerHTML = 'Live editing, baby!'
Live editing, baby!
```

Your changes should be reflected in the original page as soon as you hit ***ENTER***.

The only other thing I want to mention about Javascript Shell is the `props` function.

Example: Get properties of an element

```
var link = document.getElementsByTagName('a')[0]
props(link)
Methods of prototype: blur, focus
Fields of prototype: id, title, lang, dir, className, accessKey,
charset, coords, href, hreflang, name, rel, rev, shape, tabIndex,
target, type, protocol, host, hostname, pathname, search, port,
hash, text, offsetTop, offsetLeft, offsetWidth, offsetHeight,
offsetParent, innerHTML, scrollTop, scrollLeft, scrollHeight,
scrollWidth, clientHeight, clientWidth, style
Methods of prototype of prototype of prototype: insertBefore,
replaceChild, removeChild, appendChild, hasChildNodes, cloneNode,
normalize, isSupported, hasAttributes, getAttribute, setAttribute,
removeAttribute, getAttributeNode, setAttributeNode,
removeAttributeNode, getElementsByTagName, getAttributeNS,
setAttributeNS, removeAttributeNS, getAttributeNodeNS,
setAttributeNodeNS, getElementsByTagNameNS, hasAttribute,
hasAttributeNS, addEventListener, removeEventListener, dispatchEvent,
compareDocumentPosition, isSameNode, lookupPrefix, isDefaultNamespace,
lookupNamespaceURI, isEqualNode, getFeature, setUserData, getUserData
Fields of prototype of prototype of prototype: tagName, nodeName,
nodeValue, nodeType, parentNode, childNodes, firstChild, lastChild,
previousSibling, nextSibling, attributes, ownerDocument, namespaceURI,
prefix, localName, ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE,
CDATA_SECTION_NODE, ENTITY_REFERENCE_NODE, ENTITY_NODE,
PROCESSING_INSTRUCTION_NODE, COMMENT_NODE, DOCUMENT_NODE,
DOCUMENT_TYPE_NODE, DOCUMENT_FRAGMENT_NODE, NOTATION_NODE,
baseURI, textContent, DOCUMENT_POSITION_DISCONNECTED,
DOCUMENT_POSITION_PRECEDING, DOCUMENT_POSITION_FOLLOWING,
DOCUMENT_POSITION_CONTAINS, DOCUMENT_POSITION_CONTAINED_BY,
DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC
Methods of prototype of prototype of prototype of prototype of prototype: toString
```

To coin a phrase, "Whoa!" What's all that about? It's a list of all the properties and methods of that `<a>` element that are available to you in Javascript, grouped by levels in the DOM object hierarchy. Methods and properties that are specific to link elements (like the `blur` and `focus` methods, and the `href` and `hreflang` properties) are listed first, followed by methods and properties shared by all types of nodes (like the `insertBefore` method), and so on.

Again, this is the same information that is available in DOM Inspector... but with more typing and experimenting, and less pointing and clicking.

Warning:

Like DOM Inspector, Javascript Shell does not "follow" you as you browse. If you open Javascript Shell and then navigate somewhere else in the original window, Javascript Shell will get very confused. It's best to go where you want to go, open Javascript Shell, fiddle to your heart's content, then close Javascript Shell before doing anything else.

3.5. Other debugging tools

Here is a list of other debugging tools that I have found useful, but haven't had time to document fully.

Further reading

- Web Developer Extension <http://chrispederick.com/work/firefox/webdeveloper/> contains a plethora of functions for deconstructing pages.
- Aardvark <http://www.karmatics.com/aardvark/> interactively displays tag names, `id` and `class` attributes.
- Venkman Javascript Debugger <http://www.hacksrus.com/~ginda/venkman/> is a complete run−time Javascript debugger.
- Web Development Bookmarklets <http://www.squarefree.com/bookmarklets/webdevel.html> contains a number of useful functions you can drag to your toolbar.
- JSUnit <http://www.edwardh.com/jsunit/> is a unit testing framework for Javascript.
- js−lint <http://www.crockford.com/javascript/lint.html> checks Javascript code for common errors.

Chapter 4. Common Patterns

4.1. Executing a user script on a domain and all its subdomains

Many sites work equally well whether you include the `www.` prefix in the address or not. If you want to write a user script for such sites, you need to make sure to catch both versions of the address.

Example: Metadata tags to match a domain and all subdomains

```
// ==UserScript==
// @include http://example.com/*
// @include http://*.example.com/*
// ==/UserScript==
```

Real examples

- Butler <http://diveintomark.org/projects/butler/butler.user.js>
- Salon Auto−Pass <http://diveintogreasemonkey.org/download/salonautopass.user.js>

4.2. Testing whether a Greasemonkey function is available

New versions of Greasemonkey expose new functions to user scripts. If you plan to distribute your user scripts, you should check that the Greasemonkey functions you use actually exist.

Example: Alert the user if the `GM_xmlhttpRequest` function is not available

```
if (!GM_xmlhttpRequest) {
    alert('Please upgrade to the latest version of Greasemonkey.');
    return;
}
// more code here that uses GM_xmlhttpRequest
```

4.3. Testing whether a page includes an HTML element

You can use the `getElementsByTagName` function to test whether an HTML element exists on a page.

Example: Check if the page contains a `<textarea>`

```
var textareas = document.getElementsByTagName('textarea');
if (textareas.length) {
    // there is at least one textarea on this page
} else {
    // there are no textareas on this page
}
```

Real examples

- BetterDir <http://diveintogreasemonkey.org/download/betterdir.user.js>
- Zoom Textarea <http://diveintogreasemonkey.org/download/zoomtextarea.user.js>

4.4. Doing something for every HTML element

Sometimes you need to do something with every HTML element on a page. Firefox supports `getElementsByTagName('*')`, which returns a collection you can loop through.

Example: Loop through every element

```
var allElements, thisElement;
allElements = document.getElementsByTagName('*');
for (var i = 0; i < allElements.length; i++) {
    thisElement = allElements[i];
    // do something with thisElement
}
```

> **Note:**
>
> You should only do this if you really need to do something with *every* element. If you know in advance that you only want to operate on certain elements, it will be faster to use an XPath query to get exactly the elements you want. See Doing something for every element with a certain attribute for more information.

Real examples

- Anti−Disabler <http://diveintogreasemonkey.org/download/antidisabler.user.js>

4.5. Doing something for every instance of a specific HTML element

Sometimes you need to do something with every instance of a particular HTML element on a page. For example, you could change the font on every `<textarea>`. The easiest way to do this is to call `getElementsByTagName('tagname')`, which returns a collection you can loop through.

Example: Find all the textareas on a page

```
var allTextareas, thisTextarea;
allTextareas = document.getElementsByTagName('textarea');
for (var i = 0; i < allTextareas.length; i++) {
    thisTextarea = allTextareas[i];
    // do something with thisTextarea
}
```

Note:

You should not use this pattern to do something to every link on the page, because the `<a>` element can also be used for in−page anchors. See Doing something for every element with a certain attribute for how to find all the links on a page.

Real examples

- Zoom Textarea <http://diveintogreasemonkey.org/download/zoomtextarea.user.js>

4.6. Doing something for every element with a certain attribute

The single most powerful tool in your Greasemonkey arsenal is the `evaluate` method, which finds elements, attributes, and text on a page using a query language called XPath.

Let's say, for example, that you wanted to find all the links on a page. You might consider using `document.getElementsByTagName('a')`, but then you'll still need to check each element to see if it has an `href` attribute, because the `<a>` element can also be used for named anchors.

Instead, use Firefox's built−in XPath support to find all the `<a>` elements that have an `href` attribute.

Example: Find all the links on a page

```
var allLinks, thisLink;
allLinks = document.evaluate(
    '//a[@href]',
    document,
    null,
    XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
    null);
for (var i = 0; i < allLinks.snapshotLength; i++) {
    thisLink = allLinks.snapshotItem(i);
    // do something with thisLink
}
```

The `document.evaluate` method is the key here. It takes an XPath query as a string, then a bunch of other parameters I'll explain in a minute. This XPath query finds all the `<a>` elements that have an `href` attribute, and returns them in random order. (That is, the first one you get is not guaranteed to be the first link on the page.) Then you access each found element with the `allLinks.snapshotItem(i)` method.

XPath expressions can do wonderous things. Here's one that finds *any* element that has a `title` attribute.

Example: Find all the elements with a `title` attribute

```
var allElements, thisElement;
allElements = document.evaluate(
    '//*[@title]',
    document,
    null,
    XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
    null);
for (var i = 0; i < allElements.snapshotLength; i++) {
    thisElement = allElements.snapshotItem(i);
    switch (thisElement.nodeName.toUpperCase()) {
        case 'A':
            // this is a link, do something
            break;
        case 'IMG':
            // this is an image, do something else
            break;
        default:
            // do something with other kinds of HTML elements
    }
}
```

Tip:

Once you have a reference to an element (like `thisElement`), you can use `thisElement.nodeName` to determine its HTML tag name. If the page is served as `text/html`, the tag name is always returned as uppercase, regardless of how it was specified in the original page. However, if the page is served as `application/xhtml+xml`, the tag name is always lowercase. I always use `thisElement.nodeName.toUpperCase()` and forget about it.

Here's an XPath query that returns every `<div>` with a specific `class`.

Example: Find all `<div>`s with a `class` of `sponsoredlink`

```
var allDivs, thisDiv;
allDivs = document.evaluate(
    "//div[@class='sponsoredlink']",
    document,
    null,
    XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
    null);
for (var i = 0; i < allDivs.snapshotLength; i++) {
    thisDiv = allDivs.snapshotItem(i);
    // do something with thisDiv
}
```

Note that I used double quotes around the XPath query string, so that I could use single quotes within it.

There are lots of variations of the `document.evaluate` method. The second parameter (`document` in both of the previous examples) can be any element, and the XPath query will only return nodes that are children of that element. So if you already have a reference to an element (say, from `document.getElementById` or a member of a `document.getElementsByTagName` array), you can restrict the query to search only children of that element.

The third parameter is a reference to a namespace resolver function, which is only useful if you care about writing user scripts that work on pages served with the `application/xhtml+xml` media type. If you don't know what that means, don't worry about it; there aren't very many pages like that and you probably won't run into one. Mozilla XPath documentation <http://www−jcsu.jesus.cam.ac.uk/~jg307/mozilla/xpath−tutorial.html> explains how to use it, if you really want to know.

The fourth parameter is how you want your results returned. The previous two examples both use `XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE`, which returns elements in random order. I use this 99% of the time, but if for some reason you wanted to make sure that you got elements back in exactly the order in which they appear in the page, you can use `XPathResult.ORDERED_NODE_SNAPSHOT_TYPE` instead. Mozilla XPath documentation <http://www−jcsu.jesus.cam.ac.uk/~jg307/mozilla/xpath−tutorial.html> gives examples of some other variations as well.

The fifth parameter can be used to merge the results of two XPath queries. Pass in the result of a previous call to `document.evaluate`, and it will return the combined results of both queries. The previous two examples both use `null`, meaning that we are only interested in the single XPath query defined in the first parameter.

Got all that? XPath can be as simple or as complicated as you like. I urge you to read this excellent XPath tutorial <http://www.zvon.org/xxl/XPathTutorial/General/examples.html> to learn more about XPath syntax. As for the other parameters to `document.evaluate`, I rarely use them except as you've already seen here. In fact, you can define a function to encapsulate them.

Example: The `xpath` function

```
function xpath(query) {
    return document.evaluate(query, document, null,
        XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE, null);
}
```

Now you can simply call `xpath('//a[@href]')` to get all the links on the page, or `xpath('//*[@title]')` to get all the elements with a `title` attribute. You'll still need to use the `snapshotItem` method to access each item in the results; it's not a regular Javascript array.

Real examples

- Access Bar <http://diveintogreasemonkey.org/download/accessbar.user.js>
- BetterDir <http://diveintogreasemonkey.org/download/betterdir.user.js>
- Blogdex Display Title <http://diveintogreasemonkey.org/download/blogdex−display−title.user.js>
- Butler <http://diveintomark.org/projects/butler/butler.user.js>
- Frownies <http://diveintogreasemonkey.org/download/frownies.user.js>
- Offsite Blank <http://diveintogreasemonkey.org/download/offsiteblank.user.js>
- Rotten Reviews <http://diveintogreasemonkey.org/download/rottenreviews.user.js>
- Stop The Presses <http://diveintogreasemonkey.org/download/stopthepresses.user.js>

Further reading

- Mozilla XPath documentation <http://www−jcsu.jesus.cam.ac.uk/~jg307/mozilla/xpath−tutorial.html>
- XPath tutorial by example <http://www.zvon.org/xxl/XPathTutorial/General/examples.html>
- XPathResult reference <http://www.xulplanet.com/references/objref/XPathResult.html>

4.7. Inserting content before an element

Once you've found an element (by any means), you may wish to insert additional content before it. You can do this with the `insertBefore` method.

Example: Insert an `<hr>` before the main content

This presumes that there is an element whose ID is `"main"`.

```
var main, newElement;
main = document.getElementById('main');
if (main) {
    newElement = document.createElement('hr');
    main.parentNode.insertBefore(newElement, main);
}
```

Real examples

- Butler <http://diveintomark.org/projects/butler/butler.user.js>
- Zoom Textarea <http://diveintogreasemonkey.org/download/zoomtextarea.user.js>

4.8. Inserting content after an element

Once you've found an element, you may wish to insert additional content *after* it. You can use the `insertBefore` function for this too, combined with the `nextSibling` property.

Example: Insert an `<hr>` after the navigation bar

This presumes that there is an element whose ID is `"navbar"`.

