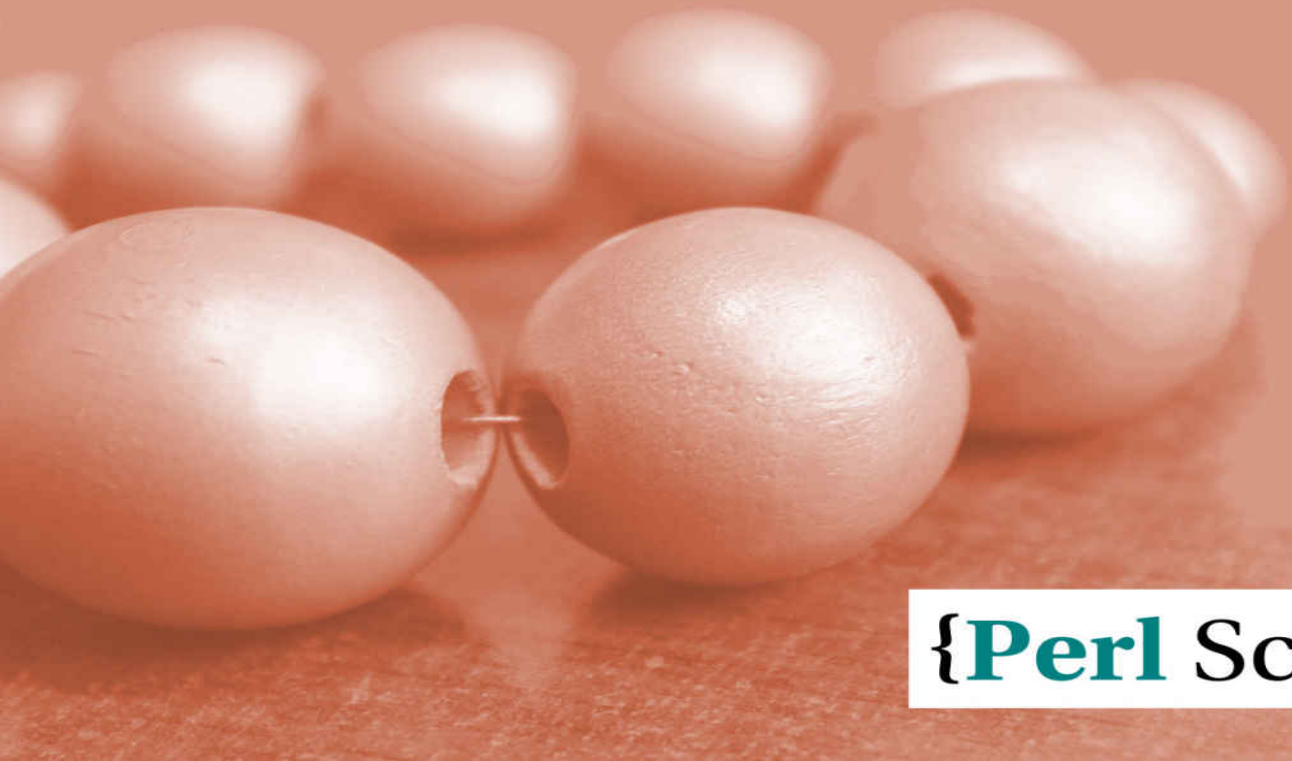
Selenium and Perl

Driving browsers with Perl

John Davies



{Perl School}

Selenium and Perl

John Davies

Selenium and Perl

1 About …

1.1 … the text

Without knowing the specifics of the devices that might be used to read this, any proliferation of fonts and styles would risk confusing the reader rather than helping. There are therefore very few fonts in use throughout the document. The one appearing here is used for the text, while anything that should be typed verbatim, such as module names and code, `will appear in this font`. Links to URLs in the text should appear differently and be clickable if this is supported by the reader's device and software. A typical link is PerlMonks. Text is emphasised very rarely. *When it is, it appears like this*.

1.2 … the code examples

All code examples are meant to be small, self-contained, complete examples. They have been tested on a small number of machines and are believed to contain no bugs, although no warranty should be inferred.

Strict and warnings are included in all examples. There is no “shebang” line, partly to reduce space and partly to be consistent with pure Windows examples that do not need one. URLs and other string literals have usually been abstracted to variables. This is to make the code clearer. It is perfectly possible to have the URL as a literal in all the examples.

Commonly used variable names include `$driver` for a Selenium WebDriver object, `$elt` for element objects, `$rtn` for values, possibly including references, returned from various calls, `$pic` for generic graphics and `$b64` for base64 encoded objects. Hungarian prefixes are sometimes used, the commonest being `ar_`, indicating an array reference and `hr_` indicating a hash reference.

All the code examples are available from the Perl School web site.

1.3 … the series

The Perl School brand has its roots in a series of low-cost Perl training courses that Dave Cross ran in 2012. By running low-cost training at the weekend, he hoped to encourage more programmers to keep their Perl knowledge up to date. These courses were run regularly for about a year before the idea was put on hold for a while.

Dave always knew that he would want to return to the Perl School brand at some point and late in 2017 he realised what the obvious next step was - low-cost Perl books. He had already developed a pipeline for creating e-books from Markdown files so it was a short step to republishing some of his training materials as books.

The first Perl School book, Perl Taster was published at the end of 2017 (just in time for the London Perl Workshop) and this is the second. Dave has plans to publish more over the coming months.

If you are interested in writing a book for the Perl School range, then please get in touch. We are @perl_school on Twitter.

1.4 … the author

John is an accountant with ingrowing computers. He started punching cards in 1974 and has more accounting qualifications than anyone could possibly need. An unrepentant bookworm, he still has Pascal and COBOL manuals from the 1970s which he has been known to quote in answer to Perl problems in 2017. He yearns for the “good old days” when documentation came in glossy manuals and didn’t make him rant. His pipe dream is for software that doesn’t make him rant. Or even for accounting laws and standards that don’t. When not on PerlMonks, he can frequently be found at Lord’s, playing bridge or backgammon or watching cricket.

2 What is Selenium WebDriver?

Selenium WebDriver is a tool for automating web interaction. It can pass actions to a web browser and can therefore be used to reproduce actions for bug reporting, running tests or taking the user to part of a web site that cannot be bookmarked. Selenium interfaces exist for several languages and the commands are usually very close, meaning that someone trained to use Selenium with one language can move to another relatively easily. This document considers Perl only. Although JavaScript injection is demonstrated, this is done using Perl.

The machine running Perl and the machine running the browser may or may not be the same. Selenium is designed to cope with both situations.

There is a chain of interfaces that must be satisfied for any code using Selenium to work. Perl will link to a browser driver, possibly via a server. The driver will link to the browser and the browser will fetch and process a web page. If any link in this chain fails, the behaviour of the Perl code will change. Programmers are advised to disable automatic updates and to install copies of any link in this chain when updating. This will enable them to test which link has changed should anything break. The one link that cannot be guaranteed is the final one, the web page. If this is under the programmer's control, all may be well. This document will show examples of code that use third party web sites. If these change between the time of writing and the time of code execution, the results may not be as described in the text. The operating system may also be a factor. Drivers and browsers may not perform identically on different operating systems.

Note that there are separate products called “Selenium IDE” and “Selenium Grid”. While they are produced by the same people, they are different packages and not covered by this document.

References to “the documentation” should be understood to mean the information on cpan.org for the relevant module. “This document” is intended to mean the document currently being read, while all other references to documentation should be accompanied by a reference of some sort.

2.1 Perl

Perl 5.6.0 is specified as the minimum version in the code of `Selenium::Remote::Driver`. This has not been tested. The earliest version of Perl used to run any of the code shown is 5.18.2. Certain examples include `use feature 'say';`. This requires a Perl version of at least 5.10 (released in 2007). If an earlier version must be used, there should be no problem replacing the `say` statements with `print` statements of the form `print "$var\n";` or even writing a sub to implement `say`.

To run the examples, `Selenium::Remote::Driver` must be installed. It comes with a large dependency chain and may be part of such a chain itself. Chains that install it include `Selenium::Chrome` and `Selenium::Firefox`. It is currently at version 1.27. This version has new documentation about WC3 Webdriver compatibility, although a large amount of knowledge is assumed.

2.2 Browsers, Servers and Drivers

A browser is a package for reading a web page. A driver is a package for controlling a browser. A server is a package for returning the output of a browser or driver to a client. The Selenium Standalone Server can act as a driver for most browsers. Some browsers have dedicated drivers written that can act as servers for Perl and other languages under certain circumstances. This section will try to explain some of these complexities. Selenium lists its recommended driver downloads on its downloads page.

The browser name may be specified as part of the constructor. By default, Selenium will use Firefox, although connecting to the Selenium Standalone Server with an invalid browser name (using the wrong case, for example) seems to default to Chrome in at least some cases.

2.2.1 Selenium Standalone Server

This requires Java. To install Java on a Linux machine that uses some flavour of Debian, the first command is

```
sudo apt install –assume-yes default-jdk.
```

This will need raised privileges. The server can be downloaded from Selenium's downloads page, the same site as recommended for driver downloads. The latest version is 3.9.1 at the time of writing. The download is a .jar file. To run the server, the command is:

```
java -jar selenium-server-standalone-3.9.1.jar
```

Adding `--help` to this command brings up some useful information.

Selenium Standalone Server can act as a driver for browsers and it can act as a server on a remote machine, unlike almost all other

drivers. It can act as an interface not only between `Selenium::Remote::Driver` on one machine and a browser on another, but also for `Selenium::Chrome` and `Selenium::Firefox` if they are preferred to `Selenium::Remote::Driver`. Its default is to instantiate a Firefox browser.

Code to test which browsers are available for a specific instance of Selenium Standalone Server appears in section 14.7 below.

2.2.2 Chrome

This includes the Chromium family. A driver is needed and the latest driver will not support some relatively new versions of Chrome. Version 59 is known to be incompatible with the latest driver. Vivaldi is a fork of Chrome, but has not been tested.

Chrome is perhaps more compatible with Selenium than any other popular browser and has been used in all examples where there is no need to demonstrate browser specifics.

2.2.3 Open a Web Page with Chrome

Enter and run the code below:

```
use strict; use warnings;
use feature 'say';
use Selenium::Chrome;
my $url = 'http://www.perlmonks.org';

my $driver = Selenium::Chrome->new();
$driver->get($url);
say $driver->get_title();
$driver->quit();
```

The code should respond “PerlMonks – The Monastery Gates”.

2.2.4 Internet Explorer

Internet Explorer version 6 has not been supported since 2014. Unfortunately, even version 11 appears to be incompatible with `Selenium::Remote::Driver`. There is a driver download available from here, but although Selenium claims it works with versions from 7 to 11, the Perl binding seems to be incompatible. Investigation reveals that it is failing to connect to the driver even though other tools can connect. All attempts to run without Selenium Standalone Server (and very many options have been tried) result in the message "Selenium server did not return proper status". Again, many attempts have been made with Selenium Standalone Server. All of these result in an error message that contains much system information but not much that is helpful. The specifics of the error message are "Could not create new session: Unable to create new service" and "Driver info: driver.version: unknown".

2.2.5 Edge

Edge uses a different driver, MicrosoftWebDriver.exe, available from Microsoft's developer web site. It must be run explicitly and be listening when the Perl code is started. The default port is 17556. It is also necessary to make a change to one of the packages in the Selenium suite, namely `Selenium::Remote::RemoteConnection`. In versions 1.26 and 1.27, line 101 must be commented out. This inserts a prefix into the status request that the driver cannot process, causing the status to be NOTOK. This is a very ugly hack and cannot be recommended for production as it may break other browsers, although it is believed to be necessary for any hack that might make Internet Explorer work. Despite the ugliness, this hack makes the following code work as expected.

```
use strict; use warnings;
use feature 'say';
use Selenium::Remote::Driver;

my $driver = Selenium::Remote::Driver->new(
    'port' => 17556,
```

```
);
$driver->get('http://www.perlmonks.org');
say $driver->get_title();
$driver->quit();
```

2.2.6 Firefox

This needs a driver called Geckodriver. Firefox has another driver, Marionette, that comes with Firefox. It is not compatible with the WebDriver protocol used by Selenium and Geckodriver is the interface. It is possible to set options to use the marionette capability directly, but doing so has not been explored in this document.

Firefox is undergoing major revisions at the time of writing. These have caused many problems with add-ins and, at the time of writing, the latest driver/browser combination does not support the sending of arbitrary keystrokes from the Perl Selenium binding under Windows. Under Debian, the `quit` method fails. It cannot be replaced by sending keystrokes. Although Debian allows keys to be sent, Selenium sends them to an element as a POST action and not to the browser itself. In version 1.26, the browser instance was not closed. In version 1.27, the instance is closed after a delay, but there is still an error raised that causes the Perl code to abort unless trapped in an `eval` block.

Firefox's geckodriver is not recognised as an installable package by Debian. It can be downloaded from github. Go to Github's Geckodriver Releases page and choose the appropriate download for the target hardware and OS. On Linux, it should be unpacked to a directory in the path to avoid having to state the location in the parameters of every object. `/usr/local/bin` seems to be the recommended directory. At the time of writing, the latest version is 0.20.1. Once both Firefox and geckodriver are installed and both executables available in the path, the following code can be run to test them out.

```
use strict; use warnings;
use feature 'say';
use Selenium::Firefox;

my $driver = Selenium::Firefox->new();
$driver->get('http://www.perlmonks.org');
say $driver->get_title();
$driver->quit();
```

Saving this as a file and executing it from the command line should result in a browser instance appearing, the PerlMonks home page being fetched and its title appearing in the terminal. At the time of writing, error messages appear as well, warning of a failure in the cleanup routine. This appears to be in the Perl binding. Behaviour has changed from 1.26, when these were warnings and could be ignored as code continued to run and 1.27, where these are errors that cause the code to crash.