```
var navbar, newElement;
navbar = document.getElementById('navbar');
if (navbar) {
    newElement = document.createElement('hr');
    navbar.parentNode.insertBefore(newElement, navbar.nextSibling);
}
```

> **Tip:**
>
> Inserting new content before `someExistingElement.nextSibling` will work even if `someExistingElement` is the last child of its parent (i.e. it has no next sibling). In this case, `someExistingElement.nextSibling` will return `null`, and the `insertBefore` function will simply append the new content after all other siblings. (If this doesn't make any sense to you, don't worry about it. The point is that this example will always work, even when it seems like it shouldn't.)

Real examples

- Blogdex Display Title <http://diveintogreasemonkey.org/download/blogdex−display−title.user.js>
- Butler <http://diveintomark.org/projects/butler/butler.user.js>

4.9. Removing an element

You can use Greasemonkey to remove entire chunks of a page in one fell swoop, with the `removeChild` function.

Example: Remove an ad sidebar

This presumes that there is an element whose ID is `"ads"`.

```
var adSidebar = document.getElementById('ads');
if (adSidebar) {
    adSidebar.parentNode.removeChild(adSidebar);
}
```

> **Note:**
>
> Removing an element with `removeChild` will also remove all the content inside it. For example, if you remove a `<table>` element, this will also remove all of its table cells (`<td>` elements).
>
> **Tip:**
>
> If all you want to do is remove ads, it's probably easier to install AdBlock <http://adblock.mozdev.org/> and import an up−to−date filter list <http://www.geocities.com/pierceive/adblock/> than to write your own user script.

Real examples

- Butler <http://diveintomark.org/projects/butler/butler.user.js>

4.10. Replacing an element with new content

Once you find an element, you can replace it with entirely new content, using the `replaceChild` function.

Example: Replace an image with its alt text

This presumes that there is an element whose ID is `"annoyingsmily"`.

```
var theImage, altText;
theImage = document.getElementById('annoyingsmily');
if (theImage) {
    altText = document.createTextNode(theImage.alt);
    theImage.parentNode.replaceChild(altText, theImage);
}
```

Note:

If you want to replace an element with a large chunk of HTML, you can construct the HTML as a string and then set the element's `innerHTML` property.

Real examples

- Frownies <http://diveintogreasemonkey.org/download/frownies.user.js>

4.11. Inserting complex HTML quickly

The `innerHTML` property allows you to construct HTML as a string and then insert it directly into the page, without constructing separate DOM objects for each HTML element and setting their attributes one by one.

Example: Insert a banner at the top of the page

```
var logo = document.createElement("div");
logo.innerHTML = '<div style="margin: 0 auto 0 auto; ' +
    'border-bottom: 1px solid #000000; margin-bottom: 5px; ' +
    'font-size: small; background-color: #000000; ' +
    'color: #ffffff;"><p style="margin: 2px 0 1px 0;"> ' +
    'YOUR TEXT HERE ' +
    '</p></div>';
document.body.insertBefore(logo, document.body.firstChild);
```

The key here is the second line, where I set `logo.innerHTML` to a string. Firefox parses the string as HTML and creates all the necessary objects, just like it does for the entire page when it first loads. Then I can insert my new `logo` (a `<div>` containing another `<div>` containing a `<p>`) into the page at any point   at the beginning of the page, at the end, or before or after any element I choose. In other words, anywhere on the page.

Real examples

- Access Bar <http://diveintogreasemonkey.org/download/accessbar.user.js>
- Butler <http://diveintomark.org/projects/butler/butler.user.js>
- BetterDir <http://diveintogreasemonkey.org/download/betterdir.user.js>

4.12. Adding images without hitting a central server

Firefox supports `data:` URLs, which are a way to embed chunks of data into a URL instead of retrieving the data from the server in a separate call. You've probably never heard of `data:` URLs because Internet Explorer doesn't support them, so no one uses them. But they can come in handy in user scripts.

Example: Add a graphical logo at the top of the page

```
var logo = document.createElement('img');
logo.src = 'data:image/gif;base64,R0lGODlhDQAOAJEAANno6wBmZgAAAAAAACH5BAAAAAAA'+
    'LAAAAAANAA4AQAIjjI8Iyw3GhACSQecutsFV3nzgNi7SVEbo06lZa66LRib2UQAAOw%3D%3D';
document.body.insertBefore(logo, document.body.firstChild);
```

In this example, the `src` of the `<img>` element is a `data:` URL which contains the entire image data in encoded form. Once the new element is inserted into the page, it will display just like any other image, but without requiring that the image be stored on a central server. In essence, you can embed images in your user scripts and distribute them as part of the same file as the rest of your code.

Tip:

Use the `data:` URI kitchen <http://software.hixie.ch/utilities/cgi/data/data> to construct your own `data:` URLs.

Real examples

- Butler <http://diveintomark.org/projects/butler/butler.user.js>
- Zoom Textarea <http://diveintogreasemonkey.org/download/zoomtextarea.user.js>

See also

- `data:` URI kitchen <http://software.hixie.ch/utilities/cgi/data/data>

4.13. Adding CSS styles

I have often found the need to add my own CSS styles to a page. You can add styles that override existing styles, or styles specific to new elements that your user script inserts.

Example: Make paragraph text larger

```
function addGlobalStyle(css) {
    var head, style;
    head = document.getElementsByTagName('head')[0];
    if (!head) { return; }
    style = document.createElement('style');
    style.type = 'text/css';
    style.innerHTML = css;
    head.appendChild(style);
}

addGlobalStyle('p { font-size: large ! important; }');
```

This function takes one argument, a string containing the style rules you want to add to the page. It works by quite literally injecting a `<style>` element into the page's `<head>` section that contains your style rules. Firefox automatically picks up the change, parses the style rules, and applies them to the page. You may include as many separate style rules as you like in a single function call; just concatenate them together as a string and pass them all at once.

> **Tip:**
>
> You can use the `addGlobalStyle` function to style elements that you insert into a page, or elements that were part of the original page. However, if you are styling existing elements, you should use the `! important` keyword for each rule you define to ensure your styles override the rules defined by the original page.

Real examples

- Access Bar <http://diveintogreasemonkey.org/download/accessbar.user.js>
- Aint It Readable <http://diveintogreasemonkey.org/download/aintitreadable.user.js>
- BetterDir <http://diveintogreasemonkey.org/download/betterdir.user.js>
- Butler <http://diveintomark.org/projects/butler/butler.user.js>
- CDReadable <http://diveintogreasemonkey.org/download/cdreadable.user.js>
- LIP <http://diveintogreasemonkey.org/download/lip.user.js>

See also

- Case study: Ain't It Readable

4.14. Getting an element's style

It is occasionally useful to get a specific element's actual style, after all CSS rules have been applied. You might naively assume that an element's `style` property will give you this, but you would be mistaken. That only returns the contents of the element's `style` attribute. To get the final style (including any styles defined in external stylesheets), you need to use the `getComputedStyle` function.

To illustrate this, let's create a simple test page. It defines a style for all `<p>` elements, but then one of the `<p>` elements overrides that with its `style` attribute.

```
<html>
<head>
<title>Style test page</title>
<style type="text/css">
p { background-color: white; color: red; }
</style>
</head>
<body>
<p id="p1">This line is red.</p>
<p id="p2" style="color: blue">This line is blue.</p>
</body>
</html>
```

Example: Get the styles defined by an element's `style` attribute

```
var p1elem, p2elem;
p1elem = document.getElementById('p1');
p2elem = document.getElementById('p2');
alert(p1elem.style.color); // will display an empty string
alert(p2elem.style.color); // will display "blue"
```

That wasn't terribly helpful, and you should never use `element.style` to get individual styles. (You can use it to *set* individual styles; see Setting an element's style.)

So what should you use instead? `getComputedStyle()`.

Example: Get the actual style of an element

```
var p1elem, p2elem, p1style, p2style;
p1elem = document.getElementById('p1');
p2elem = document.getElementById('p2');
p1style = getComputedStyle(p1elem, '');
p2style = getComputedStyle(p2elem, '');
alert(p1style.color); // will display "rgb(255, 0, 0)"
alert(p2style.color); // will display "rgb(0, 0, 255)"
```

Real examples

- Butler <http://diveintomark.org/projects/butler/butler.user.js>
- Zoom Textarea <http://diveintogreasemonkey.org/download/zoomtextarea.user.js>

4.15. Setting an element's style

If you only need to set a few styles on a single element, you can do this "manually" by setting various subproperties on the element's `style` property.

Example: Setting style on a single element

This presumes that there is an element whose ID is `"logo"`.

```
var logo = document.getElementById('logo');
logo.style.marginTop = '2em';
logo.style.backgroundColor = 'white';
logo.style.color = 'red';
```

> **Warning:**
>
> The property names of individual styles are not always obvious. Generally, they follow the same pattern, where `margin-top` becomes `someElement.style.marginTop`. But there are exceptions: the `float` property is set with `someElement.style.cssFloat`, since "float" is a reserved word in Javascript.

Real examples

- Access Bar <http://diveintogreasemonkey.org/download/accessbar.user.js>
- BetterDir <http://diveintogreasemonkey.org/download/betterdir.user.js>
- Blogdex Display Title <http://diveintogreasemonkey.org/download/blogdex−display−title.user.js>
- Zoom Textarea <http://diveintogreasemonkey.org/download/zoomtextarea.user.js>

Further reading

- CSS properties <http://www.westciv.com/style_master/academy/css_tutorial/properties/>

4.16. Post−processing a page after it renders

Because of how Firefox stores a rendered page internally, large changes to the page's DOM should be done *after* the page is finished loading. You can delay processing of a function using the `addEventListener` function.

Example: Replace an entire page with custom content

```
var newBody =
'<html>' +
'<head>' +
'<title>My New Page</title>' +
'</head>' +
'<body>' +
'<p>This page is a complete replacement of the original.</p>' +
'</body>' +
'</html>';
window.addEventListener(
    'load',
    function() { document.body.innerHTML = newBody; },
    true);
```

Looking closely at this code, you might naturally wonder why it works at all. I am declaring an anonymous function to pass as the second argument to `window.addEventListener`, which accesses the `newBody` variable defined in the outer function. This is called a "closure", and is perfectly legal in Javascript. In general, an "inner" function defined within an "outer" function can access all of the outer function's local variables −− even after the outer function finishes executing. This is a very powerful feature that makes it possible to create event handlers and other functions that contain data structures constructed at runtime.

Tip:

Zapping and replacing `document.body.innerHTML` does not change everything about the page. Anything defined in the `<head>` of the original page will still be in effect, including the page title, CSS styles, and scripts. You can modify or remove these separately.

Real examples

- Access Bar <http://diveintogreasemonkey.org/download/accessbar.user.js>
- BetterDir <http://diveintogreasemonkey.org/download/betterdir.user.js>

4.17. Matching case−insensitive attribute values

Many attribute values in HTML are case−insensitive. Many can also contain leading and trailing spaces. If you want to find all variations, you need to do a little legwork in your XPath queries.

Example: Find all forms whose method is "POST" or "post"

```
var postforms = document.evaluate(
    "//form[translate(@method, 'POST ', 'post')='post']",
    document,
    null,
    XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
    null);
```

This XPath query finds forms that submit with a POST method. First, we need to translate the `method` attribute to lowercase, using the `translate` function to convert uppercase letters to lowercase. (XPath 2.0 has a `lowercase` function, but I didn't have any success using it.) Second, we need to strip leading and trailing spaces. We can incorporate this into our call to the `translate` function by including an extra space in the first argument. Since there is no corresponding letter in the second argument, all spaces are stripped. Finally, we can compare the resulting attribute value to `'post'`.

Real examples

- ForceGet <http://diveintogreasemonkey.org/download/forceget.user.js>

4.18. Getting the current domain name

User scripts that operate on multiple domains (or all domains) frequently need to detect the domain of the current page. You can use `window.location.href` to get the full URL of the current page, or `window.location.host` to get just the domain name by itself.

Example: Get the current domain

```
var href = window.location.host;
```

Real examples

- Offsite Blank <http://diveintogreasemonkey.org/download/offsiteblank.user.js>

4.19. Rewriting links

Once you find a link on a page, you can rewrite it by setting its `href` attribute.

Example: Add a query parameter to the end of a link

This presumes that there is a link with the id `"article"`.

```
var a = document.getElementById('article');
if (a.href.match(/\?/i)) {
    // link already contains other parameters, so append "&amp;printer=1"
    a.href += '&amp;printer=1';
} else {
    // link does not contain any parameters, so append "?printer=1"
    a.href += '?printer=1';
}
```

Real examples

- Rotten Reviews <http://diveintogreasemonkey.org/download/rottenreviews.user.js>
- Stop The Presses <http://diveintogreasemonkey.org/download/stopthepresses.user.js>

4.20. Redirecting pages

You can use Greasemonkey to automatically redirect certain pages, by setting the `window.location.href` property.

Example: Redirect a site to its secure counterpart

```
window.location.href = window.location.href.replace(/^http:/, 'https:');
```

Real examples

- GMail Secure <http://diveintogreasemonkey.org/download/gmailsecure.user.js>
- Salon Auto−Pass <http://diveintogreasemonkey.org/download/salonautopass.user.js>

4.21. Intercepting user clicks

While rewriting links is easy, you can go one step further and trap every click, anywhere on the page, by using the `addEventListener` function. (This also traps clicks on links.) You can then calculate what to do: whether to allow the click to "fall through" to the clicked element, or do something else entirely.

Example: Do something when the user clicks anywhere on the page

```
document.addEventListener('click', function(event) {
    // event.target is the element that was clicked

    // do whatever you want here

    // if you want to prevent the default click action
    // (such as following a link), use these two commands:
    event.stopPropagation();
    event.preventDefault();
}, true);
```

Note the use of an anonymous function passed as an argument to the `document.addEventListener` function.

4.22. Overriding a built−in Javascript method

You can use the `prototype` property to override a native object's method.

Example: Do something before a form is submitted

```
function newsubmit(event) {
    var target = event ? event.target : this;

    // do anything you like here
    alert('Submitting form to ' + target.action);

    // call real submit function
    this._submit();
}

// capture the onsubmit event on all forms
window.addEventListener('submit', newsubmit, true);

// If a script calls someForm.submit(), the onsubmit event does not fire,
// so we need to redefine the submit method of the HTMLFormElement class.
HTMLFormElement.prototype._submit = HTMLFormElement.prototype.submit;
HTMLFormElement.prototype.submit = newsubmit;
```

There are two things going on here. First, I am adding an event listener that traps `submit` events. A `submit` event occurs when the user clicks a Submit button on a form. However, if another script manually calls the `submit()` method of a form, the `submit` event does not fire. So, the second thing I do is override the `submit` method of the `HTMLFormElement` class.

But wait, there's more. Both the event listener and the method override point to the same function, `newsubmit`. If `newsubmit` gets called via a `submit` event, the `event` argument will be populated with an event object that contains information about the event (for example, `event.target` will be the form being submitted). However, if a script manually calls the `submit` method, the `event` argument will be missing, but the global variable `this` will point to the form. Therefore, in my `newsubmit` function, I test whether `event` is null, and if so, get the target form from `this` instead.

> **Tip:**
>
> A `submit` event fires when a user submits a form in the usual way, i.e. by clicking the form's ***Submit*** button or hitting ***ENTER*** within a form. However, the `submit` event *does not* fire when the form is submitted via a script calling `aForm.submit()`. Therefore you need to do two things to capture a form submission: add an event listener to capture to `submit` event, *and* modify the prototype of the `HTMLFormElement` class to redirect the `submit()` method to your custom function.

Further reading

- Javascript event compatibility tables <http://www.quirksmode.org/js/events_compinfo.html>
- Javascript−DOM prototypes in Mozilla <http://www.mozilla.org/docs/dom/mozilla/protodoc.html>
- Displaying Event object constants <http://www.mozilla.org/docs/dom/domref/examples.html#999002>

4.23. Parsing XML

Firefox automatically parses the current page into a DOM, but you can also manually create a DOM out of any XML string, either one you constructed yourself, or XML that you retrieved from a remote location.

Example: Parse an arbitrary string as XML

```
var xmlString = '<passwd>' +
'  <user id="101">' +
'    <login>mark</login>' +
'    <group id="100"/>' +
'    <displayname>Mark Pilgrim</displayname>' +
'    <homedir>/home/mark/</homedir>' +
'    <shell>/bin/bash</shell>' +
'  </user>' +
'</passwd>'
var parser = new DOMParser();
var xmlDoc = parser.parseFromString(xmlString, "application/xml");
```

The key here is the `DOMParser` object, which contains the `parseFromString` method. (It contains other methods too, but they are not useful to us here.) Its `parseFromString` method takes two arguments: the XML string to parse, and the content type. Both arguments are required.

> **Note:**
>
> The `DOMParser`'s `parseFromString` method takes a content type as its second parameter. The method accepts `application/xml`, `application/xhtml+xml`, and `text/xml`. For reasons that are too ridiculous to go into, you should always use `application/xml`.

This pattern is especially powerful when you combine it with the `GM_xmlhttpRequest` function to parse XML from a remote source.

Example: Parse XML from a remote source

```
GM_xmlhttpRequest({
    method: 'GET',
    url: 'http://greaseblog.blogspot.com/atom.xml',
    headers: {
        'User-agent': 'Mozilla/4.0 (compatible) Greasemonkey/0.3',
        'Accept': 'application/atom+xml,application/xml,text/xml',
    },
    onload: function(responseDetails) {
        var parser = new DOMParser();
        var dom = parser.parseFromString(responseDetails.responseText,
            "application/xml");
        var entries = dom.getElementsByTagName('entry');
        var title;
        for (var i = 0; i < entries.length; i++) {
            title = entries[i].getElementsByTagName('title')[0].textContent;
            alert(title);
        }
    }
});
```

This code will load the Atom feed from http://greaseblog.blogspot.com/atom.xml, parse it into a DOM, and query the DOM to get the list of entries. For each entry, it queries the DOM again to get the entry title, then displays it in a dialog box.

See also

- Doing something for every instance of a specific HTML element
- GM_xmlhttpRequest

Chapter 5. Case Studies

5.1. Case study: GMail Secure

Forcing a site to use a secure connection

GMail Secure forces GMail <http://gmail.google.com/> to use a secure connection, by redirecting `http://gmail.google.com/` to `https://gmail.google.com/`.

Google offers GMail <http://gmail.google.com/>, their webmail service, through an unsecured connection (an `http://` address) or a secure connection (an `https://` address). I could try to remember to always use the secure connection when I'm checking my email on a public network (such as an Internet cafe), but why bother when my computer can do remember it for me? I wrote a user script that detects when I'm attempting to use GMail over an unsecure connection and redirects me to the secure site instead.

Example: Redirect GMail to an equivalent `https://` address

```
// ==UserScript==
// @name          GMailSecure
// @namespace     http://diveintogreasemonkey.org/download/
// @description   force GMail to use secure connection
// @include       http://gmail.google.com/*
// ==/UserScript==

window.location.href = window.location.href.replace(/^http:/, 'https:');
```

This user script is very simple. Most of the "work" is accomplished by the `@include` line:

```
// @include       http://gmail.google.com/*
```

This user script only runs if the `@include` matches, so if the script is running, I already know I'm in the wrong place (in the sense that I'm using GMail over an insecure connection). Given that, accomplishing my goal is a single line of code, to redirect the current page to the same URL, but with an `https://` prefix instead of `http://`.

```
window.location.href = window.location.href.replace(/^http:/, 'https:');
```

See also

- Redirecting pages

5.2. Case study: Bloglines Autoload

Automating page actions

Bloglines <http://www.bloglines.com/> is a web−based aggregator for syndicated feeds. It has a 2−pane interface; the left−hand pane displays your subscriptions, and the right−hand pane displays the content from those subscriptions. It's a very nice interface; the only thing I dislike about it is that I always use it the same way: every time I visit Bloglines, I want to see everything I haven't seen before, that is, all the unread items.

In Bloglines, displaying all unread items is just one click away. Clicking on the root level of your subscriptions in the lefthand pane, and it displays the unread items in the righthand pane. But since I always want to do this, I wrote a user script to automate that one click.

Example: Make Bloglines display all unread items automatically

```
// ==UserScript==
// @name          Bloglines Autoloader
// @namespace     http://diveintogreasemonkey.org/download/
// @description   Auto−display all new items in Bloglines
// @include       http://bloglines.com/myblogs*
// @include       http://www.bloglines.com/myblogs*
// ==/UserScript==

if (doLoadAll) {
    doLoadAll();
}
```

This user script is quite simple. Bloglines defines a function, `doLoadAll()`, which is what would get executed if I manually clicked on the root level of my subscriptions list. Calling the function displays all unread items.

However, since Bloglines uses frames, the user script will end up getting executed in each frame (since they all match the pattern I've defined in the `@include`), so I first need to check whether the `doLoadAll()` function exists in this frame:

```
if (doLoadAll) {
```

Assumes the function exists, I simply call it. Since the user script runs in the same context as the page, I can call any scripts that the original page has defined.

```
    doLoadAll();
    }
```

Download

- `bloglines−autoload.user.js` <http://diveintogreasemonkey.org/download/bloglines−autoload.user.js>

5.3. Case study: Ain't It Readable

Overriding page styles

Ain't It Cool News <http://aint−it−cool−news.com/> is a site devoted to entertainment news. The site is wonderful; I highly recommend it. Unfortunately, I can't stand how the site looks. Every headline is too large, too bold, and changes color as you move your cursor over it. So I wrote a user script that changes the styles I don't like.

Example: `aintitreadable.user.js` <http://diveintogreasemonkey.org/download/aintitreadable.user.js>

```
// ==UserScript==
// @name          Ain't It Readable
// @namespace     http://diveintogreasemonkey.org/download/
// @description   change style on aint−it−cool−news.com
// @include       http://aint−it−cool−news.com/*
// @include       http://*.aint−it−cool−news.com/*
// ==/UserScript==

function addGlobalStyle(css) {
    var head, style;
    head = document.getElementsByTagName('head')[0];
    if (!head) { return; }
    style = document.createElement('style');
    style.type = 'text/css';
    style.innerHTML = css;
    head.appendChild(style);
}

addGlobalStyle(
'h1, h2, h3, h4 {' +
'  font−size: 12px ! important;' +
'  line−height: 14px ! important;' +
'  font−weight: normal ! important;' +
'}' +
'h1:hover, h2:hover, h3:hover, h4:hover {' +
'  background−color: inherit ! important;' +
'  color: inherit ! important;' +
'}');
```

This user script is very simple. First I define a function which adds arbitrary CSS styles to the page. See Adding CSS styles for more information about this function.

```
function addGlobalStyle(css) {
    var head, style;
    head = document.getElementsByTagName('head')[0];
    if (!head) { return; }
    style = document.createElement('style');
    style.type = 'text/css';
    style.innerHTML = css;
    head.appendChild(style);
}
```

Then I simply call the function with the CSS styles I want to add: make the headlines smaller and not bold, and remove the color change when you move the cursor over them. In this case, the page defines style rules for each of these, so I use the `! important` keyword to make sure that my style rules are applied instead of the page's original rules.

Note that the function takes only one argument, a string that contains all the style rules to add. I've formatted the string for readability here, but it's still just a string.

```
addGlobalStyle(
'h1, h2, h3, h4 {' +
'  font-size: 12px ! important;' +
'  line-height: 14px ! important;' +
'  font-weight: normal ! important;' +
'}' +
'h1:hover, h2:hover, h3:hover, h4:hover {' +
'  background-color: inherit ! important;' +
'  color: inherit ! important;' +
'}');
```

Download

- `aintitreadable.user.js` <http://diveintogreasemonkey.org/download/aintitreadable.user.js>

See also

- Adding CSS styles

5.4. Case study: Offsite Blank

Forcing offsite links to open in a new window

Offsite Blank is a user script that I wrote after someone posted a request to the Greasemonkey script repository. I personally like to open links in a new tab in the current window, but other people prefer to open a separate window for each site. Offsite Blank lets you do this automatically, by forcing offsite links to open in a new window.

Example: `offsiteblank.user.js` <http://diveintogreasemonkey.org/download/offsiteblank.user.js>

```
// ==UserScript==
// @name          Offsite Blank
// @namespace     http://diveintogreasemonkey.org/download/
// @description   force offsite links to open in a new window
// @include       http://*
// @include       https://*
// ==/UserScript==

var a, thisdomain, links;
thisdomain = window.location.host;
links = document.getElementsByTagName('a');
for (var i = 0; i < links.length; i++) {
    a = links[i];
    if (a.host && a.host != thisdomain) {
        a.target = "_blank";
    }
}
```

First, I declare that this user script should run on every web page (but not, for example, on HTML documents stored on your local machine that you open from the ***File***−>***Open*** menu).

```
// @include       http://*
// @include       https://*
```

The code breaks down into four steps:

1. Get the domain of the current page.
2. Get a list of all the links on the page.
3. Compare the domain of each link to the domain of the page.
4. If the domains don't match, set the target of the link so it opens in a new window.

Getting the domain of the current page is easy. See Getting the current domain name for more information.

```
thisdomain = window.location.host;
```

Getting a list of all the links on the page is equally easy, although I should note that I am ignoring my own advice and simply using `document.getElementsByTagName('a')` instead of doing an XPath query. To−may−to, to−mah−to...

```
links = document.getElementsByTagName('a');
```

Next I loop through all the links (well, all the `<a>` elements, some of which might be links) and check whether the domain of the link matches the domain of the current page. Because some links might point to non−HTTP URLs (for example, they could point to a local file, or to an FTP server), I need to check whether `a.host` exists, then check whether it's equal to the current domain.

```
for (var i = 0; i < links.length; i++) {
    a = links[i];
    if (a.host && a.host != thisdomain) {
        ...
    }
```

If I find a link that has a domain but isn't the same as the current domain, I simply set its `target` attribute to `"_blank"` to force the link to open in a new window.