Running Firefox via Selenium Standalone Server is more problematic, as the failure to honour the `close` and `quit` commands leaves the browser open without allowing processing to continue. The only way forward is to close the browser manually, causing the client to crash.

The PaleMoon project is a fork of Firefox. No attempt has been made to drive it with Selenium.

2.2.7 Opera

The documentation on Selenium's web site about Opera is misleading. It points to a GitHub wiki page that Selenium says describes the steps needed to get Opera working. Unfortunately, the link seems to be inactive, as it redirects to the wiki home page and searching for Opera reveals there is nothing available. There is a Github page (at the time of writing) that contains drivers for common

operating systems. However, no combination of operating system and options seems to create a working implementation. The closest is Windows, which produces a window running the driver, but this rapidly disappears. The window from which the code was invoked gives the message “no such session”. Opera is not listed as one of the names of recognised browsers in the `Selenium::Remote::Driver` documentation, but Selenium Standalone Server lists it as one it understands.

2.2.8 HtmlUnit

HtmlUnit is a “headless” browser, creating nothing visible to the user, that requires Java to run. It does not support screen shots, but is very much faster than browsers that display a screen. The latest version of HtmlUnit is available from Sourceforge. Version 2.29 is dated 2017-12-28 and users should check for later versions and download the latest.

2.2.9 phantomjs

phantomjs is another headless browser, although it is no longer being maintained. It is therefore deprecated.

2.2.10 Safari

Safari requires a driver. According to the Selenium home page, the latest version is 2.48 (at the time of writing – as usual, it is wise to check for later versions) and can be downloaded from this web page. Going to the parent directory of this reveals a list of directories with later version numbers. A brief, but not exhaustive, review reveals no Safari driver in any of the later directories, but this may be subject to change. Further, it appears that since OSX Sierra, or about August 2016, the driver is included with the operating system, so it is worth checking before downloading. To check, bring up a terminal window and enter `safaridriver -h`. This will indicate whether or not the driver is present, but there appears to be no option to identify the version number. It is probably safe to assume that the version will be

appropriate for the browser, but its compatibility with `Selenium::Remote::Driver` cannot be guaranteed.

While it is necessary to have the driver installed, it need not be started explicitly. It is necessary to have Selenium Standalone Server running. This will listen on port 4444 and will start its own instance of the driver on a port of its own choosing.

Safari itself may need configuring in two respects. First, select Preferences. There is an “Advanced” tab. Near the bottom of this, there is a check box labelled “Show Develop menu in menu bar”. This must be ticked and will add an option to the Safari main menu. Second, this new menu option must be opened. Towards the bottom, the option “Allow Remote Automation” must be ticked. The following code should now run.

```
use strict; use warnings;
use feature 'say';
use Selenium::Remote::Driver;

my $driver = Selenium::Remote::Driver->new(
    'port' => 4444,
    'browser_name' => 'safari'
);
$driver->get('http://www.perlmonks.org');
say $driver->get_title();
$driver->quit();
```

A side effect of this is that an instance of Safari will be left open as a process, although the window may be closed. Manual intervention will be needed to close the instance.

2.2.11 Android

A Selenium server for Android, Selendroid, is available from the Selendroid web site. Perl on Android is possible, but not a

mainstream option. Typically, execution will use a remote server. This is discussed in section 15 below. Selendroid's documentation states that "devices must be plugged in via USB to the computer that the selendroid-standalone component is running on". This may be impractical where development uses a server or a virtual machine. There are also emulators available that can reproduce some Android functionality on a non-Android machine. However, since Selendroid is not an official part of Selenium and a large amount of Android developer knowledge is needed to run tests from Perl, the explanations would become too long for this document. Selenium Remote Server has options to emulate Android devices. Max Maischein was unable to get all the elements of Android to work together with `WWW::Mechanize::Chrome`, as he reported to the 2017 London Perl Workshop (YouTube video, 16:14).

3 Testing

The commonest application for Selenium is testing. The standard installation comes with a module, `Test::Selenium::Remote::Driver`, that provides test commands. However, the documentation states “this is an *experimental* addition to the `Selenium::Remote::Driver` distribution, and some interfaces may change”. It contains a number of references to a previous fork and the documentation is at best fragmentary. It is therefore difficult to recommend it for production.

Perl’s core module `Test::More` has been used in almost all the following examples. This document will not try to explain it. To learn more about testing, consider Langworth & chromatic, “Perl Testing – a Developer’s Notebook”, O’Reilly, ISBN 9780596100926.

The simplest demonstration of testing is to change the very first example into a test. The code below shows one way of doing this.

```
use strict; use warnings;
use Selenium::Chrome;
use Test::More;

my $url = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
is $driver->get_title(),
    'PerlMonks - The Monastery Gates',
    'Title as expected';
$driver->quit();
done_testing;
```

This test should pass. Devotees of Test Driven Development may prefer to see a test fail first. Any change to the expected value

should accomplish this.

It is possible to delay the browser for the convenience of the user by using Perl's `sleep` command. It is also possible to instruct Selenium to pause, but unlike Perl, the delay is specified in milliseconds with a default of 1000, or one second. A five second delay would therefore be specified as `$driver->pause(5000);`.

By default, Selenium will wait until the requested page has finished loading before continuing. It is therefore unlikely that this will be needed unless the HTTP "page loaded" response is misleading. This can happen when the end of HTTP transfers is merely the start of Javascript execution that does a lot of work before the user would recognise the page as having loaded.

4 Basic Browser Interactions

4.1 Following a link

```
use strict; use warnings;
use Selenium::Chrome;
use Test::More;

my $url    = 'http://www.perlmonks.org';
my $link   = 'Recently Active Threads';
my $driver = Selenium::Chrome->new();
$driver->get($url);
$driver->find_element_by_link_text($link)->click;
is $driver->get_title(),
    'Recently Active Threads',
    'Title as expected';
$driver->quit();
done_testing;
```

This example shows the use of the `click` method to select a link. There are several mouse and keyboard actions that can be implemented using Selenium. The `click` method shown did not cause the mouse to move, but that can be done if desired. The technique is demonstrated in section 9 below.

4.2 Browser Actions

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $link   = 'Recently Active Threads';
my $driver = Selenium::Chrome->new();
$driver->get($url);
$driver->find_element_by_link_text($link)->click;
is $driver->get_title(),
    'Recently Active Threads',
    'Title as expected';
$driver->go_back;
is $driver->get_title(),
    'PerlMonks - The Monastery Gates',
    'Title as expected after back';
$driver->go_forward;
is $driver->get_title(),
    'Recently Active Threads',
    'Title as expected after forward';
$driver->refresh;
is $driver->get_title(),
    'Recently Active Threads',
    'Title as expected after refresh';
$driver->quit();
done_testing;
```

This code is an extension of the previous example. After loading the “Monastery Gates” it advances to “Recently Active Threads”. After that, the browser’s “Back”, “Forward” and “Refresh” commands are invoked.

4.3 Keyboard Actions

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;
use Selenium::Remote::WDKeys 'KEYS';

my $url    = 'http://www.google.com';
my $text   = 'Perl Monks';
my $xpath  = q<//input[@name='q']>;
my $driver = Selenium::Chrome->new();
$driver->get($url);
$driver->find_element($xpath)->
    send_keys($text, KEYS->{'enter'});
is $driver->get_title(),
    'Perl Monks - Google Search',
    'Title as expected';
$driver->quit();
done_testing;
```

This example introduces another module, `WDKeys`, which comes as standard when installing Selenium. It allows the programmer to send keys beyond the standard ASCII set. In this case, the Enter key has been “hit” at the end of the text.

Element locators are described in more detail in section 5 below. The example in the documentation expresses the locator as `$driver->find_element("//input[\@name='q']");`. This is functionally identical, but requires certain characters to be escaped. The code example above avoids this. As with other literals, the xpath has been abstracted to a variable to clarify the code, but might equally well be included in line.

4.4 Take a Screen Shot

```
use strict; use warnings;
use Selenium::Chrome;
use Selenium::Screenshot;

my $url     = 'http://www.perlmonks.org';
my $outdir  = '.';
my $outfile = 'scrshot';
my $driver  = Selenium::Chrome->new();
$driver->get($url);
my $pic = Selenium::Screenshot->new(
    png => $driver->screenshot,
    folder => $outdir,
)->save(filename => $outfile);
$driver->quit();
```

This example involves a large dependency chain that has not been mentioned previously. It will not be used again, so anyone who does not want to go through the installation process will miss very little from this document.

There are several learning points here. The first is that "`Selenium::Screenshot` is a wrapper class for `Image::Compare`" according to the POD. There are accordingly many features that can be used to process the image if that is wanted. The second is that, by default, a "screenshots" directory will be created and used unless the directory is specified as above. The third is that the file name will be created automatically as a time stamp. The fourth is that the extension will be the file type, in this case ".png", and that this is immutable. The fifth is that the term "screen shot" is slightly misleading. What will appear will vary depending on the browser used. With Firefox, neither the browser decoration nor anything that appears on the screen from the operating system or other applications will appear. The file will contain the entire web page as rendered by the browser, regardless of how little would actually fit on

a screen. With Chrome, the decoration will again be lost, apart from the image of a vertical scroll bar, which will be useless. The image will be approximately the size of a screen. Finally, it should be noted that this example saves the file created to the current directory. Care should be taken when specifying directories as a path such as '~' may not use the user's home directory but create a directory called '~' off the current directory. This may be operating system or version dependent.

Taking a screen shot can be very helpful should a test fail. Reporting the name of the screen shot in the test failure report can increase the helpfulness as it will associate the image file with the test failure. This can save time if there are multiple test failures.

Newer versions of `Selenium::Remote::Driver` include a simpler method, although the same dependency chain is needed. The following code demonstrates it:

```
use strict; use warnings;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
$driver->capture_screenshot('PMHome.png');
$driver->quit();
```

There is also a method to return a screen shot in Base64 encoding. There is no plausible practical use for the following code, but it demonstrates that an equivalent file will result from both approaches. It would be invalid to test that the two files are equal as PerlMonks contains "quips" that change from one loading to the next, in addition to the possibility that new questions and answers may appear.

```
use strict; use warnings;
use Selenium::Chrome;
```

```
use MIME::Base64;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $b64 = $driver->screenshot();
my $decoded = decode_base64($b64);
open my $fh, '>', 'PMB64.png'
    or die "Can't open PMB64.png for output: $!";
binmode $fh, ':raw';
print $fh $decoded;
close $fh;
$driver->quit();
```

5 Elements in Detail

So far, this document has glossed over the issue of how to find parts of a page. The following constructs have been used:

- `$driver->find_element_by_link_text($link)`
- `$driver->find_element(q<//input[@name='q']>)`

Elements can be found singly or as groups. If singly, an object will be returned; if as groups, an array of objects. The syntax for the two is identical to the second version, save for the change from “element” to “elements”.

If nothing is found, the error message is somewhat cryptic. Depending on the installation and version, it may appear as:

```
coercion for "id" failed: When passing in
an object to the WebElement id attribute,
it must have at least one of the ELEMENT or
element-6066-11e4-a52e-4f735466cecf keys. at
/usr/local/share/perl/5.24.1/Selenium/Remote/Driver.pm  line
1034.
```

There are no additional messages indicating which line of the programmer’s source code has caused the error.

There is an improved error message in version 1.27. This states that it is unable to locate an element and gives the method and selector as received by Selenium. This may not be immediately recognisable if the selector is not a literal and there is still nothing to indicate the location in the original source.

The two examples above show not only two different ways to find an element but also the two different forms of function that can be used. In the first example, the element will be found using the link text. In the second, the strategy that will be used is to search by xpath. When `find_element` is specified without being extended using a `_by_` part, the strategy may be specified as an optional second parameter. This defaults to xpath, so the second example above could have been specified more verbosely as:

```
$driver->find_element(q<//input[@name='q']>, 'xpath')
```

The difference between the two is that the version including the strategy as part of the method will warn if nothing is found, while the version passing the locator strategy as a parameter will use croak to kill the script. Note that, while it is possible to write a different error handler and pass it to the driver object at construction time, this will NOT prevent the locator methods croaking if that is their usual behaviour. The creation of a custom error handler is described in section 14.1 below. This also includes example code for testing locator results without causing a test script to die, the normal behaviour of croak.

The `Selenium::Remote::Driver` documentation states that there are ten strategies for finding elements:

- `find_element_by_class`
- `find_element_by_class_name`
- `find_element_by_css`
- `find_element_by_id`
- `find_element_by_link`
- `find_element_by_link_text`
- `find_element_by_name`
- `find_element_by_partial_link_text`
- `find_element_by_tag_name`
- `find_element_by_xpath`

Note that some of these do not exist in Selenium bindings for other languages. No documentation has been found for them and they have been identified but not used in the sections below. Note also that the default engine of the browser will be used for many of the locators, so if the css or xpath engines of browsers differ, so will the results returned by Selenium.

5.1 Class

While this will, in both formats, return an object, its purpose is not documented and no equivalent exists in other languages that support Selenium. It requires a parameter but, given the paucity of documentation and unclear purpose, it will be ignored. It appears to be a synonym for `class_name`, but there may be edge cases that differ.

5.2 Class Name

This is one of the most commonly used locators. The class name must be given as a parameter.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt1 = $driver->find_element_by_class_name(
    'post-voterep');
my $elt2 = $driver->find_element(
    'post-voterep', 'class_name');
is_deeply $elt1, $elt2,
    'Find element format doesn\'t matter';
my $doc = $driver->find_element_by_class_name(
    'superdoc');
isn't $elt1->get_text,
    $doc->get_text,
    'Looking at different elements';
is $doc->get_text,
    'The Monastery Gates',
    'h3 is a superdoc with TMG title';
$driver->quit();
done_testing;
```

The two elements are created using the two different formats to demonstrate that they return identical objects. Any element object returned is likely to have over 400 lines of `Data::Dumper` output, making tools like `is_deeply` invaluable.

The first “superdoc” class appears in the page source as: `<h3 class="superdoc">The Monastery Gates</h3>`

The first thing done is to test it at a simplistic level (there is no `isn't_deeply` option in `Test::More`) to demonstrate that it is in fact a different object. Then its value is tested against the heading that appears.

Selenium has its own method for testing the equality of elements.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->find_element(
    'post-voterep', 'class_name');
my $doc = $driver->find_element_by_class_name(
    'superdoc');
ok !$driver->compare_elements($elt, $doc),
    'The superdoc is not a post';
$driver->quit();
done_testing;
```

The output from `compare_elements` is a boolean, so if the elements should be the same, it is better to use `is_deeply`, which will give more detailed information on the differences between the two elements. If they should be different, `compare_elements` will check the entire structure, while the cursory comparison of `get_text` in the earlier example depends on knowing that the two elements are different at that particular point. Of course, `is_deeply` is available only if tests are being run.

Only one format is permitted for finding multiple elements.

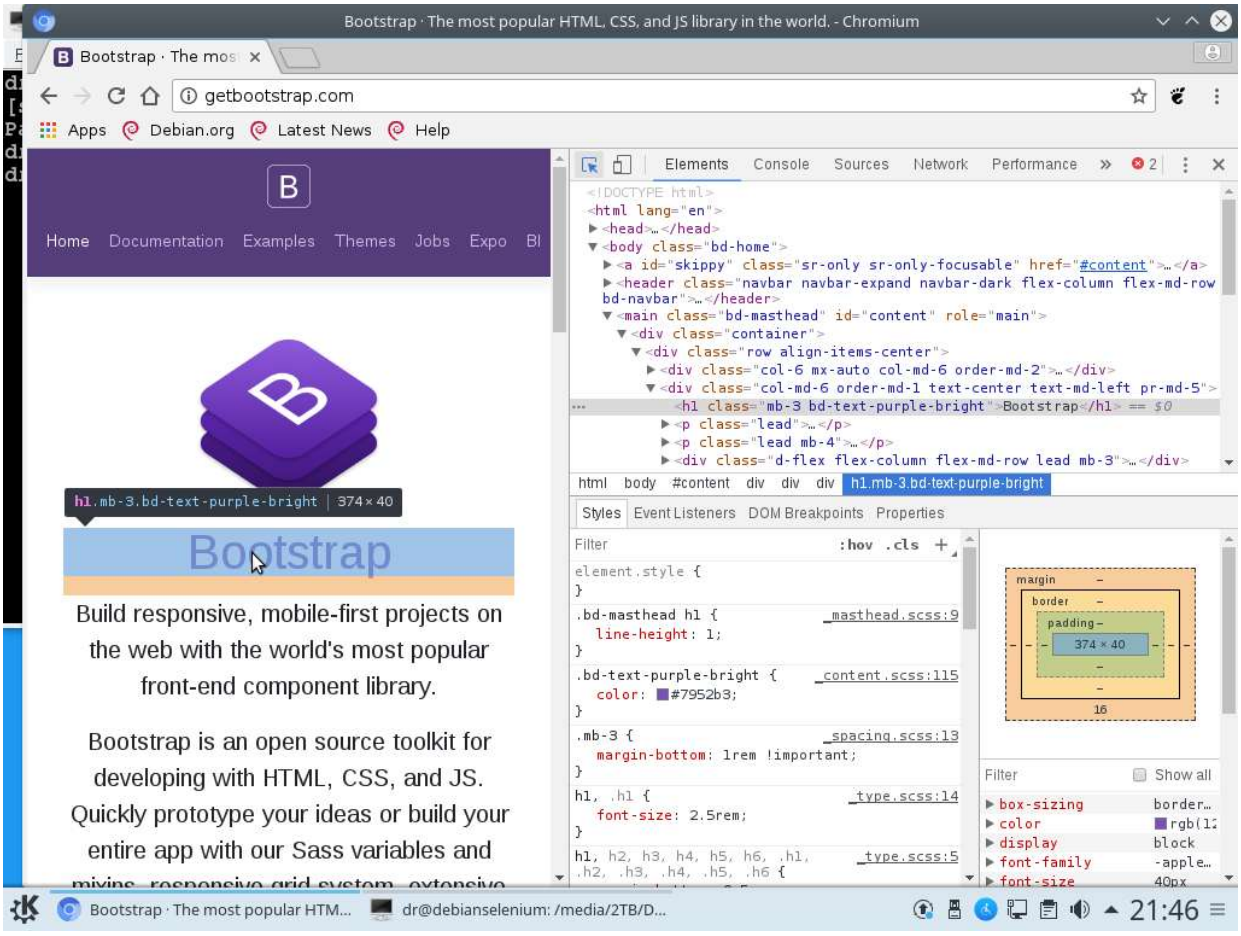
```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
```

```
my $driver = Selenium::Chrome->new();
$driver->get($url);
my @elt1 = $driver->find_elements(
    'post-voterep', 'class_name');
my @elt2 = $driver->find_elements(
    'superdoc' , 'class_name');
my $num = scalar @elt1;
cmp_ok $num, '>', 5, "More than 5 posts ($num)";
is scalar @elt2, 1, 'Only one superdoc';
$driver->quit();
done_testing;
```

5.3 CSS

This is very useful but slightly misleadingly named. In other Selenium implementations it is called CSS Selector. The point is not that it returns CSS or finds elements by looking at CSS, but it uses the same locator as CSS does. So if the CSS that is supposed to be applied is known, it can be used to find the element. This is obviously most useful on sites that have extensive CSS, although that is a growing number and the overwhelming majority of new sites. There are also browser tools that can help specify the CSS locators of an element.



In the screen shot above, the developer tools have been activated on Chrome. This can be done with Ctrl-Shift-I or by following the “More tools” menu from the three vertical dots.

Elements can be selected using the selection tool, a rectangle with an arrow. This is highlighted in blue in the screen shot above. The “Bootstrap” masthead has been selected with it, and the developer window shows the CSS selector needed.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://getbootstrap.com';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->find_element(
    'h1.mb-3.bd-text-purple-bright',
    'css');
is $elt->get_text(),
    'Bootstrap',
    'Masthead found by CSS';
$driver->quit();
done_testing;
```

CSS and xpath are the two locators that can find one element of many that match, if the index is known. So to find the fourth column of the third row of a table, for example, the CSS specified might be: `tr:nth-of-type(3) td:nth-of-type(4)`. If the index is not known but the value is, it is still possible to use `find_elements` and iterate over the return to identify the element desired.

See also section 6.5 below, which describes the css attribute of an element.

5.4 ID

IDs can be computer generated and subject to change without notice. Assuming that the ID is known, the element can be extracted in the usual way.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->find_element(
    'Voting_Booth', 'id');
unlike $elt->get_text(),
    '/No recent polls found/i',
    'There is a current poll';
$driver->quit();
done_testing;
```

5.5 Link

Like Class, this is undocumented and does not exist in other languages. It will therefore be ignored. It is better to use “Link Text” or “Partial Link Text”, which are documented for other languages and will therefore be easier to understand and research.

5.6 Link Text

This is one of the two locators used previously. The example code was `$driver->find_element_by_link_text($link)`. This could also be written as `$driver->find_element($link, 'link_text')`. The example in section 4.1 above shows its use.

5.7 Partial Link Text

This is useful if only part of the link text is known or if multiple elements are wanted.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->find_element('oting',
    'partial_link_text');
like $elt->get_text(),
    '/experience/i',
    'Link to voting & experience system';
$driver->quit();
done_testing;
```

The example above will work whether “voting” is spelled with an upper or lower case V. Users should note that, since Selenium is available for many languages, the full features of Perl’s regex engine cannot reliably be used in the Selenium binding, although they can be used in the pure Perl parts of the code where regexes have their standard meaning.

It might be expected that the code above would return the voting booth element, as in the previous example. However, this locator is for *links* and the voting booth element is not a link, although it contains links. This can be shown by changing the code as below.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
```

```
$driver->get($url);
my @elt = $driver->find_elements('oting',
    'partial_link_text');
is scalar @elt, 1, 'Only 1 voting link';
$driver->quit();
done_testing;
```

5.8 Name

This is the element's name, not the name of the tag, covered in the next section. It is common to use hidden elements to create some sort of state in HTML, and for programmer ease these elements frequently have convenient names. But names may be given to elements for other reasons.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->find_element('passwd', 'name');
is $elt->get_attribute ('maxlength'), 10,
    'Still no increase in password length';
$driver->quit();
done_testing;
```

The code introduces a new Selenium function, `get_attribute`. This DWIMs, returning the value of an attribute whose name is passed as a parameter. This code finds the password entry box on the PerlMonks home page. It does *not* use the tag name for good reason – there is an HTML input type "password" and using this as a name would risk confusion. The restriction of passwords to 10 characters is a common gripe among PerlMonks users, but the old code makes increasing it problematic. There are lots of problems with PerlMonks passwords, so don't re-use one there.

5.9 Tag Name

This is the terminology chosen by Selenium. It is the tag itself. The code below searches for the first “a” reference, a link to the hosting company.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->find_element('a',
    'tag_name');
is $elt->get_attribute('href'),
    'http://pair.com/',
    'First link is to Pair';
$driver->quit();
done_testing;
```

Again, the `get_attribute` function has been used to extract the href. However, there is no Selenium way to extract a list of what the attributes are. That would have to be done with Javascript or using an HTML parser.

Note that this is an unusual way of getting links. The “link text” and “partial link text” locators described above would be a more normal way.

5.10 XPath

This is the most complex of the Selenium locators, but it is also the most powerful. It is the default if no locator variable is passed. XPath is a large and complicated subject of its own, and will be covered only briefly. As with CSS, it is possible to use the developer tools to help. However, there is also an online tool at FreeFormatter's web site that allows users to play with xpath expressions and XML (officially, but XHTML is HTML that follows XML rules and is probably a good idea anyway) to find out what works and doesn't. The example from section 4.3 above that uses an xpath is repeated here for convenience.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;
use Selenium::Remote::WDKeys 'KEYS';

my $url    = 'http://www.google.com';
my $text   = 'Perl Monks';
my $xpath  = q<//input[@name='q']>;
my $driver = Selenium::Chrome->new();
$driver->get($url);
$driver->find_element($xpath)->
    send_keys($text, KEYS->{'enter'});
is $driver->get_title(),
    'Perl Monks - Google Search',
    'Title as expected';
$driver->quit();
done_testing;
```

Remember that, if no locator parameter is passed, xpath is used.

To repeat what was stated in section 5.3 above, CSS and xpath are the two locators that can find one element of many that match, if the index is known. So to find the fourth column of the third row of a table, for example, the xpath specified might be: `//tr[3]//td[4]`. If

the index is not known but the value is, it is still possible to use `find_elements` and iterate over the return to identify the element desired.

5.11 Active Element

Since the active element has already been located, this locator may seem redundant and, indeed, it is of little use for locating an element in the first place. However, if a bug is found, especially if that bug has been created by mouse or keyboard actions, it may be useful to interrogate the element that has really been activated to determine whether it is the one the code assumed was activated. Other locators like id and css may also result in an unexpected element being activated if the page design has changed in an unexpected way.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.google.com';
my $driver = Selenium::Chrome->new();
$driver->get($url);
is $driver->get_active_element()->
    get_attribute('name'),
    'q',
    'Google query box activated by default';
$driver->quit();
done_testing;
```

Note that there is always an active element (under normal circumstances – extremes like an empty web page have not been tried), even if a `find_element` command has failed.

Since the active element must always exist and need not be assigned to a variable, it is convenient that there is a method to send keys to it without extra work. This method is `send_keys_to_active_element` and works in exactly the same way as the `send_keys` method described in section 4.3 above.

5.12 Child Elements

An element, such as a form or a table, may be a container for other elements.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->find_element(
    'Voting_Booth', 'id');
my $count = scalar
    @{$driver->find_child_elements(
        $elt, 'input', 'tag_name')} - 3;
cmp_ok $count, '>', 2,
    "The poll has at least 2 options ($count)";
$driver->quit();
done_testing;
```

In section 5.4 above, a test was shown for the existence of a current poll on PerlMonks. This test examines the number of inputs within the voting booth. The subtraction of 3 is because of the design of this particular web page, not something that would need to be replicated for all web pages.

The `find_child_elements` method has much in common with other locators. It must be given a locator string and strategy, although as with `find_element`, this will default to xpath. But it differs in needing another parameter first, namely the specific element whose children are sought.

As with root elements, methods exist for both single and multiple elements.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->find_element(
    'Voting_Booth', 'id');
my $child = $driver->find_child_element
    ($elt, "//input[\@value='Vote']");
is $child->get_tag_name(),
    'input',
    'A voting button exists';
$driver->quit();
done_testing;
```

The parameters – two compulsory, one, the strategy, optional and defaulting to `xpath` – are the same as for `find_child_elements`. The “child” locators have no equivalent where the strategy is part of the method rather than a parameter.