```
        a.target = "_blank";
```

Download

- `offsiteblank.user.js` <http://diveintogreasemonkey.org/download/offsiteblank.user.js>

See also

- Getting the current domain name
- Doing something for every instance of a specific HTML element
- Doing something for every element with a certain attribute

5.5. Case study: Dumb Quotes

Converting smart quotes to dumb quotes

DumbQuotes was born out of a frustration shared by many webloggers: most publishing software can automatically convert straight ASCII quotes to "smart quotes", but the same software is stupid about handling "smart quotes" when an author copies and pastes them into an article. A common example is a blogger who wants to quote a passage from another site. They select a few sentences in their web browser, paste it into a web form to post to their own site, and the passage they pasted ends up looking nothing like the original site because their publishing software doesn't track character encoding properly.

Well, I can't fix every publishing system in the world, but I can fix the problem for myself by writing a user script that automatically converts smart quotes and other problematic high−bit characters to their 7−bit ASCII equivalent.

Example: `dumbquotes.user.js` <http://diveintogreasemonkey.org/download/dumbquotes.user.js>

```
// ==UserScript==
// @name          DumbQuotes
// @namespace     http://diveintogreasemonkey.org/download/
// @description   straighten curly quotes and apostrophes, simplify fancy dashes, etc.
// @include       *
// ==/UserScript==

var replacements, regex, key, textnodes, node, s;

replacements = {
    "\xa0": " ",
    "\xa9": "(c)",
    "\xae": "(r)",
    "\xb7": "*",
    "\u2018": "'",
    "\u2019": "'",
    "\u201c": '"',
    "\u201d": '"',
    "\u2026": "...",
    "\u2002": " ",
    "\u2003": " ",
    "\u2009": " ",
    "\u2013": "-",
    "\u2014": "--",
    "\u2122": "(tm)"};
regex = {};
for (key in replacements) {
    regex[key] = new RegExp(key, 'g');
}

textnodes = document.evaluate(
    "//text()",
    document,
    null,
    XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
    null);
for (var i = 0; i < textnodes.snapshotLength; i++) {
    node = textnodes.snapshotItem(i);
    s = node.data;
    for (key in replacements) {
        s = s.replace(regex[key], replacements[key]);
    }
    node.data = s;
}
```

The code breaks down into four steps:

1. Define a list of string replacements, mapping certain 8−bit characters to their 7−bit equivalents.
2. Get all the text nodes of the current page.
3. Loop through the list of text nodes.
4. In each text, node, for each 8−bit character, replace it with its 7−bit equivalent.

The first step is really two steps. Javascript's string replacement is based on regular expressions. So in order to replace 8−bit characters with 7−bit equivalents, I really need to create a set of regular expression objects.

```
replacements = {
    "\xa0": " ",
    "\xa9": "(c)",
    "\xae": "(r)",
    "\xb7": "*",
    "\u2018": "'",
    "\u2019": "'",
    "\u201c": '"',
    "\u201d": '"',
    "\u2026": "...",
    "\u2002": " ",
    "\u2003": " ",
    "\u2009": " ",
    "\u2013": "-",
    "\u2014": "--",
    "\u2122": "(tm)"};
regex = {};
for (key in replacements) {
    regex[key] = new RegExp(key, 'g');
}
```

I use the curly braces syntax to quickly create an associative array. This is equivalent to (but less typing than) assigning each key−value pair individually:

```
replacements["\xa0"] = " ";
replacements["\xa9"] = "(c)";
replacements["\xae"] = "(r)";
// and so forth
```

The individual 8−bit characters are represented by their hexadecimal values, using the escaping syntax `"\xa0"` or `"\u2018"`. Once we have an associative array of characters to strings, I loop through the array and create a list of regular expression objects. Each regular expression object will search globally for one 8−bit character. (The *`'g'`* in the second argument means search globally; otherwise each regular expression would only search and replace the first occurrence of a particular 8−bit character, and I might end up missing some.)

The next step is to get a list of all the text nodes in the current document. You might be tempted to say to yourself, "Hey, I could just use `document.body.innerHTML` and get the whole page as a single string, and search and replace in that."

```
var tmp = document.body.innerHTML;
// do a bunch of search/replace on tmp
document.body.innerHTML = tmp;
```

But this is a bad habit to get into, because the `innerHTML` property will return the entire source of the page: all the markup, all the script, all the attributes, everything. In this case it probably wouldn't cause a problem (HTML tags themselves do not contain 8−bit characters), but in other cases it could be disastrous, and very difficult to debug. You need to ask yourself what exactly you are trying to search−and−replace. If the answer is "the raw page source," then go ahead and use `innerHTML`. However, in this case, the answer is "all the text on the page," so the correct solution is to use an XPath query to get all the text nodes.

```
textnodes = document.evaluate(
    "//text()",
    document,
    null,
    XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
    null);
```

This works by using an XPath function, `text()`, which matches any text node. You're probably more familiar with querying for element nodes: a list of all the `<a>` elements, or all the `<img>` elements that have an `alt` attribute. But the DOM also contains nodes for the actual text content within elements. (There are other types of nodes too, like comments and processing instructions.) And it's the text nodes that we're interested in at the moment.

Step 3 is to loop through all the text nodes. This is exactly the same as looping through all the elements returned from an XPath query like `//a[@href]`; the only difference is that each item in the loop is a text node, not an element node.

```
for (var i = 0; i < textnodes.snapshotLength; i++) {
    node = textnodes.snapshotItem(i);
    s = node.data;
    // do replacements
    node.data = s;
```

`node` is the current text node in the loop, and `s` is a string that is the actual text of `node`. I'm going to use `s` to do the replacements, then copy the result back to the original node.

So now that I have the text of a single node, I need to actually do the replacements. Since I pre−created a list of regular expressions and a list of replacement string, this is relatively straightforward.

```
    for (key in replacements) {
        s = s.replace(regex[key], replacements[key]);
    }
```

Download

- `dumbquotes.user.js` <http://diveintogreasemonkey.org/download/dumbquotes.user.js>

See also

- Doing something for every element with a certain attribute

5.6. Case study: Frownies

Converting graphical smilies to text

Frownies came to be in response to a joke. Someone on the Greasemonkey mailing list <http://greasemonkey.mozdev.org/list.html> announced that they had developed a user script to convert ASCII "smilies" like `:-)` to their graphical equivalents. Someone else responded, wondering how long it would take for someone to do the reverse: convert graphical smilies back to text.

For the record, it took me about 20 minutes. Most of the time was spent researching publishing software that auto−generated graphical smilies, and compiling a comprehensive list of variations.

This script relies on the fact that most publishing software that generates graphical smilies puts the text equivalent in the `alt` attribute of the `<img>` element. So really what this script does is replace images with their ALT text, if the ALT text matches one of a list of pre−defined constants.

Example: `frownies.user.js` <http://diveintogreasemonkey.org/download/frownies.user.js>

```
// ==UserScript==
// @name          Frownies
// @namespace     http://diveintogreasemonkey.org/download/
// @description   convert graphical smilies to their text equivalents
// @include       *
// ==/UserScript==

var smilies, images, img, replacement;
smilies = [":)", ":-)" ":-(", ":(", ";-)", ";)", ":-D", ":D", ":-/",
    ":/", ":X", ":-X", ":\">", ":P", ":-P", ":O", ":-O", "X-(",
    "X(", ":->", ":>", "B-)", "B)", ">:)", ":((", ":(((", ":-((",
    ":))", ":-))", ":-|", ":|", "O:-)", "O:)", ":-B", ":B", "=;",
    "I)", "I-)", "|-)", "|)", ":-&", ":&", ":-$", ":$", "[-(", ":O)",
    ":@)", "3:-O", ":(|)", "@};-", "**==", "(~~)", "*-:)", "8-X",
    "8X", "=:)", "<):)", ";;)", ":*", ":-*", ":S", ":-S", "/:)",
    "/:-)", "8-|", "8|", "8-}", "8}", "(:|", "=P~", ":-?", ":?",
    "#-O", "#O", "=D>", "~:>", "%%-", "~O)", ":-L", ":L", "[-O<",
    "[O<", "@-)", "@)", "$-)", "$)", ">-)", ":-\"", ":^O", "B-(",
    "B(", ":)>-", "[-X", "[X", "\\:D/", ">:D<", "(%)", "=((", "#:-S",
    "#:S", "=))", "L-)", "L)", "<:-P", "<:P", ":-SS", ":SS", ":-W",
    ":W", ":-<", ":<", ">:P", ">:-P", ">:/", ";))", ":-@", "^:)^",
    ":-J", "(*)", ":GRIN:", ":-)", ":SMILE:", ":SAD:", ":EEK:",
    ":SHOCK:", ":???:", "8)", "8-)", ":COOL:", ":LOL:", ":MAD:",
    ":RAZZ:", ":OOPS:", ":CRY:", ":EVIL:", ":TWISTED:", ":ROLL:",
    ":WINK:", ":!:", ":?:", ":IDEA:", ":ARROW:", ":NEUTRAL:",
    ":MRGREEN:"];
images = document.evaluate(
    '//img[@alt]',
    document,
    null,
    XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
    null);
for (var i = 0; i < images.snapshotLength; i++) {
    img = images.snapshotItem(i);
    alt = img.alt.toUpperCase();
    for (var j in smilies) {
        if (alt == smilies[j]) {
            replacement = document.createTextNode(alt);
            img.parentNode.replaceChild(replacement, img);
        }
    }
}
```

The code breaks down into four steps:

1. Define a list of smilies (as text strings).
2. Find all the images on the page that contain an `alt` attribute.
3. For each image, check whether the ALT text matches any of the list of ASCII smilies.
4. If it matches, replace the `<img>` element with a text node that contains only the ASCII smily.

The first step simply defines a list, using Javascript's `[ ]` syntax.

```
smilies = [":)", ":-)" ":-(", ":(", ";-)", ";)", ":-D", ":D", ":-/",
    ":/", ":X", ":-X", ":\">", ":P", ":-P", ":O", ":-O", "X-(",
    "X(", ":->", ":>", "B-)", "B)", ">:)", ":((", ":(((", ":-((",
    ":))", ":-))", ":-|", ":|", "O:-)", "O:)", ":-B", ":B", "=;",
    "I)", "I-)", "|-)", "|)", ":-&", ":&", ":-$", ":$", "[-(", ":O)",
    ":@)", "3:-O", ":(|)", "@};-", "**==", "(~~)", "*-:)", "8-X",
    "8X", "=:)", "<):)", ";;)", ":*", ":-*", ":S", ":-S", "/:)",
    "/:-)", "8-|", "8|", "8-}", "8}", "(:|", "=P~", ":-?", ":?",
    "#-O", "#O", "=D>", "~:>", "%%-", "~O)", ":-L", ":L", "[-O<",
    "[O<", "@-)", "@)", "$-)", "$)", ">-)", ":-\"", ":^O", "B-(",
    "B(", ":)>-", "[-X", "[X", "\\:D/", ">:D<", "(%)", "=((", "#:-S",
    "#:S", "=))", "L-)", "L)", "<:-P", "<:P", ":-SS", ":SS", ":-W",
    ":W", ":-<", ":<", ">:P", ">:-P", ">:/", ";))", ":-@", "^:)^",
    ":-J", "(*)", ":GRIN:", ":-)", ":SMILE:", ":SAD:", ":EEK:",
    ":SHOCK:", ":???:", "8)", "8-)", ":COOL:", ":LOL:", ":MAD:",
    ":RAZZ:", ":OOPS:", ":CRY:", ":EVIL:", ":TWISTED:", ":ROLL:",
    ":WINK:", ":!:", ":?:", ":IDEA:", ":ARROW:", ":NEUTRAL:",
    ":MRGREEN:"];
```

Next I search the page for all `<img>` elements with an `alt` attribute, using an XPath query. See Doing something for every element with a certain attribute for more details on XPath queries.

```
images = document.evaluate(
    '//img[@alt]',
    document,
    null,
    XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
    null);
```

Step 3 loops through all these `<img>` elements, and checks whether the `alt` attribute matches any of my defined smilies. Because some smilies contain letters, I use the `toUpperCase()` method to convert the `alt` attribute to uppercase before comparing.

```
for (var i = 0; i < images.snapshotLength; i++) {
    img = images.snapshotItem(i);
    alt = img.alt.toUpperCase())
    for (var j in smilies) {
        if (alt == smilies[j]) {
            // ...
        }
    }
}
```

Finally, I create a new text node with the text of the smily, and replace the existing `<img>` element. See Replacing an element with new content for more details.