6 Element properties

6.1 Location

Elements have a `get_element_location` operator which returns a hash. The WebElement documentation mentions only two of the key / value pairs returned, namely the X and Y co-ordinates. The other keys are `scale`, `toString`, `ceil`, `translate`, `round`, `clone` and `floor`. These appear not to exist in at least some other language implementations of Selenium.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->find_element(
    'search_text', 'id');
my $hash = $elt->get_element_location();
cmp_ok $$hash{x}, ">", 10,
    "X co-ordinate reasonable ($$hash{x})";
cmp_ok $$hash{y}, ">", 50,
    "Y co-ordinate reasonable ($$hash{y})";
$driver->quit();
done_testing;
```

This tests the location of the search text input box on the PerlMonks title bar. Its position is not constant, hence the rather imprecise tests.

Another tool exists to return the location of an element, namely `get_element_location_in_view`. The documentation for this states “This is considered an internal command and should only be used to determine an element’s location for correctly generating native events”. It returns the element’s location on the screen after any scrolling has happened. There is no explanation of what would result if the element had been scrolled off screen.

6.2 Enabled

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my   $url          =   'https://www.w3schools.com/tags/tryit.asp?
filename=tryhtml_input_disabled';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $frame = $driver->find_element(
    qw{//iframe[@name='iframeResult']});
$driver->switch_to_frame ($frame);
my $elt = $driver->find_element('fname', 'name');
ok $elt->is_enabled, 'First name is enabled';
$elt = $driver->find_element('lname', 'name');
ok !$elt->is_enabled, 'Last name is disabled';
$driver->quit();
done_testing;
```

The line containing the URL is long and may appear wrapped. If copying and pasting code, take care to prevent any unwanted line break characters being pasted, as this may invalidate the example.

The example web page uses iframes. Code cannot look into an iframe unless Selenium has switched to the frame. It would be possible to do that in a single line of code, but the SSCCE (Small, Self Contained, Complete Example – see www.sscce.org) above shows the frame being located, after which `switch_to_frame` is used to change context. Only then is it possible to find the input, submit and output elements.

The use of the intermediate scalar `$elt` is not necessary; the line that creates it and the following line could be combined, but at a possible cost in clarity.

Once the frame has been selected, the code can check the first and last name input boxes to see that the right ones are enabled and disabled. A possible source of confusion is that HTML uses a “disabled” flag with “enabled” being the silent default, while Selenium names its method `is_enabled`.

6.3 Selected

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my   $url          =   'https://www.w3schools.com/tags/tryit.asp?
filename=tryhtml_select';
my $driver = Selenium::Chrome->new();
$driver->get($url);
$driver->switch_to_frame($driver->find_element(
    qw{//iframe[@name='iframeResult']}));
my $elt = $driver->find_element(
    'option', 'tag_name');
ok $elt->is_selected(), 'First option selected';
is $elt->get_attribute('value'), 'volvo',
    'First option is Volvo';
$driver->quit();
done_testing;
```

As previously, care must be taken when cutting and pasting the URL to prevent line endings causing problems.

The test of whether an element is selected is valid only for options, checkboxes and radio buttons. A boolean is returned. Two other methods, `set_selected` and `toggle`, exist but were deprecated in version 1.20 with the words “use click instead”. This deprecation seems to have been removed from version 1.23 and later, but the advice is probably still sensible.

6.4 Displayed

It is routine to have hidden elements in a web page. This may be to provide state or simply because it is easier to hide and reveal elements to change appearances than to rewrite the page to exclude them. There are two methods that report booleans for this status.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->find_element('op', 'name');
ok $elt->is_hidden(), '"op" element is hidden';
ok !$elt->is_displayed(),
    '"op" element is not displayed';
$elt = $driver->find_element('user', 'name');
ok $elt->is_displayed(),
    '"user" element is displayed';
ok !$elt->is_hidden(),
    '"user" element is not hidden';
$driver->quit();
done_testing;
```

As the code demonstrates, these are simple opposites.

6.5 CSS Attribute

This method reports the value of a CSS attribute as rendered.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://getbootstrap.com';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->find_element(
    'h1.mb-3.bd-text-purple-bright', 'css');
is $elt->get_css_attribute('font-size'), '64px',
    'Masthead font size is unchanged';
$driver->quit();
done_testing;
```

There is a list of CSS attributes at W3schools. This is the converse of the CSS selector described in section 5.3 above. That used the same locator as CSS to find an element, ignoring whatever formatting the CSS rules had imposed. This has no information on the selection mechanism but reports the formatting actually chosen.

6.6 Size

The `get_size` method returns a hash containing the size of the element. The keys for the size values are `height` and `width`. As with location, described in section 6.1 above, there are additional keys, `scale`, `toString`, `ceil`, `translate`, `round`, `clone` and `floor`, that do not exist in Selenium bindings for other languages.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->find_element(
    'search_text', 'id');
my $hr = $elt->get_size();
is $hr->{height}, 21,
    'Search text box height does not change';
is $hr->{width}, 173,
    'Search text box width does not change';
$driver->quit();
done_testing;
```

While it was necessary to write approximate tests for the location because it depended on what had appeared previously in the page, the size of the search text box should not change, meaning that the tested values can be specified precisely.

7 More browser interaction

7.1 Checkboxes

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my  $url          =  'https://www.w3schools.com/tags/tryit.asp?
filename=tryhtml_input_checked';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $frame = $driver->find_element(
    qw{//iframe[@name='iframeResult']});
$driver->switch_to_frame($frame);
$driver->find_element(
    q<//input[@value='Bike']>)->click;
$driver->find_element(
    q<//input[@value='Submit']>)->click;
$frame = $driver->find_element(
    qw{//iframe[@name='iframeResult']});
$driver->switch_to_frame($frame);
my $elt = $driver->find_element(
    "//body[\@class='w3-container']/div");
is $elt->get_text(),
    'vehicle=Bike&vehicle=Car ',
    'Both boxes selected';
$driver->quit();
done_testing;
```

As previously, care must be taken when cutting and pasting the URL to prevent line endings causing problems.

This code worked in versions up to and including 1.26. In 1.27, an error is returned indicating that there is no such element. Further investigation indicates that the contents of the page as shown by the browser are “detached” from the elements. Refreshing the browser returns the page to its state before the checkbox was ticked.

The trailing space in the string containing the expected return value is necessary as it appears in the value returned by the web site. Whether it is necessary to write code to strip trailing spaces must depend on the application. It seems unduly complicated for this example.

In the example above, the checkbox has been selected using the `click` method. It would have been possible to select it by sending a space or by any other mechanism acceptable on the web page. Indeed, a test suite might include all these if there were any danger of a different result.

7.2 Radio Buttons

Radio buttons are very much like checkboxes in the way they work, but with the obvious exception that no more than one can be active at any one time if the web page has been designed properly. There can be several sets of radio buttons with one selected in each set and it is therefore worth testing every possible combination to be sure that no button has become isolated. The code below does not do this, but shows how it might be done.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my  $url          =  'https://www.w3schools.com/html/tryit.asp?
filename=tryhtml_form_radio';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $frame = $driver->find_element(
    qw{//iframe[@name='iframeResult']});
$driver->switch_to_frame($frame);
is $driver->find_element(
    q<//input[@value='male']>)
    ->get_attribute('checked'), 'true',
    '"male" is true before click';
is $driver->find_element(
    q<//input[@value='female']>)
    ->get_attribute('checked'), undef,
    '"female" is undef before click';
is $driver->find_element(
    q<//input[@value='other']>)
    ->get_attribute('checked'), undef,
    '"other" is undef before click';
$driver->find_element(
    q<//input[@value='other']>)->click;

is $driver->find_element(
    q<//input[@value='male']>)
    ->get_attribute('checked'), undef,
    '"male" is undef after click';
is $driver->find_element(
    q<//input[@value='female']>)
    ->get_attribute('checked'), undef,
```

```
    '"female" is undef after click';
is $driver->find_element(
    q<//input[@value='other']>)
    ->get_attribute('checked'), 'true',
    '"other" is true after click';
$driver->quit();
done_testing;
```

As previously, care must be taken when cutting and pasting the URL to prevent line endings causing problems.

While it is consistent with Perl's definitions of truth, the return values of a radio button's "checked" attribute may surprise the uninitiated. If the button is selected, the value returned is a string containing 'true', while the value is undefined if not selected. undef might need handling differently from defined values.

7.3 Resizing the Browser

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $driver = Selenium::Chrome->new();
my $orighigh = $driver->
    get_window_size()->{height};
my $origwide = $driver->
    get_window_size()->{width};
my $origx    = $driver->
    get_window_position()->{x};
my $origy    = $driver->
    get_window_position()->{y};
my $changes = 0;
$driver->maximize_window;
sleep 1;

$changes++ if $driver->
    get_window_size()->{height} != $orighigh;
$changes++ if $driver->
    get_window_size()->{width}  != $origwide;
$changes++ if $driver->
    get_window_position()->{x}  != $origx;
$changes++ if $driver->
    get_window_position()->{y}  != $origy;
cmp_ok $changes, ">", 0,
    "At least one attribute has changed ($changes)";

$driver->set_window_size($orighigh, $origwide);
$driver->set_window_position($origx, $origy);
sleep 1;

is $driver->get_window_size()->{height},
    $orighigh, 'Height has original value';
is $driver->get_window_size()->{width} ,
    $origwide, 'Width has original value';
is $driver->get_window_position()->{x} ,
    $origx,    'X co-ord has original value';
is $driver->get_window_position()->{y} ,
    $origy,    'Y co-ord has original value';
$driver->quit();
done_testing;
```

This code introduces five new methods, `get_window_size`, `get_window_position`, `set_window_size`, `set_window_position` and `maximize_window`. If the browser window defaults to being maximised, a test will fail as the size and position will be the same before and after maximisation. The sleep commands seem to be important to reduce the number of test failures, but even with these, some of the tests of the restored values may fail on different implementations. This does not appear to change if a URL is retrieved. Note that `maximize_window` is known not to work with Chrome.

The `get` methods each return a hash containing two key / value pairs, `height` and `width` are returned by `get_window_size` and `x` and `y` by `get_window_position`. There are no extraneous pairs as with other hashes that are returned by some Selenium methods.

There are three further methods that are intended to emulate mobile devices. These are `get_orientation`, `set_orientation` and `set_inner_window_size`. The “orientation” methods take and return, according to the documentation, a single string in upper case which may be either `LANDSCAPE` or `PORTRAIT`. On Debian Chrome, setting returns a success code but with no visible change to the browser, while the `get_orientation` method results in a run time error message that seems to indicate a discrepancy between Selenium itself and Chrome, with Chrome having no such functionality. Any attempt to set the inner window size freezes the interface between Perl and Chrome. While Chrome will respond to manual commands, the Perl script will remain frozen until the browser instance is closed manually.

7.4 Close and Quit

These two commands are both browser dependent. `close` closes a “window”, usually meaning a tab, while `quit` closes all browsers. The documentation for `quit` is as follows:

> DELETE the session, closing open browsers. We will try to call this on our down (sic) when we get destroyed, but in the event that we are demolished during global destruction, we will not be able to close the browser. For your own unattended and/or complicated tests, we recommend explicitly calling quit to make sure you’re not leaving orphan browsers around.

It is certainly good practice generally to close all browsers, but it might be that the programmer wants to leave the browser available in a particular state, especially should a test have failed, for further work to be done. Not all browsers permit this. At least some versions of Chrome (certainly including 63.0.3239.84 on Debian 9.3) will close themselves upon termination of the Selenium caller, while at least some versions of Firefox (certainly including 52.5.2 64 bit ESR on Debian 9.3) will remain open, possibly as a result of a bug in either Selenium itself or `Selenium::Firefox`. The two code examples below were used to demonstrate the difference.