```
            replacement = document.createTextNode(alt);
            img.parentNode.replaceChild(replacement, img);
```

Download

- `frownies.user.js` <http://diveintogreasemonkey.org/download/frownies.user.js>

See also

- Doing something for every element with a certain attribute
- Replacing an element with new content

5.7. Case study: Zoom Textarea

Adding buttons to zoom textareas

Zoom Textarea alters web forms to add a toolbar above every `<textarea>` element (used for entering multiple lines of text). The toolbar lets you increase or decrease the text size of the `<textarea>`, without changing the style of the rest of the page. The buttons are fully keyboard−accessible; you can tab to them and press ***ENTER*** instead of clicking them with your mouse. (I mention this up front because accessibility matters, and also because it was harder than it sounds.)

Example: `zoomtextarea.user.js` <http://diveintogreasemonkey.org/download/zoomtextarea.user.js>

```
// ==UserScript==
// @name          Zoom Textarea
// @namespace     http://diveintogreasemonkey.org/download/
// @description   add controls to zoom textareas
// @include       *
// ==/UserScript==

var textareas, textarea;

textareas = document.getElementsByTagName('textarea');
if (!textareas.length) { return; }

function textarea_zoom_in(event) {
    var link, textarea, s;
    link = event.currentTarget;
    textarea = link._target;
    s = getComputedStyle(textarea, "");
    textarea.style.width = (parseFloat(s.width) * 1.5) + "px";
    textarea.style.height = (parseFloat(s.height) * 1.5) + "px";
    textarea.style.fontSize = (parseFloat(s.fontSize) + 7.0) + 'px';
    event.preventDefault();
}

function textarea_zoom_out(event) {
    var link, textarea, s;
    link = event.currentTarget;
    textarea = link._target;
    s = getComputedStyle(textarea, "");
    textarea.style.width = (parseFloat(s.width) * 2.0 / 3.0) + "px";
    textarea.style.height = (parseFloat(s.height) * 2.0 / 3.0) + "px";
    textarea.style.fontSize = (parseFloat(s.fontSize) − 7.0) + "px";
    event.preventDefault();
}

function createButton(target, func, title, width, height, src) {
    var img, button;
    img = document.createElement('img');
    img.width = width;
    img.height = height;
    img.style.borderTop = img.style.borderLeft = "1px solid #ccc";
    img.style.borderRight = img.style.borderBottom = "1px solid #888";
    img.style.marginRight = "2px";
    img.src = src;
    button = document.createElement('a');
    button._target = target;
    button.title = title;
    button.href = '#';
    button.onclick = func;
    button.appendChild(img);
    return button;
}

for (var i = 0; i < textareas.length; i++) {
    textarea = textareas[i];
    textarea.parentNode.insertBefore(
        createButton(
            textarea,
            textarea_zoom_in,
            'Increase textarea size',
            20,
            20,
            'data:image/gif;base64,'+
'R0lGODlhFAAUAOYAANPS1tva3uTj52NjY2JiY7KxtPf3%2BLOys6WkpmJiYvDw8fX19vb'+
'296Wlpre3uEZFR%2B%2Fv8aqpq9va3a6tr6Kho%2Bjo6bKytZqZml5eYMLBxNra21JSU3'+
'Jxc3RzdXl4emJhZOvq7KamppGQkr29vba2uGBgYdLR1dLS0lBPUVRTVYB%2Fgvj4%2BYK'+
'Bg6SjptrZ3cPDxb69wG1tbsXFxsrJy29vccDAwfT09VJRU6uqrFlZW6moqo2Mj4yLjLKy'+
's%2Fj4%2BK%2Busu7t783Nz3l4e19fX7u6vaalqNPS1MjHylZVV318ftfW2UhHSG9uccv'+
'KzfHw8qqqrNPS1eXk5tvb3K%2BvsHNydeLi40pKS2JhY2hnalpZWlVVVtDQ0URDRJmZm5'+
'mYm11dXp2cnm9vcFxcXaOjo0pJSsC%2FwuXk6AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'+
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAC'+
'H5BAAAAAAALAAAAAAUABQAAAeagGaCg4SFhoeIiYqKTSQUFwgwi4JlB0pOCkEiRQKKRxM'+
'gKwMGDFEqBYpPRj4GAwwLCkQsijwQBAQJCUNSW1mKSUALNiVVJzIvSIo7GRUaGzUOPTpC'+
'igUeMyNTIWMHGC2KAl5hCBENYDlcWC7gOB1LDzRdWlZMAZOEJl83VPb3ggAfUnDo5w%2F'+
'AFRQxJPj7J4aMhYWCoPyASFFRIAA7'),
        textarea);
    textarea.parentNode.insertBefore(
        createButton(
            textarea,
            textarea_zoom_out,
            'Decrease textarea size',
            20,
            20,
            'data:image/gif;base64,'+
'R0lGODlhFAAUAOYAANPS1uTj59va3vDw8bKxtGJiYrOys6Wkpvj4%2BPb29%2FX19mJiY'+
'%2Ff3%2BKqqrLe3uLKytURDRFpZWqmoqllZW9va3aOjo6Kho4KBg729vWJhZK%2BuskZF'+
'R4B%2FgsLBxHNydY2Mj%2Ff396amptLS0l9fX9fW2dDQ0W1tbpmZm8DAwfT09fHw8n18f'+
'uLi49LR1V5eYOjo6VBPUa6tr769wEhHSNra20pJStPS1KuqrNPS1ZmYm%2B7t77Kys8rJ'+
'y%2Fj4%2BaSjpm9uca%2BvsMjHyqalqHRzdVJRU8PDxVRTVcvKzc3Nz0pKS9rZ3evq7MC'+
'%2FwsXFxp2cnnl4e1VVVu%2Fv8ba2uM7Oz29vcbu6vZqZmnJxc9vb3PHx8uXk5mhnamJh'+
'Y1xcXZGQklZVV29vcHl4eoyLjKqpq6Wlpl1dXuXk6AAAAAAAAAAAAAAAAAAAAAAAAAAAA'+
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'+
'AAACH5BAAAAAAALAAAAAAUABQAAAeZgGaCg4SFhoeIiYqKR1IWVgcyi4JMBiQqA0heQgG'+
'KQTFLPQgMCVocBIoNNqMgCQoDVReKYlELCwUFI1glEYorOgopWSwiTUVfih8dLzRTKA47'+
'Ek%2BKBGE8GEAhFQYuPooBOWAHY2ROExBbSt83QzMbVCdQST8Ck4QtZUQe9faCABlGrvD'+
'rB4ALDBMU%2BvnrUuOBQkE4NDycqCgQADs%3D'),
        textarea);
    textarea.parentNode.insertBefore(
        document.createElement('br'),
        textarea);
}
```

The code looks complicated, and it is complicated, but for different reasons. It looks complicated because of the large multiline jibberish−looking strings in the middle of it. Those are `data:` URIs, which look like hell but are easy to generate. The real complexity lies elsewhere.

First, I get a list of all the `<textarea>` elements on the page. See Doing something for every instance of a specific HTML element for more details on this pattern. If there aren't any, there's no point continuing, so `return`.

```
textareas = document.getElementsByTagName('textarea');
```

```
if (!textareas.length) { return; }
```

Now let's skip around a little. Our "toolbar" is visually a row of buttons, but each button is really just an image of something that looks pushable, wrapped in a link that executes one of our Javascript functions.

Since we'll be creating more than one button (this script only has two, but you could easily extend it with more functionality), I created a function to encapsulate all the button−making logic.

```
function createButton(target, func, title, width, height, src) {
```

The `createButton` function takes 6 arguments:

`target`
: an element object, the `<textarea>` element that this button will control

`func`
: a function object, the Javascript function to be called when the user clicks the button with the mouse or activates it with the keyboard

`title`
: a string, the text of the tooltip when the user moves their cursor over the button

`width`
: an integer, the width of the button. This should be the width of the graphic given in the *`src`* argument.

`height`
: an integer, the height of the button. This should be the height of the graphic given in the *`src`* argument.

`src`
: a string, the URL, path, or `data:` URI of the button graphic

Creating a button is split into two halves: creating the `<img>` element, and then creating the `<a>` element around it.

I create the `<img>` element by calling `document.createElement` and setting a few attributes, including style attributes. See Setting an element's style for more details on this pattern.

```
    img = document.createElement('img');
    img._target = target;
    img.width = width;
    img.height = height;
    img.style.borderTop = img.style.borderLeft = "1px solid #ccc";
    img.style.borderRight = img.style.borderBottom = "1px solid #888";
    img.style.marginRight = "2px";
    img.src = src;
```

One thing I just want to point out quickly, because I think it's cool. You can assign the same value to two attributes at once by using this syntax:

```
    img.style.borderTop = img.style.borderLeft = "1px solid #ccc";
```

OK, moving on. The second half of creating a button is creating the link (an `<a>` element) and putting the `<img>` element inside it.

```
    button = document.createElement('a');
    button._target = target;
    button.title = title;
    button.href = '#';
    button.onclick = func;
    button.appendChild(img);
```

There are two things I want to point out here. First, I need to assign a bogus `href` attribute to the link, otherwise Firefox would treat it a named anchor and wouldn't add it to the tab index (i.e. you wouldn't be able to tab to it, making it inaccessible with the keyboard). Second, I'm setting the `_target` attribute to store a reference to the target `<textarea>`. This is perfectly legal in Javascript; you can create new attributes on an object just by assigning them a value. I'll access the custom `_target` attribute later, in the `onclick` event handler.

Now let's jump back to the `onclick` handler itself. Each handler is a function that takes one parameter, *`event`*.

```
function textarea_zoom_in(event)
```

The `event` object has several properties, only one of which I'm interested in at the moment: `currentTarget`.

```
    link = event.currentTarget;
```

If you read the documentation on the `Event` object <http://www.xulplanet.com/references/objref/Event.html>, you'll see that there are several target−related properties, including one simply called `target`. You might be tempted to use `event.target` to get a reference to the clicked link, but it behaves (in my opinion) inconsistently. When the user tabs to the button and hits ***ENTER***, `event.target` is the link, but when the user clicks the button with the mouse, `event.target` is the image inside the link! I'm sure there's a reasonable explanation for this, but it is beyond my level of understanding of the DOM event model. In any case, `event.currentTarget` returns the link in all cases, so I use that.

Next I retrieve the reference to the `<textarea>` I'm actually going to be zooming, via the custom `_target` property that I set when I created the button.

```
    textarea = link._target;
```

Now the real fun begins. (And you thought you were having fun already!) I need to get the current dimensions and font size of the `<textarea>`, so that I can make them bigger. Simply retrieving the appropriate attributes from `textarea.style` (`textarea.style.width`, `textarea.style.height`, and `textarea.style.fontSize`) will *not* work, because those only get set if the page actually defined them in a `style` attribute on the `<textarea>` itself. That's not what I want; I want the actual current style. For that I need `getComputedStyle`. See Getting an element's style for more details on this function.

```
    s = getComputedStyle(textarea, "");
    textarea.style.width = (parseFloat(s.width) * 1.5) + "px";
    textarea.style.height = (parseFloat(s.height) * 1.5) + "px";
    textarea.style.fontSize = (parseFloat(s.fontSize) + 7.0) + 'px';
```

Finally, do you remember that bogus `href` value I added to my button link to make sure it was keyboard−accessible? Well, it's now become an annoyance, because after Firefox finishes executing the `onclick` handler, it's going to try to follow that link. Since it points to a non−existent anchor, Firefox is going to jump to the top of the page, regardless of where the button is. This is annoying, and to stop it, I need to call `event.preventDefault()` before finishing my `onclick` handler.

```
    event.preventDefault();
```

The rest of the script is straightforward. I loop through all the `<textarea>` elements, create zoom buttons for each of them (each with its own `onclick` handler and button graphic), and insert the zoom buttons before the `<textarea>`. For each image, I use a `data:` URI to create an inline graphic, so users don't need to hit a central server to get the button graphics. See Inserting content before an element and Adding images without hitting a central server for more details on these patterns.

```
for (var i = 0; i < textareas.length; i++) {
    textarea = textareas[i];
    textarea.parentNode.insertBefore(
        createButton(
            textarea,
            textarea_zoom_in,
            'Increase textarea size',
            20,
            20,
            'data:image/gif;base64,'+
'R0lGODlhFAAUAOYAANPS1tva3uTj52NjY2JiY7KxtPf3%2BLOys6WkpmJiYvDw8fX19vb'+
'296Wlpre3uEZFR%2B%2Fv8aqpq9va3a6tr6Kho%2Bjo6bKytZqZml5eYMLBxNra21JSU3'+
'Jxc3RzdXl4emJhZOvq7KamppGQkr29vba2uGBgYdLR1dLS0lBPUVRTVYB%2Fgvj4%2BYK'+
'Bg6SjptrZ3cPDxb69wG1tbsXFxsrJy29vccDAwfT09VJRU6uqrFlZW6moqo2Mj4yLjLKy'+
's%2Fj4%2BK%2Busu7t783Nz3l4e19fX7u6vaalqNPS1MjHylZVV318ftfW2UhHSG9uccv'+
'KzfHw8qqqrNPS1eXk5tvb3K%2BvsHNydeLi40pKS2JhY2hnalpZWlVVVtDQ0URDRJmZm5'+
'mYm11dXp2cnm9vcFxcXaOjo0pJSsC%2FwuXk6AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'+
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAC'+
'H5BAAAAAAALAAAAAAUABQAAAeagGaCg4SFhoeIiYqKTSQUFwgwi4JlB0pOCkEiRQKKRxM'+
'gKwMGDFEqBYpPRj4GAwwLCkQsijwQBAQJCUNSW1mKSUALNiVVJzIvSIo7GRUaGzUOPTpC'+
'igUeMyNTIWMHGC2KAl5hCBENYDlcWC7gOB1LDzRdWlZMAZOEJl83VPb3ggAfUnDo5w%2F'+
'AFRQxJPj7J4aMhYWCoPyASFFRIAA7'),
        textarea);
    textarea.parentNode.insertBefore(
        createButton(
            textarea,
            textarea_zoom_out,
            'Decrease textarea size',
            20,
            20,
            'data:image/gif;base64,'+
'R0lGODlhFAAUAOYAANPS1uTj59va3vDw8bKxtGJiYrOys6Wkpvj4%2BPb29%2FX19mJiY'+
'%2Ff3%2BKqqrLe3uLKytURDRFpZWqmoqllZW9va3aOjo6Kho4KBg729vWJhZK%2BuskZF'+
'R4B%2FgsLBxHNydY2Mj%2Ff396amptLS0l9fX9fW2dDQ0W1tbpmZm8DAwfT09fHw8n18f'+
'uLi49LR1V5eYOjo6VBPUa6tr769wEhHSNra20pJStPS1KuqrNPS1ZmYm%2B7t77Kys8rJ'+
'y%2Fj4%2BaSjpm9uca%2BvsMjHyqalqHRzdVJRU8PDxVRTVcvKzc3Nz0pKS9rZ3evq7MC'+
'%2FwsXFxp2cnnl4e1VVVu%2Fv8ba2uM7Oz29vcbu6vZqZmnJxc9vb3PHx8uXk5mhnamJh'+
'Y1xcXZGQklZVV29vcHl4eoyLjKqpq6Wlpl1dXuXk6AAAAAAAAAAAAAAAAAAAAAAAAAAAA'+
'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'+
'AAACH5BAAAAAAALAAAAAAUABQAAAeZgGaCg4SFhoeIiYqKR1IWVgcyi4JMBiQqA0heQgG'+
'KQTFLPQgMCVocBIoNNqMgCQoDVReKYlELCwUFI1glEYorOgopWSwiTUVfih8dLzRTKA47'+
'Ek%2BKBGE8GEAhFQYuPooBOWAHY2ROExBbSt83QzMbVCdQST8Ck4QtZUQe9faCABlGrvD'+
'rB4ALDBMU%2BvnrUuOBQkE4NDycqCgQADs%3D'),
        textarea);
    textarea.parentNode.insertBefore(
        document.createElement('br'),
        textarea);
}
```

Download

- `zoomtextarea.user.js` <http://diveintogreasemonkey.org/download/zoomtextarea.user.js>

Further reading

- `Event` documentation <http://www.xulplanet.com/references/objref/Event.html>

See also

- Doing something for every instance of a specific HTML element
- Getting an element's style

- Setting an element's style
- Inserting content before an element
- Adding images without hitting a central server

5.8. Case study: Access Bar

Displaying accesskeys in a fixed status bar

Access Bar displays accesskeys defined on a web page. Accesskeys are an accessibility feature that allow you to jump to a specific link or form field with a keyboard shortcut defined by the page author. (Learn more about accesskeys <http://diveintoaccessibility.org/day_15_defining_keyboard_shortcuts.html>.)

Firefox supports accesskeys, but it does not display which keys are defined on a page. Thus, Access Bar was born.

Example: `accessbar.user.js` <http://diveintogreasemonkey.org/download/accessbar.user.js>