```
use strict; use warnings;
use Selenium::Chrome;
my $driver = Selenium::Chrome->new();
```

closes the browser, unlike

```
use strict; use warnings;
use Selenium::Firefox;
my $driver = Selenium::Firefox->new();
```

`Selenium::Chrome` has a method of its own that can be used instead of quit. This method is named `shutdown_binary` and claims to ensure that the binary executable is closed as well as the browser.

The nature of multiple tabs and instances is discussed in section 8 below. It is worth noting that PaleMoon, a fork of Firefox, has an option to keep the browser alive even when the last tab is deleted. The final tab of an instance does not appear to have been deleted but cleared to an empty state.

8 Using Multiple Tabs and Windows

Historically, browsers showed only a single web page. If the user wanted more open concurrently, a second instance of the browser needed to be opened in a new window. Modern browsers use tabs to combine these into a single window, giving the user an option of multiple tabs in a single window as well as multiple windows. However, Selenium's documentation sometimes refers to "window" as having the meaning of a new tab. This document will use the more generally understood meanings.

The creation of an additional instance of the browser is sometimes referred to as a popup or popunder. This is a perennial favourite of spammers and is routinely blocked by many browsers. The term "popup" is also used to mean a user interaction form created by Javascript. This is discussed in section 11.4 below.

8.1 Open a Link in a New Window

The easiest way to handle multiple windows is to use a different driver for each. It might be sensible to keep them in an array.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url  = 'http://www.perlmonks.org';
my $link = 'Recently Active Threads';
my @drivers;
push @drivers, Selenium::Chrome->new();
$drivers[0]->get($url);
my $linkurl = $drivers[0]->
    find_element_by_link_text($link)->
    get_attribute('href');
push @drivers, Selenium::Chrome->new();
$drivers[1]->get($linkurl);
is $drivers[1]->get_title(),
    'Recently Active Threads',
    'Title as expected';
sleep 5;
for (@drivers) {
    $_->quit;
}
done_testing;
```

This code is browser dependent and can cause problems with Firefox, which dislikes having multiple browser windows open at the same time unless it is done through the Firefox interface. Using a `send_keys` command is a possible workaround, although this introduces the issue of selecting and switching windows. The list of sendable keys and their names does not appear anywhere in the Perl documentation. It can be seen by looking at the source of `Selenium/Remote/WDKeys.pm`, which is little more than a list of key bindings. It certainly varies by implementation, as Apples have keys defined differently. The issue of whether a shift-type key needs to be send twice (the second time, after the affected key, being to clear the

effect) seems to be implementation dependent.

The `sleep` command is so that users can see two browsers even though there is only one test. It can be deleted if desired.

8.2 Open a Link in a New Tab

There are no documents on how to do this. It is a frequently asked question in many languages, but there are varying success rates. The approach of sending a WDKeys string is used here to open a link in a new tab.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;
use Selenium::Remote::WDKeys 'KEYS';

my $url  = 'http://www.perlmonks.org';
my $link = 'Recently Active Threads';
my $driver = Selenium::Chrome->new();
$driver->get($url);
$driver->find_element_by_link_text($link)->
    send_keys(KEYS->{'control'},
    KEYS->{'enter'});
is $driver->get_title(),
    'PerlMonks - The Monastery Gates',
    'Title as expected';
my $handles = $driver->get_window_handles;
$driver->switch_to_window($handles->[1]);
is $driver->get_title(),
    'Recently Active Threads',
    'Title of second tab as expected';
$driver->quit;
done_testing;
```

There are several learning points here. The first is that tests should not be run unless the active tab is known. This is important since some browsers allow the user to specify whether a newly opened tab automatically becomes active or not. Another option is where to put related tabs – the settings may allow the user to choose whether they appear at the end or next to the tab of the link. These differences in settings may mean that tests which pass on one machine may fail on another, even though the hardware, OS and browser versions are identical. The second learning point is that

there are Selenium commands to manipulate tab handles (remember that Selenium thinks of tabs as windows). These should be used whenever sensible in preference to sending keys. The `get_window_handles` method returns an array from which a handle can be chosen. The third is that the keys for a combination (called a *chord* by other languages) must be sent separately; `send_keys(KEYS->{'control', 'enter'})` will fail. There are web pages suggesting this approach meaning, again, that it may be implementation dependent.

Selenium has a method, `get_current_window_handle`, which can help when navigating multiple tabs.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;
use Selenium::Remote::WDKeys 'KEYS';

my $url  = 'http://www.perlmonks.org';
my $link = 'Recently Active Threads';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $orighdl = $driver->
    get_current_window_handle;
$driver->find_element_by_link_text($link)->
    send_keys(KEYS->{'control'},
        KEYS->{'enter'});
my $handles = $driver->get_window_handles;
if ($driver->get_current_window_handle
    ne $orighdl) {
    $driver->switch_to_window($handles->[0]);
}
is $driver->get_title(),
    'PerlMonks - The Monastery Gates',
    'Title as expected';
$driver->switch_to_window($handles->[1]);
is $driver->get_title(),
    'Recently Active Threads',
    'Title of second tab as expected';
$driver->quit;
done_testing;
```

It would, of course, have been perfectly possible to guarantee that the first tab opened was selected by switching to it without testing the handles. In more complicated situations where many tabs are being opened and closed in a browser with unknown settings, this may not be possible and the best way to identify which is a specific tab might well be to loop through all the handles looking for one that has been identified previously. Handles are strings, and the array reference returned by `get_window_handles` is to an array of strings.

9 Drag and Drop

A previous maintainer of `Selenium::Remote::WebDriver` has written a blog post. The implications are that drag & drop may be unreliable without Javascript, but the post was written in approximately 2015 and the code below seems to work.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url = 'https://jqueryui.com/droppable/';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $frame =
    $driver->find_element
    (q<//iframe[@class='demo-frame']>);
$driver->switch_to_frame($frame);
my $from = $driver->find_element(
    'draggable', 'id');
my $to   = $driver->find_element(
    'droppable', 'id');
my $from_loc = $from->get_element_location;
my $to_loc = $to ->get_element_location;
ok ((($$from_loc{x} != $$to_loc{x})
  or ($$from_loc{y} != $$to_loc{y})),
    'One of the starting co-ordinates differs');
$driver->move_to(element => $from);
$driver->button_down;
$driver->mouse_move_to_location (element => $to);
$driver->button_up;
my $new_loc = $from->get_element_location;
ok ((($$from_loc{x} != $$new_loc{x})
  or ($$from_loc{y} != $$new_loc{y})),
    'At least one co-ordinate has changed');
sleep 10;
$driver->quit;
done_testing;
```

The `sleep` command is so the user can see the result of the action and can be deleted if wished.

There are several learning points in this code. The first is that the web page used has been specially designed for drag and drop. If the code does not work here, it is unlikely to work anywhere.

The second point is the `get_element_location method`. This is used only for the tests, not for the dragging and dropping itself. It returns a hash that contains several other elements, but the ones needed by the code are x and y, the co-ordinates of the element. It might be expected that the final co-ordinates of the move would be identical to the destination co-ordinates, but the two objects are not the same size and there is a difference of 25 pixels on the machine used to write this document. Systems may differ. Certainly, when using drag & drop in a production test, it would not be enough to test that a single co-ordinate had changed. Something more positive would be appropriate.

The next learning point is that there is no “either – or” test in `Test::More` or `Test::Simple`, nor is there any documented way of writing such a test in either module. The two tests in this SSCCE have therefore been written using `ok`, despite it being less self-explanatory when tests fail.

Mouse actions have been introduced for the first time. `button_down` and `button_up` take no parameters and are self-explanatory. Users should make sure that there is no danger that the system will be left with the mouse button down; the effects are likely to be unpredictable, hard to diagnose and system dependent. `move_to` and `mouse_move_to_location` do exactly the same thing. The documentation shows the longer version, which makes clearer that a mouse is moving, but Laziness suggests the shorter version.

The default, when moving the mouse, is to move to the centre of the element. This is why the SSCCE ended up with the moved element having different co-ordinates from the target. However, the documentation is misleading. It implies clearly that making the second movement command

```
$driver->move_to(element => $to, x => 0, y => 0);
```

would put the element at the top left of the target. However, neither this nor any of the values that have been tried for the offsets (including negative values) results in any difference in the visible position of the moved element or the values returned by `get_element_location`. As usual, this may be implementation dependent.

The documentation for `Selenium::Remote::Driver->button_down` includes the following:

> Note that the next mouse-related command that should follow is buttonup . Any other mouse command (such as click or another call to buttondown) will yield undefined behaviour.

This seems strange. While it is understandable that a click would be undefined when the mouse button is already down, it seems illogical to suggest that the mouse cannot be moved with the button depressed. The correct operation of the code example above seems to imply that the quoted documentation is less than exhaustive.

The technique of changing to an iframe has been explained in section 6.2 above. Remember that xpath is the default location strategy.

10 Cookies

Selenium documentation on cookies is sparse, regardless of the language. Selenium can read, add and delete cookies, but only certain parameters can be added directly by Selenium, although all can be read.

10.1 Reading Cookies

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.freeformatter.com';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $ar_cook = $driver->get_all_cookies();
my $n = scalar @$ar_cook;
is $n, 2, 'Freeformatter uses 2 cookies';
my %cookies;
for (@$ar_cook) {
    $cookies{$_->{name}} = $_->{value};
}
ok exists $cookies{AWSELB},
    'Found Amazon cookie at freeformatter';
$driver->quit;
done_testing;
```

This generates an array of hashes, each element of the array being a cookie. Note that the return is an array reference, so `my @cookies = $driver->get_all_cookies();` would not result in an array but an array containing an array reference. Note also that the number of cookies may be geolocation dependent; assuming the UK, the cookies that appear after loading google.com are not google.com cookies at all; they are for google.co.uk. To see any google.com cookies, the browser would have to be persuaded not to redirect. Only then would the google.com cookie(s) be accessible.

The documentation states that there are five possible properties in each cookie hash, namely ‘name’, ‘value’, ‘path’, ‘domain’ and ‘secure’. Using `Data::Dumper` reveals two more, ‘expiry’ and ‘httpOnly’. Do not be surprised if other properties appear that are undocumented in Selenium.

10.2 Adding Cookies

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.example.com';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $ar_cook = $driver->get_all_cookies();
is scalar @$ar_cook, 0,
    'Example.com uses no cookies';
$driver->add_cookie('cookiename',
    'cookievalue',
    '/',
    '.example.com');
$ar_cook = $driver->get_all_cookies();
is scalar @$ar_cook, 1, '1 cookie now';
is $$ar_cook[0]{name}, 'cookiename',
    'The added cookie has the right name ...';
is $$ar_cook[0]{value}, 'cookievalue',
    '... and value';
$driver->quit;
done_testing;
```

The parameters for adding a cookie are positional. A fifth, the “secure” value, is optional. There is no documentation, official or otherwise, on using a hash or on adding a parameter to an existing cookie. Experimentation has failed to produce a workaround.

Cookies appear to be added to the start of the array. This appears to be consistent, but again there is no documentation. A more reliable way of testing for the addition of a cookie would be to loop through the array until the name is found. Only one cookie may exist for a name and domain combination.

Note that the positional parameters do not include the expiry. Accordingly, all cookies created using this technique are session only cookies and will be destroyed when the browser is closed.

10.3 Deleting Cookies

Cookies may be deleted in two ways. Deleting all cookies for a page can be done with a single command.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.freeformatter.com';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $ar_cook = $driver->get_all_cookies();
is scalar @$ar_cook, 2,
    'Freeformatter uses 2 cookies';
$driver->delete_all_cookies;
$ar_cook = $driver->get_all_cookies();
is scalar @$ar_cook, 0, 'All gone';
$driver->refresh;
$ar_cook = $driver->get_all_cookies();
is scalar @$ar_cook, 2,
    '... but refreshing replaces them.';
$driver->quit;
done_testing;
```

If the object is to delete a single cookie, that can be done without looping if its name is known. Note that trying to delete a cookie that does not exist is not a problem; no error occurs, although nothing is deleted.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.freeformatter.com';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $ar_cook = $driver->get_all_cookies();
is scalar @$ar_cook, 2,
    'Freeformatter uses 2 cookies';
$driver->delete_cookie_named('Cthulhu');
```

```
$ar_cook = $driver->get_all_cookies();
is scalar @$ar_cook, 2, 'No cookie? No problem';
$driver->delete_cookie_named('AWSELB');
$ar_cook = $driver->get_all_cookies();
is scalar @$ar_cook, 1,
    'Deleting an existing named cookie works';
$driver->refresh;
$ar_cook = $driver->get_all_cookies();
is scalar @$ar_cook, 2,
    '... but refreshing replaces it.';
$driver->quit;
done_testing;
```

To delete a cookie where something is known apart from the name, it would be necessary to write a loop that goes through all cookies looking for the known information and then extracts the name when the information is found. `delete_cookie_named` can then be used. There are no other commands in Selenium for deleting cookies.

11 Javascript

JavaScript can be version dependent in different browsers. Selenium code that relies on JavaScript has a worse risk of problems than Selenium code that doesn’t.

11.1 Is JS Enabled?

This is a very simple task, but if any JS work is intended, this should probably be the first test with everything else being skipped if it fails. Not all drivers or browsers support JavaScript, or JavaScript may be disabled.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $driver = Selenium::Chrome->new();
ok $driver->has_javascript,
    'Javascript is enabled';
$driver->quit;
done_testing;
```

11.2 Injecting JS

Using Selenium alone, there is no way to add a field to a cookie unless the field was one of the five positional parameters the Selenium interface understood. However, this can be done using JS. It might be necessary to use JS to create cookies rather than create them directly from the server if user approval is needed. This is compulsory in certain jurisdictions such as the EU and creating cookies before a user has taken positive action to accept them may create legal issues. This document is not legal advice. If in doubt, consult a qualified legal practitioner in the jurisdiction in question.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url = 'http://www.example.com/';
my $js = <<'END_OF_JS';
document.cookie = "testcookie=testvalue; path=/; domain=
.example.com;expires=27 Nov 2027 00:00:01 UTC";
END_OF_JS

my $driver = Selenium::Chrome->new();
SKIP: {
    skip "Can't test JS (disabled)", 5
      unless $driver->has_javascript;
    $driver->delete_all_cookies;
    $driver->get($url);
    my $ar_cook = $driver->get_all_cookies();
    is scalar @$ar_cook, 0,
      'Example.com uses no cookies';
    $driver->execute_script($js);
    $ar_cook = $driver->get_all_cookies();
    is scalar @$ar_cook, 1, '1 cookie now';
    is $$ar_cook[0]->{name}, 'testcookie',
      'The added cookie has the right name ...';
    is $$ar_cook[0]->{value}, 'testvalue',
      '... and value ...';
    is $$ar_cook[0]->{expiry}, 1827273601,
      '... and expiry!';
}
$driver->delete_all_cookies;
```