```
// ==UserScript==
// @name          Access Bar
// @namespace     http://diveintogreasemonkey.org/download/
// @description   show accesskeys defined on page
// @include       *
// ==/UserScript==

function addGlobalStyle(css) {
    var head, style;
    head = document.getElementsByTagName('head')[0];
    if (!head) { return; }
    style = document.createElement('style');
    style.type = 'text/css';
    style.innerHTML = css;
    head.appendChild(style);
}

var akeys, descriptions, a, desc, label, div;
akeys = document.evaluate(
    "//*[@accesskey]",
    document,
    null,
    XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
    null);
if (!akeys.snapshotLength) { return; }
descriptions = new Array();
desc = '';
for (var i = 0; i < akeys.snapshotLength; i++) {
    a = akeys.snapshotItem(i);
    desctext = '';
    if (a.nodeName == 'INPUT') {
        label = document.evaluate("//label[@for='" + a.name + "']",
            document,
            null,
            XPathResult.FIRST_ORDERED_NODE_TYPE,
            null).singleNodeValue;
        if (label) {
            desctext = label.title;
            if (!desctext) { desctext = label.textContent; }
        }
    }
    if (!desctext) { desctext = a.textContent; }
    if (!desctext) { desctext = a.title; }
    if (!desctext) { desctext = a.name; }
    if (!desctext) { desctext = a.id; }
    if (!desctext) { desctext = a.href; }
    if (!desctext) { desctext = a.value; }
    desc = '<strong>[' +
        a.getAttribute('accesskey').toUpperCase() + ']</strong> ';
    if (a.href) {
        desc += '<a href="' + a.href + '">' + desctext + '</a>';
    } else {
        desc += desctext;
    }
    descriptions.push(desc);
}
descriptions.sort();
div = document.createElement('div');
div.id = 'accessbar-div-0';
desc = '<div><ul><li class="first">' + descriptions[0] + '</li>';
for (var i = 1; i < descriptions.length; i++) {
    desc = desc + '<li>' + descriptions[i] + '</li>';
}
desc = desc + '</ul></div>';
div.innerHTML = desc;
document.body.style.paddingBottom = "4em";
window.addEventListener(
    "load",
    function() {
        document.body.appendChild(div);
    },
    true);
addGlobalStyle(
'#accessbar-div-0 {'+
'  position: fixed;' +
'  left: 0;' +
'  right: 0;' +
'  bottom: 0;' +
'  top: auto;' +
'  border-top: 1px solid silver;' +
'  background: black;' +
'  color: white;' +
'  margin: 1em 0 0 0;' +
'  padding: 5px 0 0.4em 0;' +
'  width: 100%;' +
'  font-family: Verdana, sans-serif;' +
'  font-size: small;' +
'  line-height: 160%;' +
'}' +
'#accessbar-div-0 a,' +
'#accessbar-div-0 li,' +
'#accessbar-div-0 span,' +
'#accessbar-div-0 strong {' +
'  background-color: transparent;' +
'  color: white;' +
'}' +
'#accessbar-div-0 div {' +
'  margin: 0 1em 0 1em;' +
'}' +
'#accessbar-div-0 div ul {' +
'  margin-left: 0;' +
'  margin-bottom: 5px;' +
'  padding-left: 0;' +
'  display: inline;' +
'}' +
'#accessbar-div-0 div ul li {' +
'  margin-left: 0;' +
'  padding: 3px 15px;' +
'  border-left: 1px solid silver;' +
'  list-style: none;' +
'  display: inline;' +
'}' +
'#accessbar-div-0 div ul li.first {' +
'  border-left: none;' +
'  padding-left: 0;' +
'}');
```

The code breaks down into six steps:

1. Define a helper function, `addGlobalStyle`
2. Find all the page elements that include an `accesskey` attribute
3. Loop through those elements and determine the most appropriate text to describe each element
4. Construct a sorted list of links to the `accesskey`−enabled elements
5. Add the sorted list to the page, using standard `ul` and `li` elements
6. Style the list so it appears as a fixed faux−status−bar at the bottom of the viewport

First things first. I need a helper function, `addGlobalStyle`, to inject my own CSS styles (used in step 6). See Adding CSS styles for more details on this pattern.

```
function addGlobalStyle(css) {
    var head, style;
    head = document.getElementsByTagName('head')[0];
    if (!head) { return; }
    style = document.createElement('style');
    style.type = 'text/css';
    style.innerHTML = css;
    head.appendChild(style);
}
```

Step 2 gets a list of all the page elements that contain an `accesskey` attribute. This is easy with Firefox's XPath support. Note that if the XPath query returns no results, I simply bail, since there will be nothing to display. See Doing something for every element with a certain attribute for more information on this pattern.

```
var akeys, descriptions, a, i, desc, label, div;
akeys = document.evaluate(
    "//*[@accesskey]",
    document,
    null,
    XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,
    null);
if (!akeys.snapshotLength) { return; }
```

Step 3, constructing the list of suitable descriptions for each `accesskey`−enabled element, is the most complicated part of the script. The problem is that `accesskey` attributes can appear in several different HTML elements.

An `input` element in a form can define an `accesskey`. `input` elements may or may not have an associated `label` that contains an associated text label for the `input` field. If so, the `label` may contain a `title` attribute that gives even more detailed information about the `input` field. Or the `label` attribute may simply contain text. Or the `input` element may have no associated label at all, in which case the `value` attribute of the `input` element is the best I can do.

On the other hand, the `label` itself can define the `accesskey`, instead of the `input` element the label describes. Again, I'll look for a description the `title` attribute of the `label` element, but fall back to the text of the label if no `title` attribute is present.

A link can also define an `accesskey` attribute. If so, the link text is the obvious choice. But if the link has no text (for example, if it only contains an image), then the link's `title` attribute is the next place to look. If the link contains no text and no `title`, I fall back to the link's `name` attribute, and failing that, the link's `id` attribute.

Ain't heuristics a bitch? Here's the full algorithm. Remember, `akeys` is an `XPathResult` object, so I need to get each result by calling `akeys.snapshotItem(i)`.

```
descriptions = new Array();
desc = '';
for (var i = 0; i < akeys.snapshotLength; i++) {
    a = akeys.snapshotItem(i);
    desctext = '';
    if (a.nodeName == 'INPUT') {
        label = document.evaluate("//label[@for='" + a.name + "']",
            document,
            null,
            XPathResult.FIRST_ORDERED_NODE_TYPE,
            null).singleNodeValue;
        if (label) {
            desctext = label.title;
            if (!desctext) { desctext = label.textContent; }
        }
    }
    if (!desctext) { desctext = a.textContent; }
    if (!desctext) { desctext = a.title; }
    if (!desctext) { desctext = a.name; }
    if (!desctext) { desctext = a.id; }
    if (!desctext) { desctext = a.href; }
    if (!desctext) { desctext = a.value; }
    desc = '<strong>[' +
        a.getAttribute('accesskey').toUpperCase() + ']</strong> ';
    if (a.href) {
        desc += '<a href="' + a.href + '">' + desctext + '</a>';
    } else {
        desc += desctext;
    }
    descriptions.push(desc);
}
```

Step 4 is simple, since Javascript arrays have a `sort` method that sorts the array in place.

```
descriptions.sort();
```

Step 5 creates the HTML to render the list of `accesskey`−enabled elements. I create a wrapper `<div>`, construct the HTML for the list of accesskeys as a string, set the wrapper `<div>`'s `innerHTML` property, and finally add it to the end of the page. Because of when user scripts are executed, complex changes to a page should be delayed until after the page is done loading, so I use `window.addEventListener` to add an `onload` event that adds the wrapper `<div>` to the page.

See Inserting complex HTML quickly for more information on the use of `innerHTML`, and Post−processing a page after it renders for information on the use of `window.addEventListener`.

```
div = document.createElement('div');
div.id = 'accessbar-div-0';
desc = '<div><ul><li class="first">' + descriptions[0] + '</li>';
for (var i = 1; i < descriptions.length; i++) {
    desc = desc + '<li>' + descriptions[i] + '</li>';
}
desc = desc + '</ul></div>';
div.innerHTML = desc;
```

```
document.body.style.paddingBottom = "4em";
window.addEventListener(
    "load",
    function() {
        document.body.appendChild(div);
    },
    true);
```

Finally, in step 6, I add my own set of CSS declarations to the page so that the HTML I'm injecting will look pretty. (Specifically, it will appear as a fixed black bar along the bottom of the page, that stays in view even as you scroll the page. This is made possible by Firefox's support for the `position:fixed` display type.) See Adding CSS styles for more information.

```
addGlobalStyle(
'#accessbar-div-0 {'+
'  position: fixed;' +
'  left: 0;' +
'  right: 0;' +
'  bottom: 0;' +
'  top: auto;' +
'  border-top: 1px solid silver;' +
'  background: black;' +
'  color: white;' +
'  margin: 1em 0 0 0;' +
'  padding: 5px 0 0.4em 0;' +
'  width: 100%;' +
'  font-family: Verdana, sans-serif;' +
'  font-size: small;' +
'  line-height: 160%;' +
'}' +
'#accessbar-div-0 a,' +
'#accessbar-div-0 li,' +
'#accessbar-div-0 span,' +
'#accessbar-div-0 strong {' +
'  background-color: transparent;' +
'  color: white;' +
'}' +
'#accessbar-div-0 div {' +
'  margin: 0 1em 0 1em;' +
'}' +
'#accessbar-div-0 div ul {' +
'  margin-left: 0;' +
'  margin-bottom: 5px;' +
'  padding-left: 0;' +
'  display: inline;' +
'}' +
'#accessbar-div-0 div ul li {' +
'  margin-left: 0;' +
'  padding: 3px 15px;' +
'  border-left: 1px solid silver;' +
'  list-style: none;' +
'  display: inline;' +
'}' +
'#accessbar-div-0 div ul li.first {' +
'  border-left: none;' +
'  padding-left: 0;' +
'}');
```

Download

- `accessbar.user.js` <http://diveintogreasemonkey.org/download/accessbar.user.js>

See also

- Adding CSS styles
- Doing something for every element with a certain attribute
- Inserting complex HTML quickly
- Post−processing a page after it renders

Chapter 6. Advanced Topics

6.1. Storing and retrieving persistent data

Greasemonkey defines two functions, `GM_setValue` and `GM_getValue`, that allow user scripts to store and retrieve "private" data that only the user script can access. (Other scripts can't access them, not even other user scripts.) You can use these functions to store script−specific configuration, to maintain a cache that persists between pages, or to keep a permanent activity log.

> **Note:**
>
> Data stored with `GM_setValue` and retrieved with `GM_getValue` are similar to browser cookies, but there are important differences. Both are stored on the local machine, but while cookies are *domain−specific* and can only be accessed from their originating domain, Greasemonkey configuration values are *script−specific* and can only be accessed by the user script that created them (regardless of the URL that the user script is currently running on). And unlike cookies, user script data are never transmitted to remote servers.

`GM_setValue` stores a script−specific configuration value, and `GM_getValue` retrieves it.

```
function GM_setValue(key, value);
```

```
function GM_getValue(key, defaultValue);
```

`key` argument is a string of no fixed format. *`value`* may be a string, boolean, or integer. The *`defaultValue`* argument of `GM_getValue` is optional; if present, it will be returned if the requested *`key`* does not exist. If no *`defaultValue`* is given and the requested *`key`* does not exist, `GM_getValue` will return `undefined`.

These functions were introduced in Greasemonkey 0.3. You should test whether they exist and degrade gracefully if they do not.

Further reading

- MyPIPsTag <http://dunck.us/code/greasemonkey/mypipstag.user.js> prompts you for a username the first time you run it.
- POST Interceptor <http://kailasa.net/prakash/greasemonkey/post−interceptor.user.js> adds menu items (with `GM_registerMenuCommand`) to toggle whether the script is active.
- MSDN Language Filter <http://blog.monstuff.com/archives/images/MSDNLanguageFilter.user.js> inserts its configuration options into the page itself.

See also

- GM_getValue
- GM_setValue

6.2. Adding items to the menubar

Greasemonkey defines a function, `GM_registerMenuCommand`, that allows user scripts to add menu items to the Firefox menu bar. Registered menu items appear in the ***User Script Commands*** submenu when the user script is active.

```
function GM_registerMenuCommand(menuText, callbackFunction);
```

`menuText` is a string, the text of the menu item to add. *`callbackFunction`* is a function object. When the menu item is selected, the `callbackFunction` function is called.

This function was introduced in Greasemonkey 0.2.6. You should test whether it exists and degrade gracefully if it does not.

Further reading

- POST Interceptor <http://kailasa.net/prakash/greasemonkey/post−interceptor.user.js> adds menu items to toggle whether the script is active.

See also

- GM_registerMenuCommand

6.3. Integrating data from other sites

Greasemonkey defines a function, `GM_xmlhttpRequest`, that allows user scripts to `GET` data from any URL or `POST` data to any URL.

See GM_xmlhttpRequest for more details and examples.

This function was introduced in Greasemonkey 0.2.6. You should test whether it exists and degrade gracefully if it does not.

Further reading

- LibraryLookup <http://weblog.infoworld.com/udell/gems/LibraryLookup.user.js> integrates Amazon.com <http://www.amazon.com/> with your local library.
- Annotate Google <http://ponderer.org/download/annotate_google.user.js> integrates Google <http://www.google.com/> and del.icio.us <http://del.icio.us/>.
- Bloglines Tweaks <http://persistent.info/greasemonkey/bloglines.user.js> adds an ***Expand*** button to article summaries in Bloglines <http://www.bloglines.com/> to retrieve and display the full article.
- Flick Batch Enhancer <http://lachnlone.no−ip.info/greasemonkey/flick_batch.user.js> uses `GM_xmlhttpRequest` to add features to Flickr <http://www.flickr.com/> using Flickr's own REST API.
- Hide Google Redirects <http://www.cs.toronto.edu/~james/greasemonkey/hide−google−redirects.user.js> reprograms Google Personal Search History <http://www.google.com/searchhistory/> to use normal `<a href="...">` links, but still track clicks by calling `GM_xmlhttpRequest` on the appropriate tracking URL.

6.4. Compiling your user script into an extension

Has your user script "outgrown" Greasemonkey's architecture? Does your user script need access to privileged Javascript functions, local files, or other Firefox features that are only available to full−fledged browser extensions? You can turn your user script into a full−fledged `XPI` with just a few clicks, thanks to Adrian Holovaty's amazing Greasemonkey compiler <http://www.letitblog.com/greasemonkey−compiler/>!

Procedure: Compile Butler into a browser extension

1. Visit the Butler user script source code <http://diveintomark.org/projects/butler/butler.user.js>. From the menu, select ***Edit***−>***Select All*** (***Ctrl−A***) to copy the user script's source code to the clipboard.
2. Visit the Greasemonkey compiler <http://www.letitblog.com/greasemonkey−compiler/>.
3. Go to the "Javascript" field. From the menu, select ***Edit***−>***Paste*** (***Ctrl−V***) to paste the Butler user script source code.
4. In the "Creator" field, enter ***`Mark Pilgrim`***.
5. In the "Version" field, enter the current version of the Butler user script (0.3 as of this writing, but check the script's metadata section for the actual version).
6. Open a new window or tab and visit GUID Generator <http://extensions.roachfiend.com/cgi−bin/guid.pl> to generate a random GUID. Copy the GUID to the clipboard, including the curly braces.
7. Switch back to the "Greasemonkey compiler" page. In the "GUID" field, paste the GUID you got from the GUID Generator.
8. In the "Homepage" field, enter ***`http://diveintomark.org/projects/butler/`***.
9. Click ***Create the Firefox extension***. Firefox should pop up a dialog titled "Opening butler.xpi". Select ***Save to disk*** and select a directory.

Tada! You now have a version of Butler as a browser extension. Before you install the Butler extension, you should first disable the Butler user script and verify that it is disabled by searching for something on Google <http://www.google.com/>. Then install the Butler extension by selecting ***File***−>***Open...*** and selecting the `butler.xpi` file you created with the Greasemonkey compiler. Be sure to restart your browser to finish the installation.

An `.xpi` file is really just a ZIP archive with a certain directory structure. You can use any ZIP program (such as 7−zip <http://www.7−zip.org/> for Windows or Stuffit Expander <http://www.stuffit.com/> for Mac OS X) to decompress the archive and look at the files that constitute the browser extension.

```
butler.xpi
|
+-- install.rdf
|
+-- chrome/
    |
    +-- butler/
        |
        +-- content/
            |
            +-- browser.xul
            |
            +-- contents.rdf
            |
            +-- javascript.js
```

There are four files involved. The two RDF files are mostly Firefox boilerplate. The other two contain the actual code.

`install.rdf`
: Metadata about the extension itself, including name, version, description, and compatible versions of Firefox.

`browser.xul`
: Bootstrap code that checks the current URL against the script's `@include` and `@exclude` parameters, and then injects and executes the script.

`contents.rdf`
: More metadata, Firefox boilerplate.

`javascript.js`
: The original user script's source code.

Basically, the Greasemonkey compiler creates a really, really stripped down version of Greasemonkey that has no user interface and automatically loads a single user script on the appropriate URLs. But now you have all the source files, and you can do anything you want: create custom dialogs, modify preference panes, register menu items outside the ***User Script Commands*** menu... go wild!

Further reading

- Extension Developer's Extension <http://ted.mielczarek.org/code/mozilla/extensiondev/> is invaluable for debugging and testing Firefox extensions.

Greasemonkey API Reference

Name

GM_log   log messages to the JavaScript Console

Synopsis

```
function GM_log(message);
```

Description

`GM_log` places output in the JavaScript Console. It is primarily used for debugging purposes.

History

`GM_log` was introduced in Greasemonkey 0.3.

See also

- Logging with `GM_log`
- Tracking crashes with JavaScript Console

Name

GM_getValue   get script−specific configuration value

Synopsis

```
returntype GM_getValue(key, defaultValue);
```

Description

`GM_getValue` retrieves a script−specific configuration value. The return value may be a string, boolean, or integer. The *`key`* argument is a string of no fixed format. The *`defaultValue`* argument is optional; if present, it will be returned if the requested *`key`* does not exist. If no *`defaultValue`* is given and the requested *`key`* does not exist, `GM_getValue` will return `undefined`.

Greasemonkey configuration values are similar to browser cookies, but there are important differences. Both are stored on the local machine, but while cookies are domain−specific and can only be accessed from their originating domain, Greasemonkey configuration values are script−specific and can only be accessed by the user script that created them (regardless of the address the user script is currently running on). And unlike cookies, configuration values are never transmitted to remote servers.

Tip:

You can see all the stored configuration values by visiting `about:config` and filtering on `greasemonkey.scriptvals`.

History

`GM_getValue` was introduced in Greasemonkey 0.3.

See also

- GM_setValue

Name

GM_setValue   set script−specific configuration value

Synopsis

```
function GM_setValue(key, value);
```

Description

`GM_setValue` stores a script−specific configuration value. The *`key`* argument is a string of no fixed format. The *`value`* argument can be a string, boolean, or integer. Both arguments are required.

Greasemonkey configuration values are similar to browser cookies, but there are important differences. Both are stored on the local machine, but while cookies are domain−specific and can only be accessed from their originating domain, Greasemonkey configuration values are script−specific and can only be accessed by the user script that created them (regardless of the address the user script is currently running on). And unlike cookies, configuration values are never transmitted to remote servers.

History

`GM_getValue` was introduced in Greasemonkey 0.3.

See also

- GM_getValue

Name

GM_registerMenuCommand   add a menu item to the ***User Script Commands*** submenu

Synopsis

`function` **`GM_registerMenuCommand`**`(` *menuText*,
callbackFunction`);`

Description

`GM_registerMenuCommand` adds a menu item to the ***Tools*->*User Script Commands*** menu. The *`menuText`* argument is a string, the text of the menu item to add. The *`callbackFunction`* argument is a function object. When the menu item is selected, the `callbackFunction` function is called.

`function` **`callbackFunction`**`(`*`e`*`);`

The *`e`* parameter is the menu selection event. I'm not sure if this is useful for anything.

Bugs

There is no way to set a menu accelerator key. Calling `GM_registerMenuCommand('Some &menu text',` *`myFunction`*`)` will create a menu item titled "`Some &menu text`". (Bug 10090 <http://bugzilla.mozdev.org/show_bug.cgi?id=10090>)

History

`GM_registerMenuCommand` was introduced in Greasemonkey 0.2.6.

Name

GM_xmlhttpRequest make an arbitrary HTTP request

Synopsis

`GM_xmlhttpRequest``(`*details*`);`

Description

`GM_xmlhttpRequest` makes an arbitrary HTTP request. The *`details`* argument is an object that can contain up to seven fields.

method
: a string, the HTTP method to use on this request. *Required.* Generally `GET`, but can be any HTTP verb, including `POST`, `PUT`, and `DELETE`.

url
: a string, the URL to use on this request. *Required.*

headers
: an associative array of HTTP headers to include on this request. Optional, defaults to an empty array. Example:

```
        headers: {'User-Agent': 'Mozilla/4.0 (compatible) Greasemonkey',
                  'Accept': 'application/atom+xml,application/xml,text/xml'}
```

data
: a string, the body of the HTTP request. Optional, defaults to an empty string. If you are simulating posting a form (`method` == `'POST'`), you must include a `Content-type` of `'application/x-www-form-urlencoded'` in the `headers` field, and include the URL−encoded form data in the `data` field.

`onload`
: a function object, the callback function to be called when the request has finished successfully.

`onerror`
: a function object, the callback function to be called if an error occurs while processing the request.

`onreadystatechange`
: a function object, the callback function to be called repeatedly while the request is in progress.

`onload` callback

The callback function for `onload` takes a single parameter, *`responseDetails`*.

`function` **`onloadCallback`**`(`*responseDetails*`)`;

`responseDetails` is an object with five fields.

status
: an integer, the HTTP status code of the response. `200` means the request completed normally.

statusText
: a string, the HTTP status text. Status text is server−dependent.

responseHeaders
: a string, the HTTP headers included in the response.

responseText
: a string, the body of the response.

readyState
: unused

`onerror` callback

The callback function for `onerror` takes a single parameter, *`responseDetails`*.

`function` **`onerrorCallback`**`(`*responseDetails*`)`;

`responseDetails` is an object with five fields.

status
: an integer, the HTTP error code of the response. `404` means the page was not found.

statusText
: a string, the HTTP status text. Status text is server−dependent.

responseHeaders
: a string, the HTTP headers included in the response.

responseText
: a string, the body of the response. The body of an HTTP error page is server−dependent.

readyState

unused

`onreadystatechange` callback

The callback function for `onreadystatechange` is called repeatedly while the request is in progress. It takes a single parameter, *`responseDetails`*.

`function` **`onreadystatechangeCallback`**`(` *responseDetails*`);`

`responseDetails` is an object with five fields. The `responseDetails.readyState` denotes what stage the request is currently in.

status
: an integer, the HTTP status code of the response. This will be `0` when `responseDetails.readyState` < 4.

statusText
: a string, the HTTP status text. This will be an empty string when `responseDetails.readyState` < 4.

responseHeaders
: a string, the HTTP headers included in the response. This will be an empty string when `responseDetails.readyState` < 4.

responseText
: a string, the body of the response. This will be an empty string when `responseDetails.readyState` < 4.

readyState
: an integer, the stage of the HTTP request.

1
: loading. The request is being prepared.

2
: loaded. The request is ready to be sent to the server, but nothing has been sent yet.

3
: interactive. The request has been sent and the client is waiting for the server to finish sending data.

4
: complete. The request is completed and all response data is available in other fields.

Examples

The following code fetches the Atom feed from http://greaseblog.blogspot.com/ and displays an alert with the results.