```
$driver->quit;
done_testing;
```

This code uses a “heredoc” to set up the JS. JS does not like line breaks within a cookie definition, so be sure that the JS is a single line if cutting & pasting this example.

There is a skip block, meaning that there will be no test failures reported if the browser has JS disabled. This has been told that there are five tests that may be skipped, which is important only if a test plan has been stated.

By default, cookies expire when the session closes, but the cookie specified has a longer date, meaning that it is automatically saved by Chrome (if its default options have not been changed). This means that the browser contains the cookie before loading the URL, affecting the number of cookies that will exist for the page. This is fixed by the `delete_all_cookies` command. This is repeated outside the skip block to ensure that no cookies are left behind after the tests have been run. Conversion of cookie date & time formats is not the subject of this document.

If an attempt is made to create a cookie with an expiry that has already passed, nothing will happen. This is shown in the next code example. Again, beware of line breaks in the heredoc if cutting and pasting.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url = 'http://www.example.com/';
my $js = <<'END_OF_JS';
document.cookie = "testcookie=testvalue; path=/; domain=
.example.com;expires=Mon, 27 Nov 2017 00:00:01 UTC";
```

```
END_OF_JS

my $driver = Selenium::Chrome->new();
SKIP: {
    skip "Can't test JS (disabled)", 2
      unless $driver->has_javascript;
    $driver->delete_all_cookies;
    $driver->get($url);
    my $ar_cook = $driver->get_all_cookies();
    is scalar @$ar_cook, 0,
      'Example.com uses no cookies';
    $driver->execute_script($js);
    $ar_cook =  $driver->get_all_cookies();
    is scalar @$ar_cook, 0, 'Cookie not added';
}
$driver->quit;
done_testing;
```

11.3 Locating by JS

If JS works, locating elements by JS is the most powerful and versatile of all the locate options. It is usually faster as the document object model (DOM – the elements of the page broken down into a tree form to aid traversal) does not have to be exported from the browser to Selenium before being processed. Further, the processing is done by the browser's compiled code. This not only adds speed but also reliability. If there is a difference between the result of a locator run in Selenium and the expected result, JS may be the best tool. Remember that browsers may behave differently, so a consistency in Selenium can expose this sort of problem.

Any JS DOM element returned should be converted automatically to a WebElement object.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url = 'http://www.perlmonks.org';
my $js = <<'END_OF_JS';
return document.getElementById('Log_In');
END_OF_JS

my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->execute_script($js);
like $elt->get_text(), '/^Log In/i',
    'Login table found';
$driver->quit;
done_testing;
```

The SSCCE above locates an element in the same way as in section 5.4 above, but using JS to execute the locator. The various locators that are available in JS are not the subject of this document.

11.4 Popups

Popups take two forms. The non-Javascript one is the creation of a popup or popunder window, an additional instance of the browser discussed in section 8 above.

There are three types of popup available from Javascript. An “alert” box requires the user to click “OK”. A confirmation box allows the user to choose “OK” or “Cancel”. A prompt box allows the user to enter data before clicking “OK” or “Cancel”. All these are treated as “alerts” by Selenium.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url = 'http://www.example.com';
my $jsAlert = <<'END_OF_ALERT';
var rtn = alert("We use cookies. OK accepts this.");
localStorage.setItem("JSSelenium", rtn);
END_OF_ALERT

my $jsGetItem = <<'END_OF_GET';
return localStorage.getItem("JSSelenium");
END_OF_GET

my $driver = Selenium::Chrome->new();
$driver->get($url);
$driver->execute_script($jsAlert);
$driver->accept_alert;
my $rtn = $driver->execute_script($jsGetItem);
is $rtn, 'undefined',
    'Accepting an alert returns nothing';
$driver->execute_script($jsAlert);
$driver->dismiss_alert;
$driver->execute_script($jsGetItem);
is $rtn, 'undefined',
    'Dismissing an alert also returns nothing';
$driver->quit;
done_testing;
```

This SSCCE creates an alert box twice, although sleep or pause commands will be needed to see it. There are several points that need explaining.

Promises are available in JS (see, for example, Google's primer for developers), but they have not been implemented in Selenium. This creates a problem in the SSCCE above. It is perfectly possible to inject JS using Selenium and to capture the value or object returned, as seen in section 11.3 above. In this case, though, the JS injected creates a popup. The value of this is not known at the point where control is returned to Perl and the `execute_script` method will return undef. Likewise, the `accept_alert` and `dismiss_alert` methods return values (1 in normal circumstances), but these are different methods and do not capture the return values of anything else. Since the return value wanted, the encoding of the user's action, is not available from either of the methods used to emulate the user interface and action, the popup's return value needs to be stored somewhere and returned in a different way.

The two obvious options are cookies and local storage. The use of cookies has already been demonstrated. The code above therefore uses the popup JS injection to store the return value of the popup *as interpreted by JavaScript* in local storage and a second injection to retrieve it and return it to Selenium. This results in a JS value (`undefined`) that approximates to Perl's `undef` being passed to Perl as a string. It is possible to view the value within JS by adding a second popup that displays the value to the end of the first piece of JS injected. That code would be `alert("<" + rtn + "> returned");` and would require a sleep followed by an additional `dismiss_alert` to clear the alert after enough time for it to be read.

The `Selenium::Remote::Driver` documentation describes two methods, `get_local_storage_item` and `delete_local_storage_item`,

for local storage operations. Unfortunately, these fail with an error message along the lines of:

```
Error while executing command: getLocalStorageItem:
Server returned error message unhandled request instead of
data at
/usr/local/share/perl/5.24.1/Selenium/Remote/Driver.pm line
327".
```

This error occurs regardless of the browser used. Until this behaviour is changed, the workaround of using a second JS injection is the simplest. Also, there is no Selenium method to write to local storage.

As mentioned previously, the values returned from the second injection are strings representing the JS values. It is therefore necessary to test them against strings rather than treat them as Booleans. JS treats all responses to an alert (using JS's definition rather than Selenium's) as being the same and does not return anything to indicate what the user actually did.

The second type of popup is the `confirm` box. Its differences from the `alert` box can be seen from the code below.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url = 'http://www.example.com';
my $jsConfirm = <<'END_OF_CONFIRM';
var rtn = confirm("We use cookies. OK lets us.");
localStorage.setItem("JSSelenium", rtn);
END_OF_CONFIRM

my $jsGetItem = <<'END_OF_GET';
return localStorage.getItem("JSSelenium");
END_OF_GET
```

```
my $driver = Selenium::Chrome->new();
$driver->get($url);
$driver->execute_script($jsConfirm);
$driver->accept_alert;
my $rtn = $driver->execute_script($jsGetItem);
is $rtn, 'true',
    "Accepting a confirmation returns true";
$driver->execute_script($jsConfirm);
$driver->dismiss_alert;
$rtn = $driver->execute_script($jsGetItem);
is $rtn, 'false',
    "Dismissing a confirmation returns false";
$driver->quit;
done_testing;
```

This works in a very similar way to the previous SSCCE, but now the `accept_alert` and `dismiss_alert` methods result in `true` and `false` strings being returned.

The third JS popup is the `prompt` box. This captures data provided by the user, although the user may enter none or hit the Cancel button.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url = 'http://www.example.com';
my $jsPrompt = <<'END_OF_PROMPT';
var rtn = prompt("How many cookies may we set?");
localStorage.setItem("JSSelenium", rtn);
END_OF_PROMPT

my $jsGetItem = <<'END_OF_GET';
return localStorage.getItem("JSSelenium");
END_OF_GET

my $driver = Selenium::Chrome->new();
$driver->get($url);
for my $str ('1', '') {
    $driver->execute_script($jsPrompt);
    $driver->send_keys_to_prompt ($str);
    $driver->accept_alert;
```

```
    my $rtn = $driver->execute_script(
        $jsGetItem);
    is $rtn, $str,
        "<$str> and accept returned correctly";
}
$driver->execute_script($jsPrompt);
$driver->dismiss_alert;
my $rtn = $driver->execute_script($jsGetItem);
is $rtn, 'null',
    "Dismissing a confirmation returns null";
$driver->quit;
done_testing;
```

Unlike confirm, the `dismiss_alert` method results in a return of `null` rather than `false`. This is a peculiarity of JS rather than Perl or Selenium. The keys are sent to the popup by the method `send_keys_to_prompt`. Another method, `send_keys_to_alert`, is synonymous. There is no implicit validation of the data entered into the prompt box. It can be empty or otherwise invalid, but it will be different from the user clicking “Cancel” unless the user types in “null” and clicks OK.

The message sent to the user can be extracted from the popup.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url = 'http://www.example.com';
my $jsAlert = <<'END_OF_ALERT';
var rtn = alert("We use cookies. OK accepts this.");
localStorage.setItem("JSSelenium", rtn);
END_OF_ALERT

my $driver = Selenium::Chrome->new();
$driver->get($url);
$driver->execute_script($jsAlert);
my $rtn = $driver->get_alert_text;
is $rtn, 'We use cookies. OK accepts this.',
    'Text generated correctly';
$driver->accept_alert;
```

```
$driver->quit;
done_testing;
```

This is pointless in the simple example above as the text is given by the JS. It may be valuable in situations where the JS is being injected by the server of the web page and the code is intended to test that the correct popup appears in certain circumstances. The popup may also contain other text along the lines of “Example.com says”. There are no Selenium tools for extracting this, which may be browser dependent.

Note that, in all the examples in this section, it is necessary to open a URL. Failing to do so will result in a hash reference being returned, as will other errors.

12 Synchronous and Asynchronous Javascript

All the examples so far have used synchronous JS. A method, `execute_async_script`, exists to handle scripts that use asynchronous features of JS. The following example uses the method without any JS to demonstrate how the method handles failures.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url = 'http://www.example.com';
my $driver = Selenium::Chrome->new();
$driver->get($url);
$driver->set_async_script_timeout(2000);
my $rtn = eval
    {$driver->execute_async_script('');};
like $$rtn{message}, '/timeout/i',
    'Empty JS times out';
print $$rtn{message} . "\n";
$driver->quit;
done_testing;
```

It is not necessary to specify the length of wait, but in this case, to avoid wasting too much time, the timeout is set to 2 seconds (2000 milliseconds). The default is 30 seconds, but can be changed as needed. After the test is run, the message returned is printed out for information purposes. The next example is based on the documentation.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;
```

```
my $url = 'http://www.perlmonks.org';
my $js = <<'END_OF_JS';
var id = arguments[0];
var callback = arguments[arguments.length-1];
var elt = document.getElementById(id);
callback(elt);
END_OF_JS

my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->execute_async_script
    ($js, 'Log_In');
like $elt->get_text, '/^Log In/',
    'Login table found';
$driver->quit;
done_testing;
```

It is easy to misunderstand the documentation. A passage reads:

> The executed script is assumed to be asynchronous and must signal that is done by invoking the provided callback, which is always provided as the final argument to the function. The value to this callback will be returned to the client.

This is clearer if the word *always* is read to mean *automatically*. It is then possible to understand that the argument that creates the callback in the second line of the JS has been appended to the list of arguments supplied by the programmer. Similarly, it is the *output* of the callback, not anything provided *to* it, that is returned to the client. The example in the documentation seems incomplete, resulting in an error message ‘Can’t call method “click” on an unblessed reference’. The example above works, but is rendered problematic by the fact that the JS in the example does nothing that is asynchronous. The following example uses a somewhat contrived asynchronous feature.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;
```

```
my $url = 'http://www.perlmonks.org';
my $js = <<'END_OF_JS';
var delay = arguments[0];
var id = arguments[1];
var callback = arguments[arguments.length-1];
setTimeout(function(){
    var elt = document.getElementById(id);
    callback(elt);
}, delay * 1000);
END_OF_JS

my $driver = Selenium::Chrome->new();
$driver->get($url);
my $elt = $driver->execute_async_script
    ($js, 5, 'Log_In');
like $elt->get_text, '/^Log In/',
    'Login table found';
$driver->quit;
done_testing;
```

This imposes a five second delay before searching for the element. It may appear that the delay is less if the browser treats a page as being fully loaded before it has finished the rendering process.

The effect of this treatment of asynchronous processes is to render them synchronous to the Perl client. It is therefore not possible to use the callback process described here to avoid the process in the prompt and confirm examples in section 11.4 above. For more tools for controlling timings, see section 14.3 below.

13 More Data from Browsers

13.1 Page Source

Selenium can return the entire page to a variable with a single method. It also has two ways of extracting the body text, one of which uses techniques previously demonstrated.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.example.com/';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $page = $driver->get_page_source;
like $page, qr/You may use this/, 'OK to use';
my $body = $driver->get_body;
my $pagelen = length($page);
my $bodylen = length($body);
cmp_ok $pagelen, ">", $bodylen,
    "Page longer than body ($pagelen>$bodylen)";
my $elt = $driver->find_element(
    'body', 'tag_name')->get_text;
is $elt, $body,
    'get_body is simpler than using the driver';
$driver->quit;
done_testing;
```

13.2 Geolocation

This feature is more browser-dependent than most. There are two methods, `get_geolocation` and `set_geolocation`. The documentation for `get_geolocation` says:

> Note that your webdriver must implement this endpoint - otherwise, it will crash your session. At the time of release, we couldn't get this to work on the desktop FirefoxDriver or desktop Chromedriver.

It is not clear what is meant by "we couldn't get this to work". The documentation for `get_geolocation` says:

> note that your driver must implement this endpoint, or else it will crash your session. At the very least, it works in v2.12 of Chromedriver.

This implies that the feature is implemented but that the transfer of data to the browser fails and that trying will NOT crash the session. Using ChromeDriver 2.33, the session returns an error if get is called before set and then crashes. The documentation describes three location parameters that can be set, latitude, longitude and altitude, but the hash returned after setting includes a fourth, accuracy. This is undocumented and there is no indication whether it is Chrome specific. The equivalent code crashes using Firefox, stating that getGeolocation is not a known command. The Chrome code is:

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;
use Data::Dumper;

my $driver = Selenium::Chrome->new();
print Dumper $driver->get_geolocation;
print "Not crashed\n";
```

```
$driver->set_geolocation( location => {
    latitude => 40.714353,
    longitude => -74.005973,
});
print Dumper $driver->get_geolocation;
$driver->quit;
```

Commenting out the first `get_geolocation` command causes the code to run.

14 Driver Management

14.1 Error Handling

As was explained in section 5 above, an error message or warning will be raised if an attempt is made to locate an element that does not exist in the target web page. This can be a problem if the test is written before the element is added to the page, as is standard practice in Test Driven Development. The correct code action then would be neither a warning nor an error but a test failure. These can be trapped using Perl's block eval function.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
my $errcroak = eval {
    my $eltcroak = $driver->
        find_element('Cthulhu', 'id');
    1};
ok $errcroak, "Test should croak";
print $@ unless $errcroak;
$errcroak = eval {
    my $eltcroak = $driver->
        find_element('Voting_Booth', 'id');
    1};
ok $errcroak, "Test should NOT croak";
print $@ unless $errcroak;
$driver->quit();
done_testing;
```

Running this will result in one pass and one failure. Programmers who are doing many such tests or using an "Object Orientated" approach might consider more advanced test modules such as `Test::Exception`, `Test::Fatal` and `Test::Class`.

The code above has tested only for errors that would otherwise cause the code to die with any subsequent tests not run. The example above demonstrates that tests will continue to run even after a croak, but does not consider an action that fails with a warning. For that, there are several approaches. One is to promote warnings to errors with `use warnings FATAL => qw( all );`. This is lexically scoped and can be turned on and off wherever needed. Another way of changing the behaviour of warnings is to use `$SIG{__WARN__}`. The code below uses another test module, `Test::Warnings`.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;
use Test::Warnings;

my $url    = 'http://www.perlmonks.org';
my $driver = Selenium::Chrome->new();
$driver->get($url);
for ('Cthulhu', 'Voting_Booth') {
    my $elt = $driver->find_element_by_id($_);
    if (0 == $elt) {
        fail "Couldn't find element $_";
    } else {
        is $elt->get_attribute('id'), $_,
            "Found $_";
    }
}
$driver->quit();
done_testing;
```

This fails two tests, throwing a failure when it reaches the end and reporting that there was at least one unexpected warning. `Test::Warnings` does not indicate which test resulted in a warning. Perl follows its usual practice of displaying the warning before the problematic test This may be confusing to testers who are used to seeing the output of tests after the “ok” or “not ok” message. Therefore, should the locator fail to return an element object, a specific failure is raised to provide greater clarity. If an element is found, there is a double check to ensure that it is the one intended.

This has the effect of executing the same number of tests whichever path is followed, preventing problems for those who prefer to specify a test plan.

`Test::Warnings` has been preferred to `Test::Warn` because it handles `done_testing` correctly. Both have features for testing whether code produces a warning when it is supposed to.

Errors in methods other than element locators can be trapped by custom error handlers if desired. The error handler can be specified at creation or later and can be cleared (i.e. reverting to the default handler) as desired. This means that the default of `croak`ing on any error, causing the code to terminate, will no longer work unless it is added explicitly to the error handler. It was explained previously that the local storage methods cause problems. This is demonstrated in the code below.

```
use strict; use warnings;
use Selenium::Chrome;
use feature 'say';

my $url    = 'http://www.example.com/';
my $driver = Selenium::Chrome->new(
    error_handler => sub {
        warn @_;
        warn 'Warned by custom error handler';
    }
);
$driver->get($url);
$driver->get_local_storage_item('JSSelenium');
$driver->error_handler(
    sub {
        say @_;
        say 'Logged by custom error handler';
    }
);
$driver->get_local_storage_item('JSSelenium');
$driver->clear_error_handler;
$driver->get_local_storage_item('JSSelenium');
say 'This line not reached';
```

This results in the following output:

```
Selenium::Chrome=HASH(0x55a2697e5b48)Error while executing
command:
getLocalStorageItem: Server returned error message
unhandled request
instead of data at
/usr/local/share/perl/5.24.1/Selenium/Remote/Driver.pm line
327.
Warned by custom error handler at DebChErrHandler.pl line
13.
Selenium::Chrome=HASH(0x55a2697e5b48)Error while executing
command:
getLocalStorageItem: Server returned error message
unhandled request
instead of data at
/usr/local/share/perl/5.24.1/Selenium/Remote/Driver.pm line
327.
Logged by custom error handler
Error while executing command: getLocalStorageItem: Server
returned
error message unhandled request instead of data at
/usr/local/share/perl/5.24.1/Selenium/Remote/Driver.pm line
327.
at /usr/local/share/perl/5.24.1/Selenium/Remote/Driver.pm
line 327.
```

This demonstrates not only the setting, changing and clearing of error handlers but also the fact that the default error handler terminates execution while the custom ones, which do not contain `croak` or `die`, do not. The hash returned is the driver object. Note that, while an error in a locator method will be handled by the custom error handler, it will subsequently be handled by its own, which will croak.

14.2 Debugging

Selenium has a built-in debugging feature that sends information to stdout.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.example.com/';
my $driver = Selenium::Chrome->new();
$driver->debug_on;
$driver->get($url);
$driver->debug_off;
$driver->refresh;
$driver->quit;
```

This returns the output below.

```
  Prepping get
  Executing get
  REQ: POST,
  http://127.0.0.1:9515/wd/hub/session/01d346fbc807a8bb112d8b
  12cb69257c/url,
  {"url":"http://www.example.com/"}
  RES:
  {"sessionId":"01d346fbc807a8bb112d8b12cb69257c","status":0,
  "value":null}
```

No data has been returned from the `refresh` command because debugging had been turned off at that point. There is no facility to set a custom debugger as there is for the error handler. It is generally possible to redirect stdout to a file if desired. For a solution within Perl, consider `Capture::Tiny`.

14.3 Timings

Timeouts for asynchronous Javascript have been discussed in section 12 above. The documentation includes references to two other methods for controlling timings.

The `set_timeout` method can be used to set three different timeouts. The format is `$driver->set_timeout(type, ms)`, where `type` is one of `script`, `implicit` and `page load` and `ms` is the time limit in milliseconds. The most likely of these to be useful in practice is the `page load`. It enables programmers to raise errors should pages take unreasonably long to load. `implicit` sets a timeout for element searches. The default is zero, and users are unlikely to need to change it in normal circumstances. `script` has no effect on asynchronous scripts, which have a different method for setting timeouts, while synchronous scripts are unlikely to have timing problems. But these tools exist for the rare occasions when they are essential.

The second method, `set_implicit_wait_timeout`, is a synonym for `set_timeout('implicit', ms)`.

14.4 Status

This is slightly misleadingly named. It returns information about the environment the driver is currently using. A hash reference is returned, containing two keys, each of which contains another hash. The documentation says that one of these, “build”, should say “when the server was built”. This is certainly not returned from all implementations. Printing the hash reference using `Data::Dumper` is suggested by the docs and gives less than a screenful of output. This method should not be confused with the `cache_status` method discussed in section 14.6 below.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $driver = Selenium::Chrome->new();
for my $key ('os', 'build') {
    ok grep(/$key/, keys %{$driver->status}),
        "Found $key";
}
is scalar keys %{$driver->status}, 2,
    'No strange keys';
$driver->quit;
done_testing;
```

14.5 Logs

The documentation states that Selenium expects at least four types of log in every browser, but will return more if they exist. To find out what is available, use the `get_log_types` method, which returns an array reference. The four that are expected are browser, client, driver and server. The Selenium documentation is more detailed and states that “logs for all these nodes may not be available in all configurations”. The following code returns 3 failures and indicates that only the browser and driver logs exist in Chrome. This is not affected by fetching a web page.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $driver = Selenium::Chrome->new();
my $reflogs = $driver->get_log_types;
my $logcount = scalar @$reflogs;
cmp_ok $logcount, '>', 3,
    "At least 4 logs available ($logcount)";
for my $log
('browser', 'client', 'driver', 'server') {
    ok grep(/$log/, @$reflogs), "Found $log";
}
$driver->quit;
done_testing;
```

Logs are read using the `get_log` method. The documentation is ambiguous about whether an array or a reference to one is returned, implying that this may be browser dependent. Chrome returns array references in all tested cases. It also states that the return will include entries “since the most recent request”, which might be taken to mean the last browser request or the last request for the log contents. Experimentation has failed to extract any data using this method. The Selenium documentation referred to above lists six log levels that may be used, but there appears to be no way to set the

log level, which may well default to “OFF”. Examination of the driver object has revealed no clues. Firefox does not recognise the `get_log_types` method while the following code for Chrome returns empty arrays in every case.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url = 'http://www.example.com/';
my $driver = Selenium::Chrome->new();
my $reflogs = $driver->get_log_types;
for (0..1) {
    for my $log (@$reflogs) {
        is scalar @{$driver->get_log ($log)}, 0,
            'Empty log returned';
    }
    $driver->get($url) if 0 == $_;
}
$driver->quit;
done_testing;
```

14.6 Cache

The behaviour of this differs substantially from the documentation. Firefox crashes, claiming that `cacheStatus` “did not match a known command”. Chrome, on the other hand, returns exactly the same output as in section 14.4 above. The documentation says that the status of the HTML5 cache should be returned, matching the Firefox error message if not its behaviour. It states that the return should be an integer from 0 to 5, having meanings listed in the table below.

```
0    UNCACHED
1    IDLE
2    CHECKING
3    DOWNLOADING
4    UPDATE_READY
5    OBSOLETE
```

The incomplete (but working), experimental code used in identifying the behaviours described is given below.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.example.com/';
my @STATUS = ('UNCACHED', 'IDLE', 'CHECKING',
              'DOWNLOADING', 'UPDATE_READY', 'OBSOLETE');
my $driver = Selenium::Chrome->new();
my $idx = $driver->cache_status;
is $STATUS[$idx], 'IDLE',
    'Nothing to see yet';
$driver->get($url);
$idx = $driver->cache_status;
use Data::Dumper;
print Dumper $idx;
$driver->quit;
done_testing;
```

This seems to be a function of changing HTML5 standards. Mozilla states:

> Deprecated
>
> This feature has been removed from the Web standards. Though some browsers may still support it, it is in the process of being dropped. Avoid using it and update existing code if possible; see the compatibility table at the bottom of this page to guide your decision. Be aware that this feature may cease to work at any time.

14.7 Capabilities

Extracting a hash of capabilities from a browser is trivial.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $driver = Selenium::Chrome->new();
is $driver->get_capabilities()->{browserName},
    'chrome', 'Correct browser';
my $keycount = scalar keys %{$driver->
    get_capabilities()};
cmp_ok $keycount, '>', 10,
    "Lots of elements ($keycount)";
$driver->quit;
done_testing;
```

There is a list of which capabilities apply to which browsers on Selenium's Github site. It also lists which are writable and which are read-only. The capabilities hash gives complete control over browser creation, meaning that there will be no assumptions or user preferences set. This can help circumvent some of the issues described in section 8 above.

There are, though, a number of issues in using the capabilities as extracted from an existing browser instance. Trying to use the hashref returned without modification would not create a browser instance. The first problem is that, instead of simple true and false values such as 1 and 0, the hashref contains two references to JSON objects which evaluate to 1 and 0. Every other capability that is true or false then points to the capability containing the JSON object. This cannot be passed back to a creation process without the references being reduced to simple values. Since the hashref may contain keys pointing to further hashes, the procedure to this must

call itself recursively.

The next issue is that it is possible for some of the keys in the hashref returned to have invalid values. In the case of Chrome running on Debian, one of these is the `unexpectedAlertBehaviour` key. In the code above, its value is a zero length string. The github documentation mentioned above lists the allowable values as "“accept”, “dismiss” and “ignore”". If none of these are used, it is also acceptable to delete the key. Using the zero length string returned, however, will result in a failure to create a browser. The code below demonstrates both the dereferencing and the correcting of the zero length string.

```
use strict; use warnings;
use Test::More;
use Selenium::Chrome;

my $url    = 'http://www.example.com/';
my $driver = Selenium::Chrome->new();
my $hr_caps = $driver->get_capabilities();
$driver->quit;
$hr_caps->{'unexpectedAlertBehaviour'}
    = 'ignore';
refeval($hr_caps);
delete $hr_caps->{acceptInsecureCerts};
$driver = Selenium::Chrome->new_from_caps(
    'desired_capabilities' => {%$hr_caps});
$driver->get($url);
is $driver->get_title(), 'Example Domain',
    'Title correct';
$hr_caps = $driver->get_capabilities();
refeval($hr_caps);
is $hr_caps->{'unexpectedAlertBehaviour'},
    'ignore', 'Capability as set';
$driver->quit;
done_testing;

sub refeval {
    my $href = shift;

    for my $key(keys %$href) {
        if ('HASH' eq ref($href->{$key})) {
            refeval($href->{$key});
```

```
        } elsif ('' ne ref($href->{$key})) {
            $href->{$key} = eval($href->{$key});
        }
    }
}
```

The code above was working in version 1.26 without deleting the the `acceptInsecureCerts` key/value pair, but the `new_from_caps` method no longer creates a session instance in 1.27 unless it is deleted.

If there are multiple sessions running from a single driver, capabilities can be extracted as an array of hashes using the `get_sessions` method. In addition to the capabilities, each top level array element will have an 'id' key, the value of which will be the session ID. This is not available if the browser sessions are in different drivers, even if these are in an array as suggested.

The capabilities functionality is the only way to establish whether the browser requested is the one actually created on a remote machine. Assuming a running instance of Selenium Standalone Server, the following code may identify problems.

```
use strict; use warnings;
use Selenium::Remote::Driver;
use Test::More;

my $url = 'http://www.perlmonks.org';
for my $name (# 'phantomjs', 'firefox',
    # 'internet explorer', 'MicrosoftEdge',
    # 'safari', 'iphone',
    'htmlunit', 'chrome') {
    my $driver = Selenium::Remote::Driver->new(
        remote_server_addr => '127.0.0.1',
        port               => '4444',
        browser_name       => $name
    );
    $driver->get($url);
    is $driver->get_title(),
        'PerlMonks - The Monastery Gates',
```

```
        "Title as expected for $name";
    is $driver->get_capabilities()->{browserName},
        $name, 'Correct browser';
    $driver->quit();
}
done_testing;
```

All the browsers listed in the `Selenium::Remote::Driver` documentation are included in the code above, although those that cause problems have been commented out. Firefox runs but will not close, while the others are proprietary. Browsers that have been commented out cause the client to crash, preventing further tests. The results are likely to vary depending on the software installed on the target machine and are certain to vary between operating systems. In earlier versions, `iphone` and `phantomjs` appeared to run, but they passed the first test (reading the page) and failed the second, running Chrome by default. This has changed in `Selenium::Remote::Driver` 1.27 and/or Selenium standalone server 3.9.1.

14.8 Available Engines

This is documented in `Selenium::Remote::Driver`, but neither Chrome nor Firefox nor Selenium Standalone Server recognise it. The documentation states that this cannot be used with Webdriver3 servers

14.9 Writing plugins to return drivers

Four files are required to demonstrate this, two of which should be put in a subdirectory called `Plugins`. These two files appear next and are almost identical. They should be saved as `Chrome.pm` and `Firefox.pm`.

```
package Plugins::Chrome;
use strict; use warnings;
use Selenium::Chrome;
use Moose;
use namespace::autoclean;

has driver => (
    is => 'rw',
);
sub BUILD {
    my $self = shift;
    my %args = %$_[0];
    $self->{driver} =
        Selenium::Chrome->new(%args);
}
__PACKAGE__->meta->make_immutable;
1;
```

Firefox.pm:

```
package Plugins::Firefox;
use strict; use warnings;
use Selenium::Firefox;
use Moose;
use namespace::autoclean;

has driver => (
    is => 'rw',
);
sub BUILD {
    my $self = shift;
    my %args = %$_[0];
    $self->{driver} =
        Selenium::Firefox->new(%args);
```

```
}
__PACKAGE__->meta->make_immutable;
1;
```

The next two files should be in the parent directory of the plugins. The first should be saved as `Launcher.pm`.

```
package Launcher;
use strict; use warnings;
use Moose;
use Module::Find;

has plug => (
    is => 'rw',
);

sub _findplugins {
    my ($self, $browsername) = @_;
    my @found = useall Plugins;
    for my $plugin (@found) {
        if ($plugin =~ /::\Q$browsername\E$/i) {
            $self->{plug} = $plugin;
            last;
        }
    }
}

sub launch {
    my ($self, $browsername, $hr_params) = @_;
    my %params;
    %params = %$hr_params if defined $hr_params;
    $self->_findplugins($browsername);
    my $object = $self->plug->new(%params);
    return $object->driver;
}
1;
```

The name of the last file is not important. It demonstrates the use of the two plugins.

```
use strict; use warnings;
use Selenium::Remote::Driver;
use Test::More;
use lib ".";
use Launcher;

my $url = 'http://www.perlmonks.org';
my $launcher = Launcher->new();
for my $browser ('Chrome', 'Firefox') {
    my $driver = $launcher->launch($browser);
    $driver->get('http://www.perlmonks.org');
    is $driver->get_title(),
        'PerlMonks - The Monastery Gates',
        "Title as expected for $browser";
    $driver->close();
}
```

What will happen when this file is run is that a launcher object will be created. Next, a loop will be entered for first the Chrome and next the Firefox browser. The launcher object will be told to launch the browser and return the driver. Then a very simple pair of commands is run through the driver to demonstrate that it is working as expected. These commands could be far more complicated and different loops might run for different browsers depending on functionality.

Within the launcher, the `launch` method sorts out any parameters that are sent and instructs an internal method, `_findplugins`, to use `Module::Find` to look through all available plugins and return the one that matches the name parameter passed in. A match is assumed; this simple example makes no provision for validation or error handling. The `launch` method then creates a new object containing a driver using the plugin found and passing any parameters specified.

The plugins are very simple Moose objects, but can be extended if needed. The point about this example is that to add a new browser, it is necessary to write a new plugin once. After that, all that is needed

is to name the plugin in the code that does the work and leave the rest to Launcher.pm. It is not necessary to recode Launcher.pm; it will search and find the plugins automatically if they are in the appropriate directory.

15 Remote Servers

Starting a browser specific driver such as chromedriver or geckodriver from the command line is possible, but does not result in a server that will accept connections from Selenium, although other tools can be used to reveal valuable debugging information. This is because these are drivers that also act as servers rather than pure servers. Selenium Standalone Server is the recommended tool for situations where the server and client are on separate machines, especially since it can be reached by browser specific driver modules like `Selenium::Chrome`. It is possible to emulate the behaviour of two machines on a single machine, and this will be done in the examples in the rest of this section.

15.1 Creating

As stated in section 2.2.1 above, the server can be started with the command

```
java -jar selenium-server-standalone-3.9.1.jar
```

This will produce a lot of information reports both at boot time and when connections are made. When booting, the last report should be that the server is up and running. By default, it will accept connections on port 4444, but this, along with the reporting level, can be changed by using options when starting the server.

15.2 Connecting

```
use strict; use warnings;
# use Selenium::Remote::Driver;
use Selenium::Chrome;
use Test::More;

my $url = 'http://www.perlmonks.org';
# my $driver = Selenium::Remote::Driver->new(
my $driver = Selenium::Chrome->new(
    remote_server_addr => '127.0.0.1',
    port               => '4444',
  # browser_name       => 'chrome'
);
$driver->get($url);
is $driver->get_title(),
    'PerlMonks - The Monastery Gates',
    'Title as expected';
$driver->quit();
done_testing;
```

This code includes several lines that have been commented out. In the version that appears above, it will make a connection to the "remote" server running on the same machine, open an instance of Chrome, load the PerlMonks home page, run a test and exit.

If it is changed to use `Selenium::Remote::Driver` instead of `Selenium::Chrome`, the same thing should happen, but with the important difference that Firefox will be used, as it is the server's default unless otherwise specified. `Selenium::Remote::Driver`'s failure correctly to communicate with Firefox causes increased problems in this configuration. While a Firefox instance will be opened, communication from Selenium standalone server fails, reporting an unknown driver version. The error results in the termination of the client code, leaving the browser instance open. This means that the `done_testing` line is never reached. Tests, too,

cannot be reached.

The behaviour of defaulting to Firefox can be overridden not only on the machine running the server but also by the client. The example above shows the simplest way to do this in the final comment line. Simply specifying the name of a recognised browser is enough to invoke that browser with its usual defaults.

When connecting, it is essential that all the necessary communications between the two machines are open. Proxy settings can be changed by giving the constructor the necessary parameters, but it is up to the programmer and systems staff to ensure that ports are open and that firewalls do not prevent packet transmission.

15.3 Uploading

Selenium contains an upload_file method that allows files to be placed on the remote server. The following code needs multiple command prompts. One is to launch the server. It is preferable to run the server in a window of its own rather than in the background, as the logging information helps to explain what is happening. The second window is needed to run the code, which pauses to require the user to hit a key. The third is to examine the uploaded file.

```
use strict; use warnings;
use Selenium::Remote::Driver;

open my $fh, '>', 'upfile.txt' or die
    "Can't open upfile.txt for output: $!";
print $fh "File contents\n";
close $fh;
my $driver = Selenium::Remote::Driver->new(
    remote_server_addr => '127.0.0.1',
    port               => '4444',
    browser_name       => 'chrome'
);
my $remote_file_name =
    $driver->upload_file('upfile.txt');
print "$remote_file_name\n";
my $wait = <STDIN>;
unlink 'upfile.txt';
$driver->quit();
```

This code creates a very simple text file in the same directory as the code. Then, after connecting to the remote server, it uploads the file, prints its remote name and pauses with the browser still open. At this point, it is possible to use the third command line window to verify that the file is as expected. On Debian, it is saved to a directory created at run time in the `/tmp/` directory. This directory is deleted, with all its contents, when the `$driver->quit` command is executed. It is this that makes a pause necessary. Having inspected the file, hitting `Enter` will allow the script to continue and close the browser.

The uploaded file and its parent directory will now have been deleted automatically and the data file created for the example is also deleted, this time explicitly.

The documentation says:

> Passing raw data as an argument past the filename will upload that rather than the file's contents.
>
> When passing raw data, be advised that it expects a zipped and then base64 encoded version of a single file. Multiple files and/or directories are not supported by the remote server.

As can be seen from the example, a simple text file can be uploaded without the need to zip or encode.

16 References

16.1 Selenium modules

- http://search.cpan.org/~teodesian/Selenium-Remote-Driver/lib/Selenium/Remote/Driver.pm
- http://search.cpan.org/~teodesian/Selenium-Remote-Driver/lib/Selenium/Chrome.pm
- http://search.cpan.org/~teodesian/Selenium-Remote-Driver/lib/Selenium/Firefox.pm
- http://search.cpan.org/~teodesian/Selenium-Remote-Driver/lib/Selenium/Remote/WebElement.pm

16.2 Other Perl modules

- http://search.cpan.org/~dagolden/Capture-Tiny/lib/Capture/Tiny.pm
- http://search.cpan.org/perldoc?Module%3A%3AFind
- http://search.cpan.org/perldoc?Test%3A%3AMore
- http://search.cpan.org/perldoc?Test%3A%3AWarnings
- https://www.youtube.com/watch?v=V3WeO-iVkAc (WWW::Mechanize::Chrome)

16.3 Selenium resources

- http://www.seleniumhq.org/docs/03_webdriver.jsp
- https://github.com/operasoftware/operachromiumdriver
- https://github.com/SeleniumHQ/selenium/wiki/Logging
- https://github.com/mozilla/geckodriver/releases
- http://selendroid.io/
- https://github.com/SeleniumHQ/selenium/wiki/DesiredCapabilities
- https://sourceforge.net/projects/htmlunit/files/htmlunit/2.29/
- http://selenium-release.storage.googleapis.com/index.html (downloads for IEServerDriver, Selenium Standalone Server & others)
- https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/

16.4 W3schools resources

- https://www.w3schools.com/js/js_cookies.asp
- https://www.w3schools.com/tags/tryit.asp?filename=tryhtml_input_checked
- https://www.w3schools.com/html/tryit.asp?filename=tryhtml_form_radio
- https://www.w3schools.com/tags/tryit.asp?filename=tryhtml_input_disabled
- https://www.w3schools.com/tags/tryit.asp?filename=tryhtml_select
- https://www.w3schools.com/js/js_popup.asp
- https://www.w3schools.com/cssref/

16.5 Other resources

- http://www.sscce.org (how to ask & answer a question)
- https://www.freeformatter.com/xpath-tester.html
- https://jqueryui.com/droppable/ (drag & drop demo)
- https://validator.w3.org/ (HTML validation)
- http://www.perlmonks.org/
- https://developers.google.com/web/fundamentals/primers/promises
- https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache

17 Index

- Selendroid 2.2.11
- Selenium
  - Grid 2
  - IDE 2
  - Selenium::Remote::RemoteConnection 2.2.5
  - Selenium::Chrome 2.1
  - Selenium::Firefox 2.1
  - Selenium::Remote::Driver 2.1
  - Selenium::Remote::WDKeys 4.3
  - Selenium::Screenshot 4.4
  - Test::Selenium::Remote::Driver 3
- Selenium Standalone Server 2.2.1, 15
- SSCCE 6.2
- Tests 3
  - cmp_ok 5.2
  - is 3
  - is_deeply 5.2
  - isn't 5.2
  - ok 5.2
  - skip 11.2
  - Test::Class 14.1
  - Test::Exception 14.1
  - Test::Fatal 14.1
  - Test::More 3
  - Test::Selenium::Remote::Driver 3
  - Test::Warn 14.1
  - Test::Warnings 14.1
  - unlike 5.4
- W3schools 6.2, 6.3, 6.5, 7.1, 7.2