```
GM_xmlhttpRequest({
    method: 'GET',
    url: 'http://greaseblog.blogspot.com/atom.xml',
    headers: {
        'User-agent': 'Mozilla/4.0 (compatible) Greasemonkey',
        'Accept': 'application/atom+xml,application/xml,text/xml',
    },
    onload: function(responseDetails) {
        alert('Request for Atom feed returned ' + responseDetails.status +
              ' ' + responseDetails.statusText + '\n\n' +
              'Feed data:\n' + responseDetails.responseText);
    }
});
```

Bugs

The `onreadystatechange` callback does not work properly with `readyState < 4`.

Notes

Unlike the `XMLHttpRequest` object, `GM_xmlhttpRequest` is not restricted to the current domain; it can `GET` or `POST` data from any URL.

History

`GM_xmlhttpRequest` was introduced in Greasemonkey 0.2.6.

Futher reading

- `XMLHttpRequest` support in Mozilla <http://www.xulplanet.com/references/objref/XMLHttpRequest.html>
- `XMLHttpRequest` support in Internet Explorer <http://msdn.microsoft.com/library/default.asp?url=/library/en−us/xmlsdk/html/xmobjpmexmlhttprequest.asp>
- `XMLHttpRequest` support in Safari <http://developer.apple.com/internet/webcontent/xmlhttpreq.html>
- HTTP status codes <http://www.w3.org/Protocols/rfc2616/rfc2616−sec10.html>
- RFC 2616 <http://www.w3.org/Protocols/rfc2616/rfc2616.html>

List of "further reading" links

1. Greasemonkey script repository <http://dunck.us/collab/GreaseMonkeyUserScripts> contains hundreds of Greasemonkey scripts. [1.3. Installing a user script]
2. `tag:` URIs <http://taguri.org/> [2.2. Describing your user script with metadata]
3. Anonymous functions in Javascript <http://novemberborn.net/sifr/explained/terminology> [2.3. Coding your user script]
4. Block Scope in Javascript <http://youngpup.net/2004/jsblockscope> and associated discussion thread <http://youngpup.net/2004/jsblockscope/comments> [2.3. Coding your user script]
5. Introduction to the DOM Inspector <http://www.brownhen.com/DI.html> [3.3. Inspecting elements with DOM Inspector]
6. Inspect Element extension [3.3. Inspecting elements with DOM Inspector]
7. Inspector Widget extension <http://www.projectit.com/freestuff.html>, an alternative to the Inspect Element extension that adds a toolbar button instead of a context menu item. [3.3. Inspecting elements with DOM Inspector]
8. Web Developer Extension <http://chrispederick.com/work/firefox/webdeveloper/> contains a plethora of functions for deconstructing pages. [3.5. Other debugging tools]
9. Aardvark <http://www.karmatics.com/aardvark/> interactively displays tag names, `id` and `class` attributes. [3.5. Other debugging tools]
10. Venkman Javascript Debugger <http://www.hacksrus.com/~ginda/venkman/> is a complete run−time Javascript debugger. [3.5. Other debugging tools]
11. Web Development Bookmarklets <http://www.squarefree.com/bookmarklets/webdevel.html> contains a number of useful functions you can drag to your toolbar. [3.5. Other debugging tools]
12. JSUnit <http://www.edwardh.com/jsunit/> is a unit testing framework for Javascript. [3.5. Other debugging tools]
13. js−lint <http://www.crockford.com/javascript/lint.html> checks Javascript code for common errors. [3.5. Other debugging tools]
14. Mozilla XPath documentation <http://www−jcsu.jesus.cam.ac.uk/~jg307/mozilla/xpath−tutorial.html> [4.6. Doing something for every element with a certain attribute]
15. XPath tutorial by example <http://www.zvon.org/xxl/XPathTutorial/General/examples.html> [4.6. Doing something for every element with a certain attribute]
16. XPathResult reference <http://www.xulplanet.com/references/objref/XPathResult.html> [4.6. Doing something for every element with a certain attribute]
17. CSS properties <http://www.westciv.com/style_master/academy/css_tutorial/properties/> [4.15. Setting an element's style]
18. Javascript event compatibility tables <http://www.quirksmode.org/js/events_compinfo.html> [4.22. Overriding a built−in Javascript method]
19. Javascript−DOM prototypes in Mozilla <http://www.mozilla.org/docs/dom/mozilla/protodoc.html> [4.22. Overriding a built−in Javascript method]
20. Displaying Event object constants <http://www.mozilla.org/docs/dom/domref/examples.html#999002> [4.22. Overriding a built−in Javascript method]
21. `Event` documentation <http://www.xulplanet.com/references/objref/Event.html> [5.7. Case study: Zoom Textarea]
22. MyPIPsTag <http://dunck.us/code/greasemonkey/mypipstag.user.js> prompts you for a username the first time you run it. [6.1. Storing and retrieving persistent data]
23. POST Interceptor <http://kailasa.net/prakash/greasemonkey/post−interceptor.user.js> adds menu items (with `GM_registerMenuCommand`) to toggle whether the script is active. [6.1. Storing and retrieving persistent data]
24. MSDN Language Filter <http://blog.monstuff.com/archives/images/MSDNLanguageFilter.user.js> inserts its configuration options into the page itself. [6.1. Storing and retrieving persistent data]
25. POST Interceptor <http://kailasa.net/prakash/greasemonkey/post−interceptor.user.js> adds menu items to toggle whether the script is active. [6.2. Adding items to the menubar]

26. LibraryLookup <http://weblog.infoworld.com/udell/gems/LibraryLookup.user.js> integrates Amazon.com <http://www.amazon.com/> with your local library. [6.3. Integrating data from other sites]
27. Annotate Google <http://ponderer.org/download/annotate_google.user.js> integrates Google <http://www.google.com/> and del.icio.us <http://del.icio.us/>. [6.3. Integrating data from other sites]
28. Bloglines Tweaks <http://persistent.info/greasemonkey/bloglines.user.js> adds an ***Expand*** button to article summaries in Bloglines <http://www.bloglines.com/> to retrieve and display the full article. [6.3. Integrating data from other sites]
29. Flick Batch Enhancer <http://lachnlone.no−ip.info/greasemonkey/flick_batch.user.js> uses `GM_xmlhttpRequest` to add features to Flickr <http://www.flickr.com/> using Flickr's own REST API. [6.3. Integrating data from other sites]
30. Hide Google Redirects <http://www.cs.toronto.edu/~james/greasemonkey/hide−google−redirects.user.js> reprograms Google Personal Search History <http://www.google.com/searchhistory/> to use normal `<a href="...">` links, but still track clicks by calling `GM_xmlhttpRequest` on the appropriate tracking URL. [6.3. Integrating data from other sites]
31. Extension Developer's Extension <http://ted.mielczarek.org/code/mozilla/extensiondev/> is invaluable for debugging and testing Firefox extensions. [6.4. Compiling your user script into an extension]
32. `XMLHttpRequest` support in Mozilla <http://www.xulplanet.com/references/objref/XMLHttpRequest.html> []
33. `XMLHttpRequest` support in Internet Explorer <http://msdn.microsoft.com/library/default.asp?url=/library/en−us/xmlsdk/html/xmobjpmexmlhttprequest.asp> []
34. `XMLHttpRequest` support in Safari <http://developer.apple.com/internet/webcontent/xmlhttpreq.html> []
35. HTTP status codes <http://www.w3.org/Protocols/rfc2616/rfc2616−sec10.html> []
36. RFC 2616 <http://www.w3.org/Protocols/rfc2616/rfc2616.html> []

List of tips

1. Many user scripts are available at the Greasemonkey script repository <http://dunck.us/collab/GreaseMonkeyUserScripts>, although there is no requirement to list your scripts there. You may host (and users may install) your script from anywhere. You don't even need a web server; you can install a user script from a local file. [1.3. Installing a user script]
2. You can specify the items of your user script metadata in any order. I like `@name`, `@namespace`, `@description`, `@include`, and finally `@exclude`, but there is nothing special about this order. [2.2. Describing your user script with metadata]
3. When you click ***Edit*** in the "Manage User Scripts" dialog, you are "live" editing a copy of your script that is buried deep inside your Firefox profile directory. I've gotten into the habit, once I've finished a "live" editing session, of going back to my text editor one last time and selecting ***File***−>***Save as...***, and saving the user script in another directory. While it's not necessary (Greasemonkey only pays attention to the copy in your profile directory), I prefer to keep the "master copy" of my scripts in another directory with the rest of my work. [2.4. Editing your user script]
4. In JavaScript Console, you can right−click (Mac users control−click) on any line and select ***Copy*** to copy it to the clipboard. [3.2. Logging with `GM_log`]
5. Once you have a reference to an element (like `thisElement`), you can use `thisElement.nodeName` to determine its HTML tag name. If the page is served as `text/html`, the tag name is always returned as uppercase, regardless of how it was specified in the original page. However, if the page is served as `application/xhtml+xml`, the tag name is always lowercase. I always use `thisElement.nodeName.toUpperCase()` and forget about it. [4.6. Doing something for every element with a certain attribute]
6. Inserting new content before `someExistingElement.nextSibling` will work even if `someExistingElement` is the last child of its parent (i.e. it has no next sibling). In this case, `someExistingElement.nextSibling` will return `null`, and the `insertBefore` function will simply append the new content after all other siblings. (If this doesn't make any sense to you, don't worry about it. The point is that this example will always work, even when it seems like it shouldn't.) [4.8. Inserting content after an element]
7. If all you want to do is remove ads, it's probably easier to install AdBlock <http://adblock.mozdev.org/> and import an up−to−date filter list <http://www.geocities.com/pierceive/adblock/> than to write your own user script. [4.9. Removing an element]
8. Use the `data:` URI kitchen <http://software.hixie.ch/utilities/cgi/data/data> to construct your own `data:` URLs. [4.12. Adding images without hitting a central server]
9. You can use the `addGlobalStyle` function to style elements that you insert into a page, or elements that were part of the original page. However, if you are styling existing elements, you should use the `!important` keyword for each rule you define to ensure your styles override the rules defined by the original page. [4.13. Adding CSS styles]
10. Zapping and replacing `document.body.innerHTML` does not change everything about the page. Anything defined in the `<head>` of the original page will still be in effect, including the page title, CSS styles, and scripts. You can modify or remove these separately. [4.16. Post−processing a page after it renders]
11. A `submit` event fires when a user submits a form in the usual way, i.e. by clicking the form's ***Submit*** button or hitting ***ENTER*** within a form. However, the `submit` event *does not* fire when the form is submitted via a script calling `aForm.submit()`. Therefore you need to do two things to capture a form submission: add an event listener to capture to `submit` event, *and* modify the prototype of the `HTMLFormElement` class to redirect the `submit()` method to your custom function. [4.22. Overriding a built−in Javascript method]
12. You can see all the stored configuration values by visiting `about:config` and filtering on `greasemonkey.scriptvals`. []

List of examples

List of procedures

Revision history

Revision History	
Revision 2005−05−09	2005−05−09
• Added Case study: Zoom Textarea. • Added Storing and retrieving persistent data. • Added Adding items to the menubar. • Added Integrating data from other sites. • Added Compiling your user script into an extension.	
Revision 2005−05−06	2005−05−06
• Added Case study: Frownies. • Changed `void` to `function` in Greasemonkey API Reference. ("You keep using that word. I do not think it means what you think it means.")	
Revision 2005−05−05	2005−05−05
• Added What is Greasemonkey?. Thanks, Dennis. • Added Case study: Dumb Quotes. • Added downloadable Palm OS" database for reading on mobile devices.	
Revision 2005−05−04	2005−05−04
• Added Parsing XML. • Added Case study: Offsite Blank.	
Revision 2005−05−03	2005−05−03
• Compressed videos further. • Added videos to downloadable HTML distribution.	
Revision 2005−05−02	2005−05−02
• Added GM_xmlhttpRequest. • Added Case study: Ain't It Readable. • Refactored Offsite Blank <http://diveintogreasemonkey.org/download/offsiteblank.user.js>. • Updated code examples to Greasemonkey 0.3 format.	
Revision 2005−05−01	2005−05−01
• Added Greasemonkey API Reference. • Incorporated copious feedback from Simon Willison. Thanks, Simon. • Incorporated copious feedback from Jeremy Dunck. Thanks, Jeremy.	
Revision 2005−04−30	2005−04−30
• Added Getting Started. • Added Your First User Script. • Added Debugging User Scripts. • Added videos (currently online only).	
Revision 2005−04−25	2005−04−25

- Initial publication

About this book

I wrote this book in DocBook XML <http://wiki.docbook.org/topic/DocBook> using Emacs <http://www.gnu.org/software/emacs/emacs.html>. SAXON <http://saxon.sourceforge.net/> converted it to HTML (with a customized version of Norman Walsh's XSL stylesheets <http://docbook.sourceforge.net/projects/xsl/index.html>). HTMLDoc <http://www.easysw.com/htmldoc/> converted it to PDF. w3m <http://w3m.sourceforge.net/> converted it to plain text. Plucker Distiller <http://www.plkr.org/> converted it to a Palm OS" database so you can read it on mobile devices. An Ant <http://ant.apache.org/> script automates the entire process.

I created the supplementary videos with CamStudio <http://www.swftools.com/tools−details.php?tool=8162413051> and TMPGEnc <http://www.tmpgenc.net/>.

Dean Edwards wrote js−highlight <http://dean.edwards.name/my/behaviors/#js−highlight.htc>, which provides automatic syntax highlighting of the Javascript examples.

If you're interested in learning more about DocBook for technical writing, you can download the XML source <http://diveintogreasemonkey.org/download/diveintogreasemonkey−2005−05−09−xml.zip>, which includes the version of DocBook and all the XSL stylesheets and build scripts I use to create the different versions of the book. You should also read the canonical book, *DocBook: The Definitive Guide* <http://www.docbook.org/tdg/en/html/docbook.html>. You can also subscribe to the DocBook mailing lists <http://www.oasis−open.org/docbook/mailinglist/>.

GNU General Public License

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.

Free Software Foundation, Inc.
 59 Temple Place, Suite 330,
 Boston,
 MA
 02111−1307
 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Version 2, June 1991

1. Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software − to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:

1. copyright the software, and
2. offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

2. Terms and conditions for copying, distribution, and modification

Section 0

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program " means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification ".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

Section 1

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

Section 2

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License.

    Note: Exception:

    If the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

Section 3

You may copy and distribute the Program (or a work based on it, under Section 2 in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

1. Accompany it with the complete corresponding machine−readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine−readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

Section 4

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Section 5

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate