

# Working with Selenium

SELENIUM TRAINING

## Working with Selenium

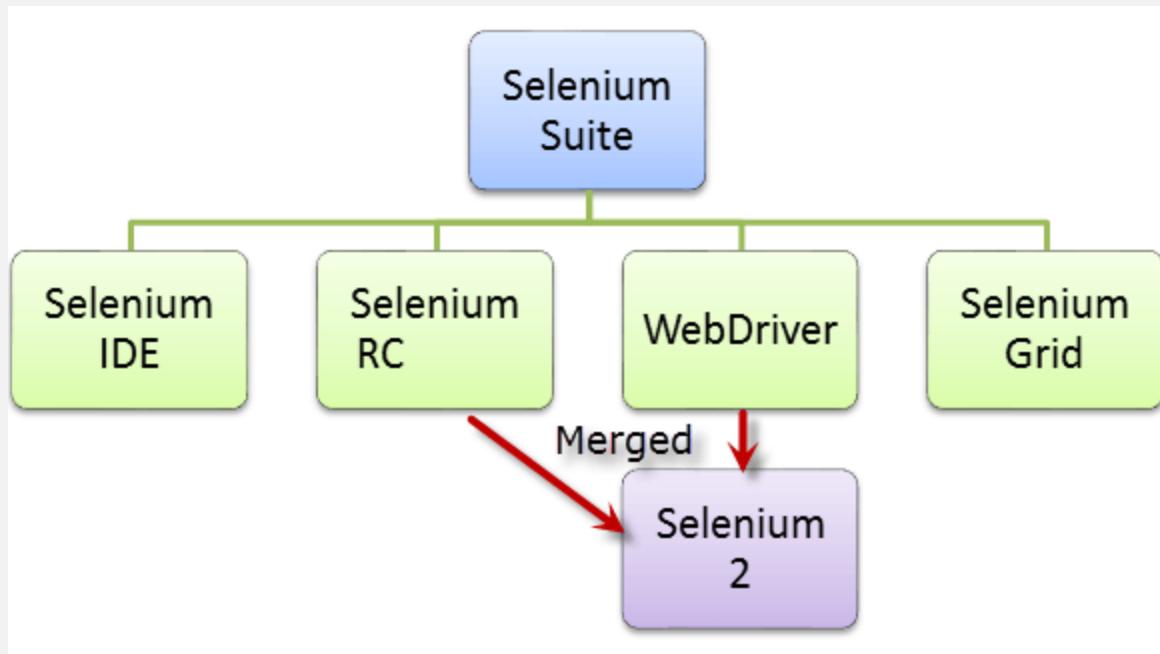
### What is Selenium?

**Selenium is a free (open source) automated testing suite for web applications across different browsers and platforms.** It is quite similar to HP Quick Test Pro (QTP) only that Selenium focuses on automating web-based applications.

Selenium is not just a single tool but a suite of software's, each catering to different testing needs of an organization.

**It has four components.**

- Selenium Integrated Development Environment (IDE)
- Selenium Remote Control (RC)
- WebDriver
- Selenium Grid

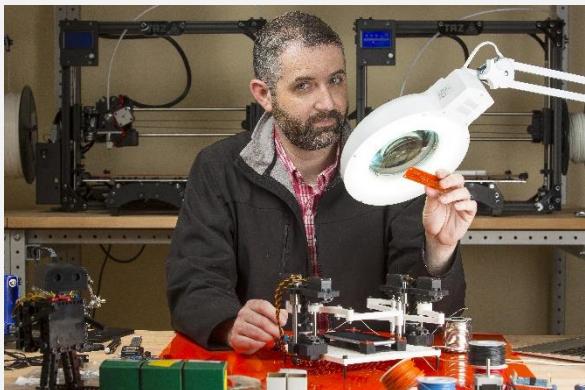


## Introduction to Selenium

At the moment, Selenium RC and WebDriver are merged into a single framework to form **Selenium 2**. Selenium 1, refers to Selenium RC.

## Who developed Selenium?

Since Selenium is a collection of different tools, it had different developers as well. Below are the key persons who made notable contributions to the Selenium Project.



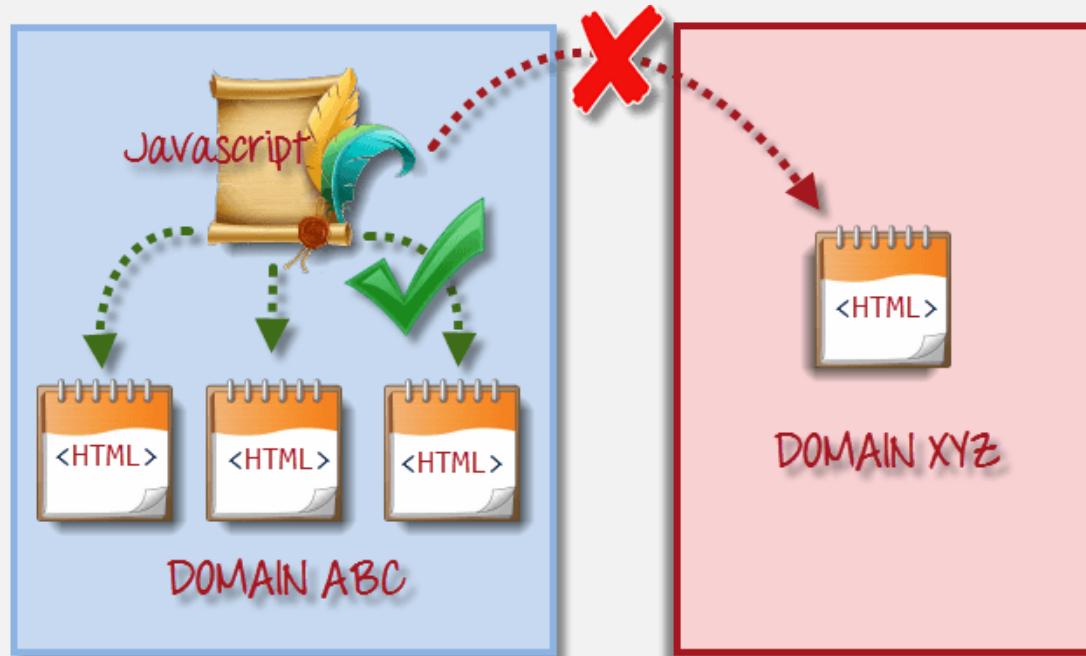
Primarily, Selenium was **created by Jason Huggins in 2004**. An engineer at ThoughtWorks, he was working on a web application that required frequent testing. Having realized that the repetitive manual testing of their application was becoming more and more inefficient, he created a JavaScript program that would automatically control the browser's actions. He named this program as the "**JavaScriptTestRunner**."

Seeing potential in this idea to help automate other web applications, he made JavaScriptRunner open-source which was later re-named as **Selenium Core**.

## The Same Origin Policy Issue

**Same Origin policy prohibits JavaScript code from accessing elements from a domain that is different from where it was launched.** Example, the HTML code in www.google.com uses a JavaScript program "randomScript.js". The same origin policy will only allow randomScript.js to access pages within google.com such as google.com/mail, google.com/login, or google.com/signup.

However, it cannot access pages from different sites such as yahoo.com/search because they belong to different domain.



under Same Origin Policy, a Javascript program can only access pages on the same domain where it belongs. It cannot access pages from different domains

## The Same Origin Policy Issue

This is the reason why prior to Selenium RC, testers needed to install local copies of both Selenium Core (a JavaScript program)

## Birth of Selenium Remote Control (Selenium RC)

**Paul Hammant**



Unfortunately; testers using Selenium Core had to install the whole application under test and the web server on their own local computers because of the restrictions imposed by the **same origin policy**. So another Thought Work's engineer, **Paul Hammant**, decided to create a server that will act as an HTTP proxy to "trick" the browser into believing that Selenium Core and the web application being tested come from the same domain. This system became known as the **Selenium Remote Control or Selenium 1**.

## Birth of Selenium Grid

**Patrick Lightbody**



Selenium Grid was developed by **Patrick Lightbody** to address the need of minimizing test execution times as much as possible. He initially called the system "**Hosted QA**." It was capable of capturing browser screenshots during significant stages, and also of **sending out Selenium commands to different machines simultaneously**.

## Birth of Selenium IDE



**Shinya Kasatani** of Japan created **Selenium IDE**, a Firefox extension that can automate the browser through a record-and-playback feature. He came up with this idea to further increase the speed in creating test cases. He donated Selenium IDE to the Selenium Project in **2006**.

## Birth of WebDriver



**Simon Stewart**

**Simon Stewart** created WebDriver circa **2006** when browsers and web applications were becoming more powerful and more restrictive with JavaScript programs like Selenium Core. **It was the first cross-platform testing framework that could control the browser from the OS level.**

## Birth of Selenium 2

In 2008, the whole Selenium Team decided to merge WebDriver and Selenium RC to form a more powerful tool called **Selenium 2**, with **WebDriver being the core**. Currently, Selenium RC is still being developed but only in maintenance mode. Most of the Selenium Project's efforts are now focused on Selenium 2.

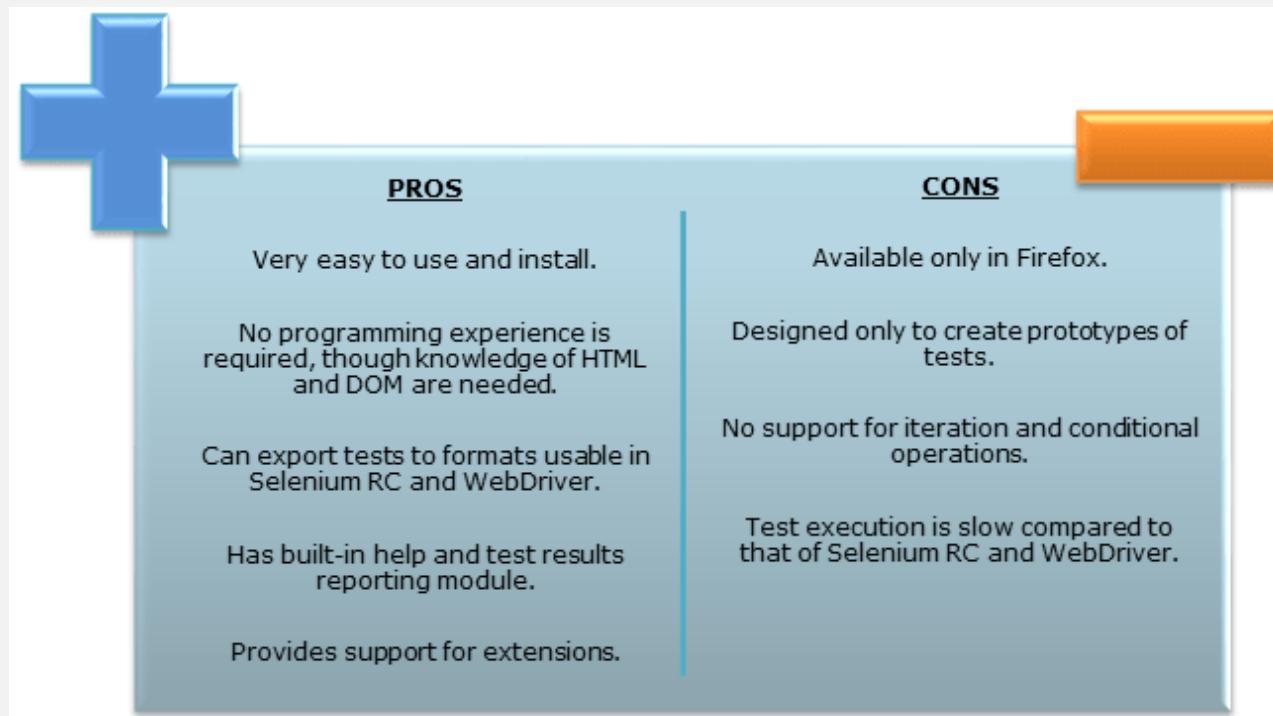
## So, Why the Name Selenium?

It came from a joke which Jason cracked one time to his team. Another automated testing framework was popular during Selenium's development, and it was by the company called **Mercury Interactive** (yes, the company who originally made QTP before it was acquired by HP). Since Selenium is a well-known antidote for Mercury poisoning, Jason suggested that name. His teammates took it, and so that is how we got to call this framework up to the present.



## Brief Introduction Selenium IDE

Selenium **Integrated Development Environment** (IDE) is the **simplest framework** in the Selenium suite and is **the easiest one to learn**. It is a **Firefox plugin** that you can install as easily as you can with other plugins. However, because of its simplicity, Selenium IDE should only be used as a **prototyping tool**. If you want to create more advanced test cases, you will need to use either **Selenium RC** or **WebDriver**.



## Brief Introduction Selenium Remote Control (Selenium RC)

Selenium RC was the **flagship testing framework** of the whole Selenium project for a long time. This is the first automated web testing tool that **allowed users to use a programming language they prefer**. As of version 2.25.0, RC can support the following programming languages:

- Java
- C#
- PHP
- Python
- Perl
- Ruby

## Pros

Cross-browser and cross-platform

Can perform looping and conditional operations

Can support data-driven testing

Has matured and complete API

Can readily support new browsers

Faster execution than IDE

## Cons

Installation is more complicated than IDE

Must have programming knowledge

Needs Selenium RC Server to be running

API contains redundant and confusing commands

Browser interaction is less realistic

Inconsistent results & Uses Javascript

Slower execution time than WebDriver

## Brief Introduction WebDriver

The WebDriver proves itself to be **better than both Selenium IDE and Selenium RC** in many aspects. It implements a more modern and stable approach in automating the browser's actions. WebDriver, unlike Selenium RC, does not rely on JavaScript for automation. **It controls the browser by directly communicating to it.**

The supported languages are the same as those in Selenium RC.

- Java
- C#
- PHP
- Python
- Perl
- Ruby

Pros	Cons
<ul style="list-style-type: none"><li>Simpler installation than Selenium RC</li><li>Communicates directly to the browser</li><li>Browser interaction is more realistic</li><li>No need for a separate component such as the RC Server</li><li>Faster execution time than IDE and RC</li></ul>	<ul style="list-style-type: none"><li>Installation is more complicated than Selenium IDE</li><li>Requires programming knowledge</li><li>Cannot readily support new browsers</li><li>Has no built-in mechanism for logging runtime messages and generating test results</li></ul>

## Selenium Grid

Selenium Grid is a tool **used together with Selenium RC to run parallel tests** across different machines and different browsers all at the same time. Parallel execution means running multiple tests at once.

### Features:

- Enables **simultaneous running of tests in multiple browsers and environments**.
- **Saves time** enormously.
- Utilizes the **hub-and-nodes** concept. The hub acts as a central source of Selenium commands to each node connected to it.

## Note on Browser and Environment Support

Because of their architectural differences, Selenium IDE, Selenium RC, and WebDriver support different sets of browsers and operating environments.

	Selenium IDE	Selenium RC	WebDriver
Browser Support	Mozilla Firefox	Mozilla Firefox Internet Explorer Google Chrome Safari Opera Konqueror Others	Internet Explorer versions 6 to 9, both 32 and 64-bit  Firefox 3.0, 3.5, 3.6, 4.0, 5.0, 6, 7 and above (current version is 16.0.1)  Google Chrome 12.0.712.0 and above (current version is 22.0.1229.94 m)  Opera 11.5 and above (current version is 12.02)  Android - 2.3 and above for phones and tablets

			(devices & emulators)
			iOS 3+ for phones (devices & emulators) and 3.2+ for tablets (devices & emulators)
<b>Operating System</b>	Windows Mac OS X Linux	Windows Mac OS X Linux Solaris	HtmlUnit 2.9 and above (current version is 2.10)  All operating systems where the browsers above can run.

## How to Choose the Right Selenium Tool for Your Need

Tool	Why Choose?
Selenium IDE	<ul style="list-style-type: none"><li>• To learn about concepts on automated testing and Selenium, including:<ul style="list-style-type: none"><li>• Selenese commands such as type, open, clickAndWait, assert, verify, etc.</li><li>• Locators such as id, name, xpath, css selector, etc.</li><li>• Executing customized JavaScript code using runScript</li><li>• Exporting test cases in various formats.</li></ul></li><li>• To create tests with little or no prior knowledge in programming.</li><li>• To create simple test cases and test suites that you can export later to RC or WebDriver.</li><li>• To test a web application against Firefox only.</li></ul>
Selenium RC	<ul style="list-style-type: none"><li>• To design a test using a more expressive language than Selenese</li><li>• To run your test against different browsers (except HtmlUnit) on different operating systems.</li><li>• To deploy your tests across multiple environments using Selenium Grid.</li><li>• To test your application against a new browser that supports JavaScript.</li><li>• To test web applications with complex AJAX-based scenarios.</li></ul>
WebDriver	<ul style="list-style-type: none"><li>• To use a certain programming language in designing your test case.</li><li>• To test applications that are rich in AJAX-based functionalities.</li><li>• To execute tests on the HtmlUnit browser.</li><li>• To create customized test results.</li></ul>
Selenium Grid	<ul style="list-style-type: none"><li>• To run your Selenium RC scripts in multiple browsers and operating systems simultaneously.</li><li>• To run a huge test suite, that need to complete in soonest time possible.</li></ul>

## A Comparison between Selenium and QTP

**Quick Test Professional (QTP)** is a proprietary automated testing tool previously owned by the company **Mercury Interactive** before it was **acquired by Hewlett-Packard in 2006**. The Selenium Tool Suite has many advantages over QTP (as of version 11) as detailed below -

Advantages of Selenium over QTP

Selenium	QTP
<b>Open source, free to use, and free of charge.</b>	<b>Commercial.</b>
<b>Highly extensible</b>	Limited add-ons
Can run tests across <b>different browsers</b>	Can only run tests in <b>Firefox , Internet Explorer and Chrome</b>
Supports <b>various operating systems</b>	Can only be used in <b>Windows</b>
Supports <b>mobile devices</b>	Supports mobile device using 3 <sup>rd</sup> party software
Can execute tests <b>while the browser is minimized</b>	Needs to have the application under test to be visible on the desktop
Can execute tests <b>in parallel</b> .	Can only execute in parallel but using Quality Center which is again a paid product.

# A Comparison between Selenium and QTP

Advantages of QTP over Selenium

QTP	Selenium
Can test <b>both web and desktop applications</b>	Can only test web applications
Comes with a <b>built-in object repository</b>	Has no built-in object repository
<b>Automates faster than Selenium</b> because it is a fully featured IDE.	Automates at a slower rate because it does not have a native IDE and only third party IDE can be used for development
Data-driven testing is easier to perform because <b>it has built-in global and local data tables</b> .	Data-driven testing is more cumbersome since you have to rely on the programming language's capabilities for setting values for your test data
<b>Can access controls within the browser</b> (such as the Favorites bar, Address bar, Back and Forward buttons, etc.)	Cannot access elements outside of the web application under test
Provides professional <b>customer support</b>	No official user support is being offered.
Has native capability to <b>export test data</b> into external formats	Has no native capability to export runtime data onto external formats
Parameterization Support is in built	Parameterization can be done via programming but is difficult to implement.
Test Reports are generated automatically	No native support to generate test /bug reports.

Though clearly, QTP has more advanced capabilities, Selenium outweighs QTP in three main areas:

- **Cost**(because Selenium is completely free)
- **Flexibility**(because of a number of programming languages, browsers, and platforms it can support)
- **Parallel testing**(something that QTP is capable of but only with use of Quality Center)

## Summary

- The entire Selenium Tool Suite is comprised of four components:
  - **Selenium IDE**, a Firefox add-on that you can only use in creating relatively simple test cases and test suites.
  - **Selenium Remote Control**, also known as **Selenium 1**, which is the first Selenium tool that allowed users to use programming languages in creating complex tests.
  - **WebDriver**, the newer breakthrough that allows your test scripts to communicate directly to the browser, thereby controlling it from the OS level.
  - **Selenium Grid** is also a tool that is used with Selenium RC to execute parallel tests across different browsers and operating systems.
- Selenium RC and WebDriver was merged to form **Selenium 2**.
- Selenium is more advantageous than QTP in terms of **costs and flexibility**. It also allows you to **run tests in parallel**, unlike in QTP where you are only allowed to run tests sequentially.

# Installing Selenium IDE & Firebug

SELENIUM TRAINING

## Installing Selenium IDE & Firebug

# Installation of Selenium IDE

What you need

- Mozilla Firefox
- Active Internet Connection

If you do not have Mozilla Firefox yet, you can download it from <http://www.mozilla.org/en-US/firefox/new>.

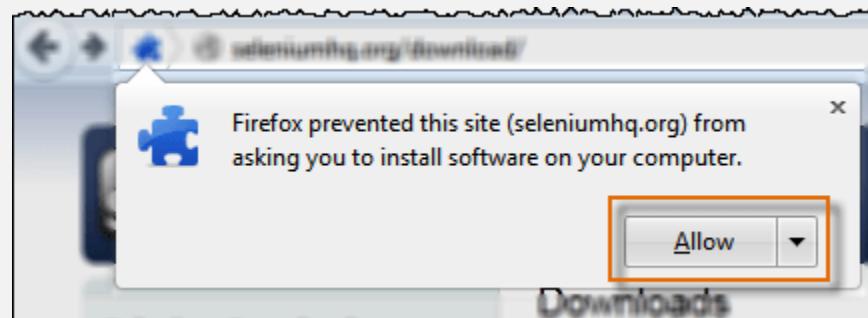
## Steps

Launch Firefox and navigate to <http://seleniumhq.org/download/>. Under the **Selenium IDE** section, click on the link that shows the current version number.



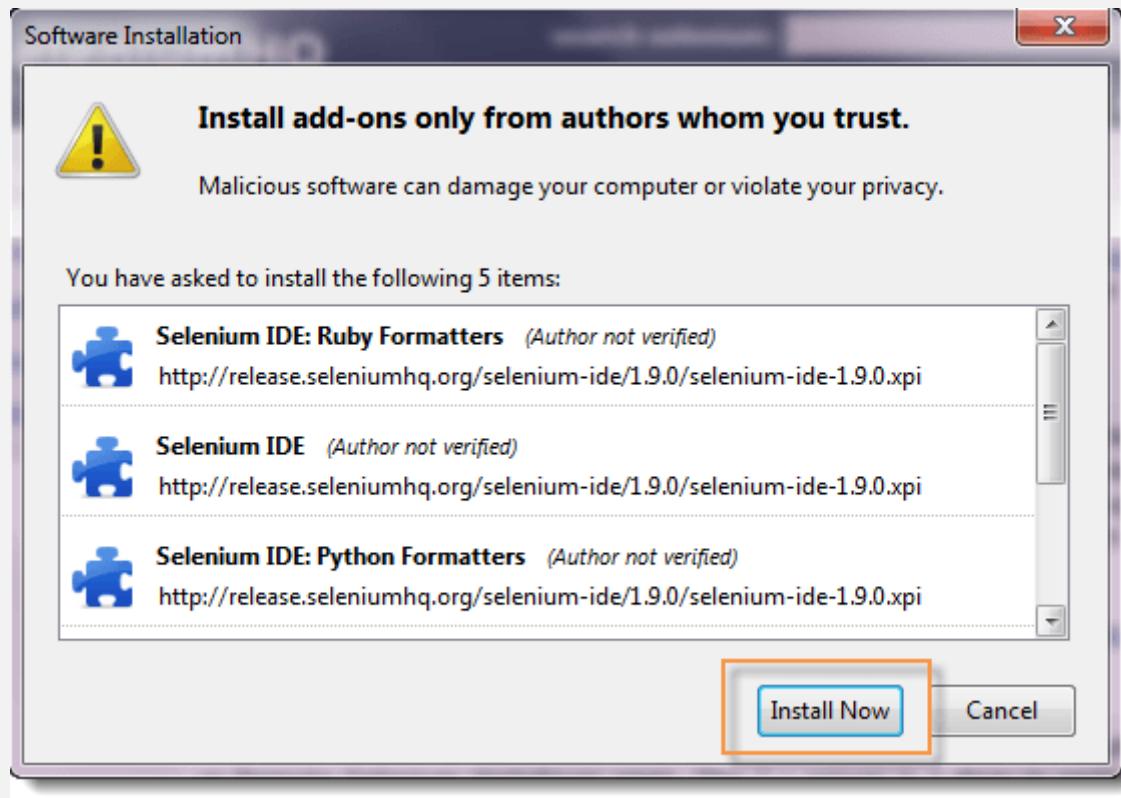
## Installing Selenium IDE & Firebug

For [security](#), a Firefox notification will pop up. Click on "Allow."



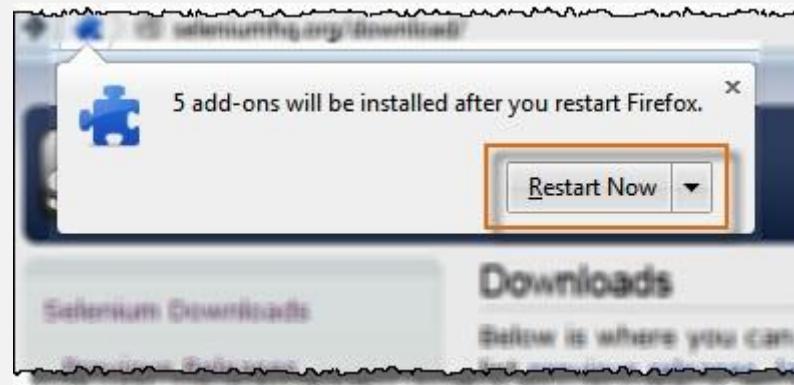
Wait until Firefox completes the download and then click "**Install Now.**"

# Installing Selenium IDE & Firebug



Wait until the installation is completed. In the pop-up window, click "**Restart Now.**"

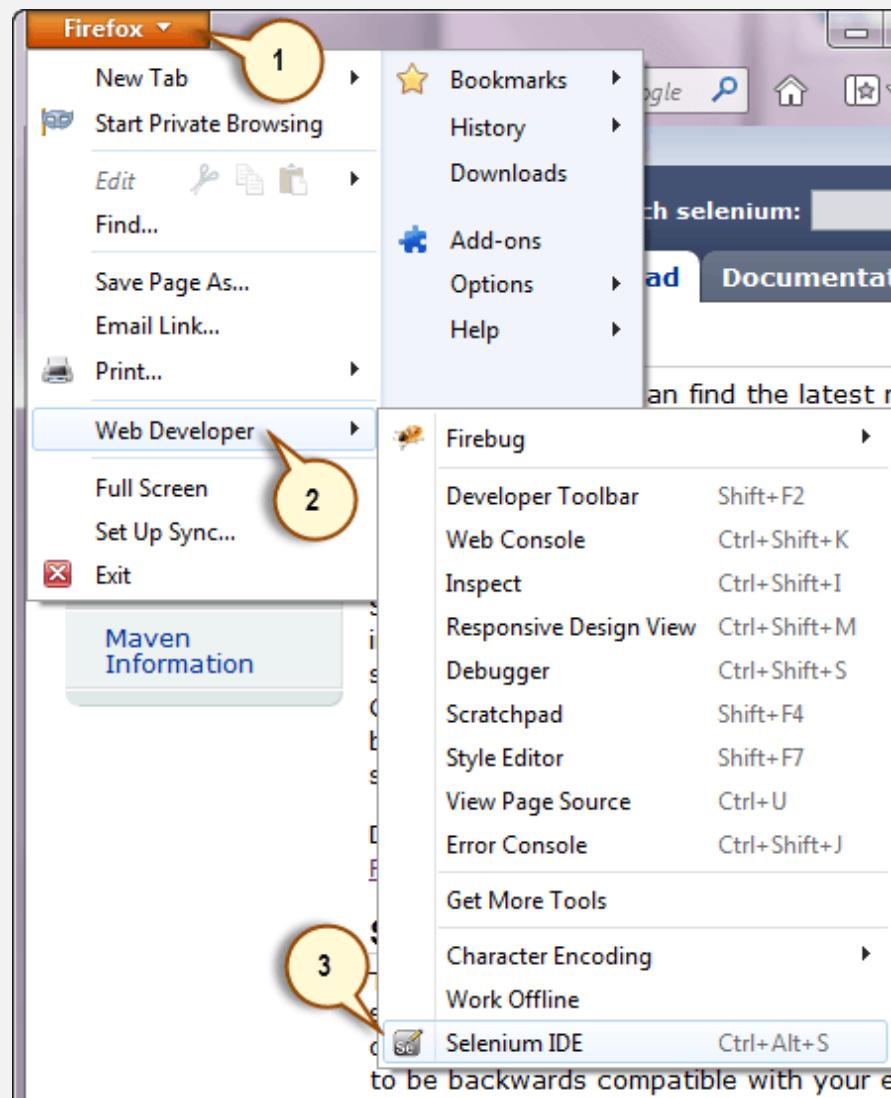
## Installing Selenium IDE & Firebug



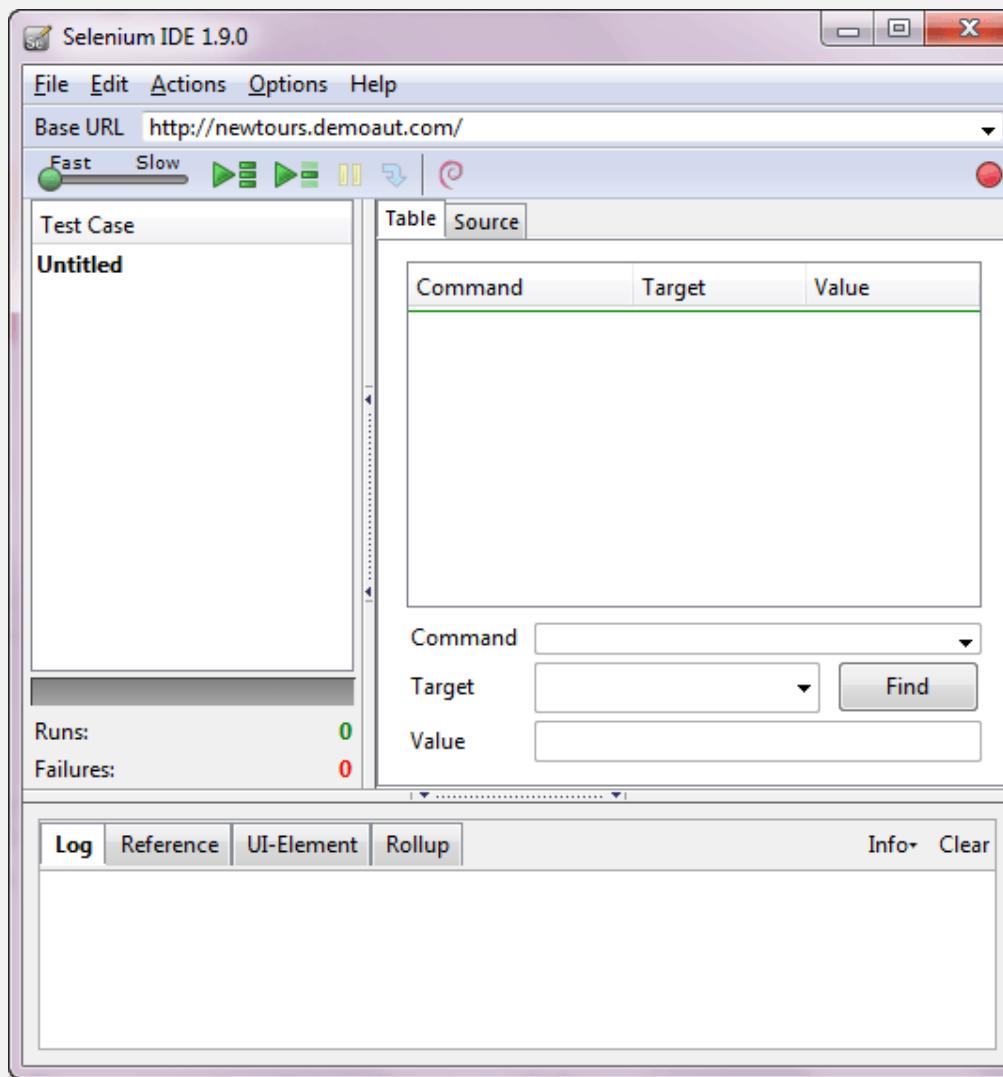
After Firefox has restarted, **launch Selenium IDE** using either of two ways:

- By pressing **Ctrl+Alt+S**
- By clicking on the **Firefox menu button > Web Developer > Selenium IDE**

# Installing Selenium IDE & Firebug



# Installing Selenium IDE & Firebug



Selenium IDE should launch as shown below

# Installation of Firebug

Firebug is a Firefox add-on that we will use to **inspect the HTML elements** of the web application under test. It will provide us the name of the element that our Selenese command would act upon.

## Step 1

Use Firefox to navigate to Firebug's download page (<https://getfirebug.com/downloads/>) and click on the download link.



## Step 2

Firefox will take you to its Firebug download section. Click the "Add to Firefox" button.

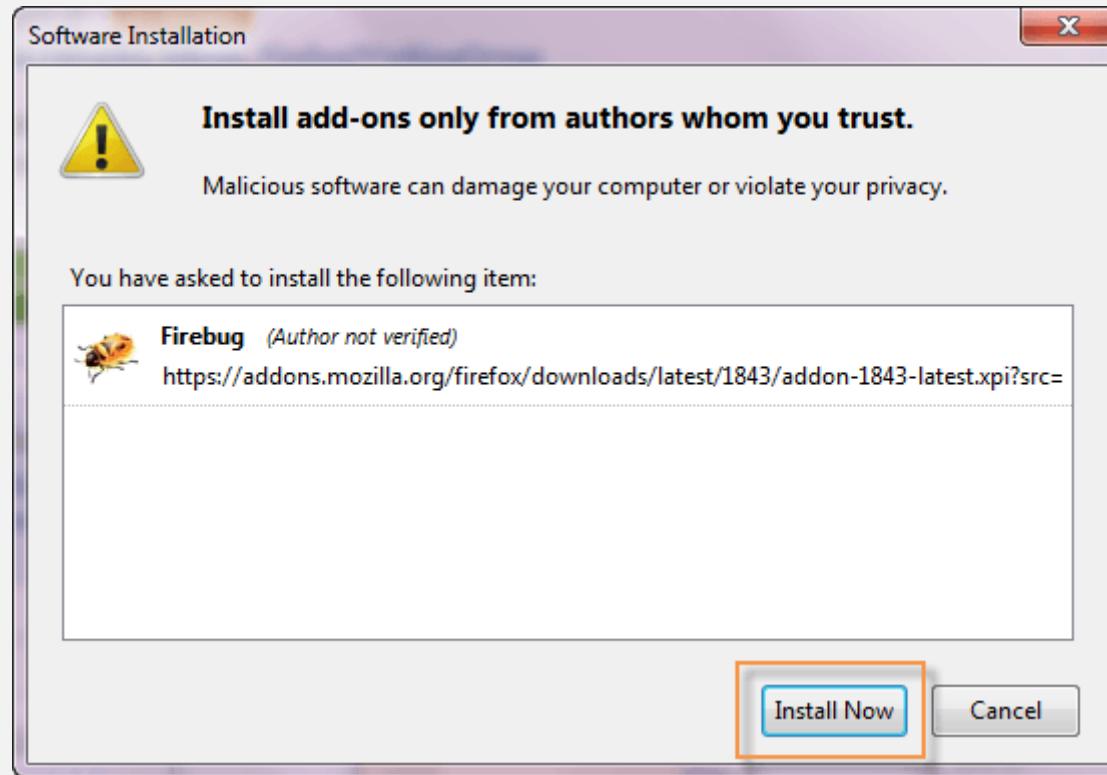
## Installation of Firebug



### Step 3

Wait for Firefox to complete downloading this add-on. On the dialog box that comes after, click "**Install Now.**"

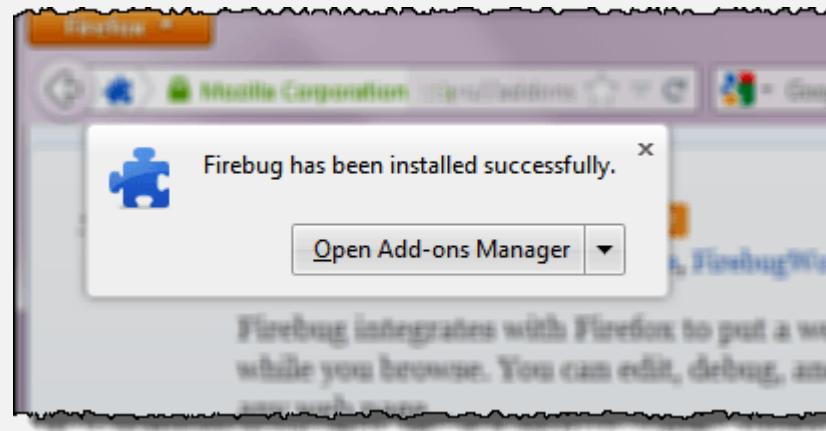
## Installation of Firebug



### Step 4

Wait for the installation to complete. A notification will pop-up saying, "Firebug has been installed successfully." You can immediately close this pop-up.

# Installation of Firebug



**Note:** In case if you do not see above pop-up, no worries! This pop-up appears for a few seconds and disappears. You do not need to restart Firefox after installing Firebug.

## Step 5

Launch Firebug by doing either of these two methods:

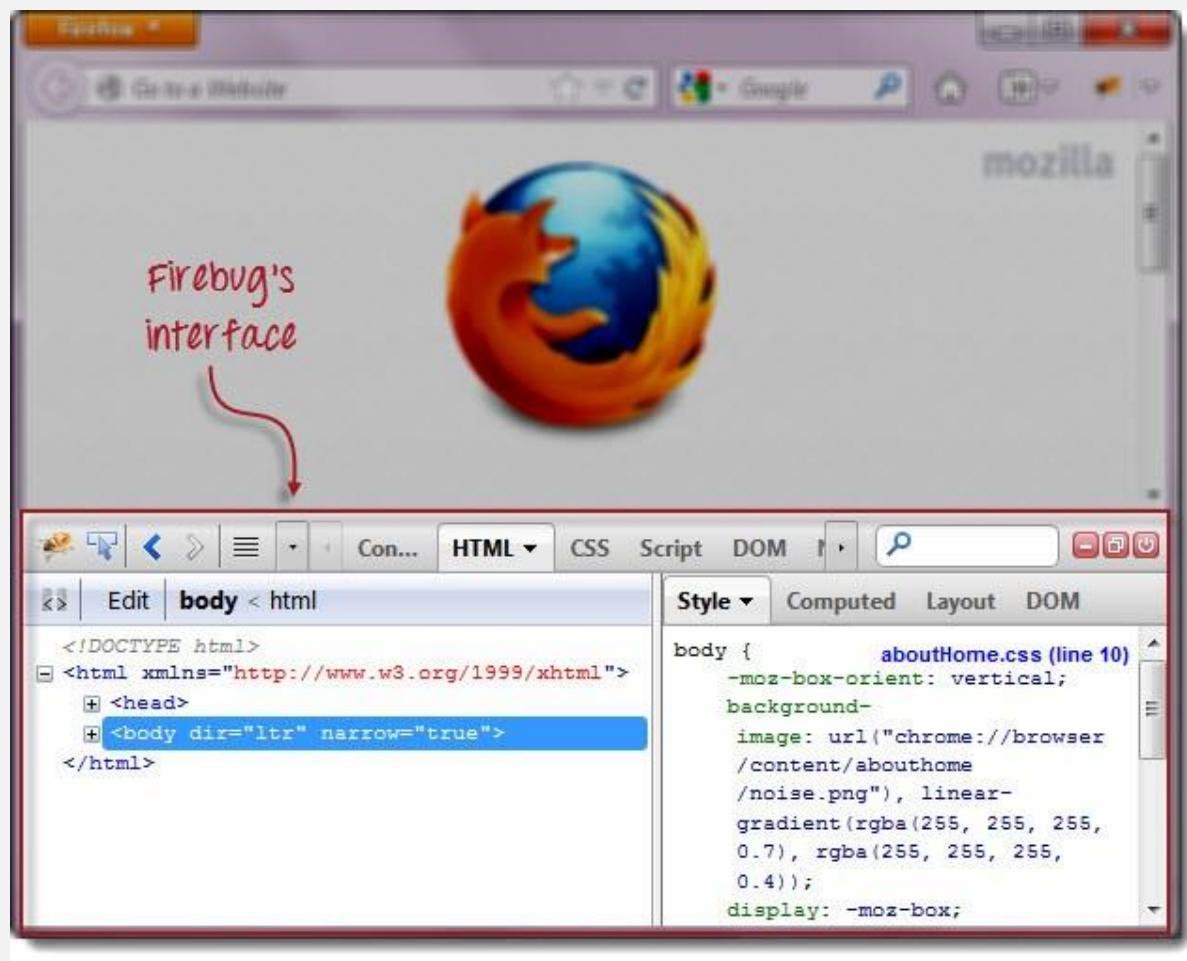
- Press **F12**
- Click on the **Firebug button** on the upper right corner of the Firefox window.



# Installation of Firebug

## Step 6

Firebug should launch at the bottom of Firefox as shown below



# Plugins

Selenium IDE can support additional Firefox add-ons or plugins created by other users. You can visit [here](#) for a list of Selenium add-ons available to date. Install them just as you do with other Firefox add-ons.

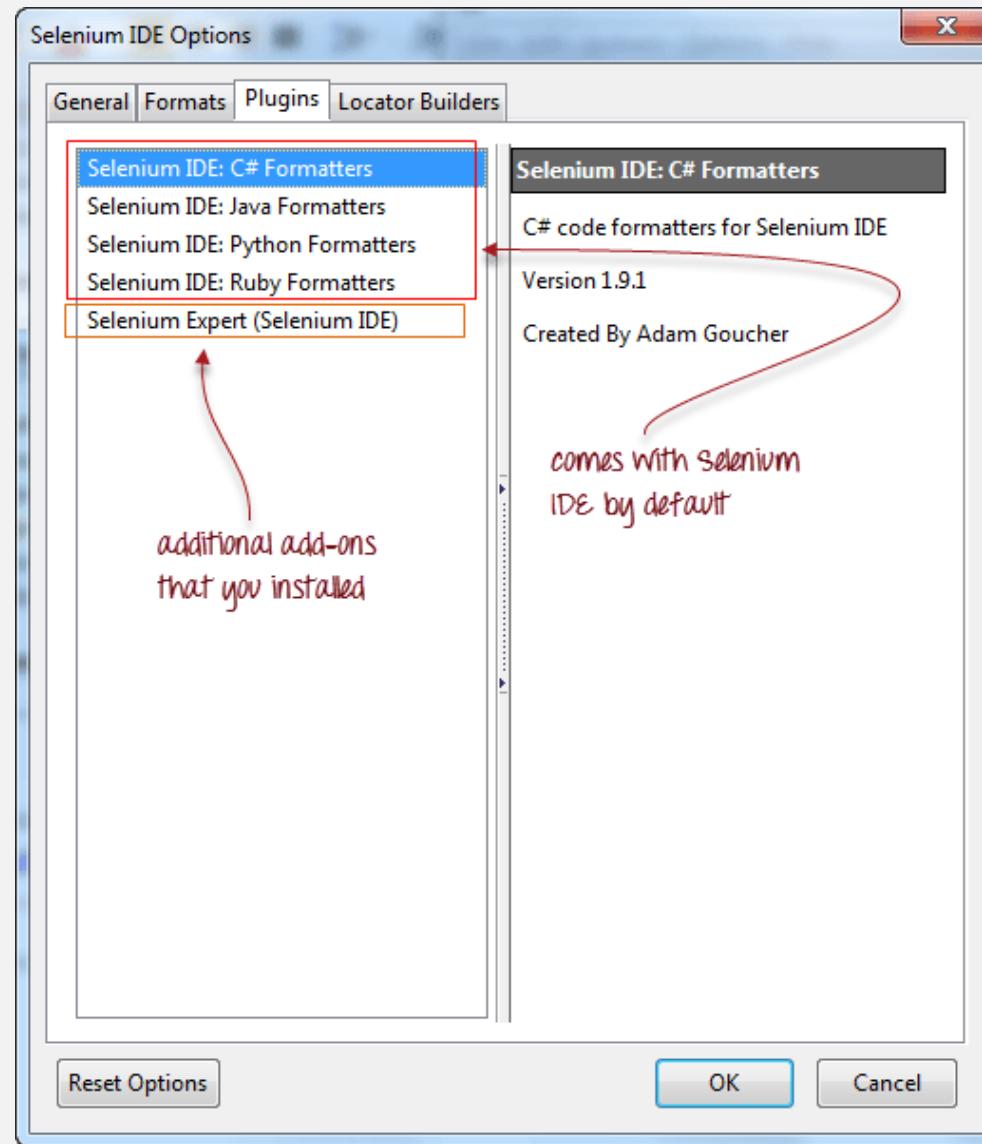
By default, Selenium IDE comes bundled with 4 plugins:

1. Selenium IDE: C# Formatters
2. Selenium IDE: [Java](#) Formatters
3. Selenium IDE: Python Formatters
4. Selenium IDE: Ruby Formatters

These four plugins are required by Selenium IDE to convert Selenese into different formats.

The Plugins tab shows a list of all your installed add-ons, together with the version number and name of the creator of each.

# Installation of Firebug



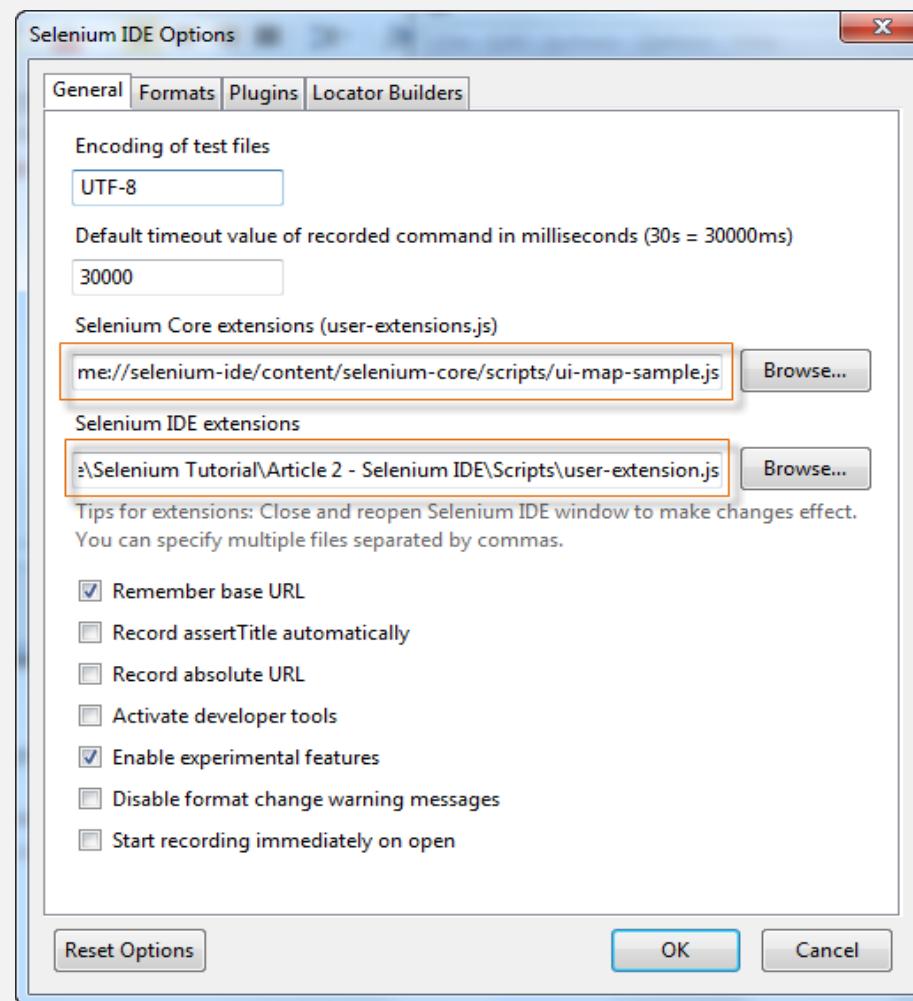
## User Extensions

Selenium IDE can support user extensions to provide advanced capabilities. User extensions are in the form of [JavaScript](#) files. You install them by specifying their absolute path in either of these two fields in the Options dialog box.

- Selenium Core extensions (`user-extensions.js`)
- Selenium IDE extensions

See example:

# User Extensions



# Introduction to Selenium IDE

SELENIUM TRAINING

## Introduction to Selenium IDE

Selenium IDE (Integrated Development Environment) is the simplest tool in the Selenium Suite. It is a Firefox add-on that creates tests very quickly through its record-and-playback functionality. This feature is similar to that of [QTP](#). It is effortless to install and easy to learn.

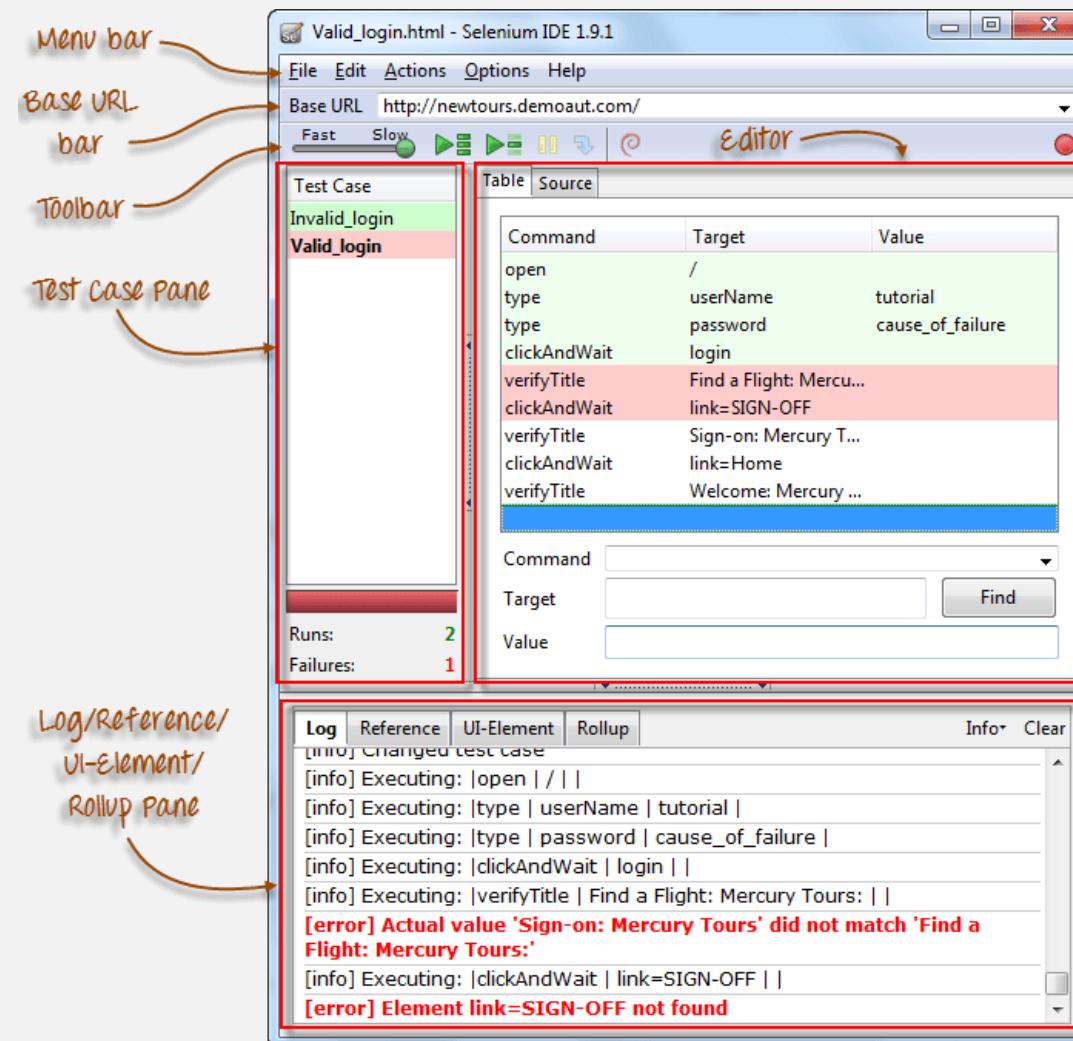
Because of its simplicity, **Selenium IDE should only be used as a prototyping tool - not an overall solution for developing and maintaining complex test suites.**

Though you will be able to use Selenium IDE without prior knowledge in programming, **you should at least be familiar with HTML, JavaScript, and the DOM (Document Object Model)** to utilize this tool to its full potential. Knowledge of JavaScript will be required when we get to the section about the Selenese command "**runScript**".

Selenium IDE supports autocomplete mode when creating tests. This feature serves two purposes:

- It helps the tester to enter commands more quickly.
- It restricts the user from entering invalid commands.

# Features of Selenium IDE



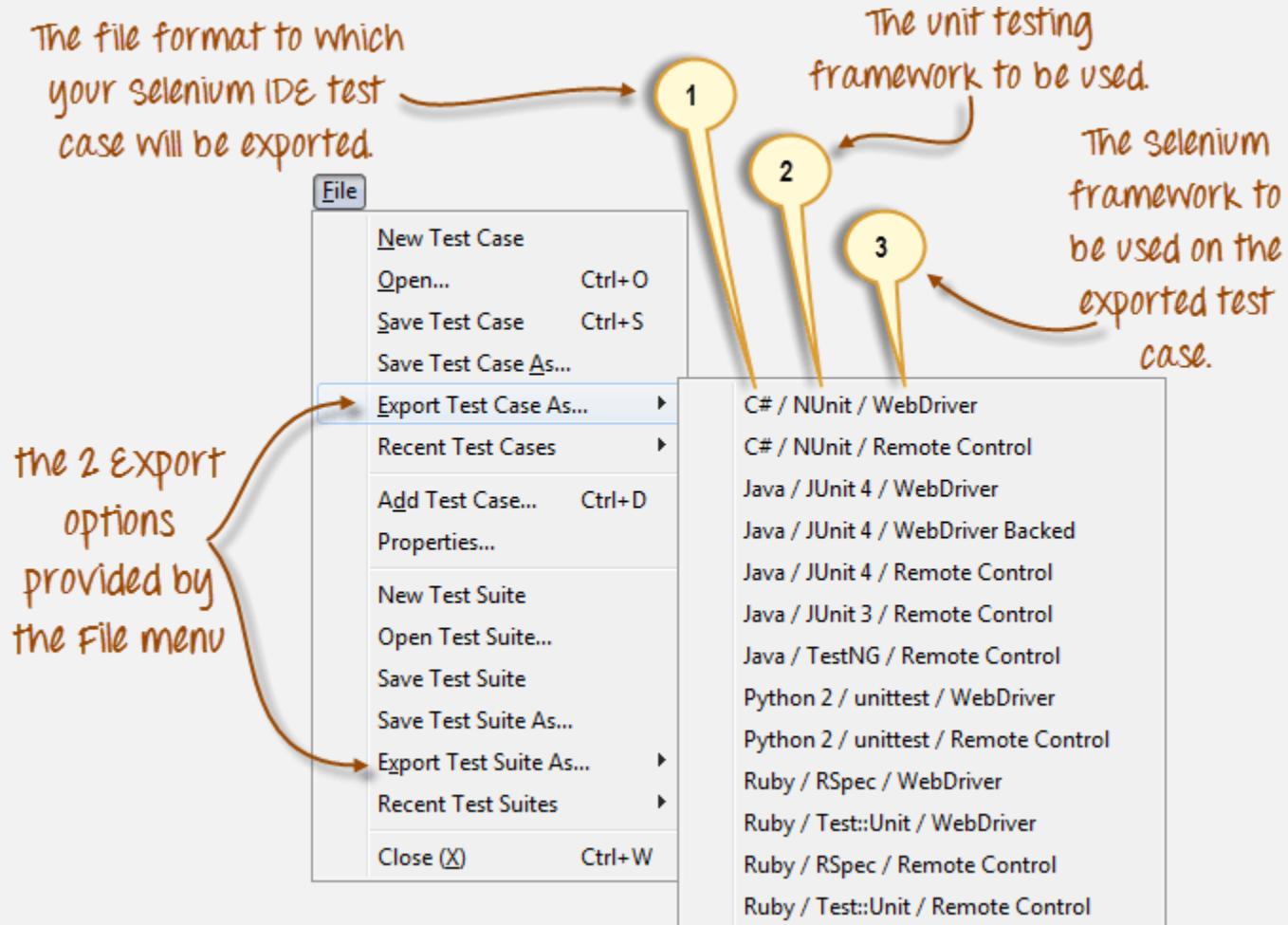
# Menu Bar

It is located at the **top most portion** of the IDE. The most commonly used menus are the File, Edit, and Options menus.

## File menu

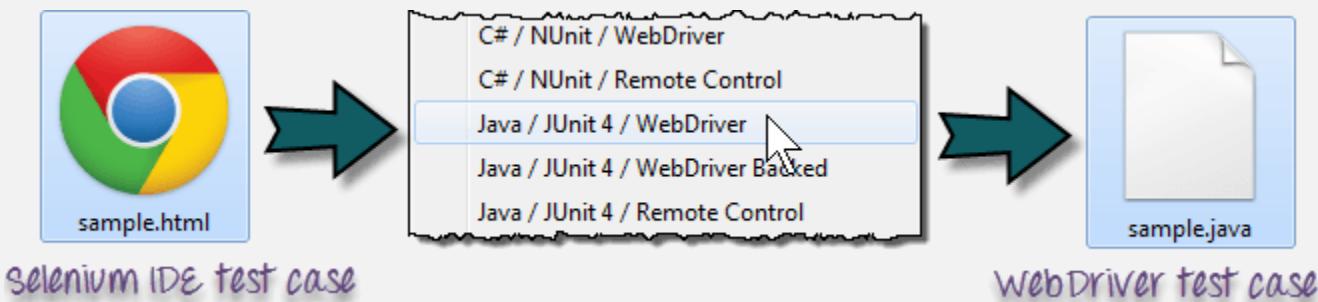
- It contains options to create, open, save, and close tests.
- Tests are **saved in HTML format**.
- The most useful option is "**Export**" because **it allows you to turn your Selenium IDE test cases into file formats that can run on Selenium Remote Control and WebDriver**
  - "**Export Test Case As...**" will export only the currently opened test case.
  - "**Export Test Suite As...**" will export all the test cases in the currently opened testsuite.

## Menu Bar



## Menu Bar

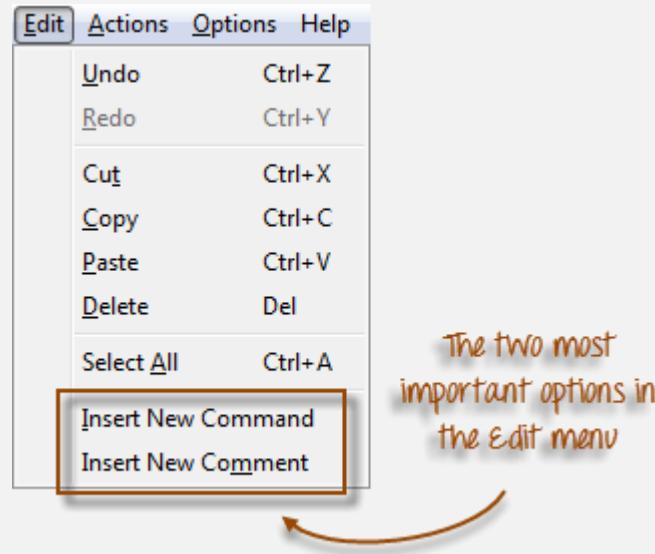
- As of **Selenium IDE v1.9.1**, test cases can be exported only to the following formats:
- .cs (C# source code)
- .java (Java source code)
- .py (Python source code)
- .rb (Ruby source code)



## Edit Menu

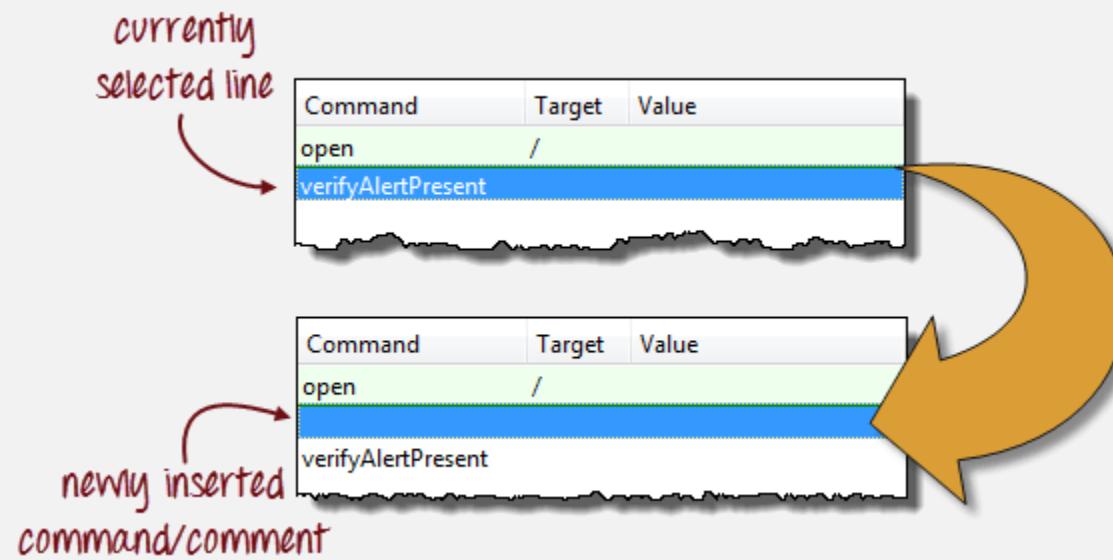
- It contains usual options like Undo, Redo, and Cut, Copy, Paste, Delete, and Select All.
- The two most important options are the "**Insert New Command**" and "**Insert New Comment**".

## Menu Bar



- The newly inserted command or comment **will be placed on top of the currently selected line.**

## Menu Bar



- Commands are colored black.
- Comments are colored purple.

## Menu Bar

Command	Target	Value
open	/	
This is a comment		
verifyAlertPresent		

commands are colored black

comments are colored purple

## Options menu

It provides the **interface for configuring various settings** of Selenium

IDE. We shall concentrate on the **Options** and **Clipboard Format** options.

### Clipboard Format

The **Clipboard Format** allows you to copy a Selenese command from the editor and paste it as a code snippet. The format of the code follows the option you selected here in Clipboard Format's list.

**HTML is the default selection.**

For example, when you choose **Java/JUnit 4/WebDriver** as your clipboard format, every Selenese command you copy from Selenium IDE's editor will be pasted as **Java code**. See the illustration below.

## Options menu

A screenshot of a software interface showing a context menu over a table row. The table has columns for Command, Target, and Value. The row selected in the menu contains the values 'type', 'name=username', and 'tutorial'. A green arrow points from this menu to a code editor window below, which displays four lines of Java code:

Command	Target	Value
type	name=username	tutorial

```
138
139     driver.findElement(By.name("username")).clear();
140     driver.findElement(By.name("username")).sendKeys("tutorial");
141
```

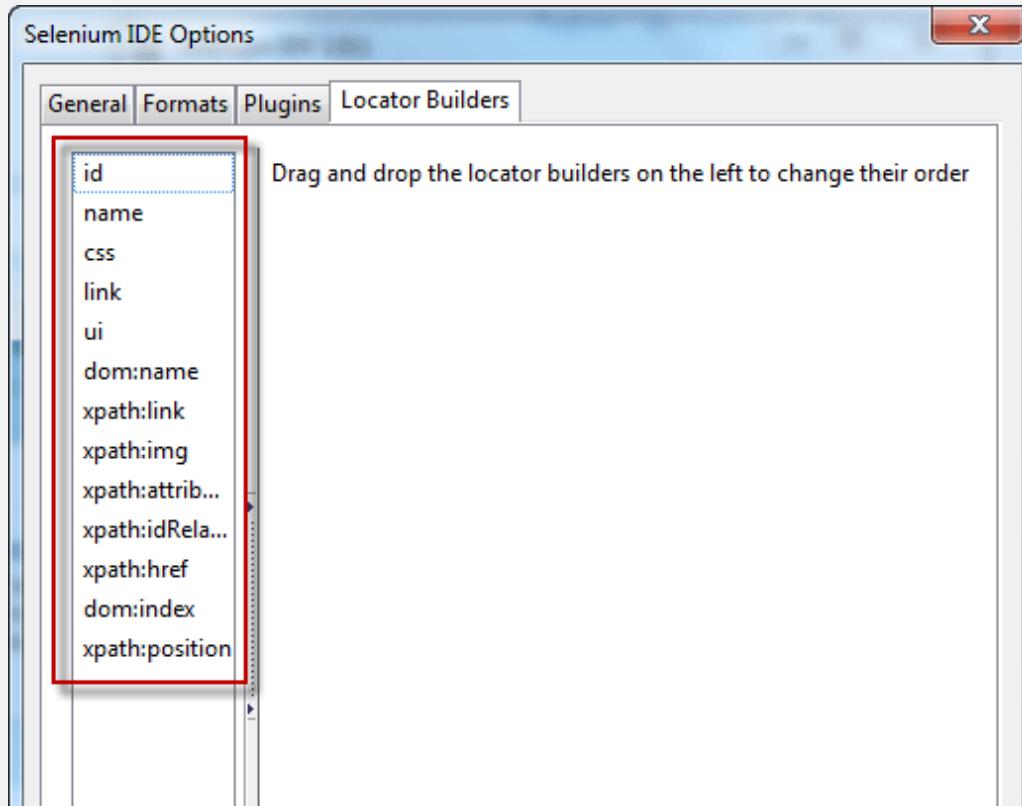
## Selenium IDE Options dialog box

You can launch the Selenium IDE Options dialog box by clicking Options > Options... on the menu bar. Though there are many settings available, we will concentrate on the few important ones.

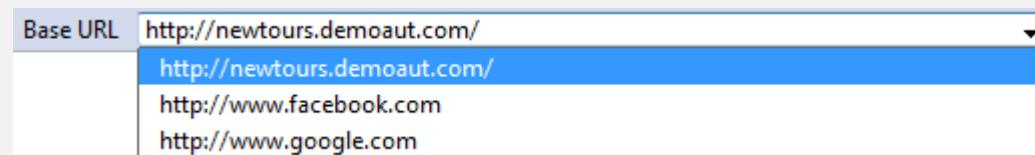
# Selenium IDE Options dialog box

- **Default Timeout Value.** This refers to the time that Selenium has to wait for a certain element to appear or become accessible before it generates an error. **Default timeout value is 3000ms.**
- **Selenium IDE extensions.** This is where you specify the extensions you want to use to extend Selenium IDE's capabilities. You can visit <http://addons.mozilla.org/en-US/firefox/> and use "Selenium" as a keyword to search for the specific extensions.
- **Remember base URL.** Keep this checked if you want Selenium IDE to remember the Base URL every time you launch it. If you uncheck this, Selenium IDE will always launch with a blank value for the Base URL.
- **Auto start record.** If you check this, Selenium IDE will immediately record your browser actions upon startup.
- **Locator builders.** This is where you specify the order by which locators are generated while recording. **Locators are ways to tell Selenium IDE which UI element a Selenese command should act upon.** In the setup below, when you click on an element with an ID attribute, that element's ID will be used as the locator since "id" is the first one in the list. If that element does not have an ID attribute, Selenium will next look for the "name" attribute since it is second in the list. The list goes on and on until an appropriate one is found.

## Selenium IDE Options dialog box



## Base URL Bar



- It has a **dropdown menu that remembers all previous values** for easy access.
- The Selenese command "**open**" will take you to the URL that you specified in the **Base URL**.
- In this tutorial series, we will be using <http://newtours.demoaut.com> as our Base URL. It is the site for Mercury Tours, a web application maintained by HP for web **testing** purposes. We shall be using this application because it contains a complete set of elements that we need for the succeeding topics.
- **The Base URL is very useful in accessing relative URLs.** Suppose that your Base URL is set to <http://newtours.demoaut.com>. When you execute the command "open" with the target value "signup", Selenium IDE will direct the browser to the sign-up page. See the illustration below.

## Base URL Bar

The screenshot shows the Selenium IDE interface with a base URL of `http://newtours.demoaut.com/`. It illustrates two scenarios for the `open` command:

**"open" used without a target**: In the command table, the `open` command is listed with no target specified. A red arrow points from this entry to a Firefox browser window where the URL bar shows the base URL `http://newtours.demoaut.com/`.

Command	Target	Value
open		

**"open" used with a target**: In the command table, the `open` command is listed with a target set to `signup`. A red arrow points from this entry to a Firefox browser window where the URL bar shows the full URL `http://newtours.demoaut.com/signup`.

Command	Target	Value
open	signup	

# Toolbar

**Playback Speed.** This controls the speed of your Test Script Execution.

**Record.** This starts/ends your recording session. Each browser action is entered as a Selenese command in the Editor.

**Play entire test suite.** This will sequentially play all the test cases listed in the Test Case Pane.

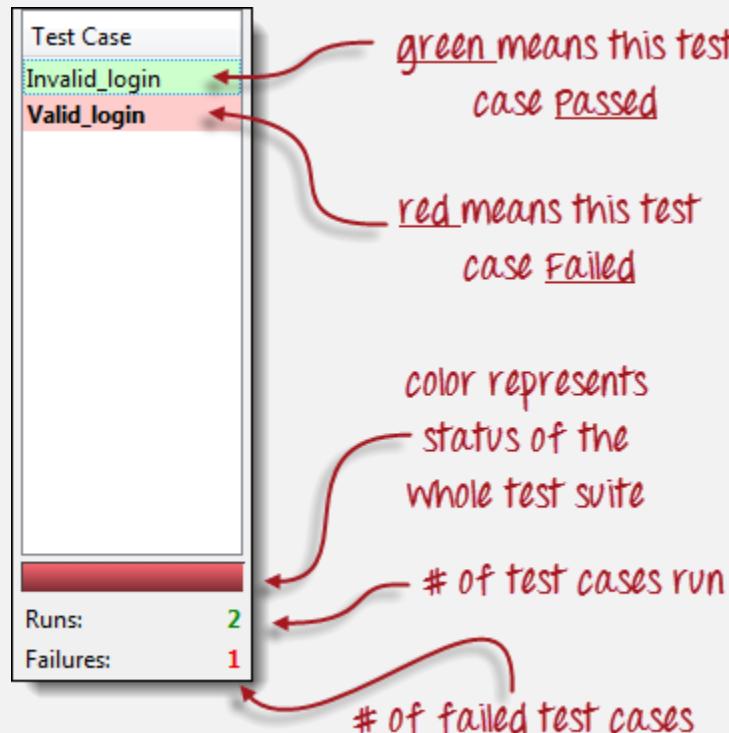
**Play current test case.** This will play only the currently selected test case in the Test Case Pane.

**Pause/Resume.** This will pause or resume your playback.

**Step.** This button will allow you to step into each command in your test script.

**Apply rollup rules.** This is an advanced functionality. It allows you to group Selenese commands together and execute them as a single action.

## Test Case Pane



In Selenium IDE, you can open **more than one test case at a time**.

The test case pane shows you the list of currently opened test cases.

When you open a test suite, the test case pane will **automatically list all the test cases** contained in it. The test case written in **bold font** is the **currently selected test case**

After playback, **each test case is color-coded** to represent if it passed or failed.

- Green color means "Passed."
- Red color means "Failed."

At the bottom portion is a summary of the number of test cases that were run and failed.

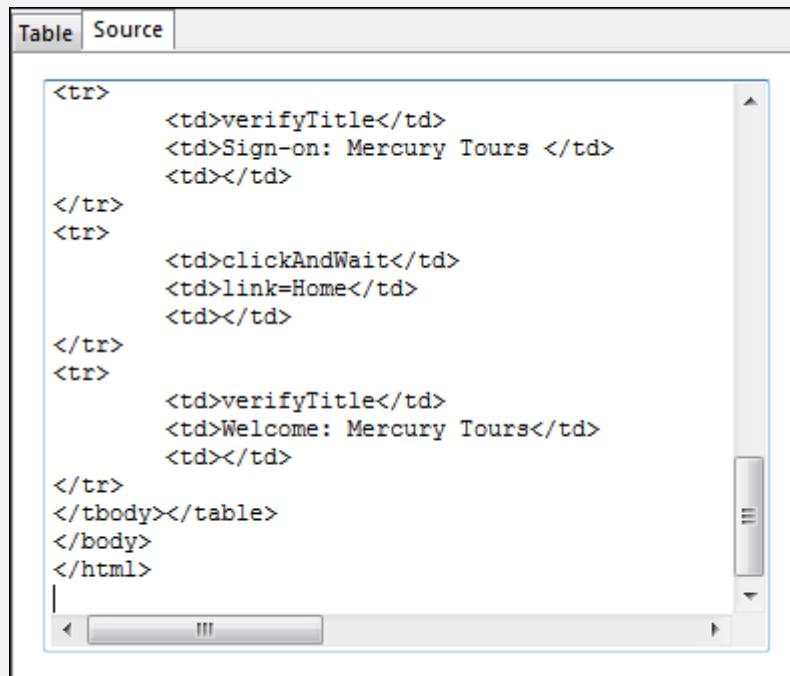
# Editor

You can think of the editor as **the place where all the action happens**. It is available in two views: Table and Source.

## Table View

- Most of the time, you will work on Selenium IDE using the **Table View**.
  - This is **where you create and modify Selenese commands**.
  - After playback, each step is color-coded.
- 
- To create steps, type the name of the command in the "Command" text box.
  - **It displays a dropdown list of commands** that match with the entry that you are currently typing.
  - Target is any parameter (like username,

## Editor



The screenshot shows the TestComplete Editor interface with the 'Source' tab selected. The main area displays an HTML script with several `<tr>` and `<td>` tags. The script includes steps like 'verifyTitle', 'Sign-on: Mercury Tours', 'clickAndWait', 'link=Home', and 'Welcome: Mercury Tours'. The code is presented in a standard text editor format with syntax highlighting.

```
<tr>
    <td>verifyTitle</td>
    <td>Sign-on: Mercury Tours </td>
    <td></td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>link=Home</td>
    <td></td>
</tr>
<tr>
    <td>verifyTitle</td>
    <td>Welcome: Mercury Tours</td>
    <td></td>
</tr>
</tbody></table>
</body>
</html>
```

Source View

- It displays the steps in HTML (default) format.
- It also allows you to edit your script just like in the Table View.

# Log Pane

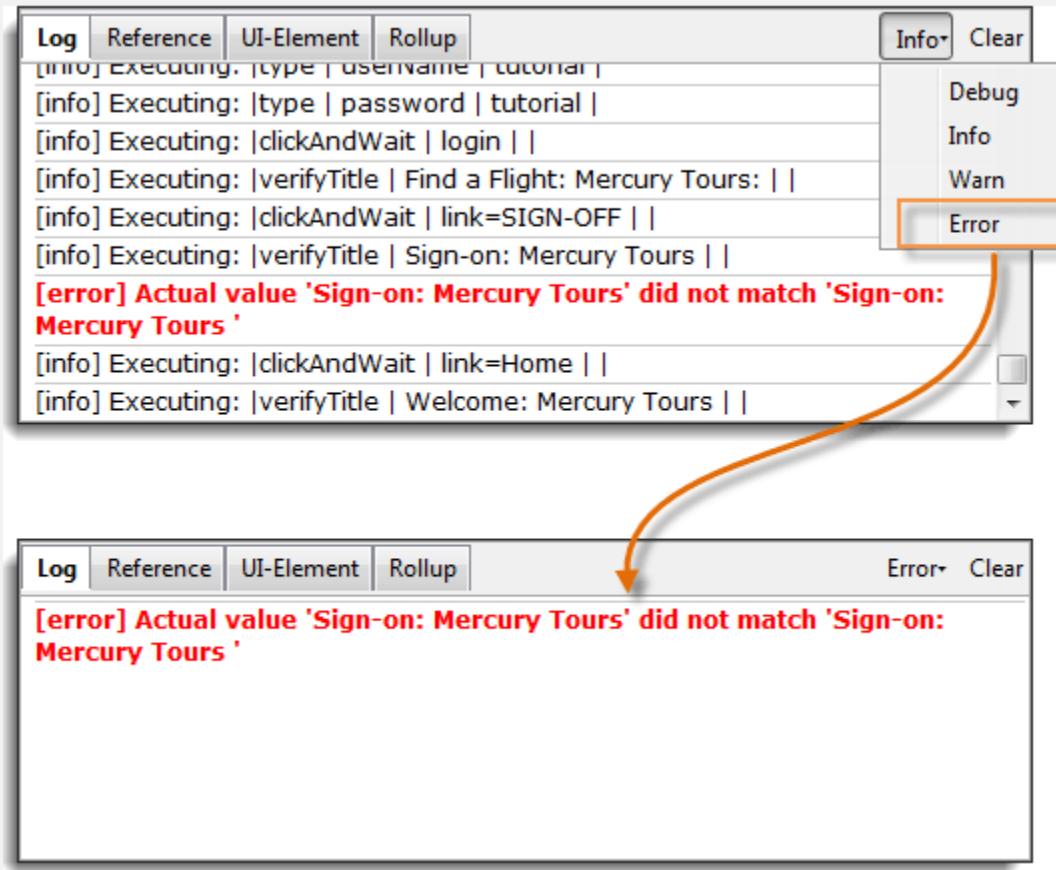
The Log Pane displays **runtime messages** during execution. It provides real-time updates as to what Selenium IDE is doing.

Logs are categorized into four types:

- Debug - By default, Debug messages are not displayed in the log panel. They show up only when you filter them. They provide technical information about what Selenium IDE is doing behind the scenes. It may display messages such as a specific module has done loading, a certain function is called, or an external JavaScript file was loaded as an extension.
- Info - It says which command Selenium IDE is currently executing.
- Warn - These are warning messages that are encountered in special situations.
- Error - These are error messages generated when Selenium IDE fails to execute a command, or if a condition specified by "verify" or "assert" command is not met.

Logs can be filtered by type. For example, if you choose to select the "Error" option from the dropdown list, the Log Pane will show error messages only.

## Log Pane



## Reference Pane

The Reference Pane shows a concise description of the currently selected Selenese command in the Editor. It also shows the description about the locator and value to be used on that command.

A screenshot of the Reference Pane in a software interface. At the top, there is a table titled "Command" with three columns: "Command", "Target", and "Value". The table contains the following rows:

Command	Target	Value
open	/	
type	userName	tutorial
type	password	tutorial
clickAndWait	login	

The row for "type" is highlighted with a blue selection bar, and a mouse cursor is positioned over the "Value" column of this row. A large orange arrow points from the right side of the table towards the detailed description below. Below the table, there is a navigation bar with tabs: Log, Reference, UI-Element, and Rollup. The "Reference" tab is currently selected. The main content area displays the following information for the "type" command:

**type(locator, value)**

Arguments:

- locator - an element locator
- value - the value to type

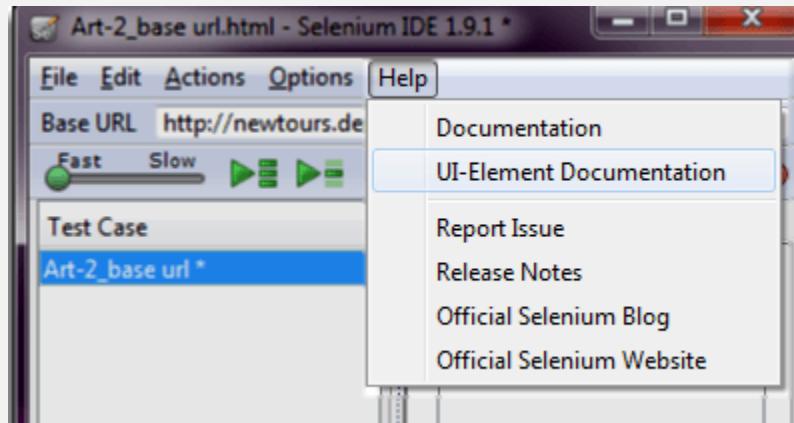
Sets the value of an input field, as though you typed it in.

Can also be used to set the value of combo boxes, check boxes, etc. In these cases, value should be the value of the option selected, not the visible text.

## UI-Element Pane

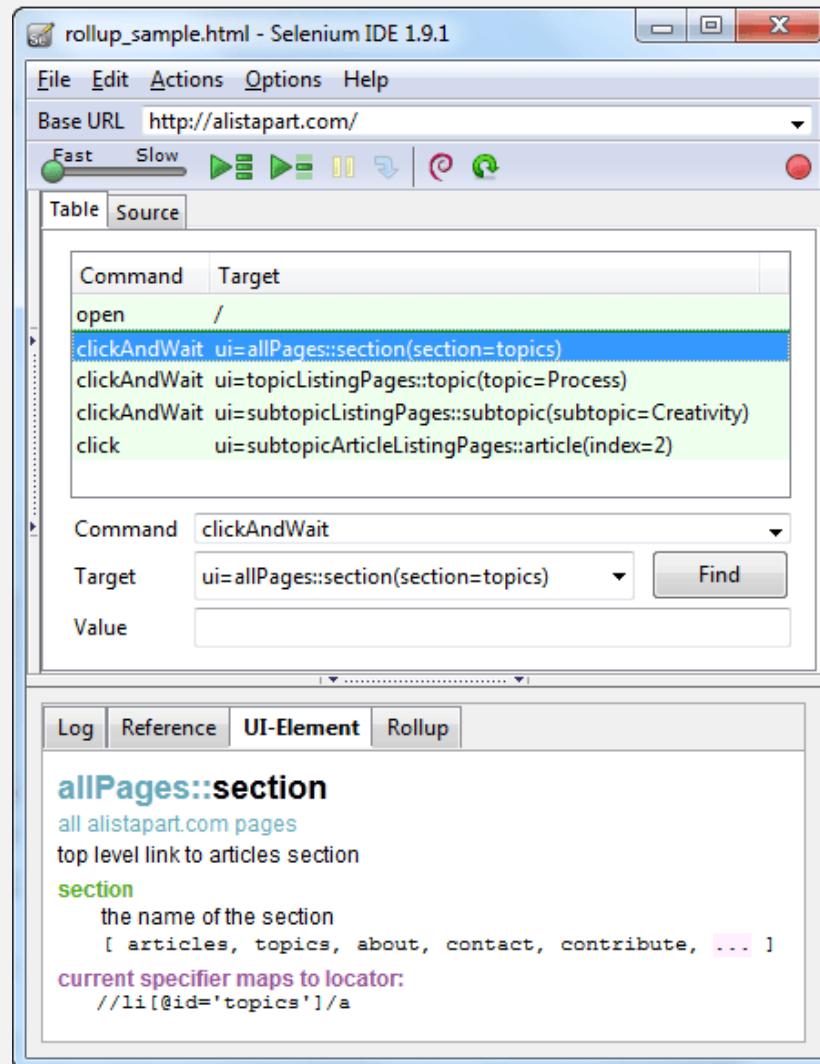
The UI-Element is for advanced Selenium users. It uses JavaScript Object Notation (JSON) to define element mappings.

The documentation and resources are found in the "UI Element Documentation" option under the Help menu of Selenium IDE.



An example of a UI-element screen is shown below.

## UI-Element Pane



# Rollup Pane

**Rollup** allows you to execute a group of commands in one step. A group of commands is simply called as a "rollup." It employs heavy use of JavaScript and UI-Element concepts to formulate a collection of commands that is similar to a "function" in programming languages.

**Rollups are reusable;** meaning, they can be used multiple times within the test case. Since rollups are groups of commands condensed into one, they contribute a lot in shortening your test script.



Command	Target	Value
open		
clickAndWait	ui=allPages::section(section=topics)	
clickAndWait	ui=topicListingPages::topic(topic=Process)	
clickAndWait	ui=subtopicListingPages::subtopic(subtopic=Creativity)/	
click	ui=subtopicArticleListingPages::article(index=2)	

Command	Target	Value
open	/	
rollup	navigate_to_subtopic_article	index=2, subtopic=Creativity

An example of how the contents of the rollup tab look like is shown below.

## Rollup Pane

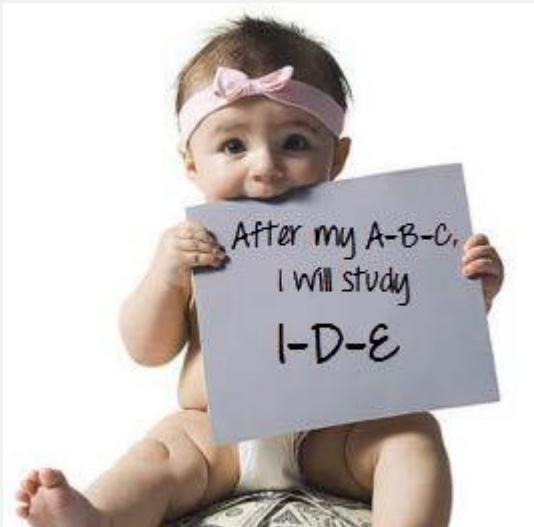
The screenshot shows the Selenium IDE 1.9.1 interface with the following details:

- Title Bar:** rollup\_sample.html - Selenium IDE 1.9.1
- Menu Bar:** File Edit Actions Options Help
- Toolbar:** Base URL http://alistapart.com/ (with dropdown arrow), Fast/Slow slider, and various icons for play, stop, pause, and refresh.
- Table View:** Shows a table with columns Command, Target, and Value. The last row is selected and highlighted in blue:

Command	Target	Value
open	/	
rollup	navigate_to_subtopic_article	index=2, subtopic=Creativity
- Form View:** Displays the command, target, and value fields corresponding to the selected row in the table:

Command	rollup
Target	navigate_to_subtopic_article
Value	index=2, subtopic=Creativity
- Log/Reference/UI-Element/Rollup Tab:** The "Rollup" tab is selected, showing detailed information about the "navigate\_to\_subtopic\_article" command:
  - name:** navigate\_to\_subtopic\_article
  - description:** navigate to an article listed under a subtopic.
  - preconditions:** current page contains the section menu (most pages should)
  - postconditions:** navigated to an article page
  - subtopic:** the subtopic whose article listing to navigate to
    - [ Browsers, CSS, Flash, HTML and XHTML, Scripting, ... ]

# Summary



Selenium IDE (Integrated Development Environment) is **the simplest tool** in the Selenium Suite. It must only be used as a **prototyping tool**.

**Knowledge of JavaScript and HTML is required for intermediate topics** such as executing the "runScript" and "rollup" commands. A **rollup** is a collection of commands that you can reuse to shorten your test scripts significantly. **Locators** are identifiers that tell Selenium IDE how to access an element.

**Firebug** (or any similar add-on) is used to obtain locator values.

The **menu bar** is used in creating, modifying, and exporting test cases into formats useable by Selenium RC and WebDriver.

The **default format for Selenese commands is HTML**.

The **"Options" menu provides access to various configurations** for Selenium IDE.

The **Base URL** is useful in accessing **relative URLs**.

- The **Test Case Pane** shows the list of currently opened test cases and a concise summary of testruns.

- The **Editor** provides the **interface for your test scripts**.
- The **Table View** shows your script **in tabular format** with "Command", "Target", and "Value" as the columns.
- The **Source View** shows your script **in HTML format**.
- The **Log** and **Reference** tabs give feedback and other useful information when executing tests.

## Summary

- The **UI-Element** and **Rollup** tabs are **for advanced Selenium IDE users only**. They both require considerable effort in coding JavaScript.
- **UI-Element** allows you to **conveniently map UI elements** using JavaScript Object Notation(JSON).

# Creating your First Selenium IDE script

SELENIUM TRAINING

## Creating your First Selenium IDE script

We will use the Mercury Tours website as our web application under test. It is an online flight reservation system that contains all the elements we need for this tutorial. Its URL is <http://newtours.demoaut.com/> and this will be our Base URL.

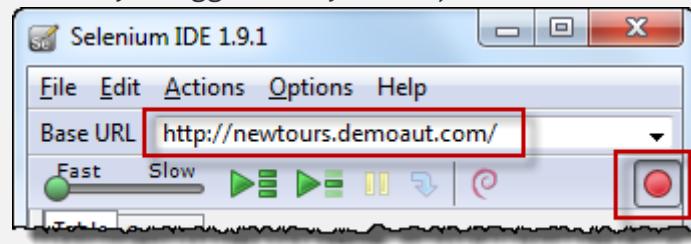
**NOTE:** The site <http://newtours.demoaut.com/> at time is down and not available to work on. We have raised this issue with HP.

# Create a Script by Recording

Let us now create our first test script in Selenium IDE using the most common method - by recording. Afterward, we shall execute our script using the playback feature.

## Step 1

- Launch Firefox and Selenium IDE.
- Type the value for our Base URL: <http://newtours.demoaut.com/>.
- Toggle the Record button on (if it is not yet toggled on by default).



## Step 2

In Firefox, navigate to <http://newtours.demoaut.com/>. Firefox should take you to the page similar to the one shown below.

# Create a Script by Recording

The screenshot shows the homepage of the Mercury Tours website. The header features a logo of a stylized planet with a ring, followed by the text "MERCURY TOURS". Below the header is a banner with the text "one cool summer" and the word "ARUBA" in a large font, accompanied by a photo of people on a beach. A navigation bar at the top right includes links for "SIGN-ON", "REGISTER", "SUPPORT", and "CONTACT". The date "Dec 4, 2012" is displayed in the top right corner.

**Featured Destination:** ARUBA

This island is surrounded by coral reefs, offers guaranteed sunshine and is blessed with beautiful beaches. Luxury resorts have taken up residence along most of the beachfronts on the southern coast, but there are still undeveloped areas on the exposed northern coast, and much of the interior is inhabited by nothing more substantial than goats.

**Specials:**

Atlanta to Las Vegas	\$398
Boston to San Francisco	\$513
Los Angeles to Chicago	\$168
New York to Chicago	\$198
Phoenix to San Francisco	\$213

**TIP #93** Always carry a travel first aid kit with bandages, antacids, aspirin, bee sting wipes, and other basic necessities.

**Find A Flight**

Registered users can sign-in here to find the lowest fare on participating airlines.

User Name:   
Password:

**Destinations**

Find detailed information about [your destination](#).

**Vacations**

Read about our [featured vacation destinations](#).

**Register**

[Register here](#) to join Mercury Tours!

**Links**

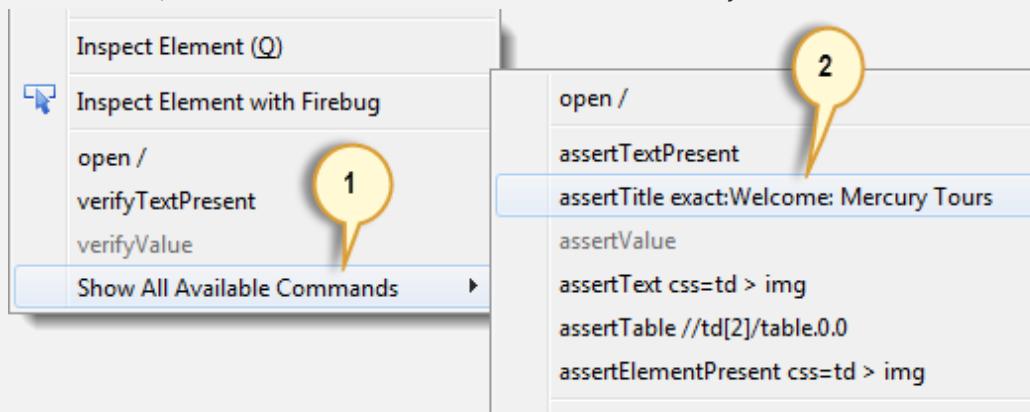
[Business Travel @ About.com](#)  
[Salon Travel](#)

© 2005, Mercury Interactive (v. 011003-1.01-058)

# Create a Script by Recording

## Step 3

- Right-click on any blank space within the page, like on the Mercury Tours logo on the upper left corner. This will bring up the Selenium IDE context menu. Note: Do not click on any hyperlinked objects or images
- Select the "Show Available Commands" option.
- Then, select "assertTitle exact: Welcome: Mercury Tours". This is a command that makes sure that the page title is correct.



## Create a Script by Recording

### Step 4

- In the "User Name" text box of Mercury Tours, type an invalid username, "invalidUN".
- In the "Password" text box, type an invalid password, "invalidPW".

# Create a Script by Recording

## Step 5

- Click on the "Sign-In" button. Firefox should take you to this page.

## Create a Script by Recording

## **Step 6**

Toggle the record button off to stop recording. Your script should now look like the one shown below.

# Create a Script by Recording

## Step 7

Now that we are done with our test script, we shall save it in a test case. In the File menu, select "Save Test Case". Alternatively, you can simply press Ctrl+S.

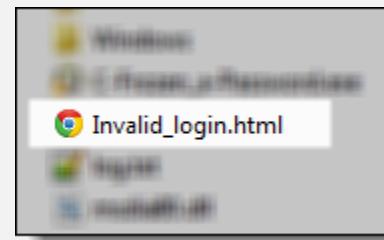
## Step 8

- Choose your desired location, and then name the test case as "Invalid\_login".
- Click the "Save" button.

## Create a Script by Recording

Step 9

Notice that the file was saved as HTML.



**Step 10.**

Go back to Selenium IDE and click the Playback button to execute the whole script. Selenium IDE should be able to replicate everything flawlessly.\*

## Create a Script by Recording

The screenshot shows a test script editor interface. On the left, a sidebar displays a 'Test Case' list with one item: 'Invalid\_login'. A red arrow points from the text 'All green' to the 'Invalid\_login' entry. In the main area, there's a table titled 'Table' with two tabs: 'Table' (selected) and 'Source'. The 'Table' tab shows a command table with four rows:

Command	Target	Value
open		
assertTitle	exact:Welcome: Mercury Tours	
type	name=userName	invalidUN

Below the table, there are two status boxes: 'Runs: 1' and 'Failures: 0'. A red box highlights the 'Runs: 1' box. A red arrow points from the text 'The test Passed!!!' to the 'Failures: 0' box.

## Introduction to Selenium Commands - Selenese

- Selenese commands can have up to a maximum of two parameters: target and value.
- Parameters are not required all the time. It depends on how many the command will need.
- For a complete reference of Selenese commands, click [here](#). →  
<http://release.seleniumhq.org/seleniumcore/1.0.1/reference.html>

# Introduction to Selenium Commands – Selenese

## 3 Types of Commands

Actions	<p>These are commands that directly interact with page elements.</p> <p>Example: the "click" command is an action because you directly interact with the element you are clicking at.</p> <p>The "type" command is also an action because you are putting values into a text box, and the text box shows them to you in return. There is a two-way interaction between you and the text box.</p>
Assessors	<p>They are commands that allow you to store values to a variable.</p> <p>Example: the "storeTitle" command is an accessor because it only "reads" the page title and saves it in a variable. It does not interact with any element on the page.</p>
Assertions	<p>They are commands that verify if a certain condition is met.</p> <h3>3 Types of Assertions</h3> <ul style="list-style-type: none"><li>• <b>Assert.</b> When an "assert" command fails, the test is stopped immediately.</li><li>• <b>Verify.</b> When a "verify" command fails, Selenium IDE logs this failure and continues with the test execution.</li><li>• <b>WaitFor.</b> Before proceeding to the next command, "waitFor" commands will first wait for a certain condition to become true.<ul style="list-style-type: none"><li>○ If the condition becomes true within the waiting period, the step passes.</li><li>○ If the condition does not become true, the step fails. Failure is logged, and test execution proceeds to the next command.</li><li>○ By default, the timeout value is set to 30 seconds. You can change this in the Selenium IDE Options dialog under the General tab.</li></ul></li></ul>

# Introduction to Selenium Commands – Selenese

## Assert vs. Verify

**ASSERT**

The screenshot shows the Selenese ASSERT tool interface. At the top, the word "ASSERT" is written in red. Below it is a table with three columns: Command, Target, and Value. The table contains the following rows:

Command	Target	Value
open		
assertTitle	Welcome: Venus Tours	
type	name=userName	invalidUN
type	name=password	invalidPW
clickAndW...	name=login	

Below the table are three input fields: "Command" (with a dropdown arrow), "Target" (with a text input and a "Find" button), and "Value" (with a text input). At the bottom of the window is a log pane with tabs for Log, Reference, UI-Element, Rollup, Info, and Clear. The Log tab is selected, showing the following log entries:

```
[info] Executing: |open || |
[info] Executing: |assertTitle | Welcome: Venus
Tours ||
[error] Actual value 'Welcome: Mercury Tours'
did not match 'Welcome: Venus Tours'
```

Red annotations are present on the left side of the screenshot:

- A red arrow points from the text "test execution was halted in this part" to the "assertTitle" row in the table.
- A red arrow points from the text "no further logs were displayed after this error message, meaning that execution indeed stopped" to the "[error]" log entry.

## VERIFY

Execution continued  
despite the error

The screenshot shows a test script editor interface with two main sections: a command table and a log window.

**Command Table:**

Command	Target	Value
open		
verifyTitle	Welcome: Venus Tours	
type	name=userName	invalidUN
type	name=password	invalidPW
clickAndW...	name=login	

**Log Window:**

```
[info] Executing: |verifyTitle| welcome: venus
Tours ||  
[error] Actual value 'Welcome: Mercury
Tours' did not match 'Welcome: Venus Tours'  
[info] Executing: |type | name=userName |
invalidUN |  
[info] Executing: |type | name=password |
invalidPW |
```

Commands after  
the Failed verify  
command were still  
executed

# Introduction to Selenium Commands – Selenese

## Common Commands

Command	Number of Parameters	Description
open	0 - 2	Opens a page using a URL.
click/clickAndWait	1	Clicks on a specified element.
type/typeKeys	2	Types a sequence of characters.
verifyTitle/assertTitle	1	Compares the actual page title with an expected value.
verifyTextPresent	1	Checks if a certain text is found within the page.
verifyElementPresent	1	Checks the presence of a certain element.
verifyTable	2	Compares the contents of a table with expected values.
waitForPageToLoad	1	Pauses execution until the page is loaded completely.
waitForElementPresent	1	Pauses execution until the specified element becomes present.

# Create a Script Manually with Firebug

Now, we shall recreate the same test case manually, by typing in the commands. This time, we will need to use Firebug.

## Step 1

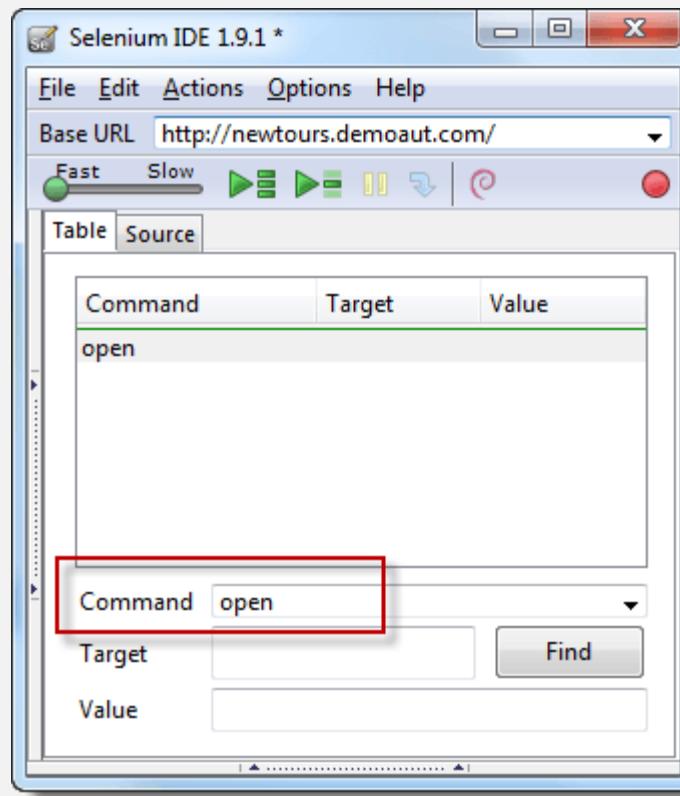
- Open Firefox and Selenium IDE.
- Type the base URL (<http://newtours.demoaut.com/>).
- The record button should be OFF.

## Step 2

Click on the topmost blank line in the Editor.

Type "open" in the Command text box and press Enter.

## Create a Script Manually with Firebug



### Step 3

- Navigate Firefox to our base URL and activate Firebug
- In the Selenium IDE Editor pane, select the second line (the line below the "open" command) and create the second command by typing "assertTitle" on the Command box.

## Create a Script Manually with Firebug

- Feel free to use the autocomplete feature.

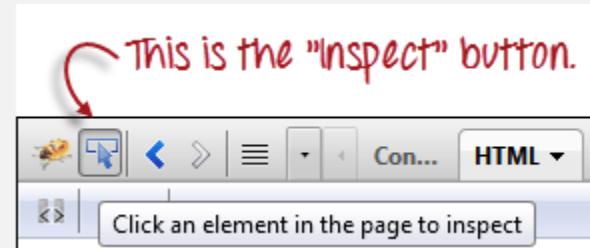
### Step 4

- In Firebug, expand the <head> tag to display the <title> tag.
- Click on the value of the <title> tag (which is "Welcome: Mercury Tours") and paste it onto the Target field in the Editor.

## Create a Script Manually with Firebug

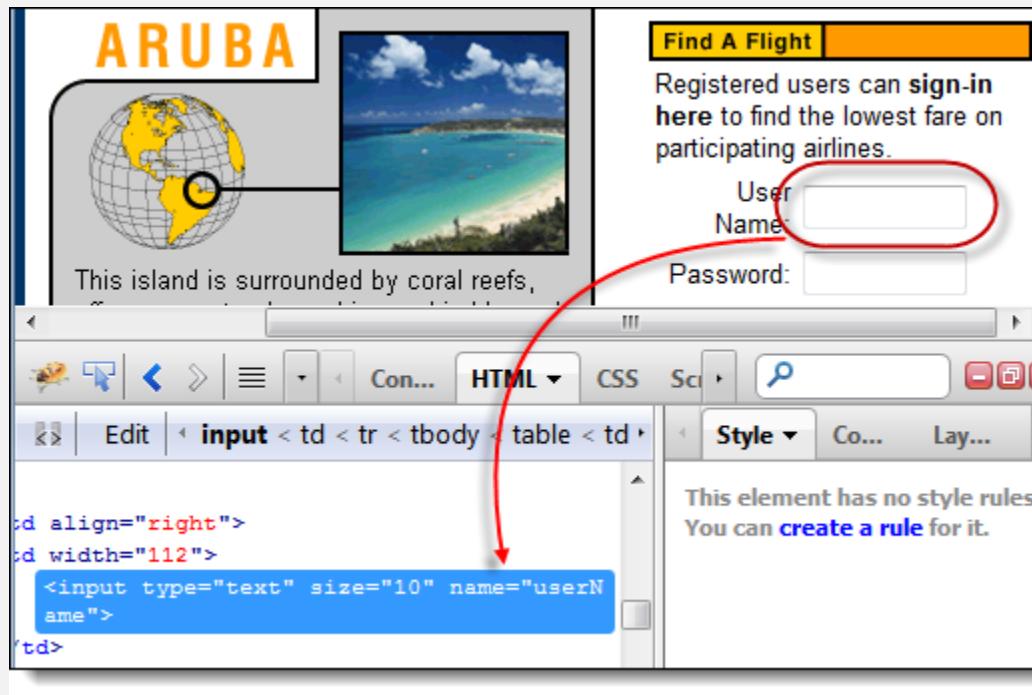
### Step 5

- To create the third command, click on the third blank line in the Editor and key-in "type" on the Command text box.



Click on the User Name text box. Notice that Firebug automatically shows you the HTML code for that element.

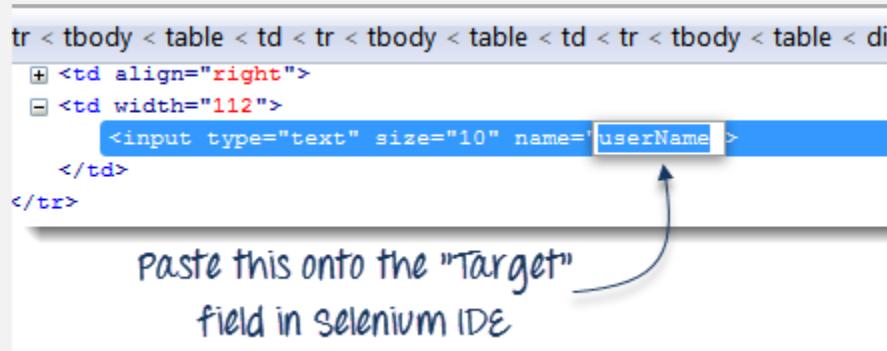
## Create a Script Manually with Firebug



### Step 6

Notice that the User Name text box does not have an ID, but it has a NAME attribute. We shall, therefore, use its NAME as the locator. Copy the NAME value and paste it onto the Target field in Selenium IDE.

## Create a Script Manually with Firebug



Still in the Target text box, prefix "userName" with "name=", indicating that Selenium IDE should target an element whose NAME attribute is "userName."

Command	Target	Value
open		
assertTitle	exact:Welcome: Mercury Tours	
type	name=userName	

Type "invalidUN" in the Value text box of Selenium IDE. Your test script should now look like the image below. We are done with the third command. Note: Instead of invalidUN, you may enter any other text string. But Selenium IDE is case sensitive and you type values/attributes exactly like in the application.

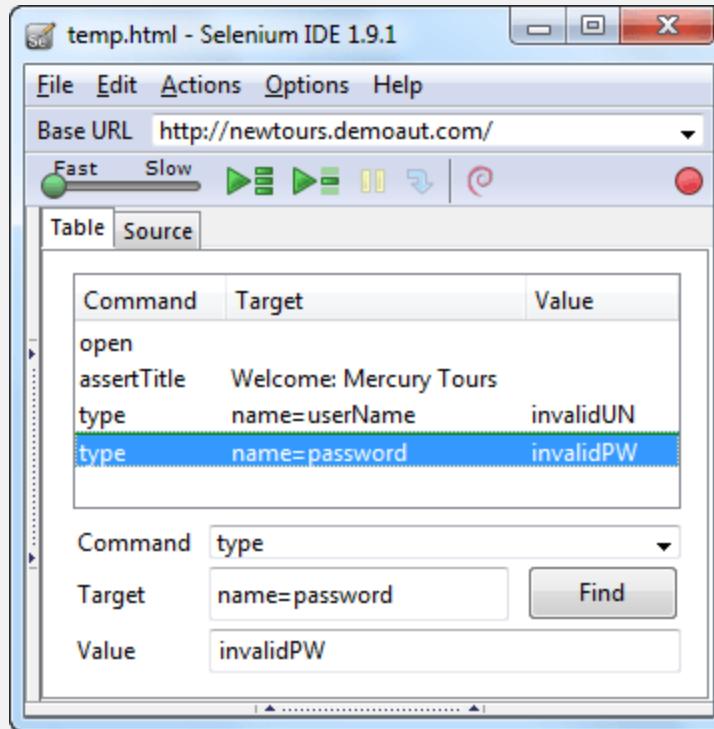
## Create a Script Manually with Firebug

### Step 7

- To create the fourth command, key-in "type" on the Command text box.
- Again, use Firebug's "Inspect" button to get the locator for the "Password" text box.

## Create a Script Manually with Firebug

- Type "invalidPW" in the Value field in Selenium IDE. Your test script should now look like the image below.



### Step 8

- For the fifth command, type "clickAndWait" on the Command text box in Selenium IDE.
- Use Firebug's "Inspect" button to get the locator for the "Sign In" button.

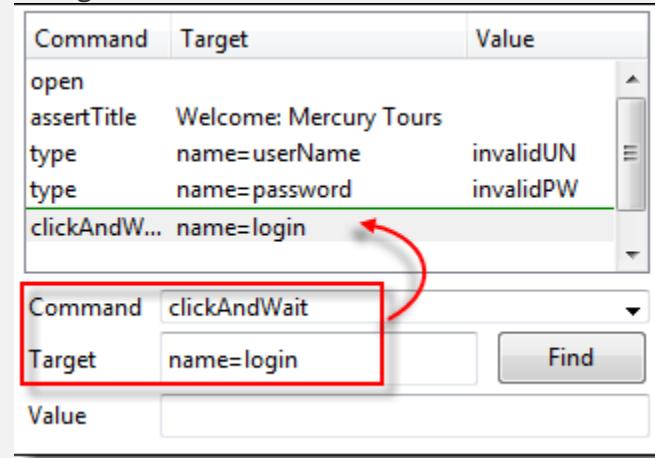
## Create a Script Manually with Firebug



```
<td width="112">
  <div align="center">
    <input width="58" type="image" height="17" border="0"
      alt="Sign-In" src="/images
      /btn_signin.gif" value="Login" name='login'>
  </div>
</td>
```

Again, the only available locator is the NAME attribute so this will be the one that we shall use.

- Paste the value of the NAME attribute ("login") onto the Target text box and prefix it with "name=".
- Your test script should now look like the image below.



Command	Target	Value
open		
assertTitle	Welcome: Mercury Tours	
type	name=userName	invalidUN
type	name=password	invalidPW
clickAndW...	name=login	

Command	Target	Value
clickAndWait	name=login	

## Create a Script Manually with Firebug

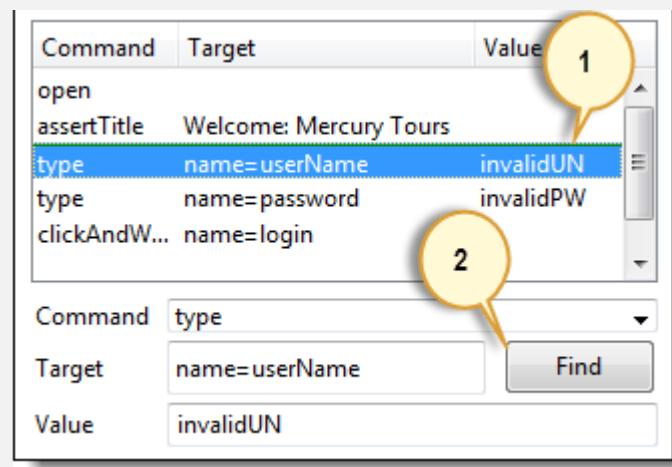
### Step 9

Save the test case in the same way as we did in the previous section.

# Using the Find Button

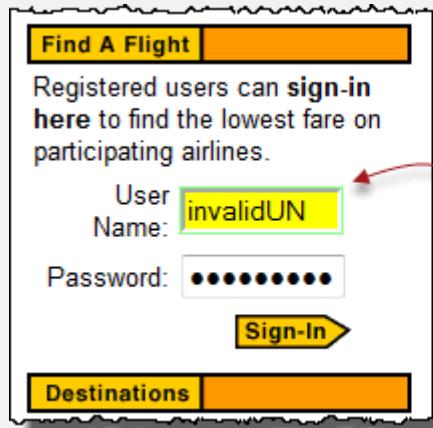
The Find button in Selenium IDE is used to verify if what we had put in the Target text box is indeed the correct UI element.

Let us use the Invalid\_login test case that we created in the previous sections. Click on any command with a Target entry, say, the third command.



Click on the Find button. Notice that the User Name text box within the Mercury Tours page becomes highlighted for a second.

## Create a Script Manually with Firebug



Every time the Find button is clicked, this element becomes highlighted with yellow and bordered with light green.

This indicates that Selenium IDE was able to detect and access the expected element correctly. If the Find button highlighted a different element or no element at all, then there must be something wrong with your script.

## Execute Command

This allows you to execute any single command without running the whole test case. Just click on the line you wish to execute and then either click on "Actions > Execute this command" from the menu bar or simply press "X" on your keyboard.

**Step 1.** Make sure that your browser is on the Mercury Tours homepage. Click on the command you wish to execute. In this example, click on the "type | userName | invalidUN" line.

**Step 2.** Press "X" on your keyboard.

**Step 3.** Observe that the text box for username becomes populated with the text "invalidUN"

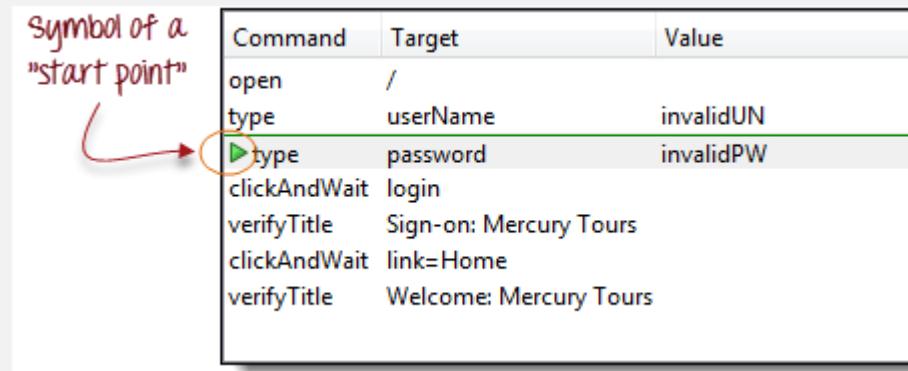
## Create a Script Manually with Firebug



Executing commands this way is highly dependent on the page that Firefox is currently displaying. This means that if you try the example above with the Google homepage displayed instead of Mercury Tours', then your step will fail because there is no text box with a "userName" attribute within Google's homepage.

## Start point

A start point is an indicator that tells Selenium IDE which line the execution will start. Its shortcut key is "S".



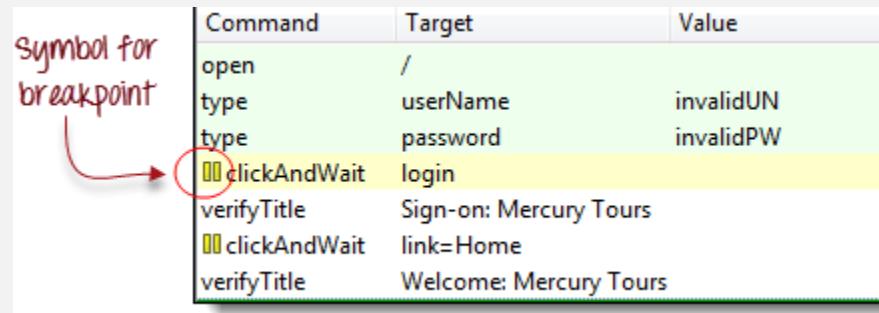
Command	Target	Value
open	/	
type	userName	invalidUN
type	password	invalidPW
clickAndWait	login	
verifyTitle	Sign-on: Mercury Tours	
clickAndWait	link=Home	
verifyTitle	Welcome: Mercury Tours	

In the example above, playback will start on the third line (type | password | invalidPW). **You can only have one start point in a single test script.**

Start point is similar to Execute Command in such that they are dependent on the currently displayed page. The start point will fail if you are on the wrong page.

# Breakpoints

Breakpoints are indicators that tell Selenium IDE where to automatically pause the test. **The shortcut key is "B".**



Command	Target	Value
open	/	
type	userName	invalidUN
type	password	invalidPW
clickAndWait	login	
verifyTitle	Sign-on: Mercury Tours	
clickAndWait	link=Home	
verifyTitle	Welcome: Mercury Tours	

The yellow highlight means that the current step is pending. This proves that Selenium IDE has paused execution on that step.

**You can have multiple breakpoints in one test case.**

## Step

It allows you to execute succeeding commands one at a time after pausing the test case. Let us use the scenario in the previous section "Breakpoints."

### **Before clicking "Step."**

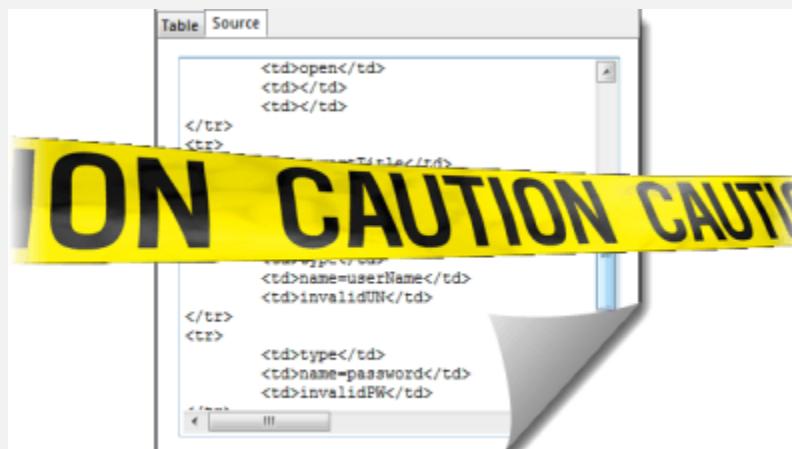
The test case pauses at the line "clickAndWait | login".

### **After clicking "Step."**

The "clickAndWait | login" line is run and pauses to the next command (verifyTitle | Sign-on: Mercury Tours).

Notice that the next line is paused even though there is no breakpoint there. This is the main purpose of the Step feature - it executes the succeeding commands one at a time to give you more time to inspect the

## Important Things to Note When Using Other Formats in Source View



Selenium IDE works well only with HTML - other formats are still in experimental mode. It is NOT advisable to create or edit tests using other formats in Source View because there is still a lot of work needed to make it stable. Below are the known bugs as of version 1.9.1.

- You will not be able to perform playback nor switch back to Table View unless you revert to HTML.
- The only way to add commands safely on the source code is by recording them.
- When you modify the source code manually, all of it will be lost when you switch to another format.
- Though you can save your test case while in Source View, Selenium IDE will not be able to open it.

The recommended way to convert Selenese tests is to use the "Export Test Case As..." option under the File menu, and not through the Source View.

# Summary

- Test scripts can be created either by recording or typing the commands and parameters manually.
- When creating scripts manually, Firebug is used to get the locator.
- The Find button is used to check that the command is able to access the correct element.
- Table View displays a test script in tabular form while Source View displays it in HTML format.
- Changing the Source View to a non-HTML format is still experimental.
- Do not use the Source View in creating tests in other formats. Use the Export features instead.
- Parameters are not required all the time. It depends upon the command.
- There are three types of commands:
  - Actions - directly interacts with page elements
  - Accessors - "reads" an element property and stores it in a variable
  - Assertions - compares an actual value with an expected one
- Assertions have three types:
  - Assert - upon failure, succeeding steps are no longer executed
  - Verify - upon failure, succeeding steps are still executed.
  - WaitFor - passes if the specified condition becomes true within the timeout period; otherwise, it will fail.

## Summary

- The most common commands are:
  - open
  - click/clickAndWait
  - type/typeKeys
  - verifyTitle/assertTitle
  - verifyTextPresent
  - verifyElementPresent
  - verifyTable
  - waitForPageToLoad
  - waitForElementPresent

# How to use Locators in Selenium IDE

SELENIUM TRAINING

## How to use Locators in Selenium IDE

**Locators tell Selenium IDE which GUI elements (say Text Box, Buttons, Check Boxes etc.) its needs to operate on.**

Identification of correct GUI elements is a prerequisite to create an automation script. But accurate identification of GUI elements is more difficult than it sounds. Sometimes, you end up working with incorrect GUI elements or no elements at all! Hence, Selenium provides a number of Locators to precisely locate a GUI element

The different types of locator are:

- ID
- Name
- Link Text
- CSS Selector
  - Tag and ID
  - Tag and class
  - Tag and attribute
  - Tag, class, and attribute
  - Inner text
- DOM (Document Object Model)
  - getElementById
  - getElementsByName
  - dom:name
  - dom: index
- XPath

# How to use Locators in Selenium IDE

There are commands that do not need a locator (such as the "open" command). However, most of them do need Locators.

**The choice of locator depends largely on your Application Under Test (AUT).** In this tutorial, we will toggle between Facebook, new tours.demoaut on the basis of locators that these applications support. Likewise in your [testing](#) project you will select any of the above listed locators based on your application support.

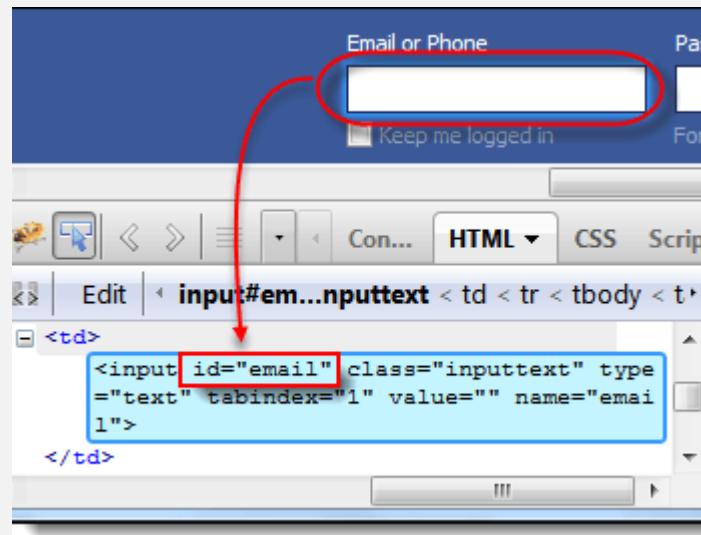
# Locating by ID

This is the most common way of locating elements since ID's are supposed to be unique for each element.

**Target Format:** id=id of the element

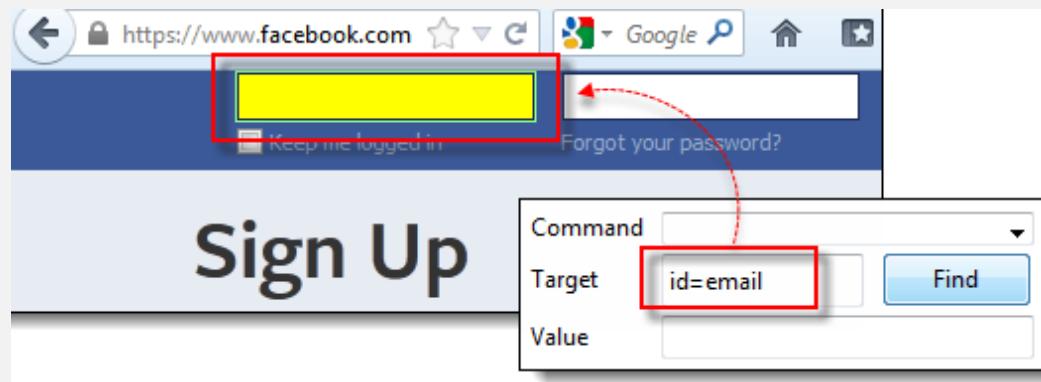
For this example, we will use Facebook as our test app because Mercury Tours does not use ID attributes.

**Step 1.** Navigate to <http://www.facebook.com>. Inspect the "Email or Phone" text box using Firebug and take note of its ID. In this case, the ID is "email".



**Step 2.** Launch Selenium IDE and enter "id=email" in the Target box. Click the Find button and notice that the "Email or Phone" text box becomes highlighted with yellow and bordered with green, meaning, Selenium IDE was able to locate that element correctly.

## Locating by ID



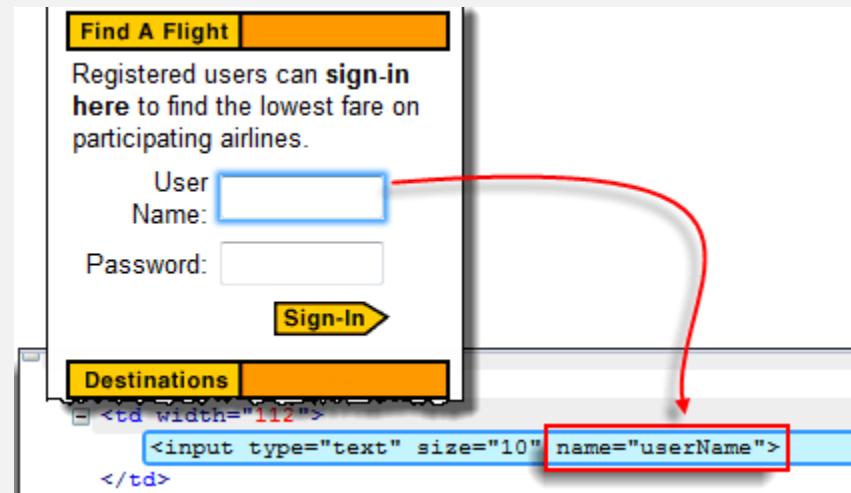
## Locating by Name

Locating elements by name are very similar to locating by ID, except that we use the "**name=**" prefix instead.

**Target Format:** `name=name of the element`

In the following demonstration, we will now use Mercury Tours because all significant elements have names.

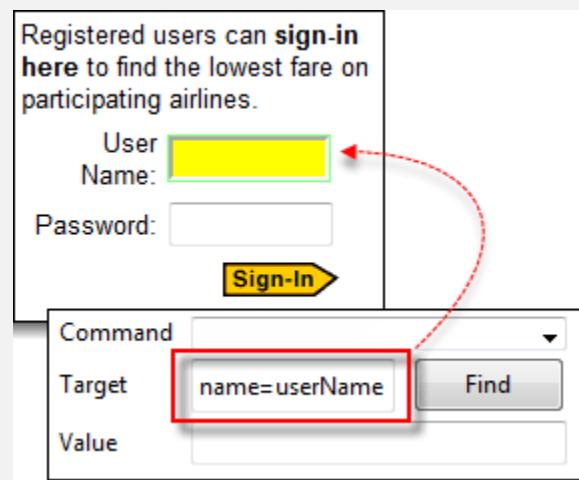
**Step 1.** Navigate to <http://newtours.demoaut.com/> and use Firebug to inspect the "User Name" text box. Take note of its name attribute.



Here, we see that the element's name is "username".

**Step 2.** In Selenium IDE, enter "name=username" in the Target box and click the Find button. Selenium IDE should be able to locate the User Name text box by highlighting it.

## Locating by Name



# Locating by Name using Filters

Filters can be used when multiple elements have the same name. **Filters are additional attributes used to distinguish elements with the same name.**

**Target Format:** name=name\_of\_the\_element filter=value\_of\_filter

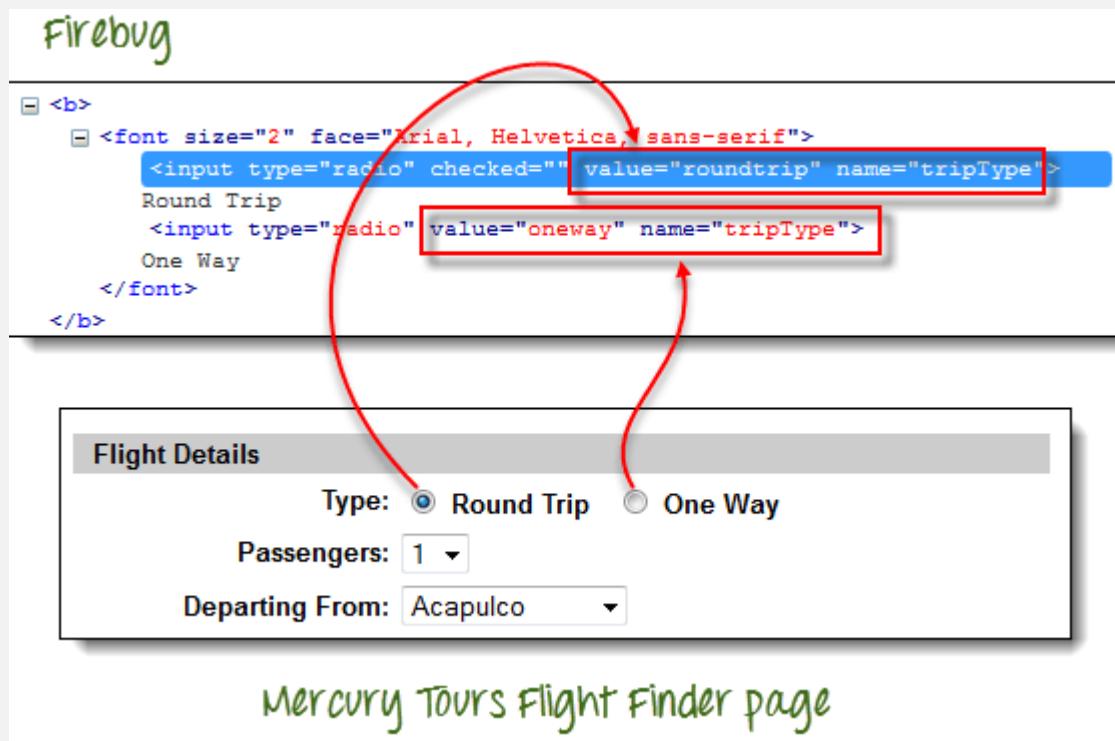
Let's see an example -

**Step 1.** Log on to Mercury Tours using "tutorial" as the username and password. It should take you to the Flight Finder page shown below.

The screenshot shows a web form titled "FLIGHT FINDER" with a yellow header bar. Below the header, a message encourages users to search for the lowest fare and visit the Hotel Finder. The form is divided into sections: "Flight Details" and "Preferences". In the "Flight Details" section, there are fields for "Type" (radio buttons for "Round Trip" and "One Way", with "Round Trip" selected), "Passengers" (dropdown menu showing "1"), "Departing From" (dropdown menu showing "Acapulco"), "On" (dropdown menus for month "December" and day "5"), "Arriving In" (dropdown menu showing "Acapulco"), and "Returning" (dropdown menus for month "December" and day "5"). In the "Preferences" section, there are radio buttons for "Service Class": "Economy class" (selected), "Business class", and "First class". There is also a dropdown menu for "Airline" with the value "No Preference". At the bottom right of the form is a "CONTINUE" button.

**Step 2.** Using Firebug, notice that the Round Trip and One Way radio buttons have the same name "tripType." However, they have different VALUE attributes so we can use each of them as our filter.

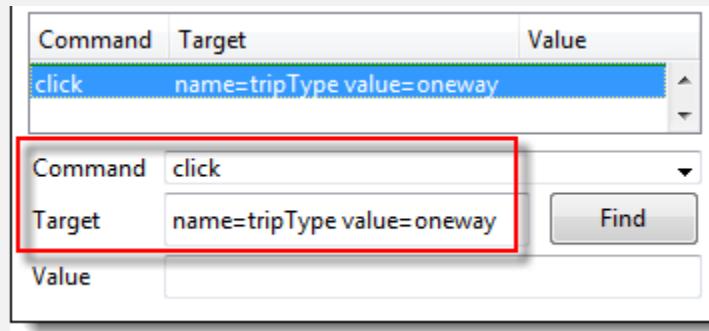
## Locating by Name using Filters



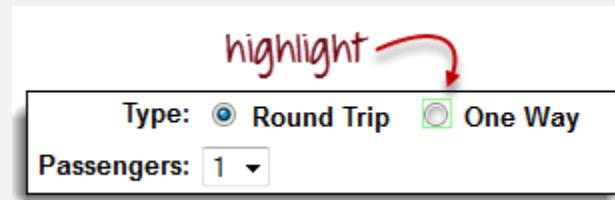
### Step 3.

- We are going to access the One Way radio button first. Click the first line on the Editor.
- In the Command box of Selenium IDE, enter the command "click".
- In the Target box, enter "name=tripType value=oneway". The "value=oneway" portion is our filter.

## Locating by Name using Filters

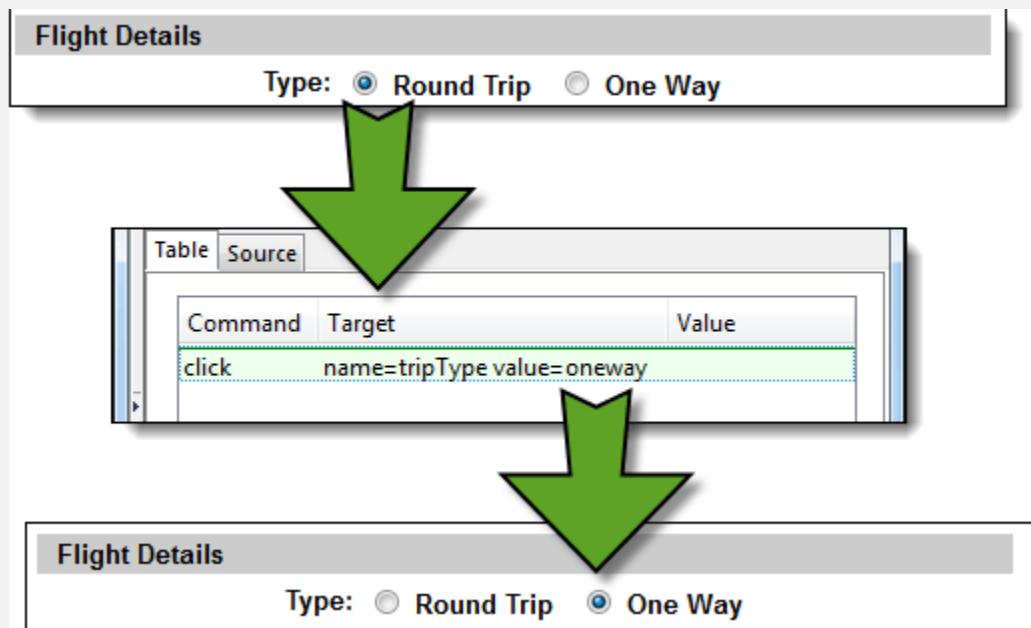


**Step 4.** Click the Find button and notice that Selenium IDE is able to highlight the One Way radio button with green - meaning that we are able to access the element successfully using its VALUE attribute.



**Step 5.** Press the "X" key in your keyboard to execute this click command. Notice that the One Way radio button became selected.

## Locating by Name using Filters



You can do the exact same thing with the Round Trip radio button, this time, using "name=tripType value=roundtrip" as your target.

## Locating by Link Text

This type of locator applies only to hyperlink texts. We access the link by prefixing our target with "link=" and then followed by the hyperlink text.

**Target Format:** link=link\_text

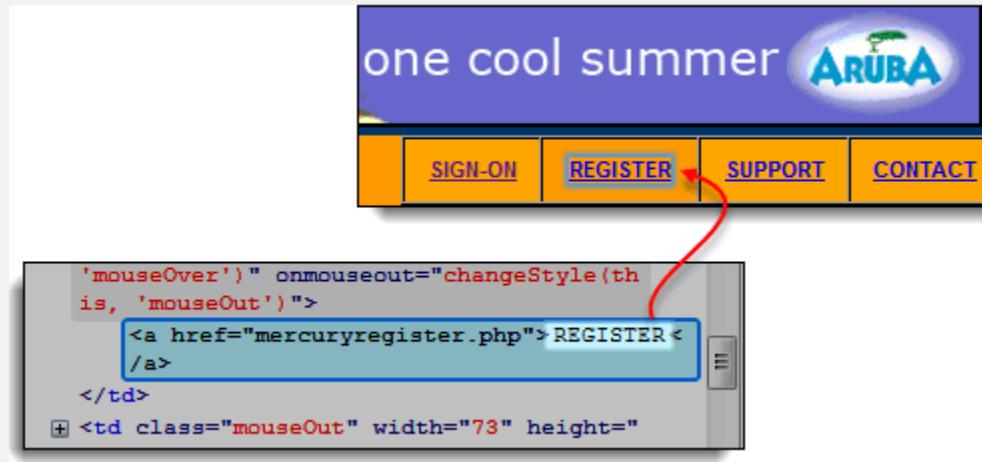
In this example, we shall access the "REGISTER" link found in the Mercury Tours homepage.

**Step 1.**

- First, make sure that you are logged off from Mercury Tours.
- Go to Mercury Tours homepage.

**Step 2.**

- Using Firebug, inspect the "REGISTER" link. The link text is found between and tags.
- In this case, our link text is "REGISTER". Copy the link text.



## Locating by Link Text

**Step 3.** Copy the link text in Firebug and paste it onto Selenium IDE's Target box. Prefix it with "link=".

Command	Target	Value
	link=REGISTER	

Command:

Target:  Find

Value:

**Step 4.** Click on the Find button and notice that Selenium IDE was able to highlight the REGISTER link correctly.



**Step 5.** To verify further, enter "clickAndWait" in the Command box and execute it. Selenium IDE should be able to click on that REGISTER link successfully and take you to the Registration page shown below.

## Locating by Link Text

To create your account, we'll need some basic information about you. This information will be used to send reservation confirmation emails, mail tickets when needed and contact you if your travel arrangements change. Please fill in the form completely.

**Contact Information**

First Name:

Last Name:

Phone:

# Locating by CSS Selector

CSS Selectors are string patterns used to identify an element based on a combination of HTML tag, id, class, and attributes. Locating by CSS Selector is more complicated than the previous methods, but it is the **most common locating strategy of advanced Selenium users because it can access even those elements that have no ID or name.**

CSS Selectors have many formats, but we will only focus on the most common ones.

- **Tag and ID**
- **Tag and class**
- **Tag and attribute**
- **Tag, class, and attribute**
- **Inner text**

When using this strategy, we always prefix the Target box with "css=" as will be shown on the following examples.

## Locating by CSS Selector - Tag and ID

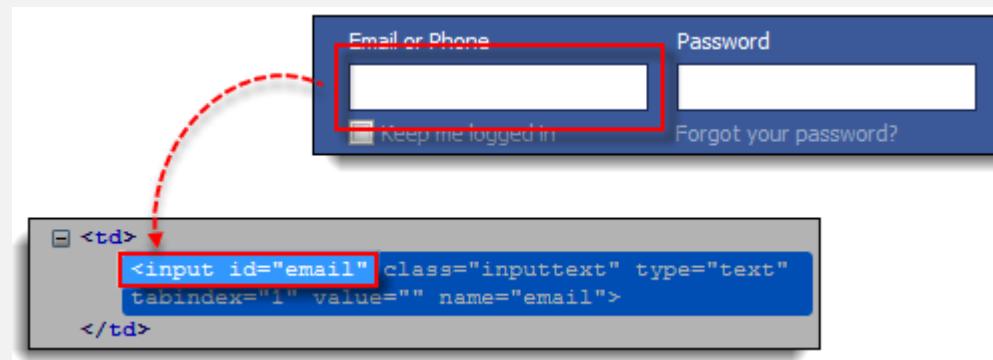
Again, we will use Facebook's Email text box in this example. As you can remember, it has an ID of "email" and we have already accessed it in the "Locating by ID" section. This time, we will use a CSS Selector with ID in accessing that very same element.

Syntax	Description
<code>css=tag#id</code>	<p>tag = the HTML tag of the element being accessed</p> <p># = the hash sign. This should always be present when using a CSS Selector with ID</p> <p>id = the ID of the element being accessed</p>

Keep in mind that the ID is always preceded by a hash sign (#).

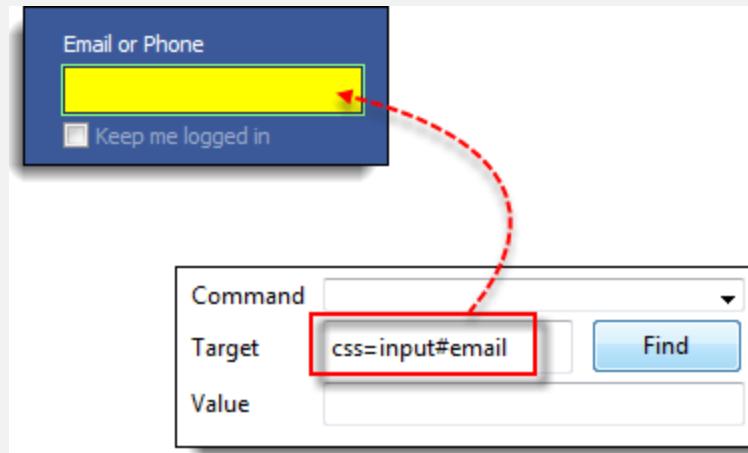
**Step 1.** Navigate to [www.facebook.com](http://www.facebook.com). Using Firebug, examine the "Email or Phone" text box.

At this point, take note that the HTML tag is "input" and its ID is "email". So our syntax will be "css=input#email".



**Step 2.** Enter "css=input#email" into the Target box of Selenium IDE and click the Find button. Selenium IDE should be able to highlight that element.

## Locating by CSS Selector - Tag and ID

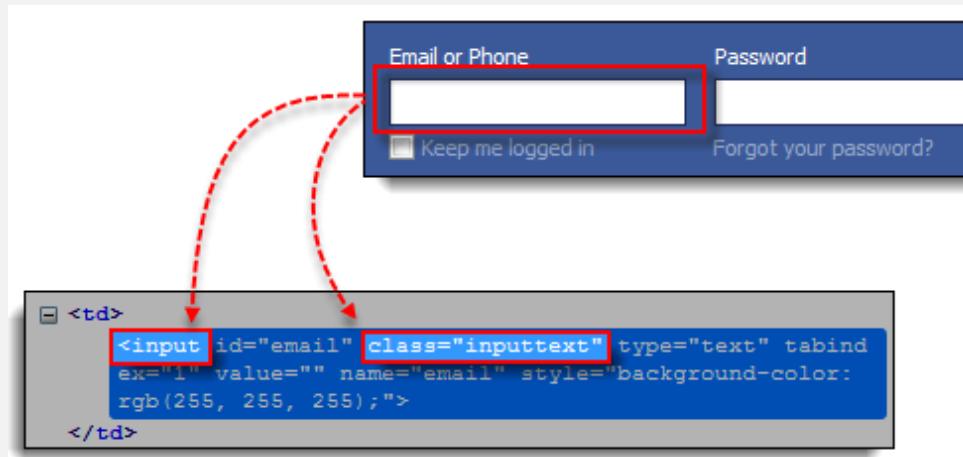


## Locating by CSS Selector - Tag and Class

Locating by CSS Selector using an HTML tag and a class name is similar to using a tag and ID, but in this case, a dot (.) is used instead of a hash sign.

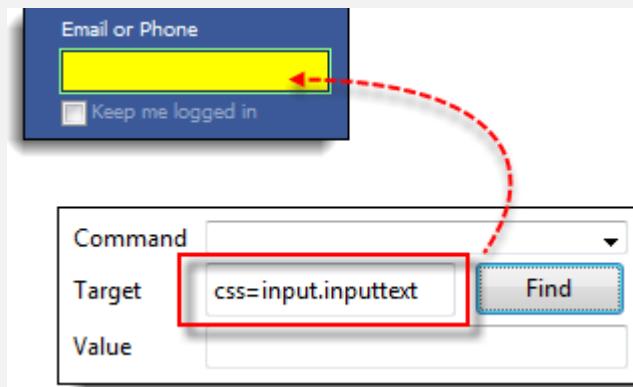
Syntax	Description
css=tag.class	<p>tag = the HTML tag of the element being accessed</p> <p>. = the dot sign. This should always be present when using a CSS Selector with class</p> <p>class = the class of the element being accessed</p>

**Step 1.** Navigate to [www.facebook.com](http://www.facebook.com) and use Firebug to inspect the "Email or Phone" text box. Notice that its HTML tag is "input" and its class is "inputtext".



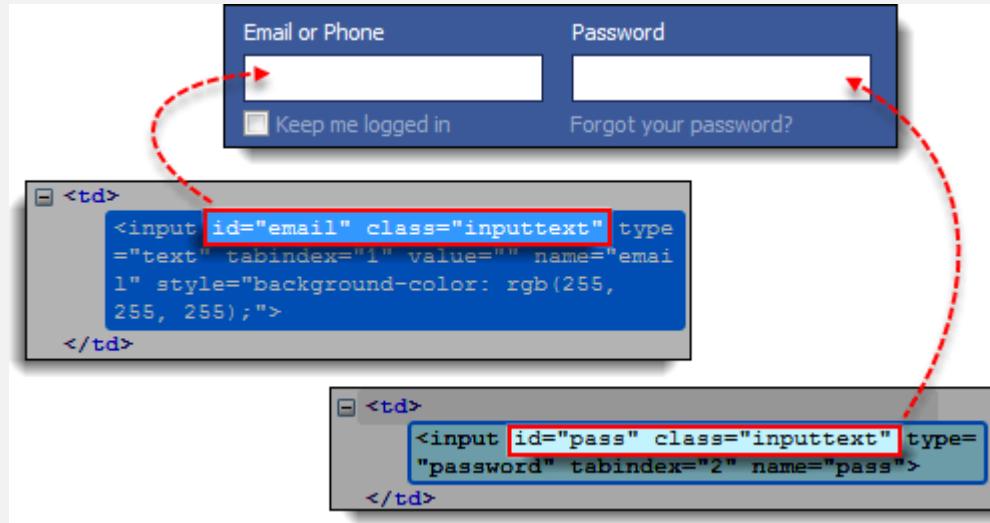
## Locating by CSS Selector - Tag and Class

**Step 2.** In Selenium IDE, enter "css=input.inputtext" in the Target box and click Find. Selenium IDE should be able to recognize the Email or Phone text box.



Take note that when multiple elements have the same HTML tag and name, only the first element in source code will be recognized. Using Firebug, inspect the Password text box in Facebook and notice that it has the same name as the Email or Phone text box.

## Locating by CSS Selector - Tag and Class



The reason why only the Email or Phone text box was highlighted in the previous illustration is that it comes first in Facebook's page source.

## Locating by CSS Selector - Tag and Class

The screenshot shows a login interface with two input fields: 'Email or Phone' and 'Password'. Handwritten text above the first field states: "'Email or Phone' Was written first in Facebook's page source". A red arrow points from this text to the first input field. Another red arrow points from the same text to the corresponding HTML code in the source view.

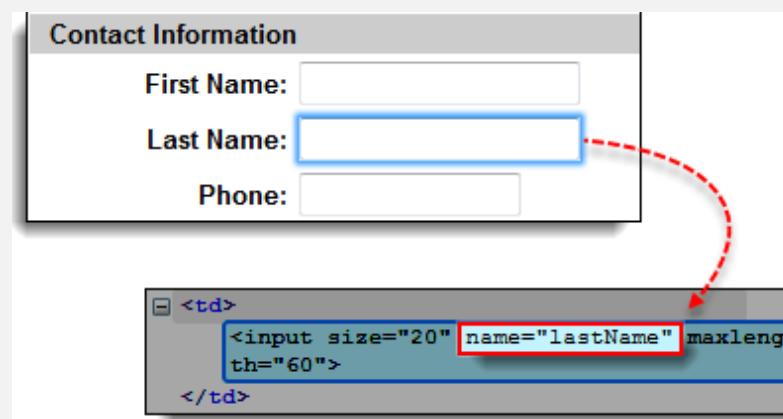
```
<td>
<input id="email" class="inputtext" type="text" tabindex="1" value="" name="email" style="background-color: rgb(255, 255, 255);">
</td>
<td>
<input id="pass" class="inputtext" type="password" tabindex="2" name="pass">
</td>
```

## Locating by CSS Selector - Tag and Attribute

This strategy uses the HTML tag and a specific attribute of the element to be accessed.

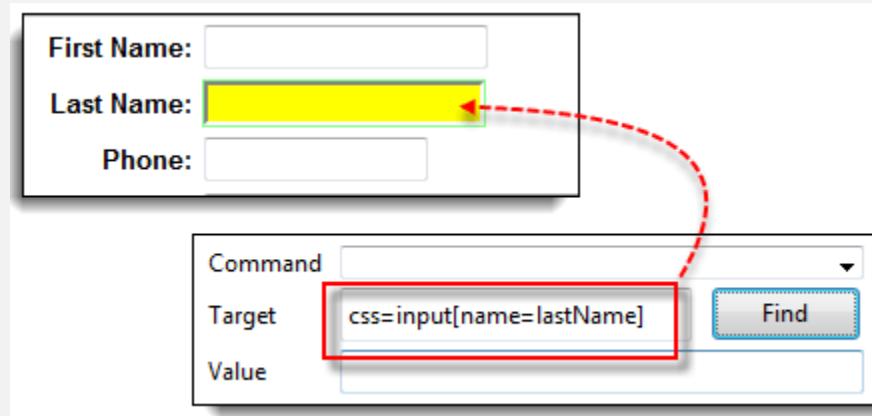
Syntax	Description
<pre>css=tag[attribute=value]</pre>	<p>tag = the HTML tag of the element being accessed</p> <p>[ and ] = square brackets within which a specific attribute and its corresponding value will be placed</p> <p>attribute = the attribute to be used. It is advisable to use an attribute that is unique to the element such as a name or ID.</p> <p>value = the corresponding value of the chosen attribute.</p>

**Step 1.** Navigate to Mercury Tours' Registration page (<http://newtours.demoaut.com/mercuryregister.php>) and inspect the "Last Name" text box. Take note of its HTML tag ("input" in this case) and its name ("lastName").



## Locating by CSS Selector - Tag and Attribute

**Step 2.** In Selenium IDE, enter "css=input[name=lastName]" in the Target box and click Find. Selenium IDE should be able to access the Last Name box successfully.

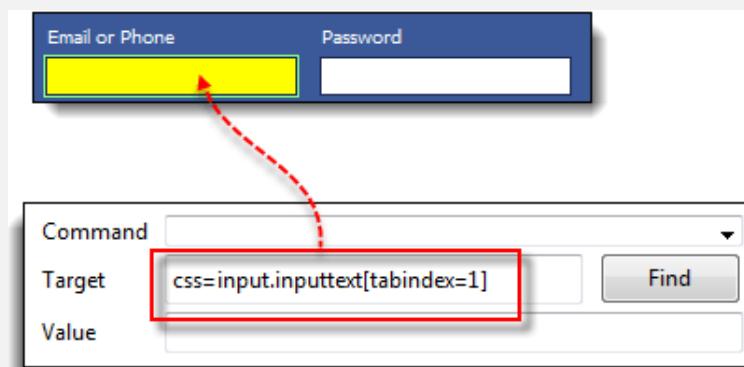


**When multiple elements have the same HTML tag and attribute, only the first one will be recognized.** This behavior is similar to locating elements using CSS selectors with the same tag and class.

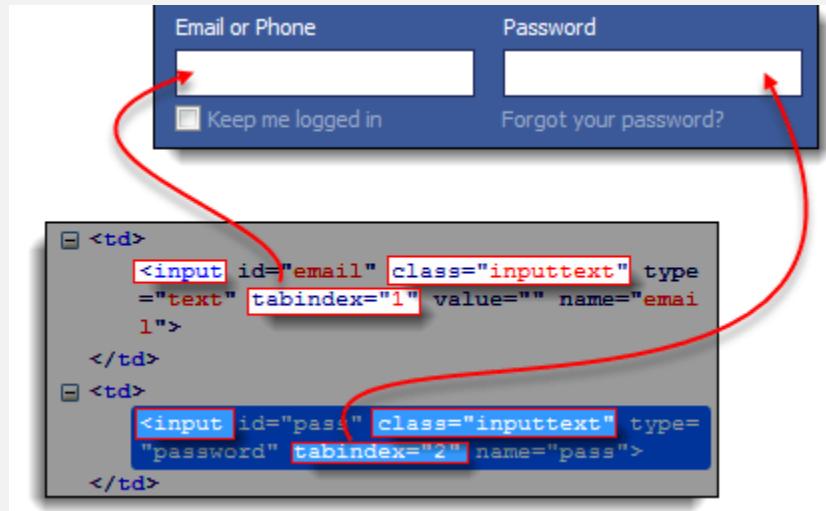
## Locating by CSS Selector - tag, class, and attribute

Syntax	Description
<code>css=tag.class[attribute=value]</code>	<p>tag = the HTML tag of the element being accessed</p> <p>. = the dot sign. This should always be present when using a CSS Selector with class</p> <p>class = the class of the element being accessed</p> <p>[ and ] = square brackets within which a specific attribute and its corresponding value will be placed</p> <p>attribute = the attribute to be used. It is advisable to use an attribute that is unique to the element such as a name or ID.</p>

**Step 1.** Navigate to [www.facebook.com](http://www.facebook.com) and use Firebug to inspect the 'Email or Phone' and 'Password' input boxes. Take note of their HTML tag, class, and attributes. For this example, we will select their 'tabindex' attributes.



## Locating by CSS Selector - tag, class, and attribute

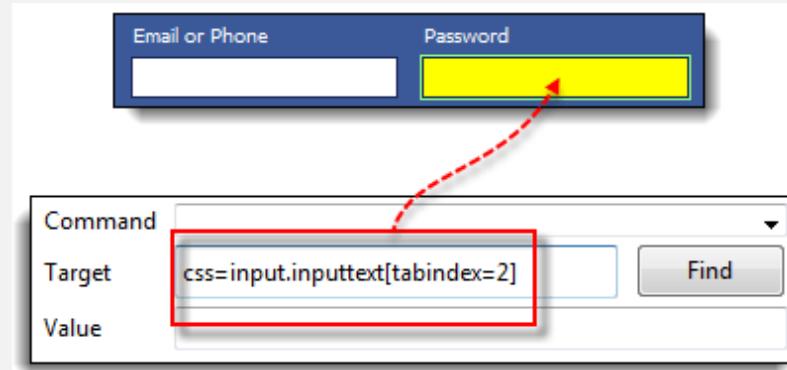


**Step 2.** We will access the 'Email or Phone' text box first, thus, we will use a tabindex value of 1.

Enter "css=input.inputtext[tabindex=1]" in Selenium IDE's Target box and click Find. The 'Email or Phone' input box should be highlighted.

## Locating by CSS Selector - tag, class, and attribute

**Step 3.** To access the Password input box, simply replace the value of the tabindex attribute. Enter "css=input.inputtext[tabindex=2]" in the Target box and click on the Find button. Selenium IDE must be able to identify the Password text box successfully.

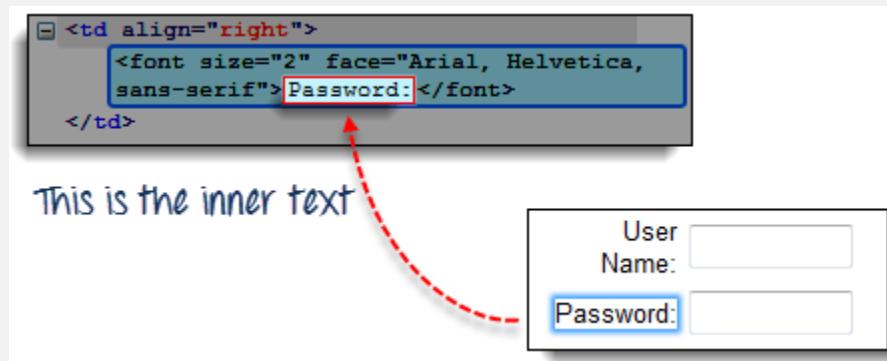


## Locating by CSS Selector - inner text

As you may have noticed, HTML labels are seldom given id, name, or class attributes. So, how do we access them? The answer is through the use of their inner texts. **Inner texts are the actual string patterns that the HTML label shows on the page.**

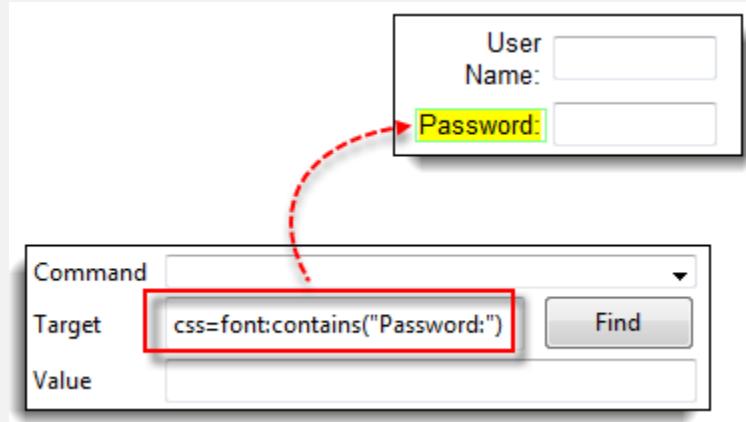
Syntax	Description
<code>css=tag:contains("inner text")</code>	tag = the HTML tag of the element being accessed inner text = the inner text of the

**Step 1.** Navigate to Mercury Tours' homepage (<http://newtours.demoaut.com/>) and use Firebug to investigate the "Password" label. Take note of its HTML tag (which is "font" in this case) and notice that it has no class, id, or name attributes.



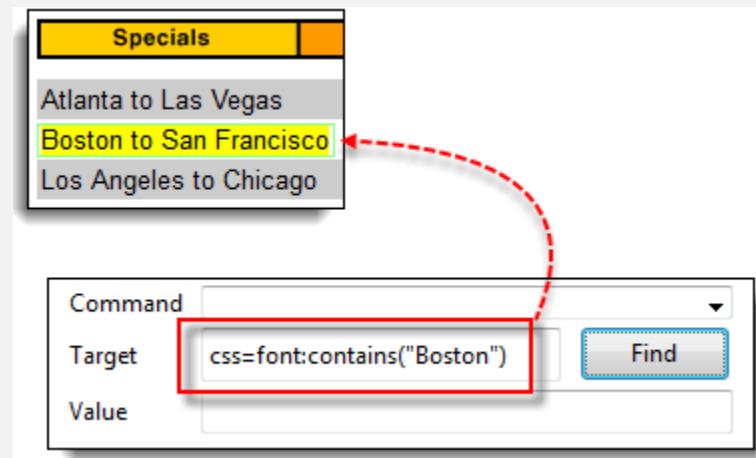
**Step 2.** Type `css=font:contains("Password:")` into Selenium IDE's Target box and click Find. Selenium IDE should be able to access the Password label as shown on the image below.

## Locating by CSS Selector - inner text



**Step 3.** This time, replace the inner text with "Boston" so that your Target will now become "css=font:contains("Boston")". Click Find. You should notice that the "Boston to San Francisco" label becomes highlighted. This shows you that Selenium IDE can access a long label even if you just indicated the first word of its inner text.

## Locating by CSS Selector - inner text



# Locating by DOM (Document Object Model)

The Document Object Model (DOM), in simple terms, is the way by which HTML elements are structured. Selenium IDE is able to use the DOM in accessing page elements. If we use this method, our Target box will always start with "dom=document..."; however, the "dom=" prefix is normally removed because Selenium IDE is able to automatically interpret anything that starts with the keyword "document" to be a path within the DOM anyway.

There are four basic ways to locate an element through DOM:

- getElementById
- getElementsByName
- dom:name (applies only to elements within a named form)
- dom:index

Locating by DOM - getElementById

Let us focus on the first method - using the getElementById method. The syntax would be:

Syntax	Description
<code>document.getElementById("id of the element")</code>	id of the element = this is the value of the ID attribute of the element to be accessed. This value should always be enclosed in a pair of parentheses ("").

**Step 1.** Navigate to [www.mail.yahoo.com](http://www.mail.yahoo.com) and use Firebug to inspect the "Stay signed in" check box. Take note of its ID.

## Locating by DOM (Document Object Model)

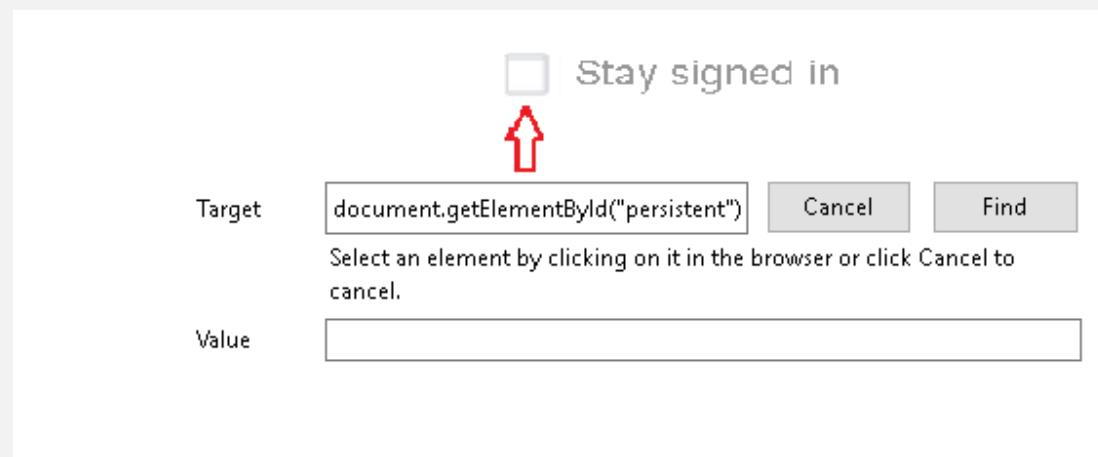
The screenshot shows a mobile sign-in interface. At the top is a text input field labeled "Enter your email". Below it is a blue button labeled "Next". Underneath the button is a checkbox labeled "Stay signed in" with a checked status. To the right of the checkbox is a link "Need help?". A red arrow points from the left towards the "Stay signed in" checkbox. Below the interface is a scrollable code editor displaying the DOM structure. The code highlights the checkbox element with a red arrow pointing to its `id="persistent"` attribute. The code is as follows:

```
+ <div id="submits" class="mbr-login-submit ">
  <div class="stay-signin-in pure-g">
    <div id="persistency" class="pure-u-1-2 mbr-checkbox ">
      <div class="mbr-text-align mbr-login-text-normal">
        <input id="persistent" class="checkbox" type="checkbox" checked="" tabindex="3" value="y" name=".persistent">
        <label for="persistent">Stay signed in</label>
      </div>
```

We can see that the ID we should use is "persistent".

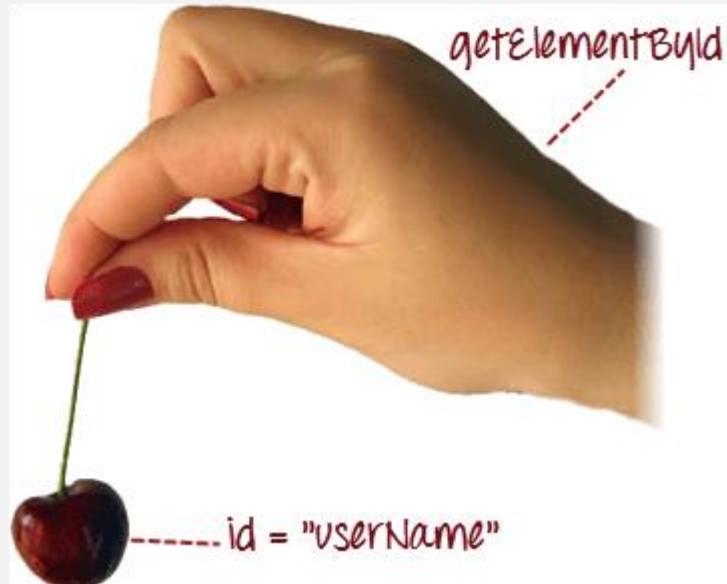
**Step 2.** Open Selenium IDE and in the Target box, enter "document.getElementById("persistent")" and click Find. Selenium IDE should be able to locate the "Keep me logged in" check box. Though it cannot highlight the interior of the check box, Selenium IDE can still surround the element with a bright green border as shown below.

## Locating by DOM (Document Object Model)



## Locating by DOM – getElementsByTagName

The getElementById method can access only one element at a time, and that is the element with the ID that you specified. The getElementsByTagName method is different. It collects an array of elements that have the name that you specified. You access the individual elements using an index which starts at 0.



**getElementById**

It will get only one element for you.

That element bears the ID that you specified inside the parentheses of getElementById().



## getElementsByName

### getElementsByName

It will get a collection of elements whose names are all the same.

Each element is indexed with a number starting from 0 just like an array

You specify which element you wish to access by putting its index number into the square brackets in getElementsByName's syntax below.

Syntax	Description
<code>document.getElementsByName("name")[index]</code>	<p>name = name of the element as defined by its 'name' attribute index = an integer that indicates which element within getElementsByName's array will be used.</p>

**Step 1.** Navigate to Mercury Tours' Homepage and login using "tutorial" as the username and password. Firefox should take you to the Flight Finder screen.

## Locating by DOM – getElementsByName

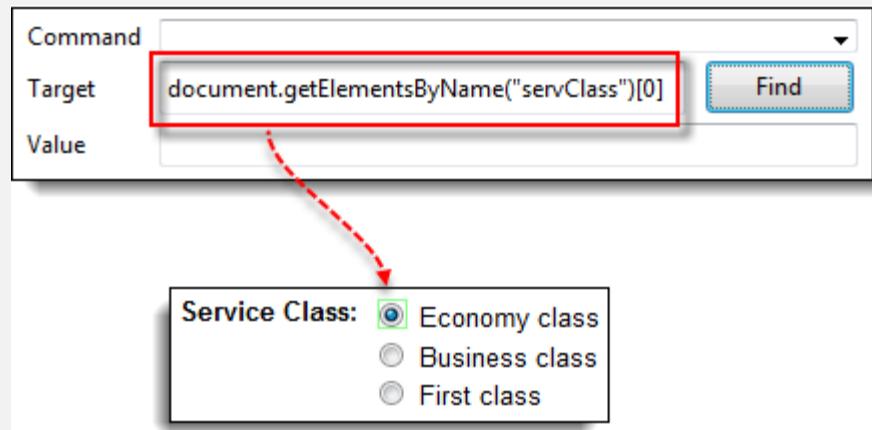
**Step 2.** Using Firebug, inspect the three radio buttons at the bottom portion of the page (Economy class, Business class, and First class radio buttons). Notice that they all have the same name which is "servClass".

The screenshot shows a 'Preferences' dialog box with a 'Service Class' section containing three radio buttons: 'Economy class' (selected), 'Business class', and 'First class'. Below this is a dropdown menu for 'Airline' with the value 'No Preference'. The Firebug DOM tree below the dialog shows the HTML structure of the 'Service Class' section. Three red arrows point from the 'Economy class' radio button in the UI to its corresponding DOM element in the tree. The DOM elements are:

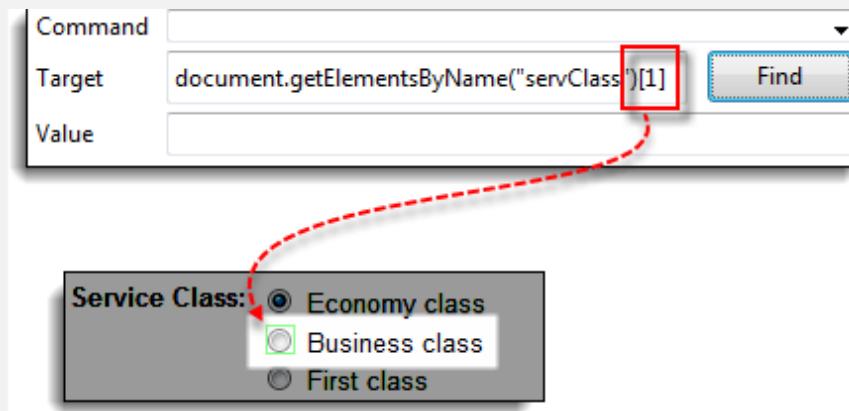
```
<td>
  <font size="2">
    <input type="radio" value="Coach" name="servClass" checked="">
    <font face="Arial, Helvetica, sans-serif">
      Economy class
      <br>
      <input type="radio" value="Business" name="servClass">
      Business class
      <br>
      <input type="radio" value="First" name="servClass">
      First class
    </font>
  </font>
</td>
```

**Step 3.** Let us access the "Economy class" radio button first. Of all these three radio buttons, this element comes first so it has an index of 0. In Selenium IDE, type "document.getElementsByName("servClass")[0]" and click the Find button. Selenium IDE should be able to identify the Economy class radio button correctly.

## Locating by DOM – getElementsByName



**Step 4.** Change the index number to 1 so that your Target will now become document.getElementsByName("servClass")[1]. Click the Find button and Selenium IDE should be able to highlight the "Business class" radio button, as shown below.



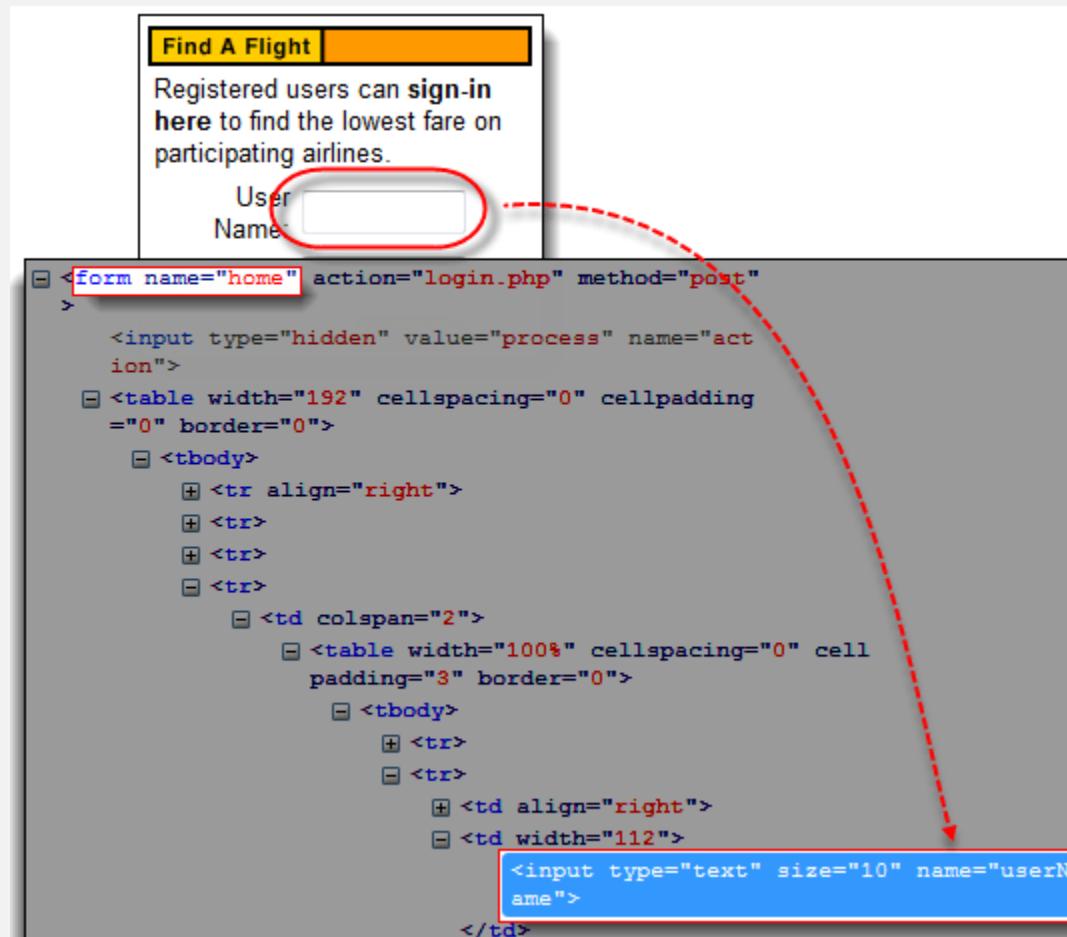
## Locating by DOM - dom:name

As mentioned earlier, this method will only apply if the element you are accessing is contained within a named form.

Syntax	Description
<code>document.forms["name of the form"].elements["name of the element"]</code>	name of the form = the value of the name attribute of the form tag that contains the element you want to access name of the element = the value of the name attribute of the element you

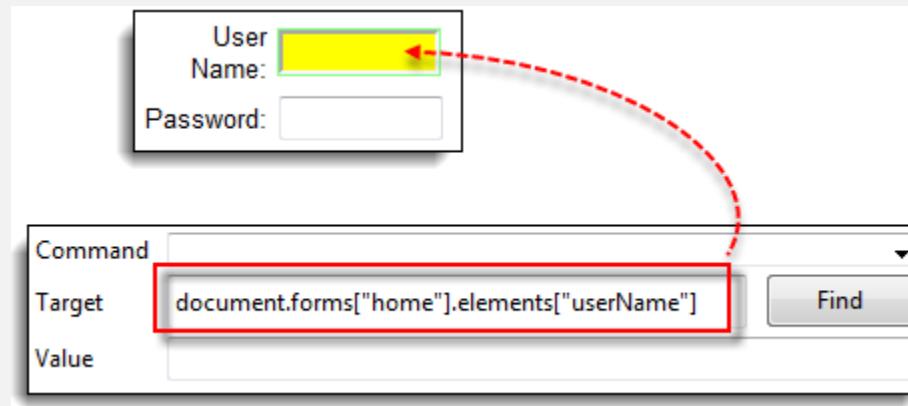
**Step 1.** Navigate to Mercury Tours homepage (<http://newtours.demoaut.com/>) and use Firebug to inspect the User Name text box. Notice that it is contained in a form named "home".

## Locating by DOM - dom:name



**Step 2.** In Selenium IDE, type "document.forms["home"].elements["userName"]" and click the Find button. Selenium IDE must be able to access the element successfully.

## Locating by DOM - dom:name



## Locating by DOM - dom:index

This method applies even when the element is not within a named form because it uses the form's index and not its name.

Syntax	Description
<code>document.forms[index of the form].elements[index of the element]</code>	index of the form = the index number (starting at 0) of the form with respect to the whole page index of the element = the index number (starting at 0) of the element with

We shall access the "Phone" text box within Mercury Tours Registration page. The form in that page has no name and ID attribute so this will make a good example.

**Step 1.** Navigate to Mercury Tours Registration page and inspect the Phone text box. Notice that the form containing it has no ID and name attributes.

## Locating by DOM - dom:index

No name and id attributes. This is the only form in the page, so its index will be 0

The screenshot shows a DOM tree for a form. The root node is a form element with action="mercurycreate\_account.php" and method="post". It contains a table with four rows (tr). The first row has a single td containing a hidden input field. The second row has two td's: the first is aligned right and the second contains an input field for first name. The third row has two td's: the first is aligned right and the second contains an input field for last name. The fourth row has two td's: the first is aligned right and the second contains an input field for phone number. Orange callout bubbles on the right side of the slide indicate the element index for each input field: element index = 0 for the first input, element index = 1 for the second, element index = 2 for the third, and element index = 3 for the fourth.

```
<form action="mercurycreate_account.php" method="post">
    <input type="hidden" value="process" name="mercury" style="background-color: rgb(255, 255, 255);"/>
    <table width="100%" cellpadding="2" border="0">
        <tbody>
            <tr bgcolor="#CCCCCC">
                <td align="right"></td>
                <td><input size="20" name="firstName" maxlength="60"></td>
            </tr>
            <tr>
                <td align="right"></td>
                <td><input size="20" name="lastName" maxlength="60"></td>
            </tr>
            <tr>
                <td align="right"></td>
                <td><input size="15" name="phone" maxlength="20" style="background-color: #FFFFFF;"></td>
            </tr>
        </tbody>
    </table>
</form>
```

element index = 0

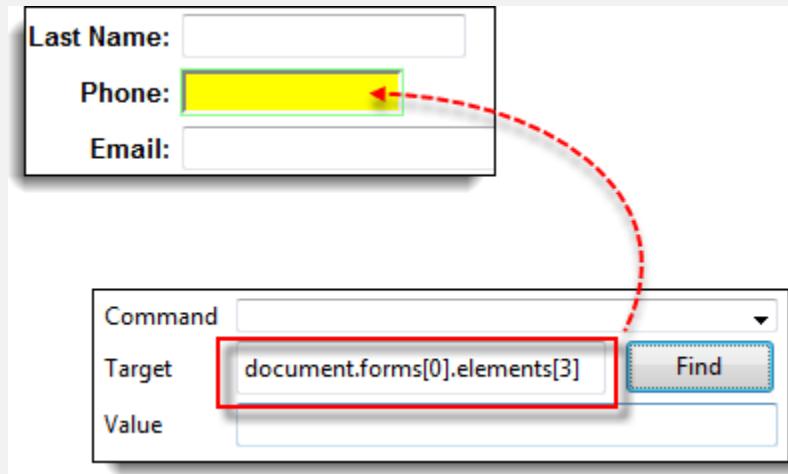
element index = 1

element index = 2

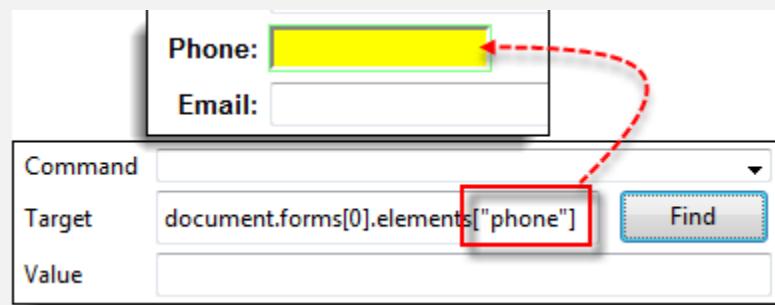
element index = 3

## Locating by DOM - dom:index

**Step 2.** Enter "document.forms[0].elements[3]" in Selenium IDE's Target box and click the Find button. Selenium IDE should be able to access the Phone text box correctly.



**Step 3.** Alternatively, you can use the element's name instead of its index and obtain the same result. Enter "document.forms[0].elements['phone']" in Selenium IDE's Target box. The Phone text box should still become highlighted.



# Locating by XPath

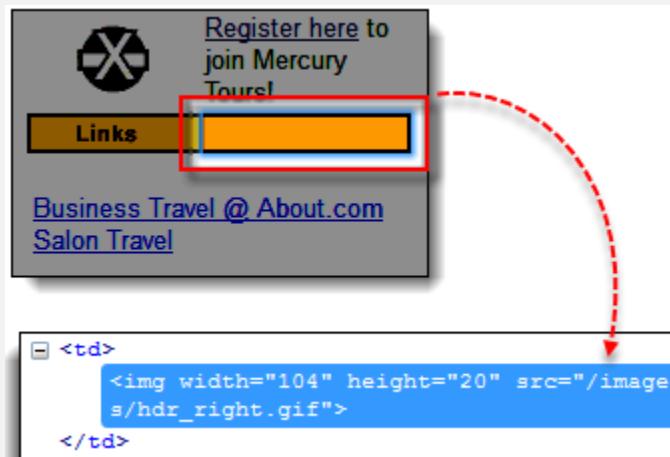
XPath is the language used when locating XML (Extensible Markup Language) nodes. Since HTML can be thought of as an implementation of XML, we can also use XPath in locating HTML elements.

**Advantage:** It can access almost any element, even those without class, name, or id attributes.

**Disadvantage:** It is the most complicated method of identifying elements because of too many different rules and considerations.

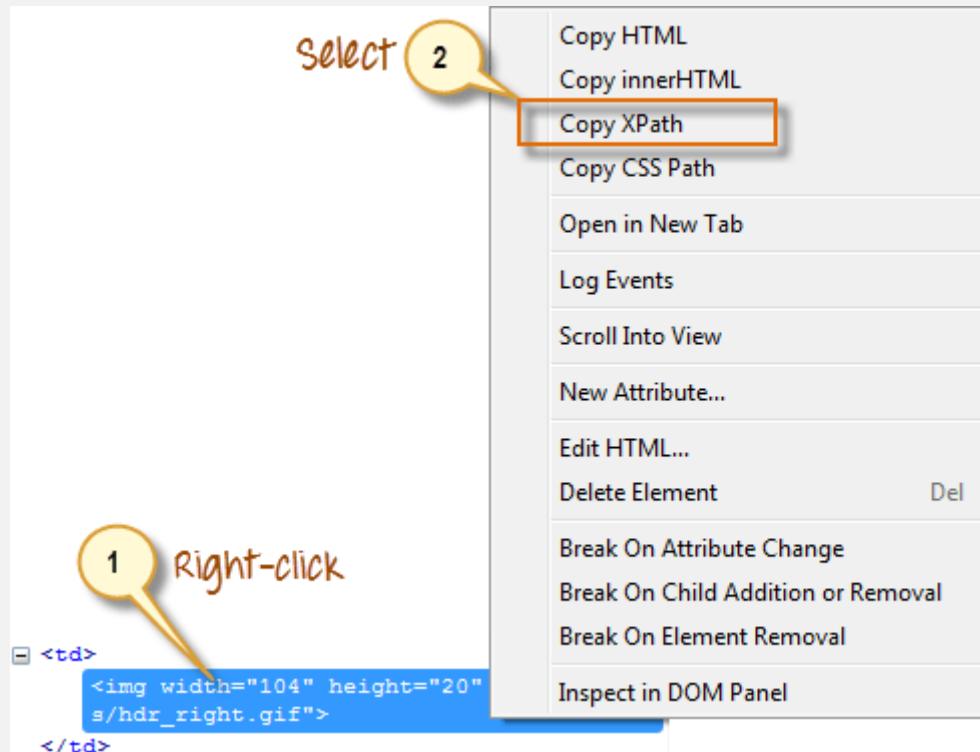
Fortunately, Firebug can automatically generate XPath locators. In the following example, we will access an image that cannot possibly be accessed through the methods we discussed earlier.

**Step 1.** Navigate to Mercury Tours Homepage and use Firebug to inspect the orange rectangle to the right of the yellow "Links" box. Refer to the image below.



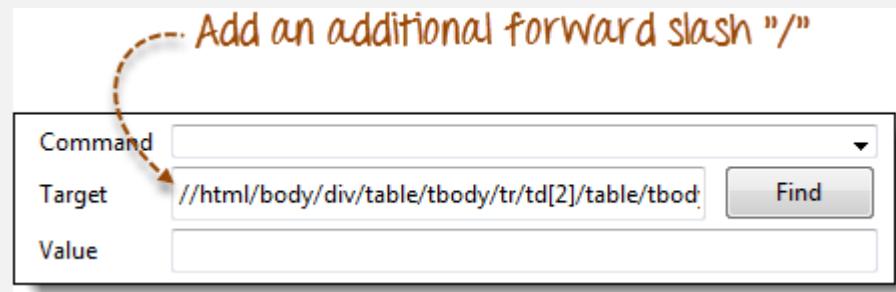
**Step 2.** Right click on the element's HTML code and then select the "Copy XPath" option.

## Locating by XPath

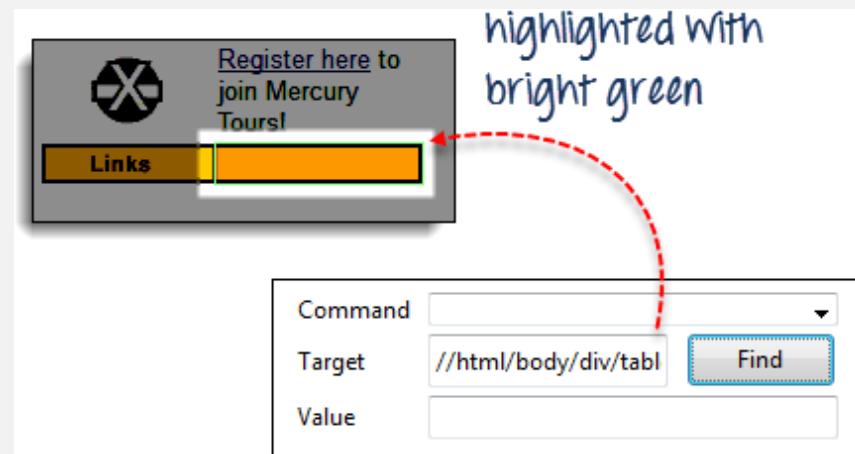


**Step 3.** In Selenium IDE, type one forward slash "/" in the Target box then paste the XPath that we copied in the previous step. **The entry in your Target box should now begin with two forward slashes "///".**

## Locating by XPath



**Step 4.** Click on the Find button. Selenium IDE should be able to highlight the orange box as shown below.



# Summary

Syntax for Locator Usage

Method	Target Syntax	Example
By ID	<code>id=id_of_the_element</code>	<code>id=email</code>
By Name	<code>name=name_of_the_element</code>	<code>name=username</code>
By Name Using Filters	<code>name=name_of_the_element filter=value_of_filter</code>	<code>name=tripType value=oneway</code>
By Link Text	<code>link=link_text</code>	<code>link=REGISTER</code>
Tag and ID	<code>css=tag#id</code>	<code>css=input#email</code>
Tag and Class	<code>css=tag.class</code>	<code>css=input.inputtext</code>
Tag and Attribute	<code>css=tag[attribute=value]</code>	<code>css=input[name=lastName]</code>
Tag, Class, and Attribute	<code>css=tag.class[attribute=value]</code>	<code>css=input.inputtext[tabindex=1]</code>

# How to enhance a script using Selenium IDE

SELENIUM TRAINING

## How to enhance a script using Selenium IDE

In this tutorial, we look at commands that will make your automation script more intelligent and complete.

# Verify Presence of an Element

We can use following two commands to verify the presence of an element:

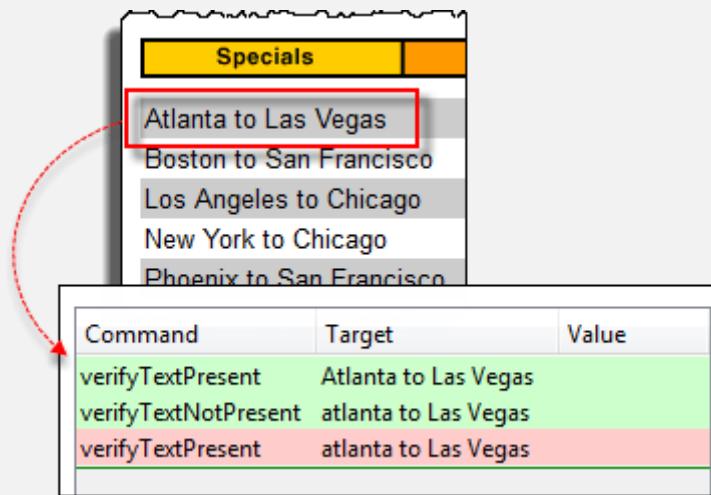
- **verifyElementPresent** - returns TRUE if the specified element was FOUND in the page; FALSE if otherwise
  - **verifyElementNotPresent** - returns TRUE if the specified element was NOT FOUND anywhere in the page; FALSE if it is present.
- The test script below verifies that the `userN`ame text box is present within the Mercury Tours homepage while the First Name text box is not. The First Name text box is actually an element present in the Registration page of Mercury Tours, not in the homepage.
- Verify Presence of a Certain Text

This step failed, thereby confirming that the element with name=firstname is not indeed present in the page

Command	Target	Value
verifyElementPresent	name=userN	
verifyElementNotPresent	name=firstname	
verifyElementPresent	name=firstname	

## Verify Presence of a Certain Text

- **verifyTextPresent** - returns TRUE if the specified text string was FOUND somewhere in the page; FALSE if otherwise
  - **verifyTextNotPresent** - returns TRUE if the specified text string was NOT FOUND anywhere in the page; FALSE if it was found
- Remember that these commands are case-sensitive.



In the scenario above, "Atlanta to Las Vegas" was treated differently from "atlanta to Las Vegas" because the letter "A" of "Atlanta" was in uppercase on the first one while lowercase on the other. When the verifyTextPresent command was used on each of them, one passed while the other failed.

# Verify Specific Position of an Element

Selenium IDE indicates the position of an element by measuring (in pixels) how far it is from the left or top edge of the browser window.

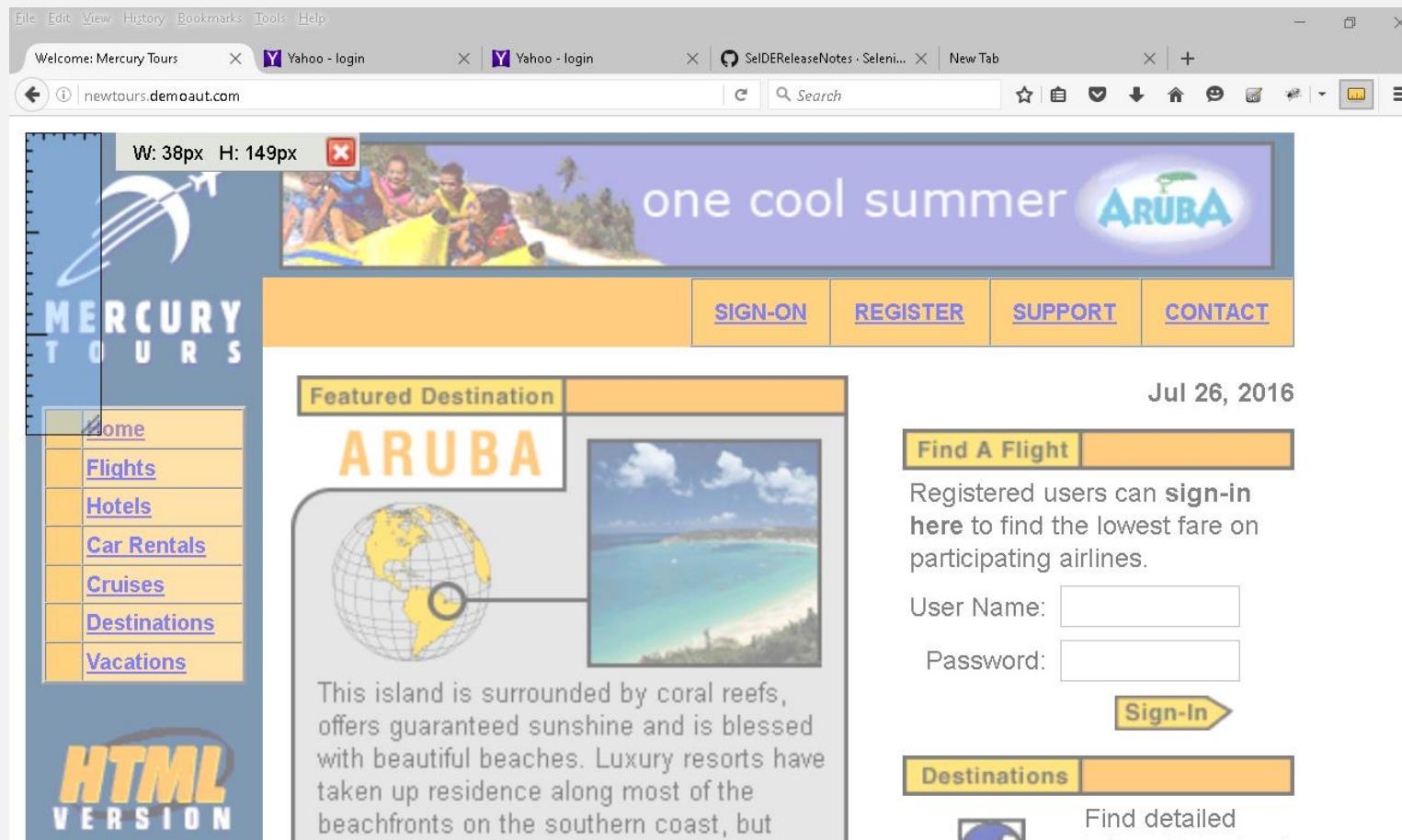
- **verifyElementPositionLeft** - verifies if the specified number of pixels match the distance of the element from the left edge of the page. This will return FALSE if the value specified does not match the distance from the left edge.
- **verifyElementPositionTop** - verifies if the specified number of pixels match the distance of the element from the top edge of the page. This will return FALSE if the value specified does not match the distance from the top edge.

# Verify Specific Position of an Element (Firefox Add – ons)

MeasureIt 0.4.15

by Kevin Freitas

Draw a ruler across any webpage to check the width, height, or alignment of page elements in pixels.



## Wait commands and Wait commands

These are commands that will wait for a new page to load before moving onto the next command. Examples are:

- clickAndWait
- typeAndWait
- selectAndWait

## Wait commands and Wait commands

PASSED

Command	Target	Value
open	/	
type	name=userName	tutorial
type	name=password	tutorial
clickAndWait	name=login	
assertTitle	Find a Flight: Mercury Tours:	

This PASSED because "clickAndWait" was used. Selenium IDE first waited for a new page to load before executing the "assertTitle" command.

FAILED

Command	Target	Value
open	/	
type	name=userName	tutorial
type	name=password	tutorial
click	name=login	
assertTitle	Find a Flight: Mercury Tours:	

This 2nd test case FAILED because "click" was used. Selenium IDE executed the "assertTitle" command without waiting for a new page to load.

## **waitFor commands**

These are commands that wait for a specified condition to become true before proceeding to the next command (irrespective of loading of a new page). These commands are more appropriate to be used on AJAX-based dynamic websites that change values and elements without reloading the whole page. Examples include:

- `waitForTitle`
- `waitForTextPresent`
- `waitForAlert`

Consider the Facebook scenario below.

## waitFor commands



We can use a combination of "click" and "waitForTextPresent" to verify the presence of the text "Providing your birthday".

## waitFor commands

Command	Target	Value
open		
click	link=Why do I need to provide my birthday?	
waitForTextPresent	Providing your birthday	
verifyTextPresent	Providing your birthday	

PASSED

We cannot use clickAndWait because no page was loaded upon clicking on the "Why do I need to provide my birthday?" link. If we do, the test will fail

## waitFor commands

since clickAndWait was used even though no page was loaded, selenium IDE basically waited for nothing

Command	Target	Value
open		
clickAndWait	link=Why do I need to provide my birthday?	
waitForTextPresent	Providing your birthday	
verifyTextPresent	Providing your birthday	

```
[info] Executing: |clickAndWait| link=Why do I need to provide my birthday? ||  
[error] Timed out after 20000ms  
[info] Executing: |waitForTextPresent| Providing your birthday ||
```



# Storing Variables and the Echo command

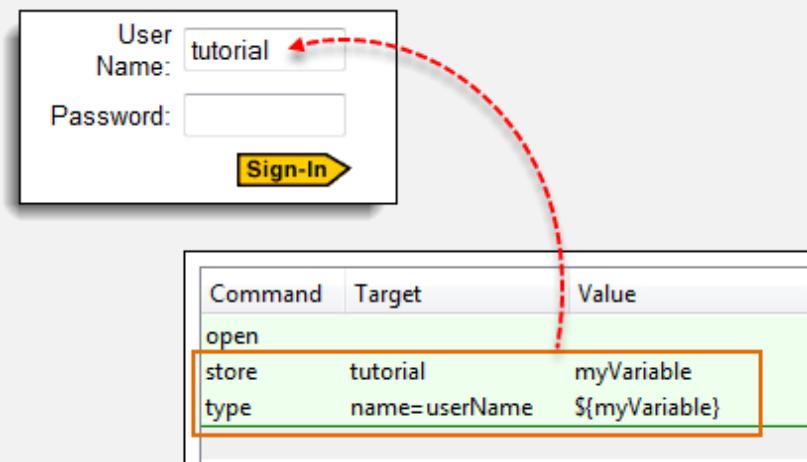
## Store

To store variables in Selenium IDE, we use the "store" command. The illustration below stores the value "tutorial" to a variable named "myVariable".



To access the variable, simply enclose it in a \${...} symbol. For example, to enter the value of "myVariable" onto the "userName" textbox of Mercury Tours, enter \${myVariable} in the Value field.

## Storing Variables and the Echo command



## StoreElementPresent

This command stores either "true" or "false" depending on the presence of the specified element. The script below stores the Boolean value "true" to "var1" and "false" to "var2". To verify, we will use the "echo" command to display the values of var1 and var2. The Base URL for the illustration below was set to Mercury Tours homepage.

A screenshot of a web page showing a login form. The form has three fields: 'User' (with placeholder 'Name'), 'Name' (with placeholder 'Name'), and 'Password'. Below the fields is a yellow 'Sign-In' button. A red dashed arrow points from the 'name' field to the text 'name=userName' above it. A purple note on the right says: 'Note: The element with name=firstname does not exist in this page'.

Command	Target	Value
open		
storeElementPresent	name=userName	var1
storeElementPresent	name=firstname	var2
echo	<code> \${var1}</code>	[info] Executing:   echo   \${var1}    [info] echo: true
echo	<code> \${var2}</code>	[info] Executing:   echo   \${var2}    [info] echo: false

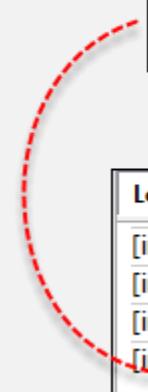
## StoreText

This command is used to store the inner text of an element onto a variable. The illustration below stores the inner text of the tag in Facebook onto a variable named 'textVar'.



Since it is the only element in the page, it is safe to use 'css=h1' as our target. The image below shows that Selenium IDE was able to save the string "Sign Up" in the 'textVar' variable by printing its value correctly.

## StoreText



Command	Target	Value
open		
storeText	css=h1	textVar
echo		<code> \${textVar}</code>

Log	Reference	UI-Element	Rollup
[info] Executing:  open			
[info] Executing:  storeText  css=h1   textVar			
[info] Executing:  echo  \${textVar}			
[info] echo: Sign Up			

# Alerts, Pop-up, and Multiple Windows

Alerts are probably the simplest form of pop-up windows. The most common Selenium IDE commands used in handling alerts are the following:

assertAlert	
assertNotAlert	retrieves the message of the alert and asserts it to a string value that you specified
assertAlertPresent	
assertAlertNotPresent	asserts if an Alert is present or not
storeAlert	retrieves the alert message and stores it in a variable that you will specify
storeAlertPresent	returns TRUE if an alert is present; FALSE if otherwise
verifyAlert	
verifyNotAlert	retrieves the message of the alert and verifies if it is equal to the string value that you specified
verifyAlertPresent	
verifyAlertNotPresent	verifies if an Alert is present or not

Remember these two things when working with alerts:

- Selenium IDE will automatically click on the OK button of the alert window and so you will not be able to see the actual alert.

## Alerts, Pop-up, and Multiple Windows

- Selenium IDE will not be able to handle alerts that are within the page's onload() function. It will only be able to handle alerts that are generated after the page has completely loaded.

In this example, we will use the storeAlert command to show that even though Selenium IDE did not show the actual alert, it was still able to retrieve its message.

**Step 1.** In Selenium IDE, set the Base URL to <http://jsbin.com>.

**Step 2.** Create the script as shown below.

## Alerts, Pop-up, and Multiple Windows

Command	Target	Value
open	/usidix/1	
click	css=input[value="Go!"]	
storeAlert	alertMessage	
echo	<code> \${alertMessage}</code>	

The screenshot shows a Firefox browser window with the URL `jsbin.com/usidix/1`. The page content is:

Click the button below to launch an alert.

When this button is clicked,  
an alert will pop up

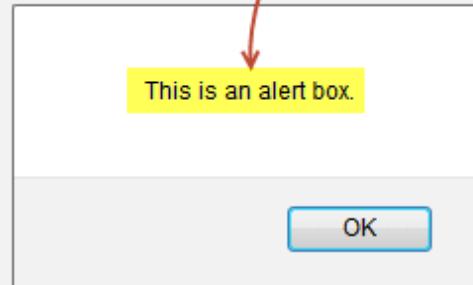
A red arrow points from the 'click' command in the test script table to the 'Go!' button on the web page. Another red arrow points from the 'alertMessage' entry in the table to the text 'an alert will pop up' on the page.

**Step 3.** Execute the script and do not expect that you will be able to see the actual alert.

## Alerts, Pop-up, and Multiple Windows

This is from the Info logs

```
[info] Executing: |open | /usidix/1 ||  
[info] Executing: |click | css=input[value="Go!"] ||  
[info] Executing: |storeAlert | alertMessage ||  
[info] Executing: |echo | ${alertMessage} ||  
[info] echo: This is an alert box.
```



This is the actual alert box when you try to click the "Go!" button manually. Notice that even though Selenium IDE prevented this alert from showing, it was still able to correctly get the alert message

# Confirmations

Confirmations are pop-ups that give you an OK and a CANCEL button, as opposed to alerts which give you only the OK button. The commands you can use in handling confirmations are similar to those in handling alerts.

- assertConfirmation/assertNotConfirmation
- assertConfirmationPresent/assertConfirmationNotPresent
- storeConfirmation
- storeConfirmationPresent
- verifyConfirmation/verifyNotConfirmation
- verifyConfirmationPresent/verifyConfirmationNotPresent
- 

However, these are the additional commands that you need to use to instruct Selenium which option to choose, whether the OK or the CANCEL button.

- chooseOkOnNextConfirmation/chooseOkOnNextConfirmationAndWait
- chooseCancelOnNextConfirmation

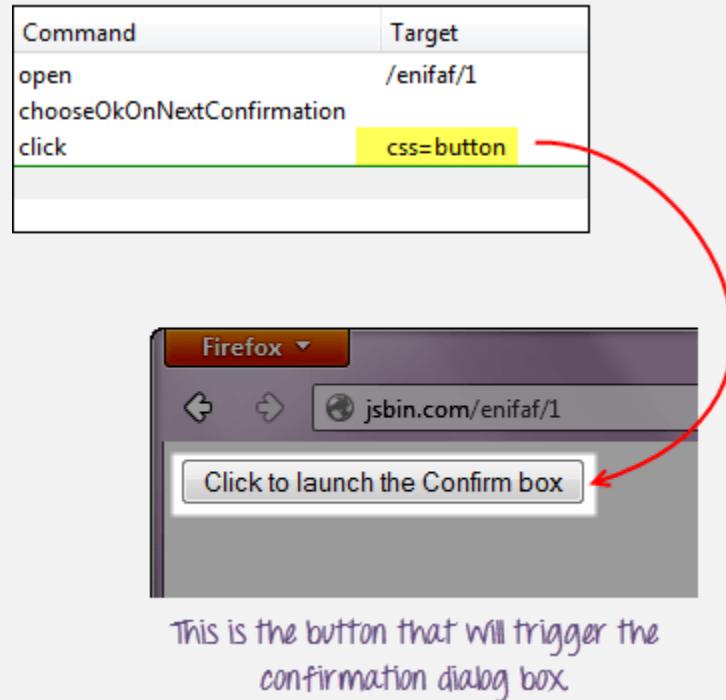
**You should use these commands before a command that triggers the confirmation box so that Selenium IDE will know beforehand which option to choose.** Again, you will not be able to see the actual confirmation box during script execution.

Let us test a webpage that has a button that was coded to show whether the user had pressed the OK or the CANCEL button.

**Step 1.** In Selenium IDE, set the Base URL to <http://jsbin.com>

**Step 2.** Create the script as shown below. This time, we will press the OK button first.

## Confirmations



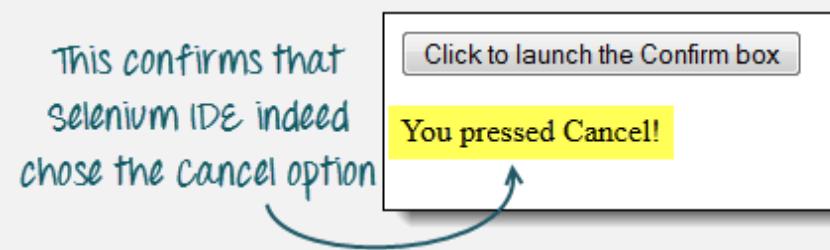
**Step 3.** Execute the script and notice that you do not see the actual confirmation, but the webpage was able to indicate which button Selenium IDE had pressed.

## Confirmations



This confirms that Selenium IDE indeed chose the OK option in the confirm dialog

**Step 4.** Replace the "chooseOkOnNextConfirmation" command with "chooseCancelOnNextConfirmation" and execute the script again.



# Multiple Windows

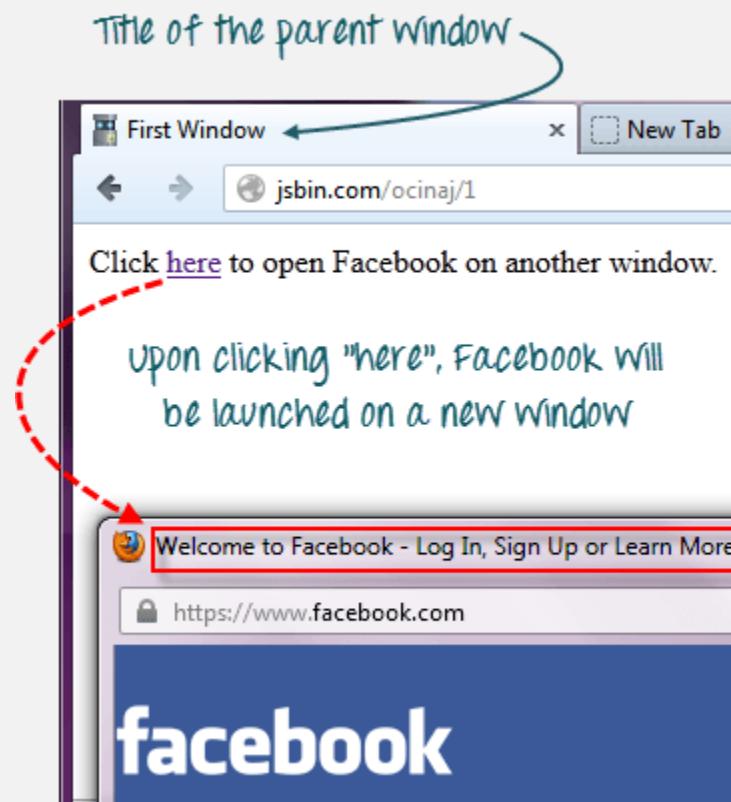
If you happen to click on a link that launches a separate window, **you must first instruct Selenium IDE to select that window first before you could access the elements within it.** To do this, you will **use the window's title as its locator.**

**We use the selectWindow command in switching between windows.**

We will use a link <http://jsbin.com/ocinaj/1> whose title is "First Window". The "here" hyperlink found in that page will open Facebook on a new window, after which we shall instruct Selenium IDE to do the following:

- Transfer control from the parent window to the newly launched Facebook window using the "selectWindow" command and its title as the locator
- Verify the title of the new window
- Select back the original window using the "selectWindow" command and "null" as its target.
- Verify the title of the currently selected window

## Multiple Windows



**Step 1.** Set the Base URL to <http://jsbin.com>.

**Step 2.** Create the script as shown below.

# Multiple Windows

Command	Target
open	/ocinaj/1
click	link=here
pause	5000
selectWindow	Welcome to Facebook - Log In, Sign Up or Learn More
verifyTitle	Welcome to Facebook - Log In, Sign Up or Learn More
selectWindow	null
verifyTitle	First Window

We need the "pause" command to wait for the newly launched window to load before we could access its title.

**Step 3.** Execute the script. Notice that the test case passed, meaning that we were able to switch between windows and verify their titles successfully.

Command	Target
open	/ocinaj/1
click	link=here
pause	5000
selectWindow	Welcome to Facebook - Log In, Sign Up or Learn More
verifyTitle	Welcome to Facebook - Log In, Sign Up or Learn More
selectWindow	null
verifyTitle	First Window

## Multiple Windows

Always remember that setting selectWindow's target to "null" will automatically select the parent window (in this case, the window where the element "link=here" is found)

# Summary

- The three most commonly used commands in verifying page elements are:
  - verifyElementPresent/verifyElementNotPresent
  - verifyTextPresent/verifyTextNotPresent
  - verifyElementPositionLeft/verifyElementPositionTop
- Wait commands are classified into two:
  - andWait commands - used when a page is expected to be loaded
  - waitFor commands - used when no new page is expected to be loaded
- The "store" command (and all its variants) are used to store variables in Selenium IDE
- The "echo" command is used to print a string value or a variable
- Variables are enclosed within a \${...} when being printed or used on elements
- Selenium IDE automatically presses the OK button when handling alerts
- When handling confirmation dialogs, you may instruct Selenium IDE which option to use:
  - chooseOkOnNextConfirmation/chooseOkOnNextConfirmationAndWait
  - chooseCancelOnNextConfirmation
- Window titles are used as locators when switching between browser windows.
- When using the "selectWindow" command, setting the Target to "null" will automatically direct Selenium IDE to select the parent window.

# Introduction to WebDriver & Comparison with Selenium RC

SELENIUM TRAINING

## Introduction to WebDriver & Comparison with Selenium RC

Now that you have learned to create simple tests in Selenium IDE, we shall now create more powerful scripts using an advanced tool called **WebDriver**.

### What is WebDriver?

WebDriver is a web automation framework that allows you to **execute your tests against different browsers**, not just Firefox (unlike Selenium IDE).



WebDriver also enables you to **use a programming language** in creating your test scripts (not possible in Selenium IDE).

- You can now use **conditional operations** like if-then-else or switch-case
- You can also perform **looping** like do-while.

Following programming languages are supported by WebDriver

- Java
- .Net
- [PHP](#)
- Python
- [Perl](#)
- Ruby

**You do not have to know all of them. You just need to be knowledgeable in one.** However, in this tutorial, we will be using Java with Eclipse as our IDE.

## WebDriver Vs Selenium RC

Before the advent of WebDriver in 2006, there was another, **automation tool called Selenium Remote Control**.

Both WebDriver and Selenium RC have following features:

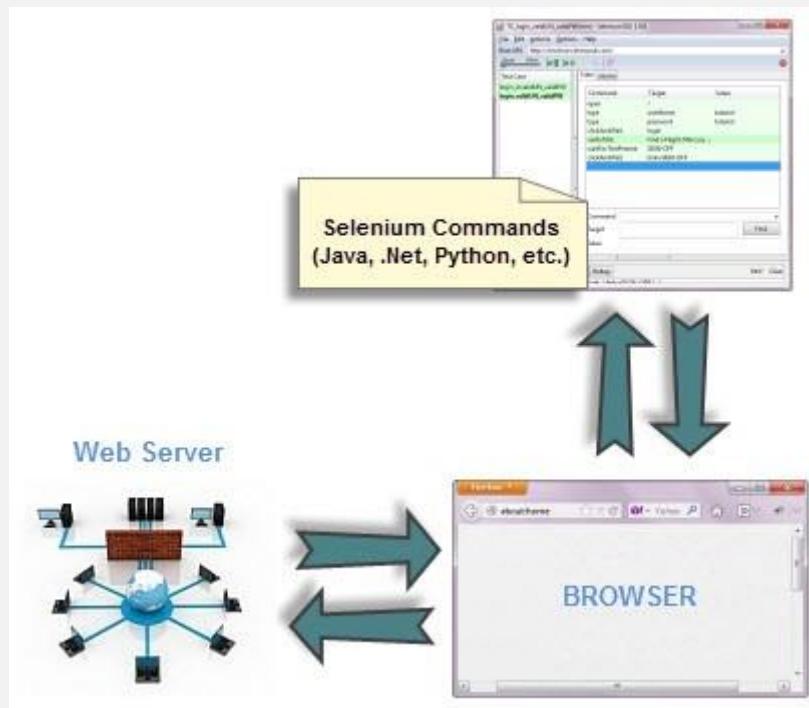
- They both allow you to **use a programming language** in designing your test scripts.
- They both allow you to **run your tests against different browsers**.

So how do they differ? Let us discuss the answers.

### Architecture

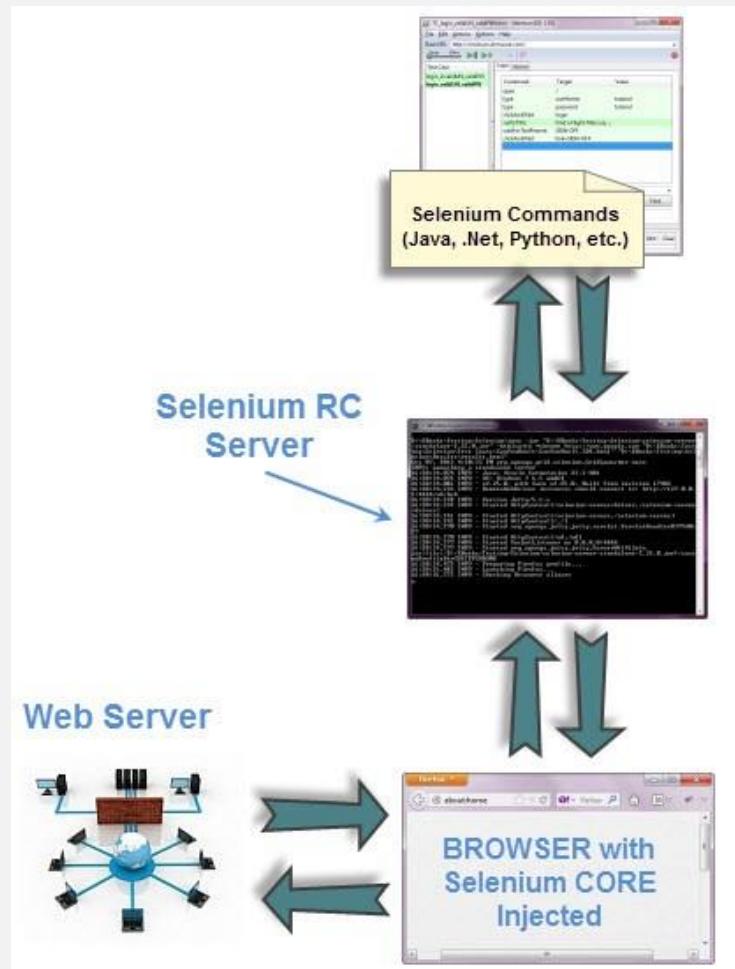
**WebDriver's architecture is simpler than Selenium RC's.**

- It controls the browser from the OS level
- All you need are your programming language's IDE (which contains your Selenium commands) and a browser.



Selenium RC's architecture is way more complicated.

- You first need to launch **a separate application called Selenium Remote Control (RC) Server** before you can start testing
- The Selenium RC Server **acts as a "middleman" between your Selenium commands and your browser**
- When you begin testing, Selenium RC Server "injects" a **Javascript program called Selenium Core** into the browser.
- Once injected, Selenium Core will start receiving instructions relayed by the RC Server from your test program.
- When the instructions are received, **Selenium Core will execute them as Javascript commands.**
- The browser will obey the instructions of Selenium Core, and will relay its response to the RC Server.
- The RC Server will receive the response of the browser and then display the results to you.
- RC Server will fetch the next instruction from your test script to repeat the whole cycle.



*Speed*



**WebDriver is faster than Selenium RC since it speaks directly to the browser uses the browser's own engine to control it.**



**Selenium RC is slower since it uses a Javascript program called Selenium Core.** This Selenium Core is the one that directly controls the browser, not you.

## Real-life Interaction



**WebDriver interacts with page elements in a more realistic way.** For example, if you have a disabled text box on a page you were testing, WebDriver really cannot enter any value in it just as how a real person cannot.



Selenium Core, just like other JavaScript codes, can access disabled elements. In the past, Selenium testers complain that Selenium Core was able to enter values to a disabled text box in their tests. Differences in API

## API



**Selenium RC's API is more matured but contains redundancies and often confusing commands.** For example, most of the time, testers are confused whether to use type or typeKeys; or whether to use click, mouseDown, or mouseDownAt. Worse, **different browsers interpret each of these commands in different ways too!**

**WebDriver's API is simpler than Selenium RC's.** It does not contain redundant and confusing commands.



## Browser Support

**WebDriver can support the headless HtmlUnit browser**

HtmlUnit is termed as "headless" because it is an invisible browser - it is GUI-less.

It is a very fast browser because no time is spent in waiting for page elements to load. This accelerates your test execution cycles. Since it is invisible to the user, it can only be controlled through automated means.

**Selenium RC cannot support the headless HtmlUnit browser.** It needs a real, visible browser to operate on.

## Limitations of WebDriver

### WebDriver Cannot Readily Support New Browsers

Remember that WebDriver operates on the OS level. Also, remember that different browsers communicate with the OS in different ways. If a new browser comes out, it may have a different process of communicating with the OS as compared to other browsers. So, **you have to give the WebDriver team quite some time to figure that new process out** before they can implement it on the next WebDriver release.

However, it is up to the WebDriver's team of developers to decide if they should support the new browser or not.

### Selenium RC Has Built-In Test Result Generator

**Selenium RC automatically generates an HTML file of test results.** The format of the report was pre-set by RC itself. Take a look at an example of this report below.

result: passed  
totalTime: 12  
numTestTotal: 2  
numTestPasses: 2  
numTestFailures: 0  
numCommandPasses: 2  
numCommandFailures: 0  
numCommandErrors: 0  
Selenium Version: 2.25  
Selenium Revision: .0

**Test Suite**

[login\\_invalidUN\\_validPW](#)  
[login\\_validUN\\_validPW](#)

**TC\_login\_invalidUN\_validPW.html**

login_invalidUN_validPW		
open	/	
type	userNmae	tutor
type	password	tutorial
clickAndWait	login	
verifyTitle	Sign-on: Mercury Tours	

**WebDriver has no built-in command that automatically generates a Test Results File.** You would have to rely on your IDE's output window, or design the report yourself using the capabilities of your programming language and store it as text, html, etc.



## Summary

- WebDriver is a tool for testing web applications **across different browsers** using different programming languages.
- You are now able to make powerful tests because WebDriver **allows you to use a programming language** of your choice in designing your tests.
- WebDriver is **faster than Selenium RC** because of its simpler architecture.
- WebDriver **directly talks to the browser** while Selenium RC needs the help of the RCServer in order to do so.
- WebDriver's API is more **concise** than Selenium RC's.
- WebDriver **can support HtmlUnit** while Selenium RC cannot.
- The only drawbacks of WebDriver are:
  - It **cannot readily support new browsers**, but Selenium RC can.
  - It **does not have a built-in command** for automatic generation of test results.

## Different Drivers

**HTMLUnit and Firefox are two browsers that WebDriver can directly automate** - meaning that no other separate component is needed to install or run while the test is being executed. For other browsers, a separate program is needed. That program is called as the **Driver Server**.

A driver server is different for each browser. For example, Internet Explorer has its own driver server which you cannot use on other browsers. Below is the list of driver servers and the corresponding browsers that use them.

Browser	Name of Driver Server	Remarks
HTMLUnit	(none)	WebDriver can drive HTMLUnit without the need of a driver server

Firefox	(none)	WebDriver can drive Firefox without the need of a driver server
Internet Explorer	Internet Explorer Driver Server	Available in 32 and 64-bit versions. Use the version that corresponds to the architecture of your IE
Chrome	ChromeDriver	Though its name is just "ChromeDriver", it is in fact a Driver Server, not just a driver. The current version can support versions higher than Chrome v.21
Opera	OperaDriver	Though its name is just "OperaDriver", it is in fact a Driver Server, not just a driver.
PhantomJS	GhostDriver	PhantomJS is another headless browser just like HTMLUnit.
Safari	SafariDriver	Though its name is just "SafariDriver", it is in fact a Driver Server, not just a driver.

# Creating Your First Script in WebDriver

## Creating your First Script in WebDriver

Using the Java class "myclass" that we created in the previous tutorial, let us try to create a WebDriver script that would:

1. fetch Mercury Tours' homepage
2. verify its title
3. print out the result of the comparison
4. Close it before ending the entire program.

### **WebDriver Code**

Below is the actual WebDriver code for the logic presented by the scenario above

## Code Challenge

[?](#)

```
1 package mypackage;
2
3 import org.openqa.selenium.WebDriver;
4 import org.openqa.selenium.firefox.FirefoxDriver;
5
6 public class myclass {
7
8     public static void main(String[] args) {
9         // declaration and instantiation of objects/variables
10        WebDriver driver = new FirefoxDriver();
11        String baseUrl = "http://newtours.demoaut.com"; String
12        expectedTitle = "Welcome: Mercury Tours"; String
13        actualTitle = "";
14
15        // launch Firefox and direct it to the Base URL
16
17        driver.get(baseUrl);
18
19    }
20}
```

```
// get the actual value of the title actualTitle =  
driver.getTitle();  
  
/*  
 * compare the actual title of the page witht the expected one and print  
 * the result as "Passed" or "Failed"  
 */  
  
if (actualTitle.contentEquals(expectedTitle)){  
    System.out.println("Test Passed!");  
  
} else {  
    System.out.println("Test Failed");  
}  
  
close Firefox driver.close();  
  
// exit the program explicitly  
System.exit(0);  
  
}
```



# Explaining the code

## Importing Packages

To get started, you need to import following two packages:

1. **org.openqa.selenium.\***- contains the WebDriver class needed to instantiate a new browser loaded with a specific driver
2. **org.openqa.selenium.firefox.FirefoxDriver** - contains the FirefoxDriver class needed to instantiate a Firefox-specific driver onto the browser instantiated by the WebDriver class

If your test needs more complicated actions such as accessing another class, taking browser screenshots, or manipulating external files, definitely you will need to import more packages.

## Instantiating objects and variables

Normally, this is how a driver object is instantiated.

```
WebDriver driver = new FirefoxDriver();
```

A FirefoxDriver class with no parameters means that the default Firefox profile will be launched by our Java program. The default Firefox profile is similar to launching Firefox in safe mode (no extensions are loaded).

For convenience, we saved the Base URL and the expected title as variables.

## Launching a Browser Session

```
driver.get(baseUrl);
```

WebDriver's **get ()** method is used to launch a new browser session and directs it to the URL that you specify as its parameter.

## Get the Actual Page Title

The WebDriver class has the `getTitle()` method that is always used to obtain the page title of the currently loaded page.

```
actualTitle = driver.getTitle();
```

## Compare the Expected and Actual Values

This portion of the code simply uses a basic Java if-else structure to compare the actual title with the expected one.

```
if (actualTitle.contentEquals(expectedTitle)) {  
    System.out.println("Test Passed!");  
} else {  
    System.out.println("Test Failed!");  
}
```

## Terminating a Browser Session

The "`close()`" method is used to close the browser window.

```
driver.close();
```

## Terminating the Entire Program

If you use this command without closing all browser windows first, your whole Java program will end while leaving the

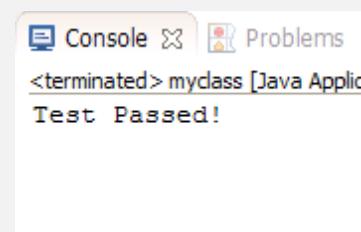
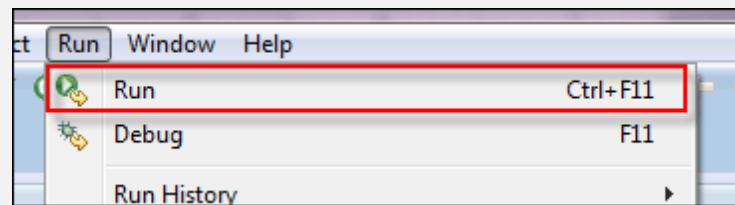
```
System.exit(0);
```

browser window open.

## Running the Test

There are two ways to execute code in Eclipse IDE.

1. On Eclipse's menu bar, click **Run > Run**.
2. Press **Ctrl+F11** to run the entire code.



If you did everything correctly, Eclipse would output "Test Passed!"

## Locating GUI Elements

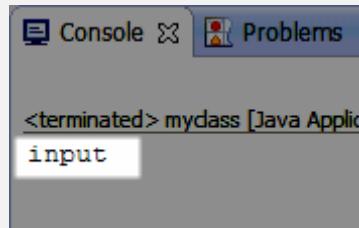
Locating elements in WebDriver is done by using the "**findElement(By.locator())**" method. The "locator" part of the code is same as any of the locators previously discussed in the Selenium IDE chapters of these tutorials. Infact, it is recommended that you locate GUI elements using IDE and once successfully identified export the code to webdriver.

Here is a sample code that locates an element by its id. Facebook is used as the Base URL.

```
?  
1  
2 package mypackage;  
3 import org.openqa.selenium.By;  
4 import org.openqa.selenium.WebDriver;  
5 import org.openqa.selenium.firefox.FirefoxDriver;  
6 public class myclass {  
7     public static void main(String[] args) {  
8         WebDriver driver = new FirefoxDriver();  
9         String baseUrl = "http://www.facebook.com";  
10        String tagName = "";  
11  
12        driver.get(baseUrl);  
13        tagName = driver.findElement(By.id("email")).getTagName();  
14        System.out.println(tagName);  
15        driver.close();  
16        System.exit(0);  
17    }  
18}
```

We used the **getTagName()** method to extract the tag name of that particular element whose id is "email". When run, this code should be able to correctly identify the tag name "input" and will print it out on Eclipse's Console window.

# Locating GUI Elements



Summary for locating elements

Variation	Description	Sample
By.className	finds elements based on the value of the "class" attribute	findElement(By.className("someClassName"))
By.cssSelector	finds elements based on the driver's underlying CSS Selector engine	findElement(By.cssSelector("input#email"))
By.id	locates elements by the value of	findElement(By.id("someId"))

	their "id" attribute	
<b>By.linkText</b>	finds a link element by the exact text it displays	<code>findElement(By.linkText("REGISTRATION"))</code>
<b>By.name</b>	locates elements by the value of the "name" attribute	<code>findElement(By.name("someName"))</code>
<b>By.partialLinkText</b>	locates elements that contain the given link text	<code>findElement(By.partialLinkText("REG"))</code>
<b>By.tagName</b>	locates elements by their tag name	<code>findElement(By.tagName("div"))</code>
<b>By.xpath</b>	locates elements via XPath	<code>findElement(By.xpath("//html/body/div/table/tbody/tr/td[2]/table/tbody/tr[4]/td/table/tbody/tr/td[2]/table/tbody/tr[2]/td[3]/form/table/tbody/tr[5]"))</code>

## Locating GUI Elements

### Note on Using findElement(By.cssSelector())

**By.cssSelector()** does not support the "contains" feature. Consider the Selenium IDE code below -

-- In selenium IDE, this step passed

Command	Target	Value
open		
storeText	css=font:contains("Password:")	var
echo		\${var}

Log Reference UI-Element Rollup

```
[info] Executing: |open| |
[info] Executing: |storeText|
css=font:contains("Password:") | var |
[info] Executing: |echo| ${var} |
[info] echo: Password:
```

The inner text was successfully printed

In Selenium IDE above, the entire test passed. However in the WebDriver script below, the same test generated an error because WebDriver does not support the "contains" keyword when used in the By.cssSelector() method.

## Locating GUI Elements

```
1 package mypackage;  
2  
3 import org.openqa.selenium.*;  
4 import org.openqa.selenium.firefox.FirefoxDriver;  
5  
6 public class myclass {  
7  
8     public static void main(String[] args) {  
9         WebDriver driver = new FirefoxDriver();  
10        String baseUrl = "http://newtours.demoaut.com";  
11        String var = "";  
12  
13        driver.get(baseUrl);  
14        var = driver.findElement(By.cssSelector("font:contains('Password:')")).getText();  
15        System.out.println(var);  
16  
17        driver.close();  
18        System.exit(0);  
19    }  
20}  
21
```

this line caused an error because the 'contains' keyword is not supported by By.cssselector() in WebDriver

Eclipse IDE reports that error was caused by line 14, the line where By.cssselector() was used

```
at mypackage.myclass.main(myclass.java:14)  
Caused by: org.openqa.selenium.remoteErrorHandler$UnknownServerException: An invalid or illegal string was specified
```



# Common Commands

## Instantiating Web Elements

Instead of using the long "driver.findElement(By.locator())" syntax every time you will access a particular element, we can instantiate a WebElement object for it. The WebElement class is contained in the "org.openqa.selenium.\*" package.

```
WebElement myElement = driver.findElement(By.id("username"));
myElement.sendKeys("tutorial");
```

## Clicking on an Element

Clicking is perhaps the most common way of interacting with web elements. **The click() method is used to simulate the clicking of any element.** The following example shows how click() was used to click on Mercury Tours' "Sign-In" button.

```
driver.findElement(By.name("login")).click();
```

Following things must be noted when using the click() method.

- **It does not take any parameter/argument.**
- The method **automatically waits for a new page to load** if applicable.
- The element to be clicked-on, **must be visible** (height and width must not be equal to zero).

## Get Commands

Get commands fetch various important information about the page/element. Here are some important "get" commands you must be familiar with.

## Common Commands

<b>get()</b> <i>Sample usage:</i>	<ul style="list-style-type: none"><li>• It automatically opens a new browser window and fetches the page that you specify inside its parentheses.</li><li>• It is the counterpart of Selenium IDE's "open" command.</li><li>• The parameter must be a <b>String</b> object.</li></ul>
<b>getTitle()</b> <i>Sample usage:</i>	<ul style="list-style-type: none"><li>• Needs no parameters</li><li>• Fetches the title of the current page</li><li>• Leading and trailing white spaces are trimmed</li><li>• Returns a null string if the page has no title</li></ul>
<b>getPageSource()</b> <i>Sample usage:</i>	<ul style="list-style-type: none"><li>• Needs no parameters</li><li>• Returns the <b>source code of the page</b> as a String value</li></ul>
<b>getCurrentUrl()</b> <i>Sample usage:</i>	<ul style="list-style-type: none"><li>• Needs no parameters</li><li>• Fetches the string representing the <b>current URL</b> that the browser is looking at</li></ul>
<b>getText()</b> <i>Sample usage:</i>	<ul style="list-style-type: none"><li>• Fetches the <b>inner text</b> of the element that you specify</li></ul>

# Common Commands

## Navigate commands

These commands allow you to refresh, go-into and switch back and forth between different web pages.

<b>navigate().to()</b> <i>Sample usage:</i>	<ul style="list-style-type: none"><li>• It automatically <b>opens a new browser window and fetches the page</b> that you specify inside its parentheses.</li><li>• <b>It does exactly the same thing as the get() method.</b></li></ul>
<b>navigate().refresh()</b> <i>Sample usage:</i>	<ul style="list-style-type: none"><li>• Needs no parameters.</li><li>• <b>It refreshes</b> the current page.</li></ul>
<b>navigate().back()</b> <i>Sample usage:</i>	<ul style="list-style-type: none"><li>• Needs no parameters</li><li>• Takes you <b>back by one page</b> on the browser's history.</li></ul>
<b>navigate().forward()</b> <i>Sample usage:</i>	<ul style="list-style-type: none"><li>• Needs no parameters</li><li>• Takes you <b>forward by one page</b> on the browser's history.</li></ul>

## Common Commands

### Closing and Quitting Browser Windows

<b>close()</b> <i>Sample usage:</i>	<ul style="list-style-type: none"><li>• Needs no parameters</li><li>• <b>It closes only the browser window that WebDriver is currently controlling.</b></li></ul>
<b>quit()</b> <i>Sample usage:</i>	<ul style="list-style-type: none"><li>• Needs no parameters</li><li>• <b>It closes all windows that WebDriver has opened.</b></li></ul>

## Common Commands



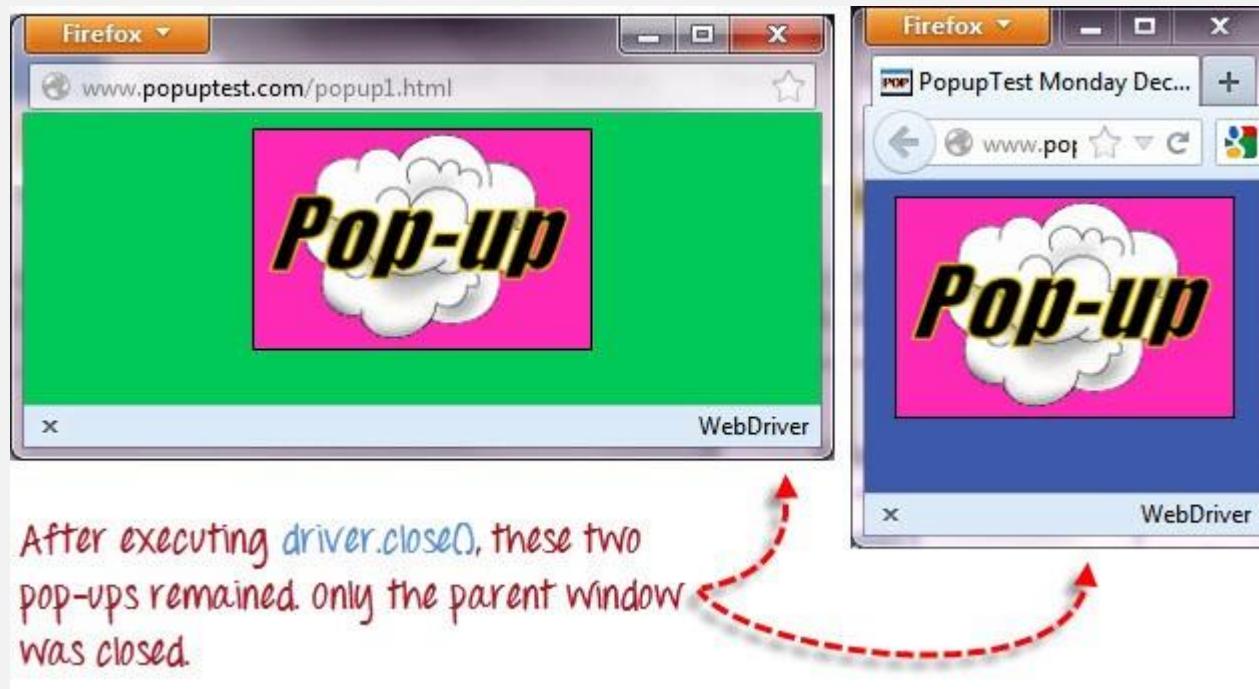
To clearly illustrate the difference between `close()` and `quit()`, try to execute the code below. It uses a webpage that automatically pops up a window upon page load and opens up another after exiting.

### Using `close()`

```
public static void main(String[] args) {  
    WebDriver driver = new FirefoxDriver();  
    driver.get("http://www.popuptest.com/popuptest2.html");  
    driver.close();  
}
```

Notice that only the parent browser window was closed and not the two pop-up windows.

## Common Commands



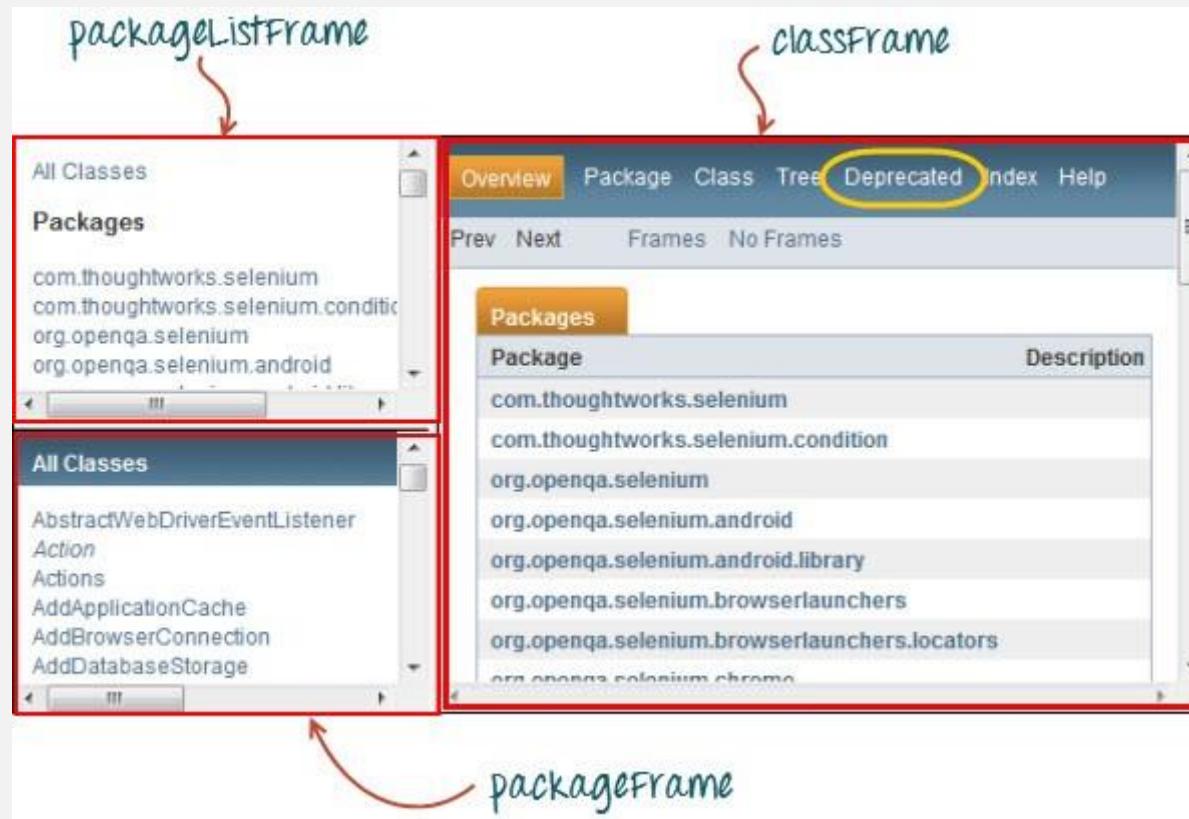
But if you use `quit()`, all windows will be closed - not just the parent one. Try running the code below and you will notice that the two pop-ups above will automatically be closed as well.

## Common Commands

```
?  
1 package mypackage;  
2  
3 import org.openqa.selenium.WebDriver;  
4 import org.openqa.selenium.firefox.FirefoxDriver;  
5  
6 public class myclass  
7  
8     public static void main(String[] args) {  
9         WebDriver driver = new FirefoxDriver();  
10  
11         driver.get("http://www.popuptest.com/popuptest2.html");  
12         driver.quit(); // using QUIT all windows will close  
13     }  
14}  
15
```

## Switching Between Frames

To access GUI elements in a Frame, we should first direct WebDriver to focus on the frame or pop-up window first before we can access elements within them. Let us take, for example, the web page <https://docs.oracle.com/javase/8/docs/api/>



This page has 3 frames whose "name" attributes are indicated above. We wish to access the "Deprecated" link encircled above in yellow. In order to do that, we must first instruct WebDriver to switch to the "classFrame" frame using the "**switchTo().frame()**" method. We will use the name attribute of the frame as the parameter for the "frame()" part.

## Switching Between Frames

```
1
2 package mypackage;
3
4 import org.openqa.selenium.By;
5 import org.openqa.selenium.WebDriver;
6 import org.openqa.selenium.firefox.FirefoxDriver;
7
8 public class myclass {
9
10    public static void main(String[] args) {
11        WebDriver driver = new FirefoxDriver();
12
13        driver.get("https://docs.oracle.com/javase/8/docs/api/");
14        driver.switchTo().frame("classFrame");
15        driver.findElement(By.linkText("Deprecated")).click();
16
17    }
18}
```

After executing this code, you will see that the "classFrame" frame is taken to the "Deprecated API" page, meaning that our code was successfully able to access the "Deprecated" link.

## Switching Between Pop-up Windows

WebDriver allows pop-up windows like alerts to be displayed, unlike in Selenium IDE. To access the elements within the alert (such as the message it contains), we must use the "`switchTo().alert()`" method. In the code below, we will use this method to access the alert box and then retrieve its message using the "`getText()`" method, and then automatically close the alert box using the "`switchTo().alert().accept()`" method.

First, head over to <http://jsbin.com/usidix/1> and manually click the "Go!" button there and see for yourself the message text.



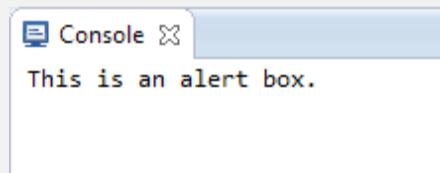
Let's see the WebDriver code to do this!

## Switching Between Pop-up Windows

```
?  
1  
2 package mypackage;  
3  
4 import org.openqa.selenium.By;  
5 import org.openqa.selenium.WebDriver;  
6 import org.openqa.selenium.firefox.FirefoxDriver;  
7  
8 public class myclass {  
9     public static void main(String[] args) {  
10         WebDriver driver = new FirefoxDriver();  
11         String alertMessage = "";  
12  
13         driver.get("http://jsbin.com/usidix/1");  
14         driver.findElement(By.cssSelector("input[value='Go!']")).click();  
15         alertMessage = driver.switchTo().alert().getText();  
16         driver.switchTo().alert().accept();  
17  
18         System.out.println(alertMessage);  
19         driver.quit();  
20     }  
21 }  
22
```

## Switching Between Pop-up Windows

On the Eclipse console, notice that the printed alert message is:



# Waits

There are two kinds of waits.

1. Implicit wait - used to set the default waiting time throughout the program
2. Explicit wait - used to set the waiting time for a particular instance only

## Implicit Wait

- It is simpler to code than Explicit Waits.
- It is usually declared in the instantiation part of the code.
- You will only need one additional package to import.

To start using an implicit wait, you would have to import this package into your code.

```
import java.util.concurrent.TimeUnit;
```

Then on the instantiation part of your code, add this.

```
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```



this means that you are setting 10 seconds as your default wait time. You can change "10" and "SECONDS" to any number and time unit you want.

## Explicit Wait

**Explicit waits are done using the WebDriverWait and ExpectedCondition classes.** For the following example, we shall wait up to 10 seconds for an element whose id is "username" to become visible before proceeding to the next command. Here are the steps.

# Waits

## Step 1

Import these two packages:

```
import org.openqa.selenium.support.ui.ExpectedConditions;  
import org.openqa.selenium.support.ui.WebDriverWait;
```

## Step 2

Declare a WebDriverWait variable. In this example, we will use "myWaitVar" as the name of the variable.

The diagram shows the Java code for creating a WebDriverWait variable. A red dashed arrow points from the text "WebDriver instance that will use the Explicit wait" to the line "WebDriver driver = new FirefoxDriver();". A blue curved arrow points from the text "number of seconds to wait" to the value "10" in the line "WebDriverWait myWaitVar = new WebDriverWait(driver, 10);".

```
WebDriver instance that  
will use the Explicit wait  
WebDriver driver = new FirefoxDriver();  
WebDriverWait myWaitVar = new WebDriverWait(driver, 10);  
number of seconds to wait
```

## Step 3

Use myWaitVar with ExpectedConditions on portions where you need the explicit wait to occur. In this case, we will use explicit wait on the "username" (Mercury Tours HomePage) input before we type the text "tutorial" onto it.

```
myWaitVar.until(ExpectedConditions.visibilityOfElementLocated(By.id("username")));
driver.findElement(By.id("username")).sendKeys("tutorial");
```

## Conditions

The following methods are used in conditional and looping operations --

- **isEnabled()** is used when you want to verify whether a certain element is enabled or not before executing a command.

for convenience, we saved the element with id="username" as an instance of the WebElement class. The WebElement class is contained in the package `org.openqa.selenium.*`

```
WebElement txtbox_username = driver.findElement(By.id("username"));
if(txtbox_username.isEnabled()){
    txtbox_username.sendKeys("tutorial");
}
```

- **isDisplayed()** is used when you want to verify whether a certain element is displayed or not before executing a command.

```
do{
    //do something here
}while (driver.findElement(By.id("username")).isDisplayed());
```

- **isSelected()** is used when you want to verify whether a certain **check box, radio button, or option in a drop-down box** is selected. It does not work on other elements.

```
//"one-way" and "two-way" are radio buttons
if (driver.findElement(By.id("one-way")).isSelected()) {
    driver.findElement(By.id("two-way")).click();
}
```

## Using ExpectedConditions

The ExpectedConditions class offers a wider set of conditions that you can use in conjunction with WebDriverWait's until() method. Below are some of the most common ExpectedConditions methods.

- **alertIsPresent()** - waits until an alert box is displayed.

```
if (myWaitVar.until(ExpectedConditions.alertIsPresent()) != null) {  
    System.out.println("alert is present!");  
}
```

- **elementToBeClickable()** - waits until an element is visible and, at the same time, enabled. The sample code below will wait until the element with id="username" to become visible and enabled first before assigning that element as a WebElement variable named "txtUserName".

```
WebElement txtUserName = myWaitVar.until(ExpectedConditions  
    .elementToBeClickable(By.id("username")));
```

- **frameToBeAvailableAndSwitchToIt()** - waits until the given frame is already available, and then automatically switches to it.

This will automatically switch to the "viewIFRAME" frame once it becomes available.

```
myWaitVar.until(ExpectedConditions  
    .frameToBeAvailableAndSwitchToIt("viewIFRAME"));
```

## Catching Exceptions

When using isEnabled(), isDisplayed(), and isSelected(), WebDriver assumes that the element already exists on the page. Otherwise, it will throw a **NoSuchElementException**. To avoid this, we should use a try-catch block so that the program will not be interrupted.

```
?  
1 WebElement txtbox_username = driver.findElement(By.id("username"));  
2 try{  
3     if(txtbox_username.isEnabled()){  
4         txtbox_username.sendKeys("tutorial");  
5     }  
6 }  
7 catch(NoSuchElementException nsee){  
8     System.out.println(nsee.toString());  
9 }  
10
```

If you use explicit waits, the type of exception that you should catch is the "TimeoutException".

```
WebDriverWait myWaitVar = new WebDriverWait(driver, 3);  
try {  
    myWaitVar.until(ExpectedConditions.visibilityOfElementLocated(By  
        .id("username")));  
    driver.findElement(By.id("username")).sendKeys("tutorial");  
} catch (TimeoutException toe) {  
    System.out.println(toe.toString());  
}
```

## Summary

- To start using the WebDriver API, you must import at least these two packages.
  - `org.openqa.selenium.*`
  - `org.openqa.selenium.firefox.FirefoxDriver`
- The `get()` method is the equivalent of Selenium IDE's "open" command.
- Locating elements in WebDriver is done by using the `findElementBy()` method.
- The following are the available options for locating elements in WebDriver:
  - `By.className`
  - `By.cssSelector`
  - `By.id`
  - `By.linkText`
  - `By.name`
  - `By.partialLinkText`
  - `By.tagName`
  - `By.xpath`
- The `By.cssSelector()` does not support the "contains" feature.
- You can instantiate an element using the `WebElement` class.
- Clicking on an element is done by using the `click()` method.
- WebDriver provides these useful **get commands**:
  - `get()`
  - `getTitle()`
  - `getPageSource()`
  - `getCurrentUrl()`
  - `getText()`
- WebDriver provides these useful **navigation commands**
  - `navigate().forward()`
  - `navigate().back()`
  - `navigate().to()`
  - `navigate().refresh()`
- The `close()` and `quit()` methods are used to close browser windows. `Close()` is used to close a single window; while `quit()` is used to close all windows associated to the parent window that the WebDriver object was controlling.

## Summary

- The **switchTo().frame()** and **switchTo().alert()** methods are used to direct WebDriver's focus onto a frame or alert, respectively.
- **Implicit waits** are used to set the waiting time throughout the program, while **explicit waits** are used only on specific portions.
- You can use the **isEnabled()**, **isDisplayed()**,**isSelected()**, and a combination of **WebDriverWait** and **ExpectedConditions** Methods when verifying the state of an element. However, they do not verify if the element exists.
- When **isEnabled()**, **isDisplayed()**,or **isSelected()** was called while the element was not existing, WebDriver will throw a **NoSuchElementException**.
- When **WebDriverWait** and **ExpectedConditions** methods were called while the element was not existing, WebDriver would throw a **TimeoutException**.

# Accessing Forms Using Selenium WebDriver

## Accessing Forms using Selenium WebDriver

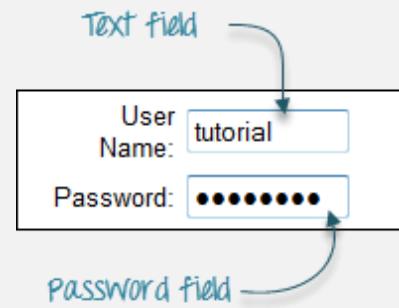
In this tutorial, we will learn how to access forms and its elements using WebDriver

# Accessing Form Elements

## Input Box

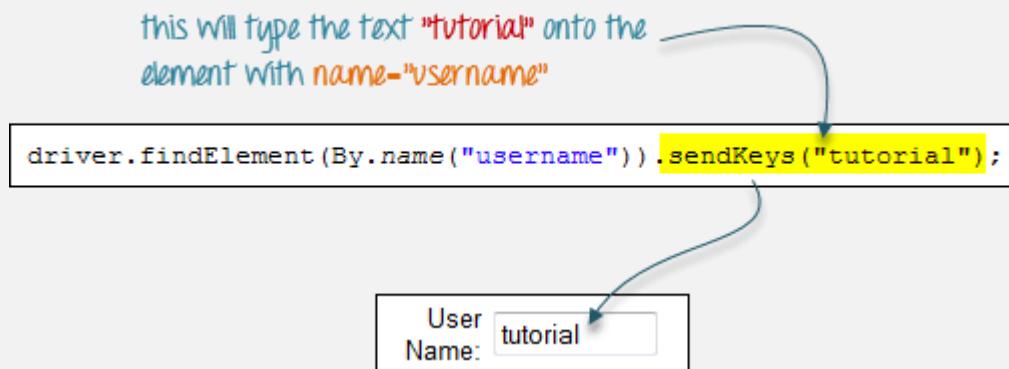
Input boxes refer to either of these two types:

1. **Text Fields**- text boxes that accept typed values and show them as they are.
2. **Password Fields**- text boxes that accept typed values but mask them as a series of special characters (commonly dots and asterisks) to avoid sensitive values to be displayed.



# Entering Values in Input Boxes

The `sendKeys()` method is used to enter values into input boxes.



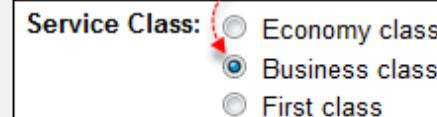
## Deleting Values in Input Boxes

The `clear()` method is used to delete the text in an input box. **This method does not need any parameter.** The code snippet below will clear out the text "tutorial" in the User Name text box.

```
driver.findElement(By.name("userName")).clear();
```

## Radio Button

```
driver.findElement(By.cssSelector("input[value='Business']")).click();
```



Toggling a radio button on is done using the `click()` method.

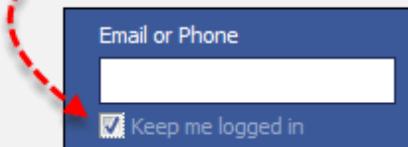
# Check Box

Toggling a check box on/off is also done using the `click()` method.

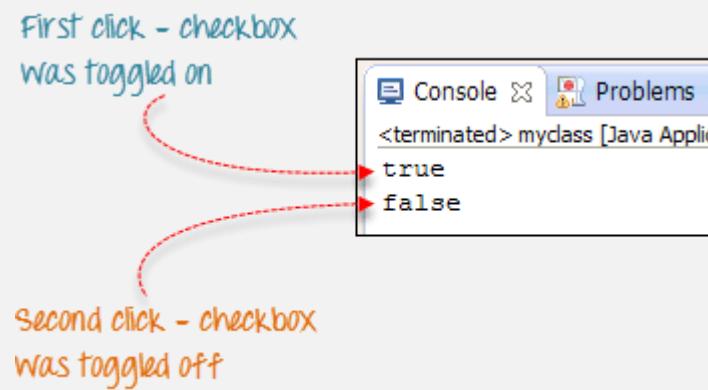
The code below will click on Facebook's "Keep me logged in" check box twice and then output the result as TRUE when it is toggled on, and FALSE if it is toggled off.

```
public static void main(String[] args) {
    WebDriver driver = new FirefoxDriver();
    String baseURL = "http://www.facebook.com";

    driver.get(baseURL);
    WebElement chkFBPersist = driver.findElement(By.id("persist_box"));
    for(int i=0; i<2; i++){
        chkFBPersist.click();
        System.out.println(chkFBPersist.isSelected());
    }
    driver.quit();
}
```



# Check Box



# Links

Links also are accessed by using the `click()` method.

Consider the below link found in Mercury Tours' homepage.

You can access this link using `linkText()` or `partialLinkText()` together with `click()`. Either of the two lines below will be able to access the "Register here" link shown above.

```
driver.findElement(By.linkText("Register here")).click();
```

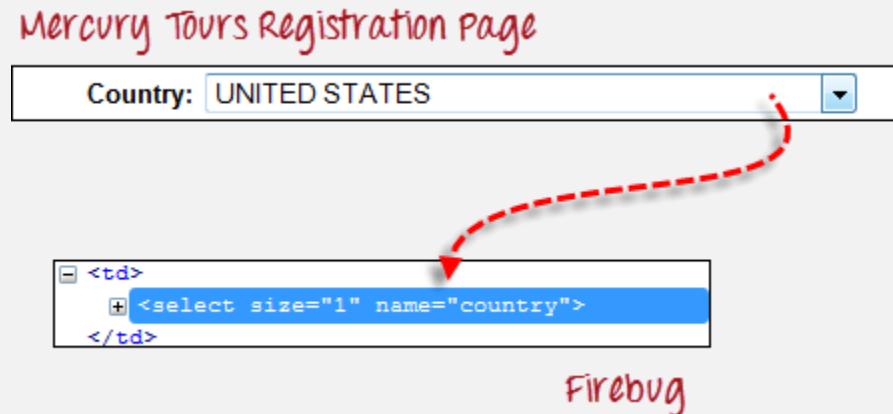
```
driver.findElement(By.partialLinkText("here")).click();
```

# Drop-Down Box

Before we can control drop-down boxes, we must do following two things:

1. Import the package `org.openqa.selenium.support.ui.Select`
2. Instantiate the drop-down box as a "Select" object in WebDriver

As an example, go to Mercury Tours' Registration page (<http://newtours.demoaut.com/mercuryregister.php>) and notice the "Country" drop-down box there.



Step 1

Import the "Select" package.

```
import org.openqa.selenium.support.ui.Select;
```

## Check Box

### Step 2

Declare the drop-down element as an instance of the Select class. In the example below, we named this instance as "drpCountry".

```
Select drpCountry = new Select(driver.findElement(By.name("country")));
```

### Step 3

We can now start controlling "drpCountry" by using any of the available Select methods. The sample code below will select the option "ANTARCTICA".

```
drpCountry.selectByVisibleText("ANTARCTICA");
```

## Selecting Items in a Multiple SELECT element

page source

```
<select id="fruits" multiple="">
  <option value="banana">Banana</option>
  <option value="apple">Apple</option>
  <option value="orange">Orange</option>
  <option value="grape">Grape</option>
</select>
```

HTML page



We can also use the `selectByVisibleText()` method in selecting multiple options in a multi SELECT element. As an example, we will take <http://jsbin.com/osebed/2> as the base URL. It contains a drop-down box that allows multiple selections at a time

## Selecting Items in a Multiple SELECT element

The code below will select the first two options using the selectByVisibleText() method.

```
public static void main(String[] args) {  
    WebDriver driver = new FirefoxDriver();  
  
    driver.get("http://jsbin.com/osebed/2");  
    Select fruits = new Select(driver.findElement(By.id("fruits")));  
    fruits.selectByVisibleText("Banana");  
    fruits.selectByIndex(1);  
}
```



# Select Methods

The following are the most common methods used on drop-down elements.

Method	Description
<b>selectByVisibleText()</b> and <b>deselectByVisibleText()</b>  <i>Example:</i>  <code>drpCountry.selectByVisibleText ("ANTARCTICA");</code>	Selects/deselects the option that displays the text matching the parameter.  <b>Parameter:</b> The exactly displayed text of a particular option
<b>selectByValue()</b> and <b>deselectByValue()</b>  <i>Example:</i>  <code>drpCountry.selectByValue ("234");</code>	Selects/deselects the option whose "value" attribute matches the specified parameter.  <b>Parameter:</b> value of the "value" attribute  Remember that not all drop-down options have the same text and "value", like in the example below.  <code>&lt;option value="10"&gt;ANGUILLA &lt;/option&gt; &lt;option value="234"&gt;ANTARCTICA &lt;/option&gt; &lt;option value="1"&gt;ANTIGUA AND BARBUDA &lt;/option&gt;</code>
<b>selectByIndex()</b> and <b>deselectByIndex()</b>	Selects/deselects the option at the given index.  <b>Parameter:</b> the index of the option to be selected.

<p><i>Example:</i></p> <pre>drpCountry.selectByIndex(0);</pre>	
<p><b>isMultiple()</b></p> <p><i>Example:</i></p> <pre>if (drpCountry.isMultiple()) {     //do something here }</pre>	<p>Returns TRUE if the drop-down element allows multiple selections at a time; FALSE if otherwise.</p> <p><b>Needs parameters needed</b></p>
<p><b>deselectAll()</b></p> <p><i>Example:</i></p> <pre>drpCountry.deselectAll();</pre>	<p>Clears all selected entries. This is only valid when the drop-down element supports multiple selections.</p> <p><b>No parameters needed</b></p>

# Submitting a Form

The `submit()` method is used to submit a form. This is an alternative to clicking the form's submit button. You can use `submit()` on any element within the form, not just on the submit button itself.

**When `submit()` is used, WebDriver will look up the DOM to know which form the element belongs to, and then trigger its submit function.**

# Summary

- The table below summarizes the commands to access each type of element discussed above.

Element	Command	Description
Input Box	<i>sendKeys()</i>	used to enter values onto text boxes
	<i>clear()</i>	used to clear text boxes of its current value
Check Box, Radio Button,	<i>click()</i>	used to toggle the element on/off
Links	<i>click()</i>	used to click on the link and wait for page load to complete before proceeding to the next command.
Drop-Down Box	<i>selectByVisibleText()/ deselectByVisibleText()</i>	selects/deselects an option by its displayed text
	<i>selectByValue()/ deselectByValue()</i>	selects/deselects an option by the value of its "value" attribute
	<i>selectByIndex()/ deselectByIndex()</i>	selects/deselects an option by its index

	<i>deselectByIndex()</i>	
	<i>isMultiple()</i>	returns TRUE if the drop-down element allows multiple selection at a time; FALSE if otherwise
	<i>deselectAll()</i>	deselects all previously selected options
<b>Submit Button</b>	<i>submit()</i>	

- WebDriver allows selection of more than one option in a multiple SELECT element.
- To control drop-down boxes, you must first import the org.openqa.selenium.support.ui.Select package and then create a Select instance.
- You can use the submit() method on any element within the form. WebDriver will automatically trigger the submit function of the form where that element belongs to.

# Accessing Links & Tables Using Selenium WebDriver

## Accessing Links & Tables using Selenium WebDriver

In this tutorial, we are going to learn about accessing links & Tables using WebDriver

# Accessing Links

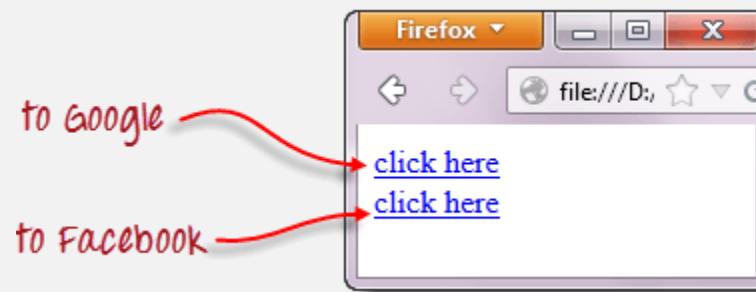
## Links Matching a Criterion

Links can be accessed using an exact or partial match of their link text. The examples below provide scenarios where multiple matches would exist, and would explain how WebDriver would deal with them.

## Exact Match

Accessing links using their exact link text is done through the **By.linkText()** method. However, if there are two links that have the very same link text, this method will only access the first one. Consider the HTML code below

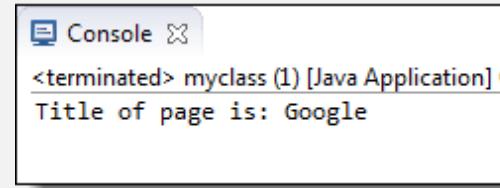
```
<html>
  <head>
    <title>Sample</title>
  </head>
  <body>
    <a href="http://www.google.com">click here</a>
    <br>
    <a href="http://www.fb.com">click here</a>
  </body>
</html>
```



## Exact Match

When you try to run the WebDriver code below, you will be accessing the first "click here" link

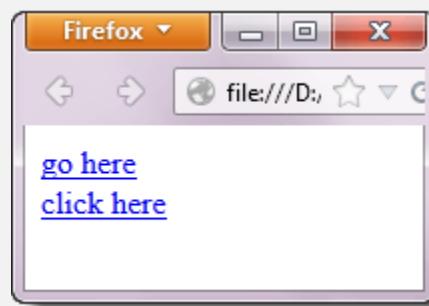
As a result, you will automatically be taken to Google.



## Partial Match

Accessing links using a portion of their link text is done using the `By.partialLinkText()` method. If you specify a partial link text that has multiple matches, only the first match will be accessed. Consider the HTML code below.

```
<html>
  <head>
    <title>Partial Match</title>
  </head>
  <body>
    <a href="http://www.google.com">go here</a>
    <br>
    <a href="http://www.fb.com">click here</a>
  </body>
</html>
```

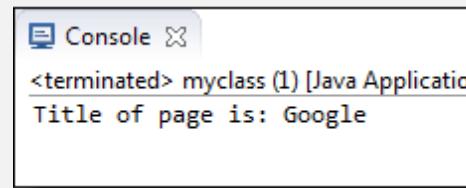


## Partial Match

When you execute the WebDriver code below, you will still be taken to Google.

```
public static void main(String[] args) {
    String baseUrl = "file:///D:/partial_match.html";
    WebDriver driver = new FirefoxDriver();

    driver.get(baseUrl);
    driver.findElement(By.partialLinkText("here")).click();
    System.out.println("Title of page is: " + driver.getTitle());
    driver.quit();
}
```



## Case-sensitivity

The parameters for `By.linkText()` and `By.partialLinkText()` are both case-sensitive, meaning that capitalization matters. For example, in Mercury Tours' homepage, there are two links that contain the text "egis" - one is the "REGISTER" link found at the top menu, and the other is the "Register here" link found at the lower right portion of the page.

The link at the top menu



The link at the lower right portion of the page



Though both links contain the character sequence "egis", the "`By.partialLinkText()`" method will access these two links separately depending on the capitalization of the characters. See the sample code below.

## Case-sensitivity

```
public static void main(String[] args) {
    String baseUrl = "http://newtours.demoaut.com/";
    WebDriver driver = new FirefoxDriver();

    driver.get(baseUrl);

    String theLinkText = driver.findElement(By
        .partialLinkText("egis"))
        .getText();
    System.out.println(theLinkText);
    theLinkText = driver.findElement(By
        .partialLinkText("EGIS"))
        .getText();
    System.out.println(theLinkText);

    driver.quit();
}
```

Console X  
<terminated> myclass (1)  
Register here  
REGISTER



# All Links

One of the common procedures in web **testing** is to test if all the links present within the page are working. This can be conveniently done using a combination of the **Java for-each loop** and the **By.tagName("a")** method. The WebDriver code below checks each link from the Mercury Tours homepage to determine those that are working and those that are still under construction.

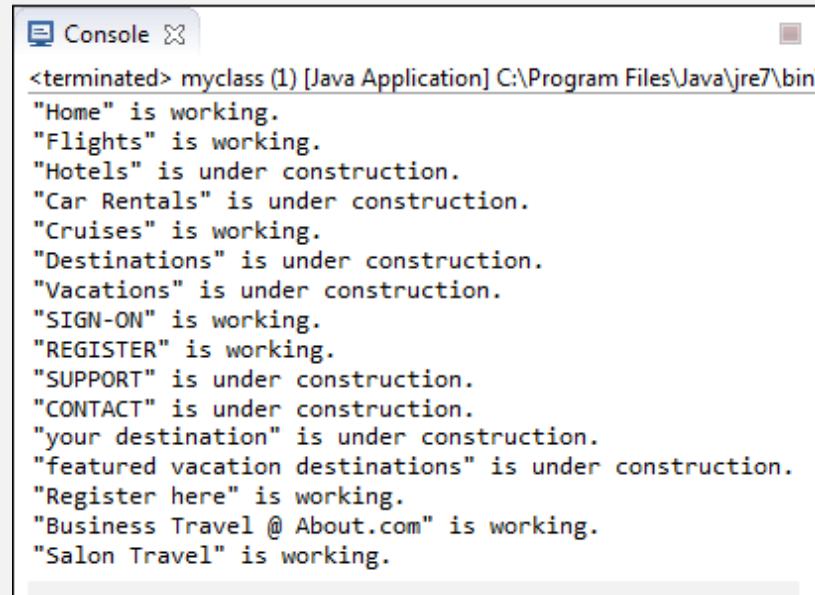
```
?  
1 package practice_webdriver;  
2  
3 import java.util.List;  
4  
5  
6  
7 import java.util.concurrent.TimeUnit;  
8 import org.openqa.selenium.*;  
9 import org.openqa.selenium.firefox.FirefoxDriver;  
10 import org.openqa.selenium.support.ui.ExpectedConditions;  
11 import org.openqa.selenium.support.ui.WebDriverWait;  
12  
13 public class AllLinks {  
14  
15     public static void main(String[] args) {  
16         String baseUrl = "http://newtours.demoaut.com/";  
17         WebDriver driver = new FirefoxDriver();  
18         String underConsTitle = "Under Construction: Mercury Tours";  
19         driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
```

```
21     driver.get(baseUrl);
22
23     List<WebElement> linkElements = driver.findElements(By.tagName("a"));
24     String[] linkTexts = new String[linkElements.size()];
25     int i = 0;
26
27     //extract the link texts of each link element
28     for (WebElement e : linkElements) {
29         linkTexts[i] = e.getText();
30         i++;
31     }
32
33     //test each link
34     for (String t : linkTexts) {
35         driver.findElement(By.linkText(t)).click();
36         if (driver.getTitle().equals(underConstTitle)) {
37             System.out.println("'" + t + "'"
38                     + " is under construction.");
39         } else {
40             System.out.println("'" + t + "'"
41                     + " is working.");
42         }
43     }
44     driver.quit();
45 }
46 }
```



## All Links

The output should be similar to the one indicated below.



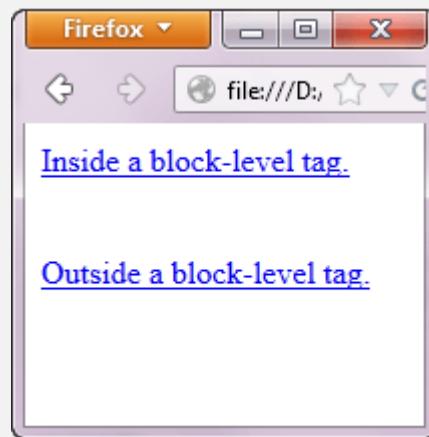
```
Console <terminated> myclass (1) [Java Application] C:\Program Files\Java\jre7\bin\j
"Home" is working.
"Flights" is working.
"Hotels" is under construction.
"Car Rentals" is under construction.
"Cruises" is working.
"Destinations" is under construction.
"Vacations" is under construction.
"SIGN-ON" is working.
"REGISTER" is working.
"SUPPORT" is under construction.
"CONTACT" is under construction.
"your destination" is under construction.
"featured vacation destinations" is under construction.
"Register here" is working.
"Business Travel @ About.com" is working.
"Salon Travel" is working.
```

## Links Outside and Inside a Block

The latest HTML5 standard allows the `<a>` tags to be placed inside and outside of block-level tags like `<div>`, `<p>`, or `<h1>`. The `"By.linkText()"` and `"By.partialLinkText()` methods can access a link located outside and inside these block-level elements. Consider the HTML code below.

```
<body>
  <p>
    <a href="http://www.google.com">Inside a block-level tag.</a>
  </p>

  <br>
  <a href="http://www.fb.com">
    <div>
      <span>Outside a block-level tag.</span>
    </div>
  </a>
</body>
```

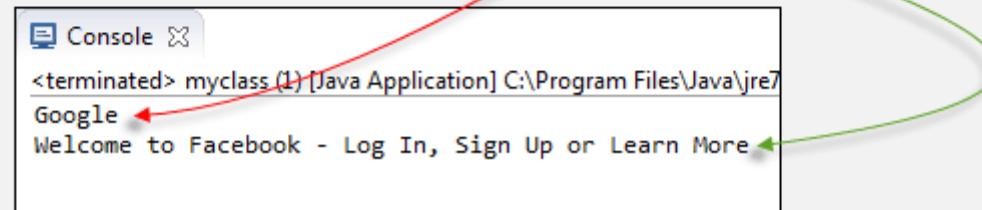


# Links Outside and Inside a Block

The WebDriver code below accesses both of these links using By.partialLinkText() method.

```
public static void main(String[] args) {
    String baseUrl = "file:///D:/Links%20Outside%20and%20Inside%20a%20Block.html";
    WebDriver driver = new FirefoxDriver();

    driver.get(baseUrl);
    driver.findElement(By.partialLinkText("Inside")).click();
    System.out.println(driver.getTitle());
    driver.navigate().back();
    driver.findElement(By.partialLinkText("Outside")).click();
    System.out.println(driver.getTitle());
    driver.quit();
}
```



The output above confirms that both links were accessed successfully because their respective page titles were retrieved correctly.

# Accessing Image Links

Image links are images that act as references to other sites or sections within the same page. Since they are images, we cannot use the By.linkText() and By.partialLinkText() methods because image links basically have no link texts at all. In this case, we should resort to using either By.cssSelector or By.xpath. The first method is more preferred because of its simplicity.

In the example below, we will access the "Facebook" logo on the upper left portion of Facebook's Password Recovery page.



## Accessing Image Links

We will use By.cssSelector and the element's "title" attribute to access the image link. And then we will verify if we are taken to Facebook's homepage.

```
?  
1 package practice_webdriver;  
2  
3  
4  
5 import org.openqa.selenium.*;  
6 import org.openqa.selenium.firefox.FirefoxDriver;  
7  
8 public class ImageLink {  
9  
10    public static void main(String[] args) {  
11        String baseUrl = "https://www.facebook.com/login/identify?ctx=recover";  
12        WebDriver driver = new FirefoxDriver();  
13  
14        driver.get(baseUrl);  
15        //click on the "Facebook" logo on the upper left portion  
16        driver.findElement(By.cssSelector("a[title=\"Go to Facebook Home\"]")).click();  
17  
18        //verify that we are now back on Facebook's homepage
```

```
19     if (driver.getTitle().equals("Welcome to Facebook - Log In, Sign Up")) {  
20         System.out.println("We are back at Facebook's homepage");  
21     } else {  
22         System.out.println("We are NOT in Facebook's homepage");  
23     }  
24     driver.quit();  
25 }  
26 }
```

Result



## Reading a Table

There are times when we need to access elements (usually texts) that are within HTML tables. However, it is very seldom for a web designer to provide an id or name attribute to a certain cell in the table. Therefore, we cannot use the usual methods such as "By.id()", "By.name()", or "By.cssSelector()". In this case, the most reliable option is to access them using the "By.xpath()" method.

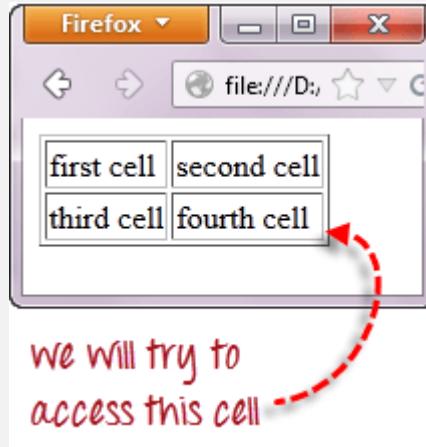
## XPath Syntax

Consider the HTML code below.

```
<html>
  <head>
    <title>Sample</title>
  </head>
  <body>
    <table border="1">
      <tbody>
        <tr>
          <td>first cell</td>
          <td>second cell</td>
        </tr>
        <tr>
          <td>third cell</td>
          <td>fourth cell</td>
        </tr>
      </tbody>
    </table>
  </body>
</html>
```

# XPath Syntax

We will use XPath to get the inner text of the cell containing the text "fourth cell".



## Step 1 - Set the Parent Element (table)

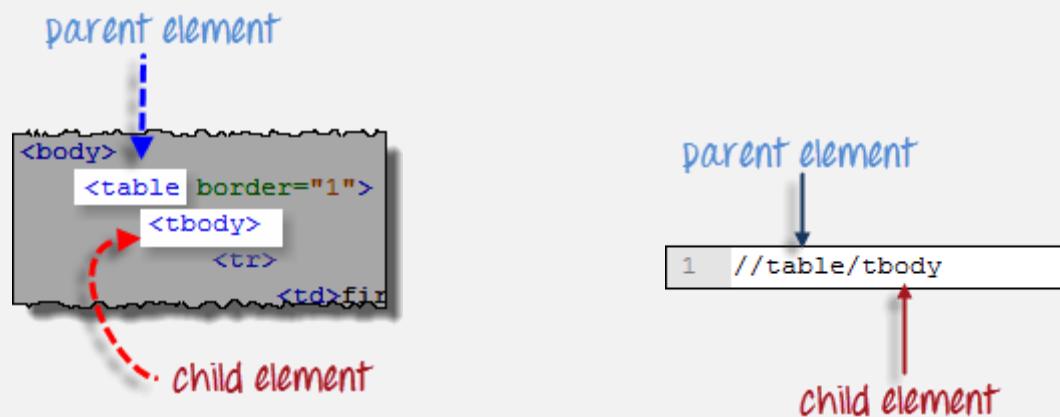
**XPath locators in WebDriver always start with a double forward slash "://" and then followed by the parent element.** Since we are dealing with tables, the parent element should always be the `<table>` tag. The first portion of our XPath locator should therefore start with "`//table`".

```
//table
```

## Step 2 - Add the child elements

The element immediately under `<table>` is `<tbody>` so we can say that `<tbody>` is the "child" of `<table>`. And also, `<table>` is the "parent" of `<tbody>`. All child elements in XPath are placed to the right of their parent element, separated with one forward slash "/" like the code shown below.

## XPath Syntax



### Step 3 - Add Predicates

The `<tbody>` element contains two `<tr>` tags. We can now say that these two `<tr>` tags are "children" of `<tbody>`. Consequently, we can say that `<tbody>` is the parent of both the `<tr>` elements.

Another thing we can conclude is that the two `<tr>` elements are siblings. **Siblings refer to child elements having the same parent.**

To get to the `<td>` we wish to access (the one with the text "fourth cell"), we must first access the **second** `<tr>` and not the first. If we simply write `"//table/tbody/tr"`, then we will be accessing the first `<tr>` tag.

So, how do we access the second `<tr>` then? The answer to this is to use **Predicates**.

Predicates are numbers or HTML attributes enclosed in a pair of square brackets "[ ]" that distinguish a child element from its *siblings*. Since the <tr> we need to access is the second one, we shall use "[2]" as the predicate.

## XPath Syntax

1 //table/tbody/tr[2]

The [2] predicate  
denotes that we are  
accessing the 2nd <tr> of  
the parent <tbody>

If we won't use any predicate, XPath will access the first sibling. Therefore, we can access the first <tr> using either of these XPath codes.

//table/tbody/tr

this will automatically access the first  
<tr> because no predicate was used

//table/tbody/tr[1]

this will access the first <tr> because  
the predicate [1] explicitly says it

## XPath Syntax

### Step 4 - Add the Succeeding Child Elements Using the Appropriate Predicates

The next element we need to access is the second <td>. Applying the principles we have learned from steps 2 and 3, we will finalize our XPath code to be like the one shown below.

```
//table/tbody/tr[2]/td[2]
```

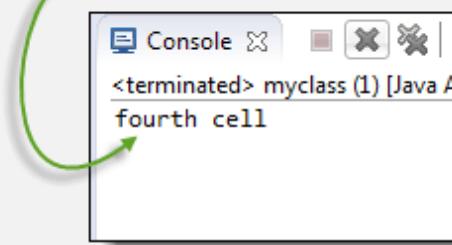
Now that we have the correct XPath locator, we can already access the cell that we wanted to and obtain its inner text using the code below. It assumes that you have saved the HTML code above as "newhtml.html" within your C Drive.

```
public static void main(String[] args) {
    String baseUrl = "file:///C:/newhtml.html";
    WebDriver driver = new FirefoxDriver();

    driver.get(baseUrl);
    String innerText = driver.findElement(
        By.xpath("//table/tbody/tr[2]/td[2]")).getText();
    System.out.println(innerText);
    driver.quit();
}
```

## XPath Syntax

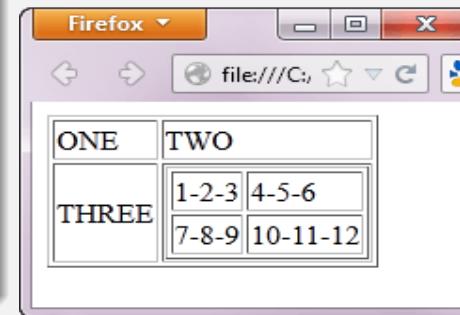
the inner text was  
successfully retrieved  
using XPath



# Accessing Nested Tables

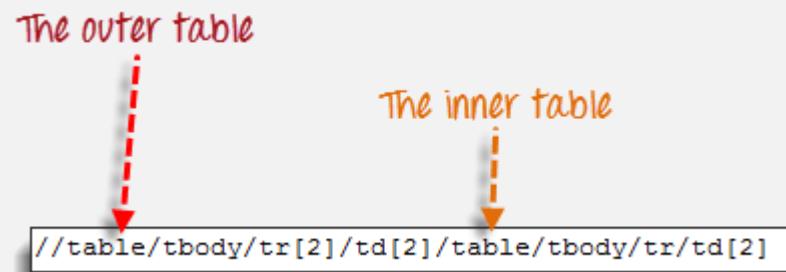
The same principles discussed above applies to nested tables. **Nested tables** are tables located within another table. An example is shown below.

```
<html>
  <head>
    <title>Sample</title>
  </head>
  <body>
    <!--outer table-->
    <table border="1">
      <tbody>
        <tr>
          <td>ONE</td>
          <td>TWO</td>
        </tr>
        <tr>
          <td>THREE</td>
          <td>
            <!--inner table-->
            <table border="1">
              <tbody>
                <tr>
                  <td>1-2-3</td>
                  <td>4-5-6</td>
                </tr>
                <tr>
                  <td>7-8-9</td>
                  <td>10-11-12</td>
                </tr>
              </tbody>
            </table>
          </td>
        </tr>
      </tbody>
    </table>
  </body>
</html>
```



## Accessing Nested Tables

To access the cell with the text "4-5-6" using the "//parent/child" and predicate concepts from the previous section, we should be able to come up with the XPath code below.



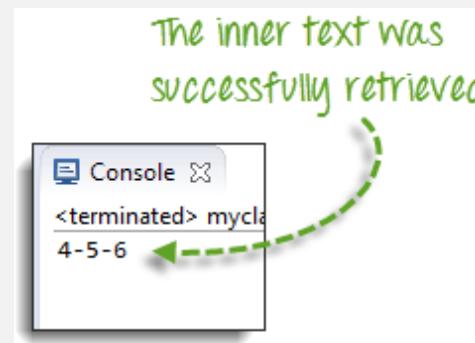
The WebDriver code below should be able to retrieve the inner text of the cell which we are accessing.

```
public static void main(String[] args) {
    String baseUrl = "file:///C:/newhtml.html";
    WebDriver driver = new FirefoxDriver();

    driver.get(baseUrl);
    String innerText = driver.findElement(By
        .xpath("//table/tbody/tr[2]/td[2]/table/tbody/tr/td[2]"))
        .getText();
    System.out.println(innerText);
    driver.quit();
}
```

## Accessing Nested Tables

The output below confirms that the inner table was successfully accessed.



The inner text was  
successfully retrieved

Console <terminated> mycl  
4-5-6

A green dashed arrow points from the text "The inner text was successfully retrieved" to the number "4-5-6" in the console output.

## Using Attributes as Predicates

If the element is written deep within the HTML code such that the number to use for the predicate is very difficult to determine, we can use that element's unique attribute instead.

In the example below, the "New York to Chicago" cell is located deep into Mercury Tours homepage's HTML code.

Specials	
Atlanta to Las Vegas	\$398
Boston to San Francisco	\$513
Los Angeles to Chicago	\$168
New York to Chicago	\$198
Phoenix to San Francisco	\$213



## Using Attributes as Predicates

```
<body>
  <div>
    <table height="100%" cellspacing="0" cellpadding="0" border="0">
      <tbody>
        <tr>
          +<td valign="top" bgcolor="#003366">
          <td valign="top">
            <table cellspacing="0" cellpadding="0" border="0">
              <tbody>
                +<tr>
                +<tr>
                +<tr>
                +<tr>
                <td>
                  <table cellspacing="0" cellpadding="0" border="0">
                    <tbody>
                      <tr>
                        <td width="14"> </td>
                      <td>
                        <table width="492" cellspacing="0" cellpadding="0" border="0">
                          <tbody>
                            <tr> </tr>
                            <tr>
                              <td width="273" valign="top">
                                +<p>
                                <table width="100%" cellspacing="0" cellpadding="0" border="0">
                                  <tbody>
                                    +<tr>
                                    +<tr>
                                    <tr valign="top">
                                      <td height="101">
                                        <table width="270" cellspacing="0" cellpadding="0" border="0">
                                          <tbody>
                                            <tr bgcolor="#CCCCCC">
                                              <td width="80%">
                                                <font size="2"> Vegas </font>
                                              </td>
                                              +<td width="20%">
```

this is the `<table>` that holds the New York to Chicago cell. Notice that it is buried deep into the HTML code, and the number to use as predicate is difficult to determine.



## Using Attributes as Predicates

In this case, we can use the table's unique attribute (`width="270"`) as the predicate. **Attributes are used as predicates by prefixing them with the `@` symbol.** In the example above, the "New York to Chicago" cell is located in the first `<td>` of the fourth `<tr>`, and so our XPath should be as shown below.

```
//table[@width="270"]/tbody/tr[4]/td
```

```
By.xpath("//table[@width=\"270\"]/tbody/tr[4]/td"))
```

use the escape characters here

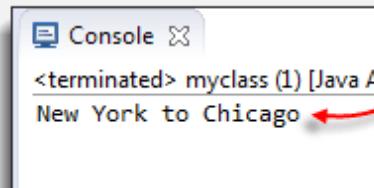
Remember that when we put the XPath code in Java, we should use the escape character backward slash "`\`" for the double quotation marks on both sides of "270" so that the string argument of `By.xpath()` will not be terminated prematurely

## Using Attributes as Predicates

We are now ready to access that cell using the code below.

```
public static void main(String[] args) {  
    String baseUrl = "http://newtours.demoaut.com/";  
    WebDriver driver = new FirefoxDriver();  
  
    driver.get(baseUrl);  
    String innerText = driver.findElement(By  
        .xpath("//table[@width=\"270\"]/tbody/tr[4]/td"))  
        .getText();  
    System.out.println(innerText);  
    driver.quit();  
}
```

the inner text was  
successfully retrieved.



## Shortcut: Use Firebug

If the number or attribute of an element is extremely difficult or impossible to obtain, the quickest way to generate the XPath code is thru Firebug.

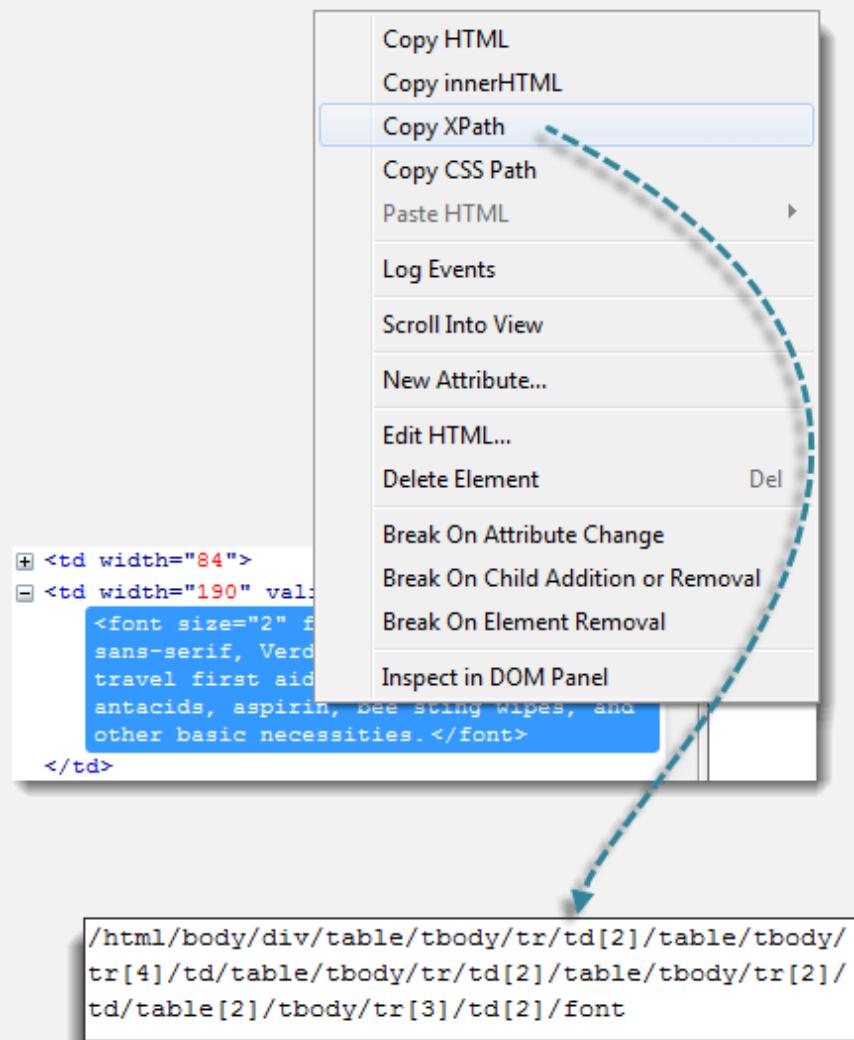
Consider the example below from Mercury Tours homepage.



### Step 1

Use Firebug to obtain the XPath code.

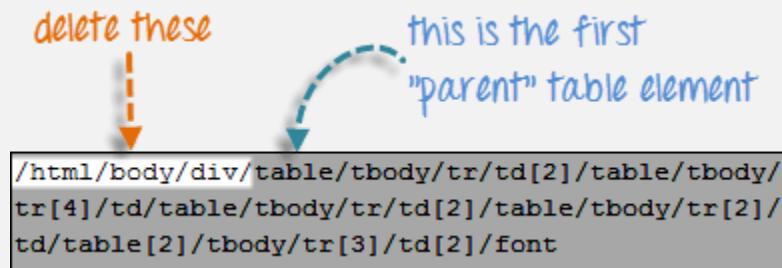
## Shortcut: Use Firebug



## Shortcut: Use Firebug

### Step 2

Look for the first "table" parent element and delete everything to the left of it.



### Step 3

Prefix the remaining portion of the code with double forward slash "://" and copy it over to your WebDriver code

## Shortcut: Use Firebug

The remaining portion of the code, trimmed  
and prefixed with "://"

```
//table/tbody/tr/td[2]/table/tbody/tr[4]/td/table/  
tbody/tr/td[2]/table/tbody/tr[2]/td/table[2]/tbody  
/tr[3]/td[2]/font
```



```
By.xpath("//table/tbody/tr/td[2]"  
+ "/table/tbody/tr[4]/td"  
+ "/table/tbody/tr/td[2]"  
+ "/table/tbody/tr[2]/td"  
+ "/table[2]/tbody/tr[3]/td[2]/font"))
```

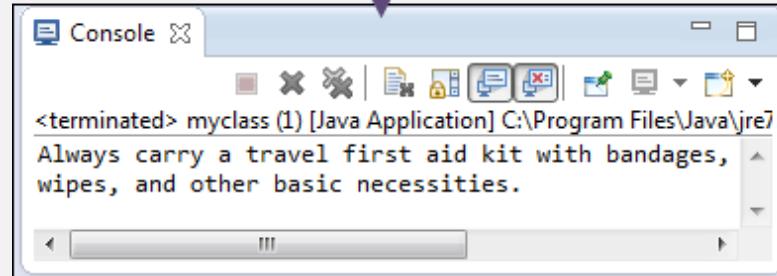
When pasted onto the By.xpath() method

## Shortcut: Use Firebug

The WebDriver code below will be able to successfully retrieve the inner text of the element we are accessing.

```
public static void main(String[] args) {
    String baseUrl = "http://newtours.demoaut.com/";
    WebDriver driver = new FirefoxDriver();

    driver.get(baseUrl);
    String innerText = driver.findElement(
        By.xpath("//table/tbody/tr/td[2]"
        + "/table/tbody/tr[4]/td"
        + "/table/tbody/tr/td[2]"
        + "/table/tbody/tr[2]/td"
        + "/table[2]/tbody/tr[3]/td[2]/font"))
        .getText();
    System.out.println(innerText);
    driver.quit();
}
```



The screenshot shows a Java application's console window titled "Console". The title bar includes standard window controls (minimize, maximize, close) and tabs. Below the title bar is a toolbar with various icons. The main area of the window displays the output of a terminated process named "myclass (1) [Java Application] C:\Program Files\Java\jre7". The output text is:  
<terminated> myclass (1) [Java Application] C:\Program Files\Java\jre7  
Always carry a travel first aid kit with bandages,  
wipes, and other basic necessities.

## Summary

- Accessing links using their exact match is done using By.linkText() method.
- Accessing links using their partial match is done using By.partialLinkText() method.
- If there are multiple matches, By.linkText() and By.partialLinkText() will only select the first match.
- Pattern matching using By.linkText() and By.partialLinkText() is case-sensitive.
- The By.tagName("a") method is used to fetch all links within a page.
- Links can be accessed by the By.linkText() and By.partialLinkText() whether they are inside or outside block-level elements.
- Accessing image links are done using By.cssSelector() and By.xpath() methods.
- By.xpath() is commonly used to access table elements.

# Keyboard Mouse Events, Uploading Files - Webdriver

## Keyboard Mouse Events, Uploading Files - Webdriver

In this tutorial, we will learn handling keyboard and mouse in Webdriver and deal with file uploads and downloads.

## Handling Keyboard & Mouse Events



Handling special keyboard and mouse events are done using the **Advanced User Interactions API**. It contains the **Actions** and the **Action** classes that are needed when executing these events. The following are the most commonly used keyboard and mouse events provided by the Actions class.

# Handling Keyboard & Mouse Events

Method	Description
<code>clickAndHold()</code>	Clicks (without releasing) at the current mouse location.
<code>contextClick()</code>	Performs a context-click at the current mouse location.
<code>doubleClick()</code>	Performs a double-click at the current mouse location.
<code>dragAndDrop(source, target)</code>	Performs click-and-hold at the location of the source element, moves to the location of the target element, then releases the mouse.  <b>Parameters:</b>  <i>source</i> - element to emulate button down at.  <i>target</i> - element to move to and release the mouse at.
<code>dragAndDropBy(source, x-offset, y-offset)</code>	Performs click-and-hold at the location of the source element, moves by a given offset, then releases the mouse.  <b>Parameters:</b>  <i>source</i> - element to emulate button down at.  <i>xOffset</i> - horizontal move offset.

	<i>yOffset</i> - vertical move offset.
<b>keyDown(modifier_key)</b>	Performs a modifier key press. Does not release the modifier key - subsequent interactions may assume it's kept pressed.  <b>Parameters:</b>  <i>modifier_key</i> - any of the modifier keys (Keys.ALT, Keys.SHIFT, or Keys.CONTROL)
<b>keyUp(modifier _key)</b>	Performs a key release.  <b>Parameters:</b>  <i>modifier_key</i> - any of the modifier keys (Keys.ALT, Keys.SHIFT, or Keys.CONTROL)
<b>moveByOffset(x-offset, y-offset)</b>	Moves the mouse from its current position (or 0,0) by the given offset.  <b>Parameters:</b>  <i>x-offset</i> - horizontal offset. A negative value means moving the mouse left.  <i>y-offset</i> - vertical offset. A negative value means moving the mouse up.
<b>moveToElement(toElement)</b>	Moves the mouse to the middle of the element. <b>Parameters:</b>  <i>toElement</i> - element to move to.
<b>release()</b>	Releases the depressed left mouse button at the current mouse location

**sendKeys(onElement, charsequence)**

Sends a series of keystrokes onto the element. **Parameters:**

*onElement* - element that will receive the keystrokes, usually a text field

*charsequence* - any string value representing the sequence of keystrokes to be sent

# Handling Keyboard & Mouse Events

In the following example, we shall use the `moveToElement()` method to mouse-over on one Mercury Tours' table rows. See the example below.



The cell shown above is a portion of a `<TR>` element. If not hovered, its color is `#FFC455` (orange). After hovering, the cell's color becomes transparent. It becomes the same color as the blue background of the whole orange table.

## Step 1

Import the **Actions** and **Action** classes.

```
import org.openqa.selenium.interactions.Action;  
import org.openqa.selenium.interactions.Actions;
```

# Keyboard & Mouse Events

## Step 2

Instantiate a new Actions object.

```
Actions builder = new Actions(driver);
```

## Step 3

Instantiate an Action using the Actions object in step 2.

```
Action mouseOverHome = builder  
    .moveToElement(link_Home)  
    .build();
```

In this case, we are going to use the `moveToElement()` method because we are simply going to mouse-over the "Home" link. The `build()` method is always the final method used so that all the listed actions will be compiled into a single step.

## Step 4

Use the `perform()` method when executing the Action object we designed in Step 3.

```
mouseOverHome.perform();
```

# Handling Keyboard & Mouse Events

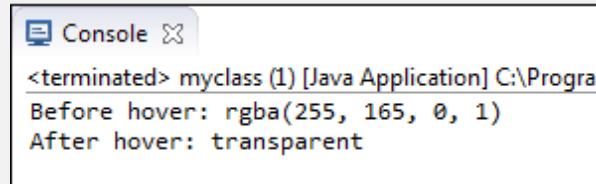
Below is the whole WebDriver code to check the background color of the <TR> element before and after the mouse-over.

```
?  
1 package mypackage;  
2  
3 import org.openqa.selenium.*;  
4 import org.openqa.selenium.firefox.FirefoxDriver;  
5  
6 import org.openqa.selenium.interactions.Action;  
7 import org.openqa.selenium.interactions.Actions;  
8  
9 public class myclass {  
10  
11 public static void main(String[] args) {  
12 String baseUrl = "http://newtours.demoaut.com/";  
13     WebDriver driver = new FirefoxDriver();  
14  
15     driver.get(baseUrl);  
16     WebElement link_Home = driver.findElement(By.linkText("Home"));  
17     WebElement td_Home = driver  
18         .findElement(By  
19             .xpath("//html/body/div")
```

```
20         + "/table/tbody/tr/td"
21         + "/table/tbody/tr/td"
22         + "/table/tbody/tr/td"
23         + "/table/tbody/tr"));
24
25     Actions builder = new Actions(driver);
26
27     Action mouseOverHome = builder
28         .moveToElement(link_Home)
29         .build();
30
31     String bgColor = td_Home.getCssValue("background-color");
32     System.out.println("Before hover: " + bgColor);
33
34     mouseOverHome.perform();
35
36     bgColor = td_Home.getCssValue("background-color");
37     System.out.println("After hover: " + bgColor);
38
39     driver.quit();
40 }
```

# Handling Keyboard & Mouse Events

The output below clearly states that the background color became transparent after the mouse-over.



A screenshot of a Java application's console window titled "Console". The window shows the output of a program named "myclass (1) [Java Application] C:\Program". The output text is:  
<terminated> myclass (1) [Java Application] C:\Program  
Before hover: rgba(255, 165, 0, 1)  
After hover: transparent

## Building a Series of Multiple Actions

You can build a series of actions using the **Action** and **Actions** classes. Just remember to close the series with the **build()** method. Consider the sample code below.

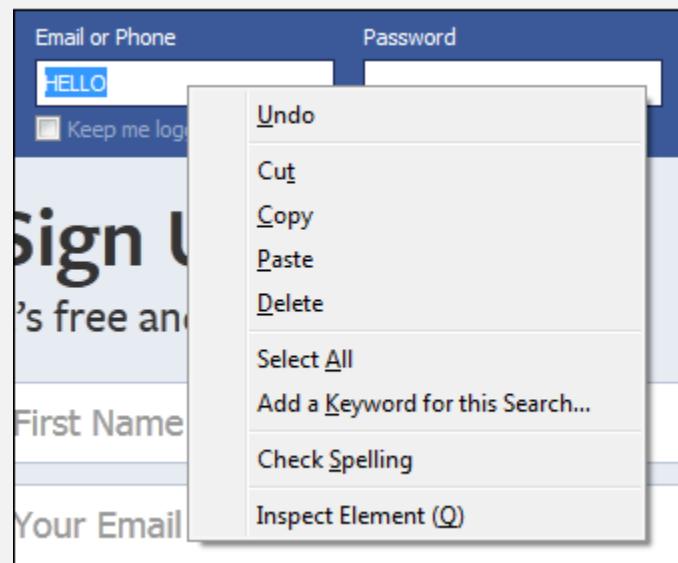
```
public static void main(String[] args) {  
    String baseUrl = "http://www.facebook.com/";  
    WebDriver driver = new FirefoxDriver();  
  
    driver.get(baseUrl);  
    WebElement txtUsername = driver.findElement(By.id("email"));  
  
    Actions builder = new Actions(driver);  
    Action seriesOfActions = builder  
        .moveToElement(txtUsername)  
        .click()  
        .keyDown(txtUsername, Keys.SHIFT)  
        .sendKeys(txtUsername, "hello")  
        .keyUp(txtUsername, Keys.SHIFT)  
        .doubleClick(txtUsername)  
        .contextClick()  
        .build();  
  
    seriesOfActions.perform();  
}
```

this will type "hello" in uppercase

this will highlight the text "HELLO"

this will bring up the context menu

## Building a Series of Multiple Actions



# Cross Browser Testing Using Selenium

## Cross Browser Testing using Selenium

### What is Cross Browser Testing?

**Cross Browser Testing** is a type of functional test to check that your web application works as expected in different browsers.



# Why do we need Cross Browser Testing?

Web based applications are totally different from windows applications. A web application can be opened in any browser by the end user. For example some people prefer to open <http://twitter.com> in **Firefox browser**, while other's can be using **Chrome browser** or **IE**.

In the diagram below you can observe that in **IE**, the login box of twitter is not showing curve at all corners but we are able to see it in chrome browser.



So we need to ensure that the web application will work as expected in all popular browsers, so that more people can access it and use it. This motive can be fulfilled with Cross Browser Testing of the product.

## Why do we need Cross Browser Testing?

### Reason Cross Browser Issues

1. Font size mismatch in different browsers.
2. [JavaScript](#) implementation can be different.
3. CSS,HTML validation difference can be there.
4. Some browser still not supporting HTML5.
5. Page alignment and div size.
6. Image orientation.
7. Browser incompatibility with OS. Etc.

# How to perform Cross Browser Testing

If we are using Selenium WebDriver, we can automate test cases using Internet Explorer, FireFox, Chrome, Safari browsers.

To execute test cases with different browsers in the same machine at same time we can integrate TestNG [framework](#) with Selenium WebDriver.

Your testing.xml will look like that,

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="TestSuite" thread-count="2" parallel="tests" >
  <test name="ChromeTest">
    <parameter name="browser" value="Chrome" />
    <classes>
      <class name="parallelTest.CrossBrowserScript">
        </class>
    </classes>
  </test>
  <test name="FirefoxTest">
    <parameter name="browser" value="Firefox" />
    <classes>
      <class name="parallelTest.CrossBrowserScript">
        </class>
    </classes>
  </test>
</suite>
```

Test will run parallel

1st Test name is 'ChromeTest'

Parameter is 'Chrome'

2st Test name is 'FirefoxTest'

Parameter is 'Firefox'

This is the TestNGXML for Cross Browser Testing

## How to perform Cross Browser Testing

This testing.xml will map with the test case which will look like that

```
Parameter will be pass from testNG.xml  
-----  
@BeforeTest  
@Parameters("browser")  
public void setup(String browser) throws Exception{  
    if(browser.equalsIgnoreCase("firefox")){  
        driver = new FirefoxDriver();  
    }  
    else if(browser.equalsIgnoreCase("chrome")){  
        System.setProperty("webdriver.chrome.driver", ".\\chromedriver.exe");  
        driver = new ChromeDriver();  
    }  
}
```

check parameter value  
and create webDriver  
according it

Here because the testing.xml has two Test tags ('ChromeTest','FirefoxTest'),this test case will execute two times for 2 different browsers.

First Test 'ChromeTest' will pass the value of parameter 'browser' as 'chrome' so ChromeDriver will be executed. This test case will run on Chrome browser.

Second Test 'FirefoxTest' will pass the value of parameter 'browser' as 'firefox' so FirefoxDriver will be executed. This test case will run on FireFox browser.

## How to perform Cross Browser Testing

Complete Code:

Guru99CrossBrowserScript.java

```
1 package parallelTest;  
2  
3 import java.util.concurrent.TimeUnit;  
4  
5 import org.openqa.selenium.By;  
6  
7 import org.openqa.selenium.WebDriver;  
8  
9 import org.openqa.selenium.WebElement;  
10  
11 import org.openqa.selenium.chrome.ChromeDriver;  
12  
13 import org.openqa.selenium.firefox.FirefoxDriver;  
14  
15 import org.testng.annotations.BeforeTest;
```

```
16
17 import org.testng.annotations.Parameters;
18
19 import org.testng.annotations.Test;
20
21 public class Guru99CrossBrowserScript {
22
23     WebDriver driver;
24
25
26
27 /**
28
29 * This function will execute before each Test tag in testng.xml
30
31 * @param browser
32
33 * @throws Exception
34
35 */
36
37 @BeforeTest
38
39 @Parameters("browser")
40
public void setup(String browser) throws Exception{
```

```
42
43     //Check if parameter passed from TestNG is 'firefox'
44
45     if(browser.equalsIgnoreCase("firefox")){
46
47         //create firefox instance
48
49         driver = new FirefoxDriver();
50
51     }
52
53     //Check if parameter passed as 'chrome'
54
55     else if(browser.equalsIgnoreCase("chrome")){
56
57         //set path to chromedriver.exe You may need to download it from
58         http://code.google.com/p/selenium/wiki/ChromeDriver
59
60         System.setProperty("webdriver.chrome.driver","C:\\chromedriver.exe");
61
62         //create chrome instance
63
64         driver = new ChromeDriver();
65
66     }
```

```
68 else if(browser.equalsIgnoreCase("ie")){
69
70         //set path to IEdriver.exe You may need to download it from
71
72         // 32 bits http://selenium-release.storage.googleapis.com/2.42/IEDriverServer_Win32_2.42.0.zip
73
74         // 64 bits http://selenium-release.storage.googleapis.com/2.42/IEDriverServer_x64_2.42.0.zip
75
76         System.setProperty("webdriver.ie.driver","C:\\IEdriver.exe");
77
78         //create chrome instance
79
80         driver = new InternetExplorerDriver();
81
82     }
83
84     else{
85
86         //If no browser passed throw exception
87
88         throw new Exception("Browser is not correct");
89
90     }
91
92     driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
93
```

```
94     }
95
96     }
97
98     @Test
99
100    public void testParameterWithXML() throws InterruptedException{
101
102        driver.get("http://demo.guru99.com/V4/");
103
104        //Find user name
105
106        WebElement userName = driver.findElement(By.name("uid"));
107
108        //Fill user name
109
110        userName.sendKeys("guru99");
111
112        //Find password
113
114        WebElement password = driver.findElement(By.name("password"));
115
116        //Fill password
117
118        password.sendKeys("guru99");
119
```

```
    }}
```

## How to perform Cross Browser Testing

testing.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
4
5 <suite name="TestSuite" thread-count="2" parallel="tests"  >
6
7 <test name="ChromeTest">
8
9 <parameter name="browser" value="Chrome" />
10
11<classes>
12
13<class name="parallelTest.CrossBrowserScript">
14
15</class>
16
17</classes>
```

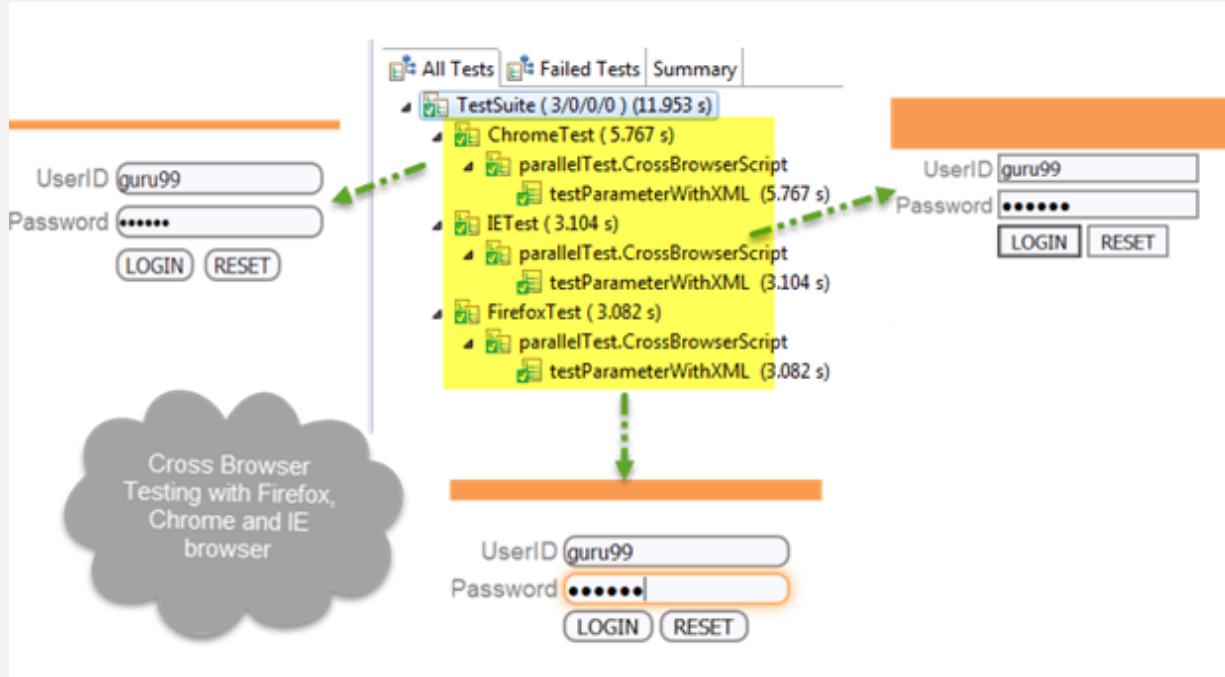
```
18
19 </test>
20
21 <test name="FirefoxTest">
22
23 <parameter name="browser" value="Firefox" />
24
25 <classes>
26
27 <class name="parallelTest.CrossBrowserScript">
28
29 </class>
30
31 </classes>
32
33 </test>
34
35 <test name="IETest">
36
37 <parameter name="browser" value="IE" />
38
39 <classes>
40
41 <class name="parallelTest.CrossBrowserScript">
42
43 </class>
```

```
44  
45 </classes>  
46  
47 </test>  
48  
49 </suite>
```

---

# How to perform Cross Browser Testing

NOTE: To run the test, Right click on the **testing.xml**, Select Run As and Click TestNG



## **Summary**

1. Cross browser Testing is a technique to test web application with different webbrowsers.
2. Selenium can support different type of browsers for automation.
3. Selenium can be integrated with Testing to perform Cross Browser Testing.
4. From parameters in testing.xml we can pass browser name and in test case we can create WebDriver reference accordingly.

# Page Object Model (POM) & Page Factory in Selenium

## Page Object Model (POM) & Page Factory in Selenium

Before we learn about Page Object Model, let's understand -

### Why POM?

Starting a UI Automation in Selenium WebDriver is **NOT** a tough task. You just need to find elements, perform operations on it. Consider this simple script to login into a website:

As you can observe, all we are doing is finding elements and filling values for those elements.

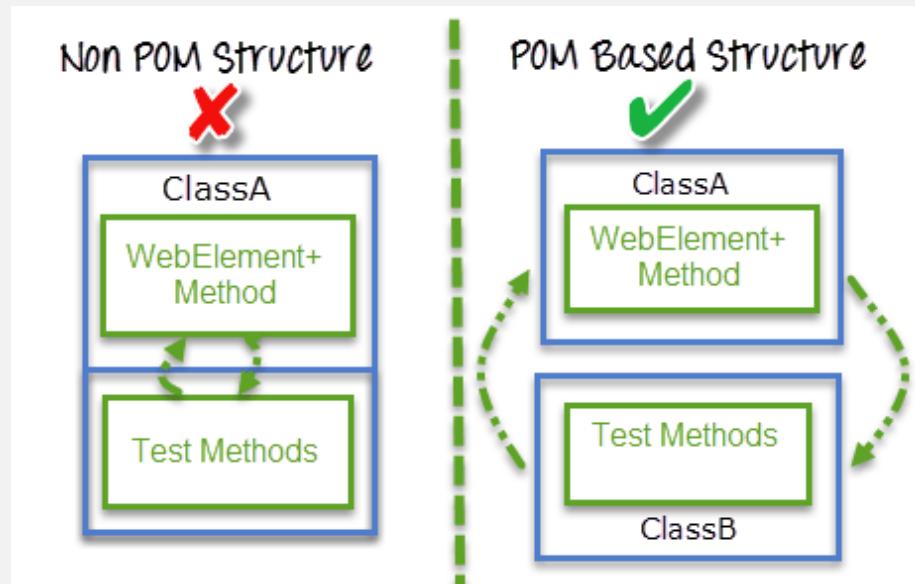
This is a small script. Script maintenance looks easy. But with time test suite will grow. As you add more and more lines to your code, things become tough.

## Why POM?

The chief problem with script maintenance is that if 10 different scripts are using the same page element, with any change in that element, you need to change all 10 scripts. This is time consuming and error prone.

A better approach to script maintenance is to create a separate class file which would find web elements, fill them or verify them. This class can be reused in all the scripts using that element. In future if there is change in the web element, we need to make change in just 1 class file and not 10 different scripts.

This approach is called **Page Object Model (POM)**. It helps make code **more readable, maintainable, and reusable**.



## What is POM?

- **Page Object Model** is a design pattern to create **Object Repository** for web UI elements.
- Under this model, for each web page in the application there should be corresponding page class.
- This Page class will find the WebElements of that web page and also contains Page methods which perform operations on those WebElements.
- Name of these methods should be given as per the task they are performing i.e., if a loader is waiting for payment gateway to appear, POM method name can be `waitForPaymentScreenDisplay()`.

## Advantages of POM

1. Page Object Pattern says operations and flows in the UI should be separated from verification. This concept makes our code cleaner and easy to understand.
2. Second benefit is the **object repository is independent of testcases**, so we can use the same object repository for a different purpose with different tools. For example, we can integrate POM with TestNG/JUnit for functional testing and at the same time with JBehave/[Cucumber](#) for acceptance testing.
3. Code becomes less and optimized because of the reusable page methods in the POM classes.
4. **Methods get more realistic names** which can be easily mapped with the operation happening in UI. i.e. if after clicking on the button we land on the home page, the method name will be like 'gotoHomePage()'.

# How to implement POM?

Simple POM:

It's the basic structure of Page object model (POM) where all Web Elements of the **AUT** and the method that operate on these Web Elements are maintained inside a class file. Task like **verification** should be **separate** as part of Test methods.

```
public class Guru99Login { 1 Page class in object repository
    WebDriver driver;
    By user99GuruName = By.name("uid");
    By password99Guru = By.name("password");
    By titleText = By.className("barone");
    By login = By.name("btnLogin");

    public Guru99Login(WebDriver driver){
        this.driver = driver;
    }
    //Set user name in textbox
    public void setUserName(String strUserName){
2        driver.findElement(user99GuruName).sendKeys(strUserName); 3
    }
}
```

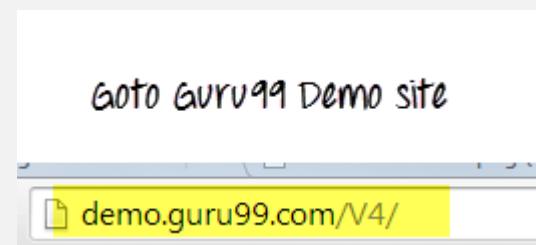
Find Web Element

Performing operation on web element

## Complete Example

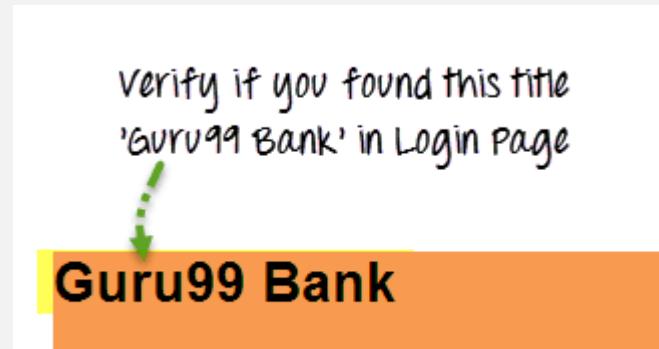
**TestCase:** Go to Guru99 Demo Site.

Step 1) Go to Guru99 Demo Site

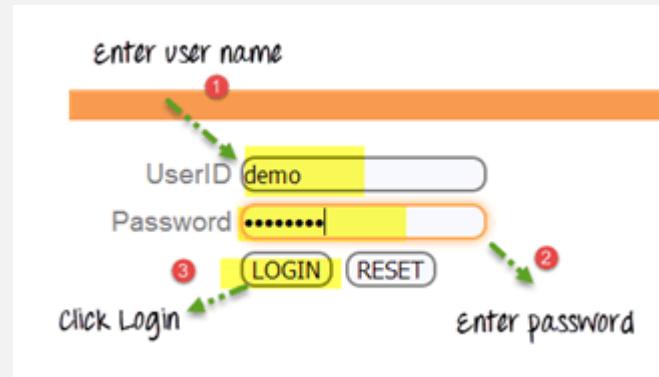


## How to implement POM?

Step 2) On home page check text "Guru99 Bank" is present



Step 3) Login into application



## How to implement POM?

Step 4) Verify that the Home page contains text as "Manger Id:

demo"

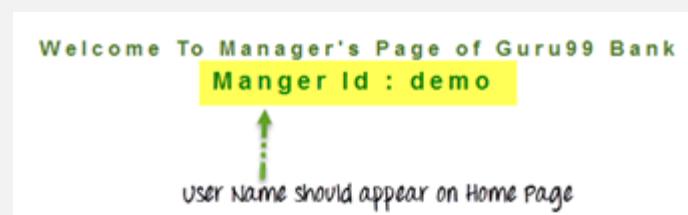
Here are we are dealing with 2 pages

1. Login Page
2. Home Page (shown once you

login)

3. Accordingly we create 2 POM

classes



## How to implement POM?

### Guru99 Login page POM

```
1 package pages;
2
3 import org.openqa.selenium.By;
4
5 import org.openqa.selenium.WebDriver;
6
7 public class Guru99Login {
8
9     WebDriver driver;
10
11     By user99GuruName = By.name("uid");
12
13     By password99Guru = By.name("password");
14
15     By titleText =By.className("barone");
16
17     By login = By.name("btnLogin");
18
19
20
```

```
21 public Guru99Login(WebDriver driver){  
22  
23     this.driver = driver;  
24  
25 }  
26  
27 //Set user name in textbox  
28  
29 public void setUserName(String strUserName){  
30  
31     driver.findElement(user99GuruName).sendKeys(strUserName);;  
32  
33 }  
34  
35  
36  
37 //Set password in password textbox  
38  
39 public void setPassword(String strPassword){  
40  
41     driver.findElement(password99Guru).sendKeys(strPassword);  
42  
43 }  
44  
45 }
```

```
47 //Click on login button
48
49 public void clickLogin(){
50
51     driver.findElement(login).click();
52
53 }
54
55
56
57 //Get the title of Login Page
58
59 public String getLoginTitle(){
60
61     return driver.findElement(titleText).getText();
62
63 }
64
65 /**
66
67 * This POM method will be exposed in test case to login in the application
68
69 * @param strUserName
70
71 * @param strPassword
```

```
73     * @return  
74  
75     */  
76  
77     public void loginToGuru99(String strUserName, String strPasword){  
78  
79         //Fill user name  
80  
81         this.setUserName(strUserName);  
82  
83         //Fill password  
84  
85         this.setPassword(strPasword);  
86  
87         //Click Login button  
88  
89         this.clickLogin();  
90  
91  
92     }  
93  
94  
95 }
```

## Guru99 Home Page POM

```
1 package pages;
2
3 import org.openqa.selenium.By;
4
5 import org.openqa.selenium.WebDriver;
6
7 public class Guru99HomePage {
8
9     WebDriver driver;
10
11     By homePageUserName = By.xpath("//table//tr[@class='heading3']");
12
13
14
15     public Guru99HomePage(WebDriver driver){
16
17         this.driver = driver;
18
19     }
20
21
22
```

```
23 //Get the User name from Home Page
24
25     public String getHomePageDashboardUserName(){
26
27         return driver.findElement(homePageUserName).getText();
28
29     }
30
31 }
```

### Guru99 Simple POM Test case

```
1 package test;
2
3 import java.util.concurrent.TimeUnit;
4
5 import org.openqa.selenium.WebDriver;
6
7 import org.openqa.selenium.firefox.FirefoxDriver;
8
9 import org.testng.Assert;
10
11 import org.testng.annotations.BeforeTest;
12
13 import org.testng.annotations.Test;
14
15 import pages.Guru99HomePage;
```

```
16
17 import pages.Guru99Login;
18
19 public class Test99GuruLogin {
20
21     WebDriver driver;
22
23     Guru99Login objLogin;
24
25     Guru99HomePage objHomePage;
26
27     @BeforeTest
28
29     public void setup(){
30
31         driver = new FirefoxDriver();
32
33         driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
34
35         driver.get("http://demo.guru99.com/V4/");
36
37     }
38
39 }
40
41 /**

```

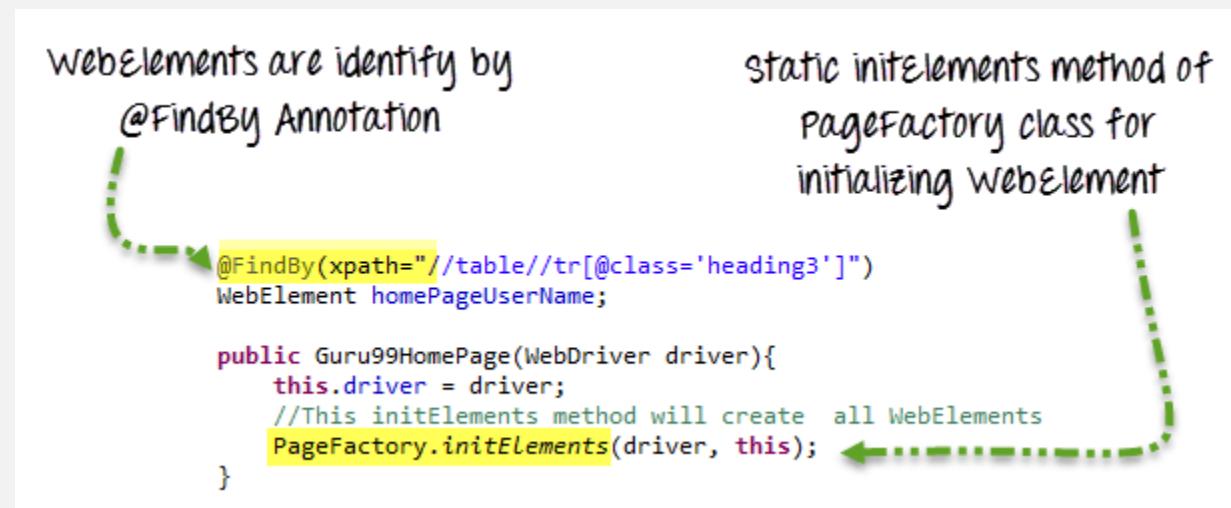
```
42
43     * This test case will login in http://demo.guru99.com/V4/
44
45     * Verify login page title as guru99 bank
46
47     * Login to application
48
49     * Verify the home page using Dashboard message
50
51     */
52
53     @Test(priority=0)
54
55     public void test_Home_Page_Appear_Correct(){
56
57         //Create Login Page object
58
59         objLogin = new Guru99Login(driver);
60
61         //Verify login page title
62
63         String loginpageTitle = objLogin.getLoginTitle();
64
65         Assert.assertTrue(loginpageTitle.toLowerCase().contains("guru99 bank"));
66
67         //login to application
```

```
68
69     objLogin.loginToGuru99("mgr123", "mgr!23");
70
71     // go the next page
72
73     objHomePage = new Guru99HomePage(driver);
74
75     //Verify home page
76
77     Assert.assertTrue(objHomePage.getHomePageDashboardUserName().toLowerCase().contains("manger id : mgr123"));
78
79 }
```

## What is Page Factory?

Page Factory is an inbuilt page object model concept for Selenium WebDriver but it is **very** optimized.

Here as well we follow the concept of separation of Page Object repository and Test methods. Additionally with the help of PageFactory class we use annotations **@FindBy** to find WebElement. We use initElements method to initialize web elements



`@FindBy` can accept `tagName`, `partialLinkText`, `name`, `linkText`, `id`, `css`, `className`, `xpath` as attributes.

Let's look at the same example as above using Page Factory

## What is Page Factory?

### Guru99 Login page with Page Factory

?

```
1 package PageFactory;  
2  
3 import org.openqa.selenium.WebDriver;  
4  
5 import org.openqa.selenium.WebElement;  
6  
7 import org.openqa.selenium.support.FindBy;  
8  
9 import org.openqa.selenium.support.PageFactory;  
10  
11 public class Guru99Login {  
12  
13     /**  
14      * All WebElements are identified by @FindBy annotation  
15     */  
16  
17 }
```

```
18  
19     WebDriver driver;  
20  
21     @FindBy(name="uid")  
22  
23     WebElement user99GuruName;  
24  
25       
26  
27     @FindBy(name="password")  
28  
29     WebElement password99Guru;  
30  
31       
32  
33     @FindBy(className="barone")  
34  
35     WebElement titleText;  
36  
37       
38  
39     @FindBy(name="btnLogin")  
40  
41     WebElement login;  
42  
43     
```

```
44
45     public Guru99Login(WebDriver driver){
46
47         this.driver = driver;
48
49         //This initElements method will create all WebElements
50
51         PageFactory.initElements(driver, this);
52
53     }
54
55     //Set user name in textbox
56
57     public void setUserName(String strUserName){
58
59         user99GuruName.sendKeys(strUserName);
60
61
62
63     }
64
65     //Set password in password textbox
66
67     public void setPassword(String strPassword){
```

```
70
71     password99Guru.sendKeys(strPassword);
72
73 }
74
75 /**
76
77 //Click on login button
78
79 public void clickLogin(){
80
81     login.click();
82
83 }
84
85 /**
86
87 //Get the title of Login Page
88
89 public String getLoginTitle(){
90
91     return titleText.getText();
92
93 }
94
95 /**
```

```
96
97     * This POM method will be exposed in test case to login in the application
98
99     * @param strUserName
100
101    * @param strPassword
102
103    * @return
104
105   */
106
107  public void loginToGuru99(String strUserName, String strPasword){
108
109      //Fill user name
110
111      this.setUserName(strUserName);
112
113      //Fill password
114
115      this.setPassword(strPasword);
116
117      //Click Login button
118
119      this.clickLogin();
120
121
```

```
122
123    }
124
125 }
```

## Guru99 Home Page with Page Factory

```
?  
1 package PageFactory;  
2  
3 import org.openqa.selenium.WebDriver;  
4  
5 import org.openqa.selenium.WebElement;  
6  
7 import org.openqa.selenium.support.FindBy;  
8  
9 import org.openqa.selenium.support.PageFactory;  
10  
11 public class Guru99HomePage {  
12  
13     WebDriver driver;  
14  
15     @FindBy(xpath="//table//tr[@class='heading3']")  
16  
17     WebElement homePageUserName;  
18
```

```
19
20
21     public Guru99HomePage(WebDriver driver){
22
23         this.driver = driver;
24
25         //This initElements method will create all WebElements
26
27         PageFactory.initElements(driver, this);
28
29     }
30
31
32
33     //Get the User name from Home Page
34
35     public String getHomePageDashboardUserName(){
36
37         return homePageUserName.getText();
38
39     }
40
41 }
```

## Guru99 TestCase with Page Factory concept

```
1 package test;  
2  
3 import java.util.concurrent.TimeUnit;  
4  
5 import org.openqa.selenium.WebDriver;  
6  
7 import org.openqa.selenium.firefox.FirefoxDriver;  
8  
9 import org.testng.Assert;  
10  
11 import org.testng.annotations.BeforeTest;  
12  
13 import org.testng.annotations.Test;  
14  
15 import PageFactory.Guru99HomePage;  
16  
17 import PageFactory.Guru99Login;  
18  
19 public class Test99GuruLoginWithPageFactory {  
20  
21     WebDriver driver;
```

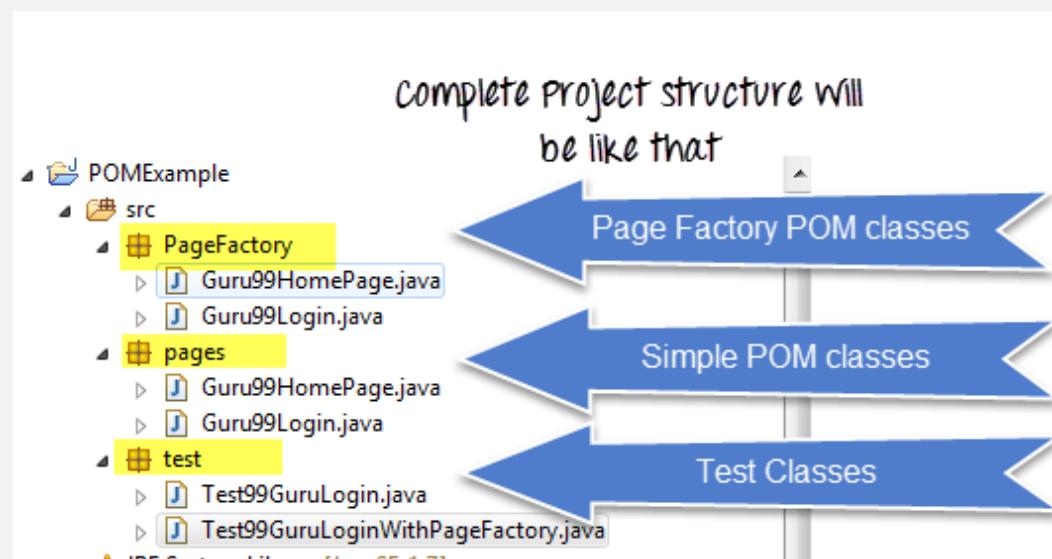
```
22
23     Guru99Login objLogin;
24
25     Guru99HomePage objHomePage;
26
27     }
28
29     @BeforeTest
30
31     public void setup(){
32
33         driver = new FirefoxDriver();
34
35         driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
36
37         driver.get("http://demo.guru99.com/V4/");
38
39     }
40
41     /**
42
43     * This test go to http://demo.guru99.com/V4/
44
45     * Verify login page title as guru99 bank
46
47     * Login to application
```

```
48
49     * Verify the home page using Dashboard message
50
51     */
52
53     @Test(priority=0)
54
55     public void test_Home_Page_Appear_Correct(){
56
57         //Create Login Page object
58
59         objLogin = new Guru99Login(driver);
60
61         //Verify login page title
62
63         String loginpageTitle = objLogin.getLoginTitle();
64
65         Assert.assertTrue(loginpageTitle.toLowerCase().contains("guru99 bank"));
66
67         //login to application
68
69         objLogin.loginToGuru99("mgr123", "mgr!23");
70
71         // go the next page
72
73         objHomePage = new Guru99HomePage(driver);
```

```

74
75     //Verify home page
76
77     Assert.assertTrue(objHomePage.getHomePageDashboardUserName().toLowerCase().contains("manger id : mgr123"));
78
79 }
80
81
82
83 }
```

Complete Project Structure will look like the diagram:



## AjaxElementLocatorFactory

One of the key advantage of using Page Factory pattern is AjaxElementLocatorFactory Class.

It is working on lazy loading concept, i.e. a timeout for a Web Element will be assigned to the Object page class with the help of AjaxElementLocatorFactory .

after 100 sec if element is not visible to perform an operation , timeout exception will appear

```
AjaxElementLocatorFactory factory = new AjaxElementLocatorFactory(driver, 100);  
PageFactory.initElements(factory, this);
```

This is a lazy loading, wait will start only if we perform operation on control

Here, when an operation is performed on an element, the ‘wait’ for its visibility starts from that moment only. If the element is not found in the given time interval, test case execution will throw 'NoSuchElementException' exception.

# **Summary**

1. Page Object Model is an Object repository design pattern in Selenium WebDriver.
2. POM creates our testing code maintainable, reusable.
3. Page Factory is an optimized way to create object repository in POM concept.
4. AjaxElementLocatorFactory is a lazy load concept in Page Factory pattern to identify WebElements only when they are used in any operation.

# Taking Screenshot Using WebDriver

## Taking Screenshot using WebDriver

It is very important to take screenshot when we execute a test script. When we execute huge number of test scripts, and if some test fails, we need to check why the test has failed.

It helps us to debug and identify the problem by seeing the screen shot.

In selenium WebDriver, we can take the screen shot using the below

```
command. File scrFile =  
((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
```

The below example explains how to take the screen shot when the test fails.

```
import java.io.File;  
import org.apache.commons.io.FileUtils;  
import org.openqa.selenium.By;  
import org.openqa.selenium.OutputType;  
import org.openqa.selenium.TakesScreenshot;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.firefox.FirefoxDriver;  
import org.testng.annotations.Test;  
  
public class takeScreenShotExample{  
    WebDriver driver;  
}
```

```
@Test

public void openBrowser() throws Exception {

    driver = new FirefoxDriver();

    driver.manage().window().maximize();

    driver.get( "http://www.google.com" );

    try {

        //the below statement will throw an exception as the element is not found, Catch block will get
        //executed and takes the screenshot.

        driver.findElement(By.id("testing")).sendKeys("test");

        //if we remove the below comment, it will not return exception and screen shot method will not get
        //executed.

        //driver.findElement(By.id("gbqfq")).sendKeys("test");

    }

    catch (Exception e){

        System.out.println( "I'm in exception" );

    }

    //calls the method to take the screenshot.
```

```
    getscreenshot();  
}  
}  
  
public void getscreenshot() throws Exception  
{  
    File scrFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);  
  
    //The below method will save the screen shot in d drive with name "screenshot.png"  
    FileUtils.copyFile(scrFile, new File( "D:\\screenshot.png" ));  
}  
}
```



# Handling Date Time Picker Using Selenium – HTML5

## Handling Date Time Picker using Selenium – HTML5

For Date Time selection, HTML5 has a new type of control shown below.

The screenshot shows a web page with a blue border. At the top, there are two buttons: "Select Date" and "Select time". Below them is a link "Open this page in Chrome". The main form area contains the text "Birthday (date and time):" followed by an input field with a red border containing "mm / dd / yyyy". To its right is another input field with a red border containing "-- : --". To the right of these fields is a small input field and a "Submit" button. Two green arrows point downwards from the "Select Date" and "Select time" labels towards their respective input fields.

Above page can be accessed [here](#)

If we see the DOM of the Date, Time Picker control, there will be only one input box for both date and time.

## Handling Date Time Picker using Selenium

The screenshot shows a browser window with the following elements:

- A header with "Open this page in Chrome" and "Date Time" buttons.
- A form field labeled "Birthday (date and time):" containing "mm / dd / yyyy : -- : --". The first two input fields are highlighted with red boxes.
- A "Submit" button.
- A status bar at the bottom with "There is only one control for both Date and Time" written in cursive.
- An open developer tools panel showing the HTML source code:

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <h3>Open this page in Chrome</h3>
    <br>
    Birthday (date and time): "
    <input type="datetime-local" name="bdaytime">
    <input type="submit">
  </form>
</body>
</html>
```

Dashed green arrows point from the status bar text to the date and time input fields in the browser and the corresponding code in the developer tools.

So to handle this type of control first we will fill date with separating with delimiter, i.e. if date is 09/25/2013, then we will pass 09252013 to the input box. Once done, we will shift focus from date to time by pressing 'tab' & fill time.

If we need to fill 02:45 PM, we will pass it a '0245PM' to the same input box.

## Handling Date Time Picker using Selenium

The code looks like this -

```
1 import java.util.List;
2
3 import java.util.concurrent.TimeUnit;
4
5 import org.openqa.selenium.By;
6
7 import org.openqa.selenium.Keys;
8
9 import org.openqa.selenium.WebDriver;
10
11 import org.openqa.selenium.WebElement;
12
13 import org.openqa.selenium.chrome.ChromeDriver;
14
15 import org.testng.annotations.Test;
16
17 public class DateTimePicker {
18
19     @Test
20
21     public void dateTimePicker(){
22
```

```
23     System.setProperty("webdriver.chrome.driver", "chromedriver.exe");
24
25     WebDriver driver = new ChromeDriver();
26
27     driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
28
29     driver.get("http://demo.guru99.com/selenium");
30
31     //Find the date time picker control
32
33     WebElement dateBox = driver.findElement(By.xpath("//form//input[@name='bdaytime']"));
34
35     //Fill date as mm/dd/yyyy as 09/25/2013
36
37     dateBox.sendKeys("09252013");
38
39     //Press tab to shift focus to time field
40
41     dateBox.sendKeys(Keys.TAB);
42
43     //Fill time as 02:45 PM
44
45     dateBox.sendKeys("0245PM");
46
47 }
```

49 }

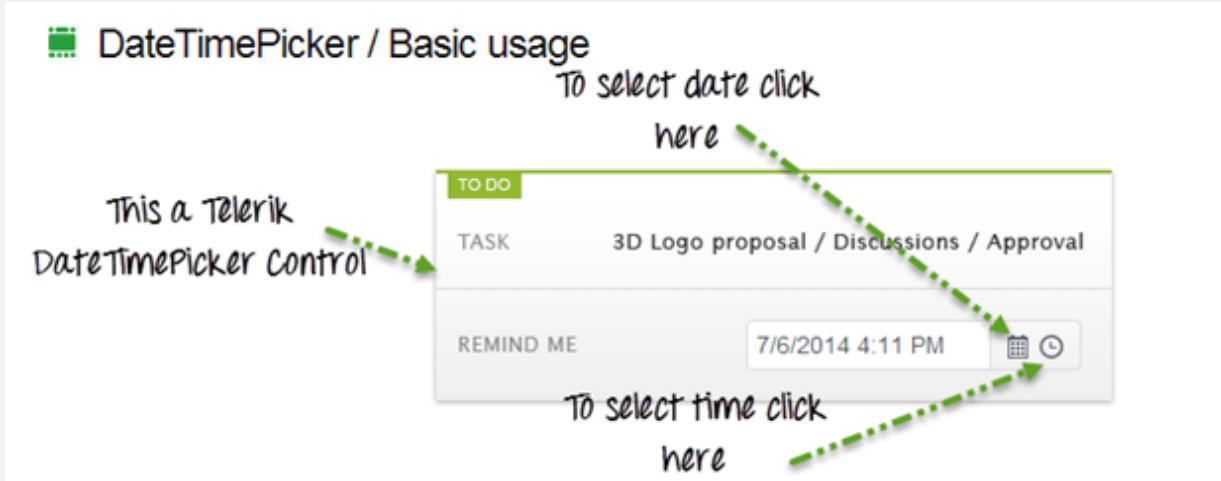
Output will be like-

**Open this page in Chrome**

Birthday (date and time):

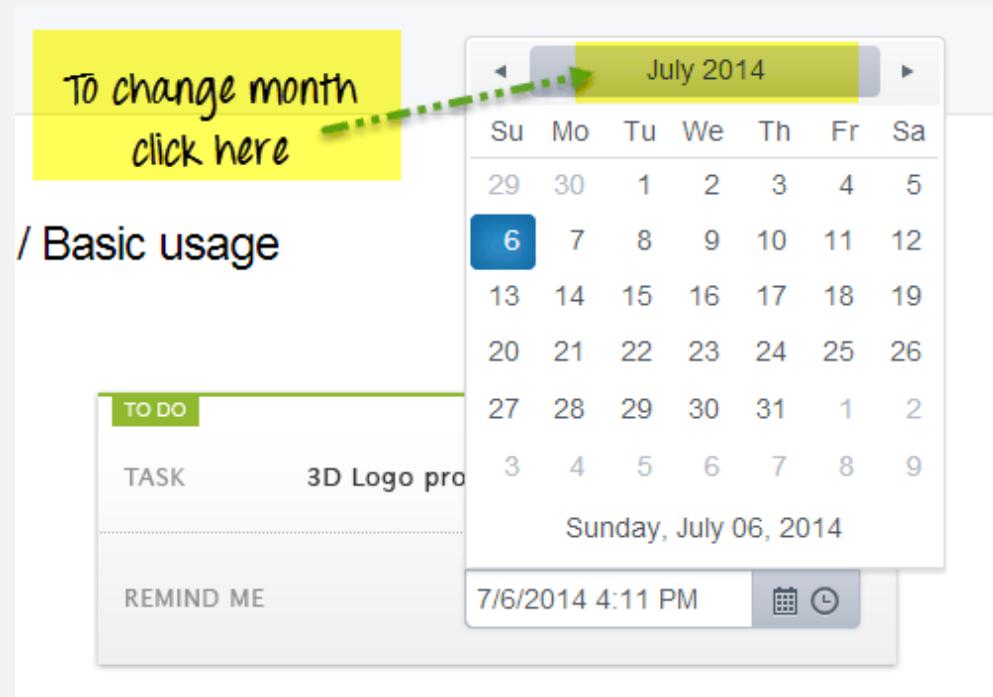
for 09/25/2013 02:45 PM ,  
output of Date Time Picker

Let's look at another example. We will use Telerik DateTimePicker control. Can be accessed [here](#)



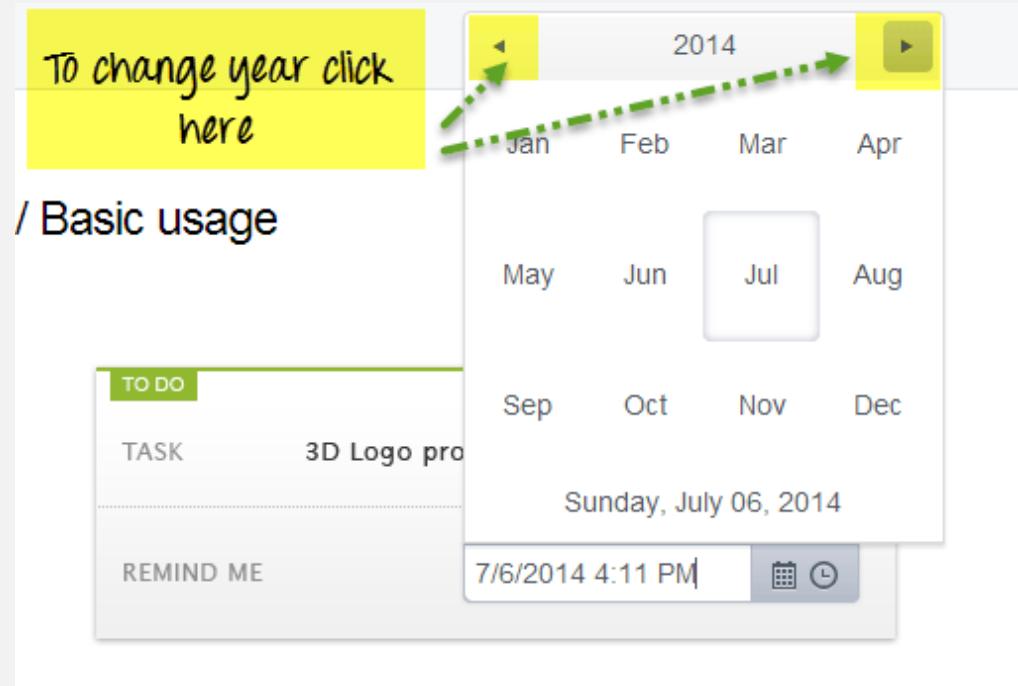
# Handling Date Time Picker using Selenium

Here if we need to change the month, we have to click on the middle of the calendar header.



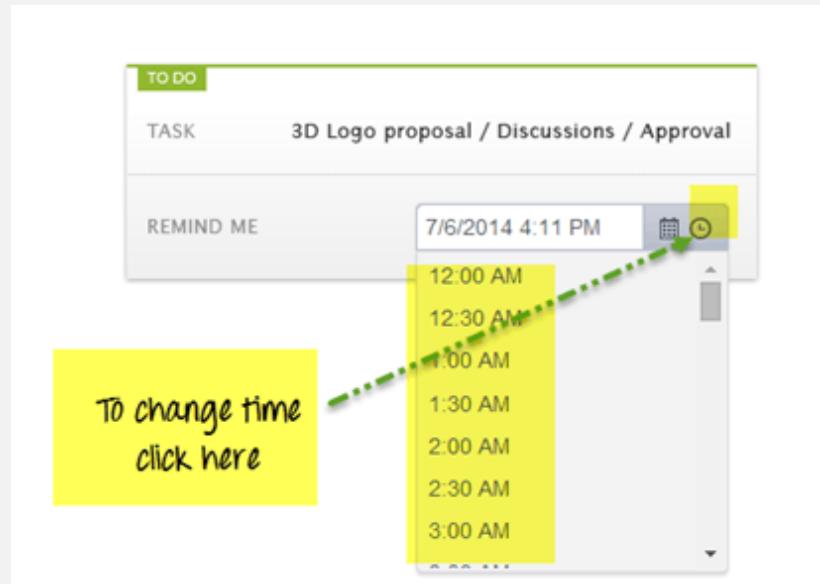
## Handling Date Time Picker using Selenium

Similarly if we need to change the year then we can do it by clicking next or previous links on the calendar.



## Handling Date Time Picker using Selenium

And finally for changing the time we can select correct time from the dropdown (Note: Here time is selected in a gap of 30 min. i.e., 12:00, 12:30 , 1:00, 1:30 etc.).



# Handling Date Time Picker using Selenium

A complete example looks like-

```
1 import java.util.Calendar;
2
3 import java.util.List;
4
5 import java.util.concurrent.TimeUnit;
6
7 import org.openqa.selenium.By;
8
9 import org.openqa.selenium.WebDriver;
10
11 import org.openqa.selenium.WebElement;
12
13 import org.openqa.selenium.firefox.FirefoxDriver;
14
15 import org.testng.annotations.Test;
16
17 public class DatePicker {
18
19     @Test
20
21     @Test
22 }
```

```
23     public void testDatePicker() throws Exception{  
24  
25  
26  
27     //Date and Time to be set in textbox  
28  
29     String dateTime ="12/07/2014 2:00 PM";  
30  
31  
32  
33     WebDriver driver = new FirefoxDriver();  
34  
35     driver.manage().window().maximize();  
36  
37  
38  
39     driver.get("http://demos.telerik.com/kendo-ui/datetimepicker/index");  
40  
41  
42  
43     driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);  
44  
45  
46  
47     //button to open calendar  
48
```

```
49     WebElement selectDate = driver.findElement(By.xpath("//span[@aria-controls='datetimepicker_dateview']"));
50
51
52     selectDate.click();
53
54     //button to move next in calendar
55
56
57     WebElement nextLink = driver.findElement(By.xpath("//div[@id='datetimepicker_dateview']//div[@class='k-
58 header']//a[contains(@class,'k-nav-next')]"));
59
60     //button to click in center of calendar header
61
62     WebElement midLink = driver.findElement(By.xpath("//div[@id='datetimepicker_dateview']//div[@class='k-
63 header']//a[contains(@class,'k-nav-fast')]"));
64
65     //button to move previous month in calendar
66
67     WebElement previousLink = driver.findElement(By.xpath("//div[@id='datetimepicker_dateview']//div[@class='k-
68 header']//a[contains(@class,'k-nav-prev')]"));
69
70     //Split the date time to get only the date part
71
72     String date_dd_MM_yyyy[] = (dateTime.split(" ")[0]).split("/");
73
74     //get the year difference between current year and year to set in calendar
```

```
75
76     int yearDiff = Integer.parseInt(date_dd_MM_yyyy[2])- Calendar.getInstance().get(Calendar.YEAR);
77
78     midLink.click();
79
80     if(yearDiff!=0){
81
82         //if you have to move next year
83
84         if(yearDiff>0){
85
86             for(int i=0;i<yearDiff;i++){
87
88                 System.out.println("Year Diff->" +i);
89
90             nextLink.click();
91
92         }
93
94     }
95
96     //if you have to move previous year
97
98     else if(yearDiff<0){
99
100        for(int i=0;i<(yearDiff*(-1));i++){
```

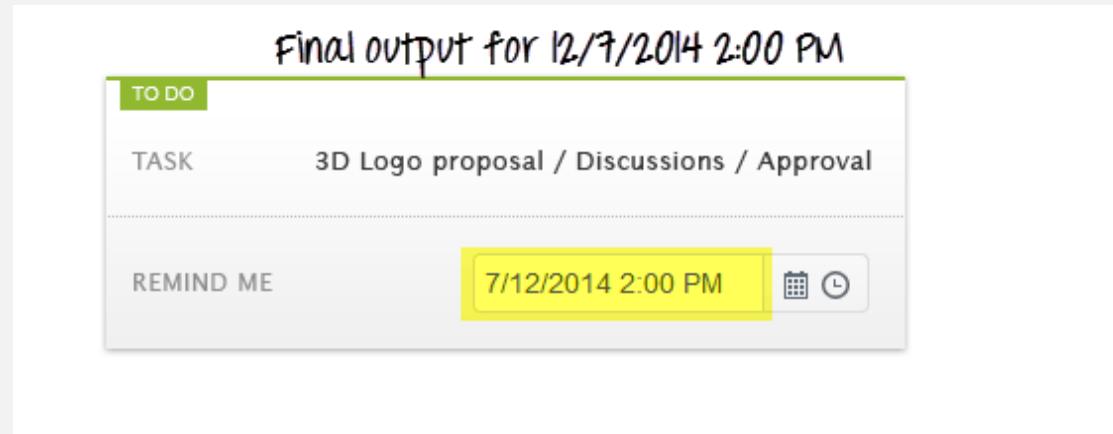
```
101
102         System.out.println("Year Diff->" + i);
103
104         previousLink.click();
105
106     }
107
108 }
109
110 }
111
112
113
114 Thread.sleep(1000);
115
116 //Get all months from calendar to select correct one
117
118 List<WebElement> list_AllMonthToBook =
119 driver.findElements(By.xpath("//div[@id='datetimepicker_dateview']//table//tbody//td[not(contains(@class,'k-other-
120 month'))]"));
121
122
123
124 list_AllMonthToBook.get(Integer.parseInt(date_dd_MM_yyyy[1]) - 1).click();
125
126
```

```
127
128     Thread.sleep(1000);
129
130     //get all dates from calendar to select correct one
131
132     List<WebElement> list_AllDateToBook =
133 driver.findElements(By.xpath("//div[@id='datetimepicker_dateview']//table//tbody//td[not(contains(@class, 'k-other-
134 month'))]"));
135
136
137
138     list_AllDateToBook.get(Integer.parseInt(date_dd_MM_yyyy[0])-1).click();
139
140
141
142
143
144     ///FOR TIME
145
146     WebElement selectTime = driver.findElement(By.xpath("//span[@aria-controls='datetimepicker_timeview']"));
147
148     //click time picker button
149
150     selectTime.click();
151
152     //get list of times
```

```
153
154     List<WebElement> allTime = driver.findElements(By.xpath("//div[@data-role='popup'][contains(@style,'display:
155 block')]/ul//li[@role='option']"));
156
157
158
159     dateTime = dateTime.split(" ")[1]+" "+dateTime.split(" ")[2];
160
161     //select correct time
162
163     for (WebElement webElement : allTime) {
164
165         if(webElement.getText().equalsIgnoreCase(dateTime))
166
167             {
168
169                 webElement.click();
170
171             }
172
173         }
174
175     }
176 }
```

## Handling Date Time Picker using Selenium

Output will be like



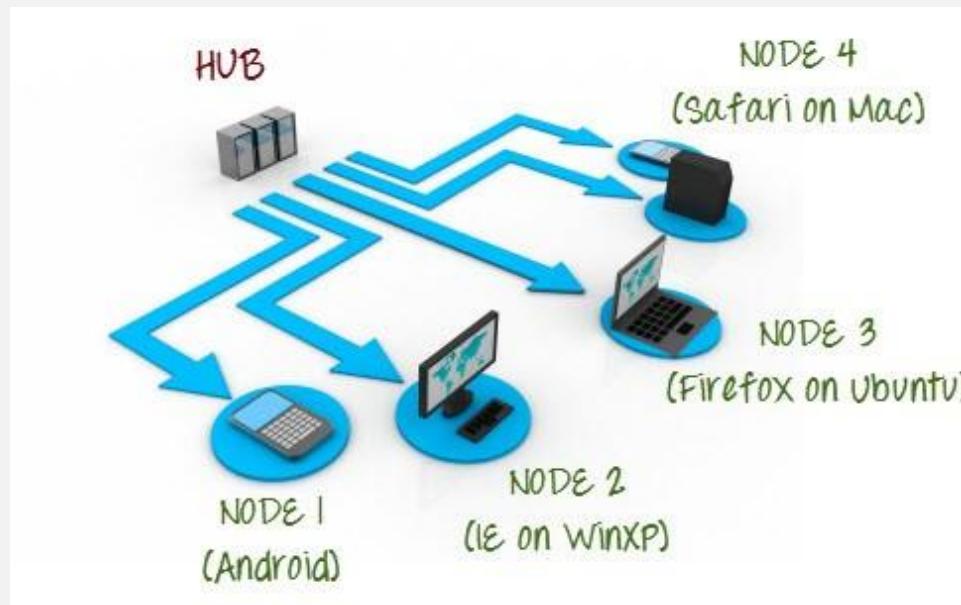
# Introduction to Selenium Grid

## Introduction to Selenium Grid

### What is Selenium Grid?

Selenium Grid is a part of the Selenium Suite that specializes on running multiple tests across different browsers, operating systems, and machines in parallel.

Selenium Grid has 2 versions - the older Grid 1 and the newer Grid 2. We will only focus on Grid 2 because Grid 1 is gradually being deprecated by the Selenium Team.



Selenium Grid uses a hub-node concept where you only run the test on a single machine called a **hub**, but the execution will be done by different machines called **nodes**.

## When to Use Selenium Grid?

You should use Selenium Grid when you want to do either one or both of following:

- **Run your tests against different browsers, operating systems, and machines all at the same time.** This will ensure that the application you are [testing](#) is fully compatible with a wide range of browser-O.S combinations.
- **Save time in the execution of your test suites.** If you set up Selenium Grid to run, say, 4 tests at a time, then you would be able to finish the whole suite around 4 times faster.

## Grid 1.0 vs. Grid 2.0

Following are the main differences between Selenium Grid 1 and 2.

### Grid 1

Selenium Grid 1 has its own remote control that is different from the Selenium RC server. They are two different programs.

You need to install and configure Apache Ant first before you can use Grid 1.

Can only support Selenium RC commands/scripts.

You can only automate one browser per remote control.

### Grid 2

Selenium Grid 2 is now bundled with the Selenium Server jar file

You do not need to install Apache Ant in Grid 2.

Can support both Selenium RC and WebDriver scripts.

One remote control can automate up to 5 browsers.

## **What is a Hub and Node?**

### **The Hub**

- The hub is the central point where you load your tests into.
- There should only be one hub in a grid.
- The hub is launched only on a single machine, say, a computer whose O.S is Windows 7 and whose browser is IE.
- The machine containing the hub is where the tests will be run, but you will see the browser being automated on the node.

### **The Nodes**

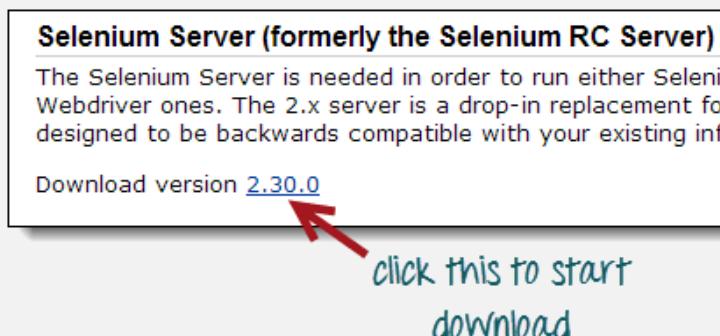
- Nodes are the Selenium instances that will execute the tests that you loaded on the hub.
- There can be one or more nodes in a grid.
- Nodes can be launched on multiple machines with different platforms and browsers.
- The machines running the nodes need not be the same platform as that of the hub.

## How to Install and Use Grid 2.0?

In this section, you will use 2 machines. The first machine will be the system that will run the hub while the other machine will run a node. For simplicity, let us call the machine where the hub runs as "Machine A" while the machine where the node runs will be "Machine B". It is also important to note their IP addresses. Let us say that Machine A has an IP address of 192.168.1.3 while Machine B has an IP of 192.168.1.4.

### Step 1

Download the Selenium Server by [here](#).



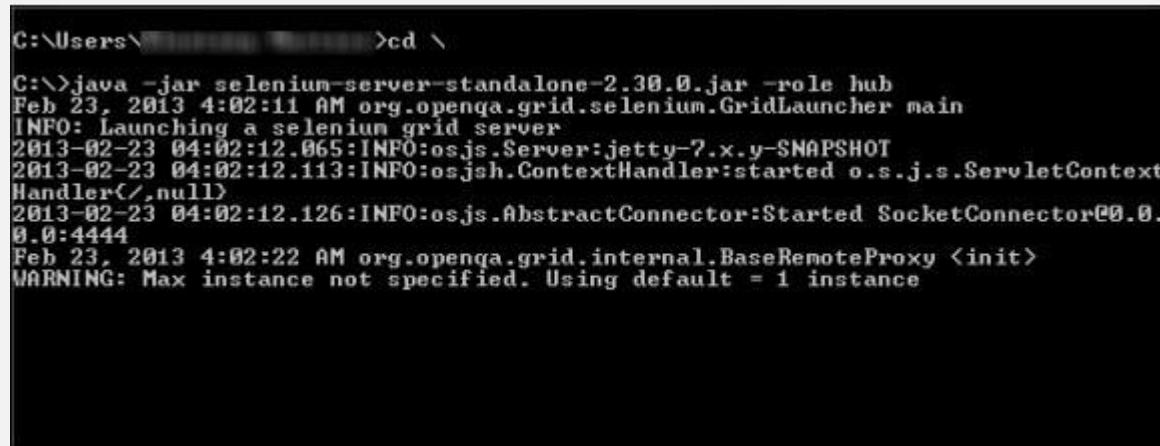
### Step 2

You can place the Selenium Server .jar file anywhere in your Hard Drive. But for the purpose of this tutorial, place it on the C drive of both Machine A and Machine B. After doing this, you are now done installing Selenium Grid. The following steps will launch the hub and the node.

### Step 3

- We are now going to launch a hub. Go to Machine A. Using the command prompt, navigate to the root of Machine A's -C drive, because that is the directory where we placed the Selenium Server.

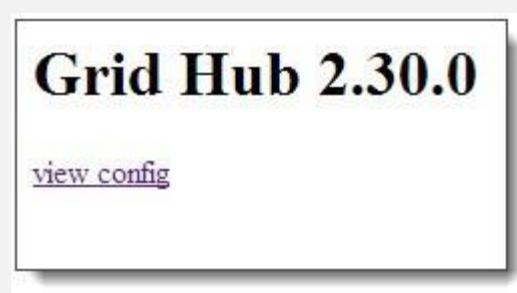
- On the command prompt, type **`java -jar selenium-server-standalone-2.30.0.jar -role hub`**
- The hub should successfully be launched. Your command prompt should look similar to the image below



```
C:\>java -jar selenium-server-standalone-2.30.0.jar -role hub
Feb 23, 2013 4:02:11 AM org.openqa.grid.selenium.GridLauncher main
INFO: Launching a selenium grid server
2013-02-23 04:02:12.065:INFO:osjs.Server:jetty-7.x.y-SNAPSHOT
2013-02-23 04:02:12.113:INFO:osjsh.ContextHandler:started o.s.j.s.ServletContext
Handler</, null>
2013-02-23 04:02:12.126:INFO:osjs.AbstractConnector:Started SocketConnector@0.0.
0.0:4444
Feb 23, 2013 4:02:22 AM org.openqa.grid.internal.BaseRemoteProxy <init>
WARNING: Max instance not specified. Using default = 1 instance
```

#### Step 4

Another way to verify whether the hub is running is by using a browser. Selenium Grid, by default, uses Machine A's port 4444 for its web interface. Simply open up a browser and go to <http://localhost:4444/grid/console>



Also, you can check if Machine B can access the hub's web interface by launching a browser there and going to where "iporhostnameofmachineA" should be the IP address or the hostname of the machine where the hub is running. Since Machine A's IP address is 192.168.1.3, then on the browser on Machine B you should type <http://192.168.1.3:4444/grid/console>

## Step 5

- Now that the hub is already set up, we are going to launch a node. Go to Machine B and launch a command prompt there.
- Navigate to the root of Drive C and type the code below. We used the IP address 192.168.1.3 because that is where the hub is running. We also used port 5566 though you may choose any free port number you desire.

```
C:\> java -jar selenium-server-standalone-2.30.0.jar -role  
webdriver -hub http://192.168.1.3:4444/grid/register -port 5566
```



IP address of the machine  
where the hub is running

- When you press Enter, your command prompt should be similar to the image below.

```
C:\>java -jar selenium-server-standalone-2.30.0.jar -role webdriver -hub http://192.168.1.3:4444/grid/register -port 5566
Feb 23, 2013 4:34:39 PM org.openqa.grid.selenium.GridLauncher main
INFO: Launching a selenium grid node
16:34:40.733 INFO - Java: Oracle Corporation 23.7-b01
16:34:40.733 INFO - OS: Windows XP 5.1 x86
16:34:40.733 INFO - v2.30.0, with Core v2.30.0. Built from revision dcief9c
16:34:40.874 INFO - RemoteWebDriver instances should connect to: http://127.0.0.1:5566/wd/hub
16:34:40.874 INFO - Version Jetty/5.1.x
16:34:40.874 INFO - Started HttpContext[/selenium-server/driver,/selenium-server/driver]
16:34:40.889 INFO - Started HttpContext[/selenium-server,/selenium-server]
16:34:40.889 INFO - Started HttpContext[/,/]
16:34:40.889 INFO - Started org.openqa.jetty.servlet.ServletHandler@ed3512

16:34:40.889 INFO - Started HttpContext[/wd,/wd]
16:34:40.905 INFO - Started SocketListener on 0.0.0.0:5566
16:34:40.905 INFO - Started org.openqa.jetty.Server@1f77497
16:34:40.905 INFO - using the json request : {"class":"org.openqa.grid.common.RegistrationRequest", "capabilities":[{"platform":"XP", "seleniumProtocol":"WebDriver", "browserName":"firefox", "maxInstances":5}, {"platform":"XP", "seleniumProtocol":"WebDriver", "browserName":"chrome", "maxInstances":5}, {"platform":"WINDOWS", "seleniumProtocol":"WebDriver", "browserName":"internet explorer", "maxInstances":1}], "configuration":{"port":5566, "register":true, "host":"192.168.1.4", "proxy":"org.openqa.grid.selenium.proxy.DefaultRemoteProxy", "maxSession":5, "role":"webdriver", "hubHost":"192.168.1.3", "registerCycle":5000, "hub":"http://192.168.1.3:4444/grid/register", "hubPort":4444, "url":"http://192.168.1.4:5566", "remoteHost":"http://192.168.1.4:5566"}}
16:34:40.905 INFO - starting auto register thread. Will try to register every 500 ms.
16:34:40.905 INFO - Registering the node to hub :http://192.168.1.3:4444/grid/register
16:34:46.171 INFO - Executing: org.openqa.selenium.remote.server.handler.Status@ie5a771 at URL: /status
16:34:46.186 INFO - Done: /status
```

## Step 6

Go to the Selenium Grid web interface and refresh the page. You should see something like this.

## Grid Hub 2.30.0

DefaultRemoteProxy

listening on http://192.168.1.4:5566

test session time out after 300 sec.

Supports up to 5 concurrent tests from:



[view config](#)

At this point, you have already configured a simple grid. You are now ready to run a test remotely on Machine B.

## Designing Test Scripts That Can Run on the Grid

To design test scripts that will run on the grid, we need to use **DesiredCapabilites** and the **RemoteWebDriver** objects.

- **DesiredCapabilites** is used to set the type of **browser** and **OS** that we will automate
- **RemoteWebDriver** is used to set which node (or machine) that our test will run against.

## Using the DesiredCapabilites Object

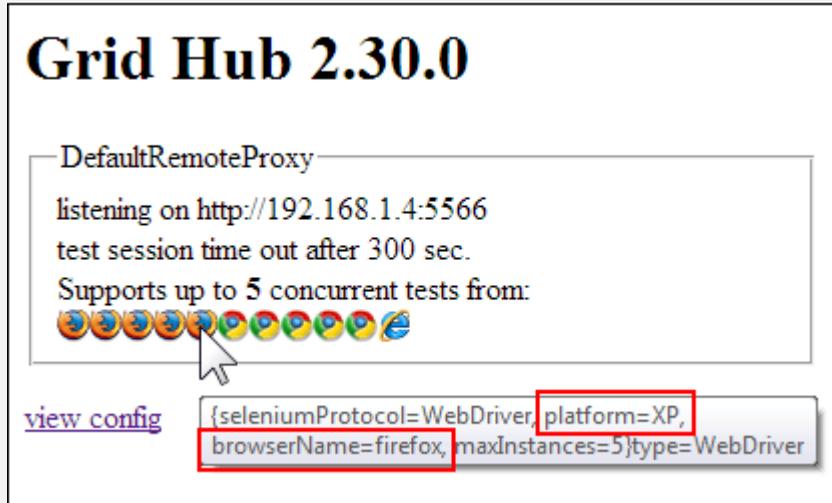
To use the **DesiredCapabilites** object, you must first import this package

To use the **RemoteWebDriver** object, you must import these packages.

```
import org.openqa.selenium.remote.DesiredCapabilities;  
import java.net.MalformedURLException;  
import java.net.URL;  
import org.openqa.selenium.remote.RemoteWebDriver;
```

## Using the RemoteWebDriver Object

Go to the Grid's web interface and hover on an image of the browser that you want to automate. Take note of the **platform** and the **browserName** shown by the tooltip.



In this case, the platform is "XP" and the browserName is "firefox".

We will use the platform and the browserName in our WebDriver as shown below (of course you need to import the necessary packages first).

```
DesiredCapabilities capability = DesiredCapabilities.firefox();
capability.setBrowserName("firefox");
capability.setPlatform(Platform.XP);
```

Import the necessary packages for RemoteWebDriver and then pass the DesiredCapabilities object that we created above as a parameter for the RemoteWebDriver object.

We used `RemoteWebDriver` and not `FirefoxDriver`

```
 WebDriver driver;
driver = new RemoteWebDriver(
    new URL("http://192.168.1.4:5566/wd/hub"), capability);
```

IP address and port on Machine B

## Running a Sample Test Case on the Grid

Below is a simple WebDriver TestNG code that you can create in Eclipse on Machine A. Once you run it, automation will be performed on Machine B.

```
import org.openqa.selenium.*;
import org.openqa.selenium.remote.DesiredCapabilities;
import java.net.MalformedURLException;
import java.net.URL;
import org.openqa.selenium.remote.RemoteWebDriver;
import org.testng.Assert;
import org.testng.annotations.*;

public class Grid_2 {
    WebDriver driver;
    String baseUrl, nodeURL;

    @BeforeTest
    public void setUp() throws MalformedURLException {
        baseUrl = "http://newtours.demoaut.com/";
        nodeURL = "http://192.168.1.4:5566/wd/hub";
        DesiredCapabilities capability = DesiredCapabilities.firefox();
        capability.setBrowserName("firefox");
        capability.setPlatform(Platform.XP);
        driver = new RemoteWebDriver(new URL(nodeURL), capability);
    }

    @AfterTest
    public void afterTest() {
        driver.quit();
    }

    @Test
    public void simpleTest() {
        driver.get(baseUrl);
        Assert.assertEquals("Welcome: Mercury Tours", driver.getTitle());
    }
}
```

The test should pass.

All suites

## Default suite

**Info**

- C:\Users\...\_AppData\Local\Temp\testng-eclipse-1896183272\testing-customsuite.xml
- 1 test
- 0 groups
- Times
- Reporter output
- Ignored methods
- Chronological view

**Results**

- 1 method, 1 passed
- Passed methods (hide)
  - simpleTest

Methods in chronological order		
Grid_2		
setUp	0 ms	
simpleTest	9374 ms	
afterTest	11570 ms	

## Summary

- Selenium Grid is used to run multiple tests simultaneously in different browsers and platforms.
- Grid uses the hub-node concept.
  - The hub is the central point wherein you load your tests.
  - Nodes are the Selenium instances that will execute the tests that you loaded on the hub.
- To install Selenium Grid, you only need to download the Selenium Server jar file - the same file used in running Selenium RC tests.
- There are 2 ways to verify if the hub is running: one was through the command prompt, and the other was through a browser
- To run test scripts on the Grid, you should use the DesiredCapabilities and the RemoteWebDriver objects.
  - DesiredCapabilites is used to set the type of browser and OS that we will automate
  - RemoteWebDriver is used to set which node (or machine) that our test will run against.

# Menu Selection Example



## Menu Selection Example

Hover the mouse on a menu and click on a resulting dropdown menu option

This piece of code allows you find and click on a drop down menu resulting from hovering the mouse over a main menu option. The issue with a menu resulting from hovering the mouse is that the sub menu elements are not available for selection until the drop down menu appears. As long as focus remains on the menu, one can choose from the drop down.

In the program below, I have used the Actions class to initiate mouse movement. The steps taken are:

1. Load the website
2. Identify the main menu option that results in a drop down menu
3. Configure and move the mouse to this identified menu link
4. Wait for 5 seconds (this allows the drop down appear)
5. Locate and click on a link on the resulting drop down
6. Done!

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.interactions.Actions;

import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

public class HoverAndClickOnAMenu {
    public static void main(String[] args) {

        //initialize browser
        WebDriver driver = new FirefoxDriver();
        driver.get("http://www.ticketmaster.co.uk");
```

```
//locate the menu to hover over using its xpath  
WebElement menu = driver.findElement(By.xpath("//*[@id='music']]"));  
  
//Initiate mouse action using Actions class
```

```
Actions builder = new Actions(driver);

// move the mouse to the earlier identified menu option
builder.moveToElement(menu).build().perform();

// wait for max of 5 seconds before proceeding. This allows sub menu appears properly before trying to
click on it

WebDriverWait wait = new WebDriverWait(driver, 5);
wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[@id='subNav_music_menu']/tbody/tr[2]/t
d[1]/a[1]"))); // until this submenu is found

//identify menu option from the resulting menu display and click
WebElement menuOption =
driver.findElement(By.xpath("//*[@id='subNav_music_menu']/tbody/tr[2]/td[1]/a[1]"));
menuOption.click();

}
```

# Test Your Mobile Web Apps with WebDriver



## Test Your Mobile Web Apps with WebDriver

Mobile testing has come a long way since the days when testing mobile web applications was mostly manual and took days to complete.

Selenium WebDriver is a browser automation tool that provides an elegant way of testing web applications. WebDriver makes it easy to write automated tests that ensure your site works correctly when viewed from an Android or iOS browser. For those of you new to WebDriver, here are a few basics about how it helps you test your web application. WebDriver tests are end-to-end tests that exercise a web application just like a real user would.

Now let's talk about mobile! WebDriver provides a touch API that allows the test to interact with the web page through finger taps, flicks, finger scrolls, and long presses. It can rotate the display and provides a friendly API to interact with HTML5 features such as local storage, session storage and application cache.

Mobile WebDrivers use the remote WebDriver server, following a client/server architecture. The client piece consists of the test code, while the server piece is the application that is installed on the device.

## Get Started

WebDriver for Android and iPhone can be installed following [these instructions](#). →  
<https://github.com/SeleniumHQ/selenium/wiki/WebDriver-For-Mobile-Browsers>

Once you've done that, you will be ready to write tests.

Let's start with a basic example using [www.google.com](http://www.google.com) to give you a taste of what's possible. The test below opens [www.google.com](http://www.google.com) on Android and issues a query for "weather in san francisco".

The test will verify that Google returns search results and that the first result returned is giving the weather in San Francisco.

```
public void testGoogleCanGiveWeatherResults() {  
  
    // Create a WebDriver instance with the activity in which we want the test to  
    // run. WebDriver driver = new AndroidDriver(getActivity());  
  
    // Let's open a web page  
    driver.get("http://www.google.com");  
}  
  
// Lookup for the search box by its name  
  
WebElement searchBar = driver.findElement(By.name("q"));  
  
// Enter a search query and submit  
  
searchBar.sendKeys("weather in san francisco"); searchBar.submit();  
  
// Making sure that Google shows 11 results  
  
WebElement resultSection = driver.findElement(By.id("ires"));  
  
List<WebElement> searchResults =  
resultSection.findElements(By.tagName("li")); assertEquals(11,  
searchResults.size());  
  
// Let's ensure that the first result shown is the weather  
widget WebElement weatherWidget = searchResults.get(0);  
  
assertTrue(weatherWidget.getText().contains("Weather for San Francisco, CA"));  
}
```

Now let's see our test in action! When you launch your test through your favorite IDE or using the command line, WebDriver will bring up a WebView in the foreground allowing you to see your web application as the test code is executing. You will see [www.google.com](http://www.google.com) loading, and the search query being typed in the search box.



We mentioned above that the WebDriver supports creating advanced gestures to interact with the device. Let's use WebDriver to throw an image across the screen by flicking horizontally, and ensure that the next image in the gallery is displayed.

```
WebElement toFlick = driver.findElement(By.id("image"));

// 400 pixels left at normal speed

Action flick = getBuilder(driver).flick(toFlick, 0, -400, FlickAction.SPEED_NORMAL)      .build();
flick.perform();

WebElement secondImage = driver.findElement("secondImage");

assertTrue(secondImage.isDisplayed());
```

```
assertEquals(landscapeSize, secondImage.getSize()) ((Rotatable)
driver).rotate(ScreenOrientation.PORTRAIT); assertEquals(portraitSize, secondImage.getSize());
```

Let's take a look at the local storage on the device, and ensure that the web application has set some key/value pairs.

```
// Get a handle on the local storage object
LocalStorage local = ((WebStorage) driver).getLocalStorage();
// Ensure that the key "name" is mapped
assertEquals("testUser", local.getItem("name"));
```

What if your test reveals a bug? You can easily take a screenshot for help in future debugging:

```
File tempFile = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
```

## High Level Architecture

WebDriver has two main components: the server and the tests themselves. The server is an application that runs on the phone, tablet, emulator, or simulator and listens for incoming requests.

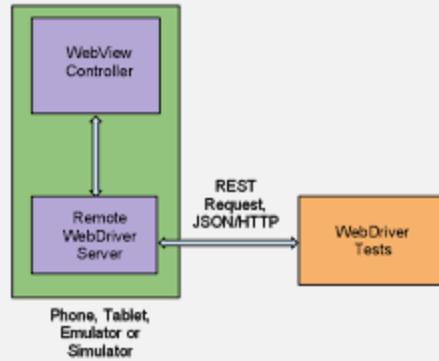
It runs the tests against a WebView (the rendering component of mobile Android and iOS) configured like the browsers. Your tests run on the client side, and can be written in any languages supported by WebDriver, including Java and Python.

The WebDriver tests communicate with the server by sending [RESTful JSON requests over HTTP](#). The tests and server pieces don't have to be on the same physical machine, although they can be.

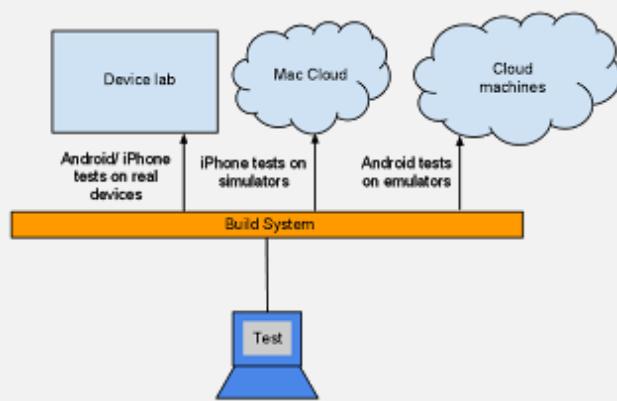
<https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol>

For Android you can also [run the tests using the Android test framework](#) instead of the remote WebDriver server.

<http://android-developers.blogspot.com/2011/10/introducing-android-webdriver.html>



## Infrastructure Setup



# Working with Ajax controls Using WebDriver



## Working with Ajax controls using WebDriver

AJAX stands for Asynchronous JavaScript and AJAX allows the Web page to retrieve small amounts of data from the server without reloading the entire page. In AJAX driven web applications, data is retrieved from server without refreshing the page.

When we perform any action on Ajax controls, using Wait commands will not work as the page is not actually refreshed here. Pausing the test execution using threads for a certain period of time is also not a good approach as web element might appear later or earlier than the stipulated period of time depending on the system's responsiveness, load or other uncontrolled factors of the moment, leads to test failures.

The best approach would be to wait for the required element in a dynamic period and then continue the test execution as soon as the element is found/visible.

This can be achieved with WebDriverWait in combination with ExpectedCondition , the best way to wait for an element dynamically, checking for the condition every second and continuing to the next command in the script as soon as the condition is met.

There are many methods which are available to use with `wait.until(ExpectedConditions.anyCondition)`; The below is the image for the number of methods which are available.

The below are the few which we use regularly when testing an application.

**Syntax:**

```
WebDriverWait wait = new WebDriverWait(driver, waitTime);  
wait.until(ExpectedConditions.presenceOfElementLocated(locator));
```

The above statement will check for the element presence on the DOM of a page. This does not necessarily mean that the element is visible.

**Syntax:**

```
WebDriverWait wait = new WebDriverWait(driver, waitTime);  
wait.until(ExpectedConditions.visibilityOfElementLocated(locator));
```

The above syntax will for the element present in the DOM of a page is visible.

Sometimes we may also need to check if the element is invisible or not. To do this we need use the below :

**Syntax:**

```
WebDriverWait wait = new WebDriverWait(driver, waitTime);  
wait.until(ExpectedConditions.invisibilityOfElementLocated(locator));
```

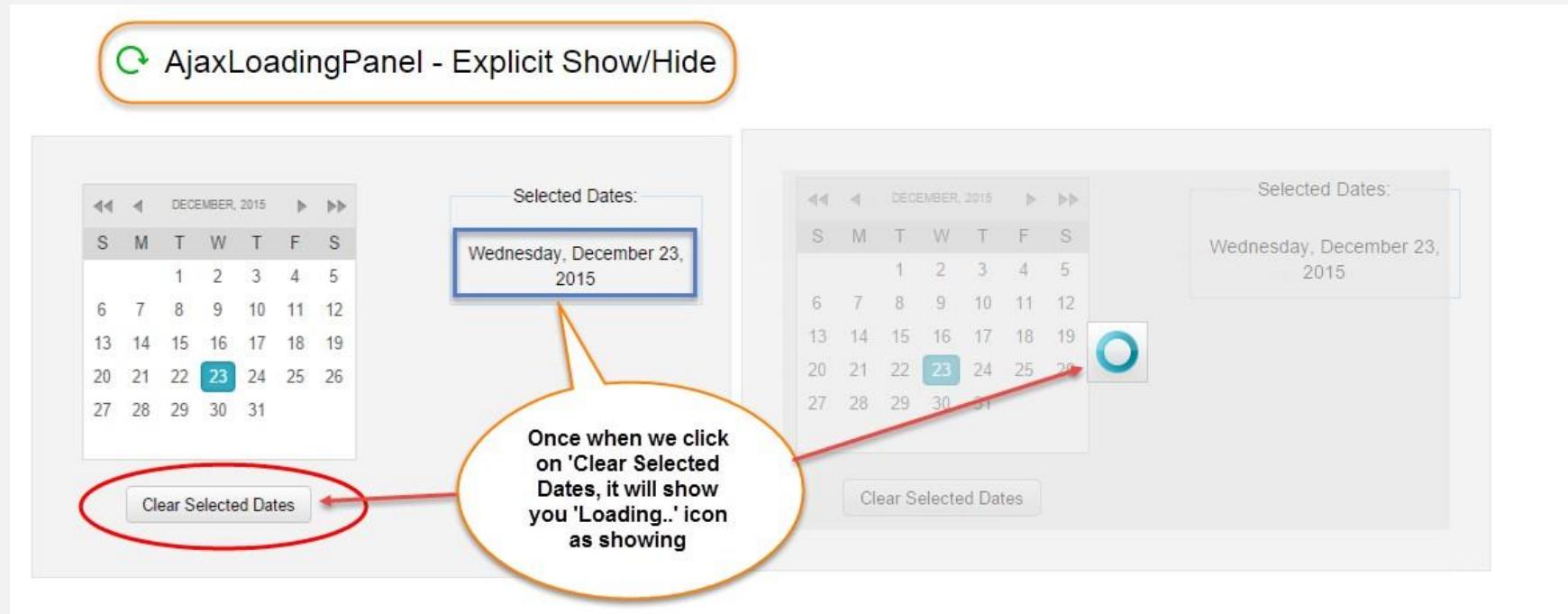
Some times you will get an exception as ""org.openqa.selenium.WebDriverException: Element is not clickable at point (611, 419). Other element would receive the click''. The below one is used to wait for the element to be clickable.

**Syntax:**

```
WebDriverWait wait = new WebDriverWait(driver, waitTime);  
wait.until(ExpectedConditions.elementToBeClickable(locator));
```

## Example

Below is the example program to handle Ajax controls with Wait statements. Based on the application loading time, we can increase or decrease the wait time. Look at the below image on which we perform operations:



```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.testng.Assert;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

public class AjaxExample {

    private String URL = "http://demos.telerik.com/aspnet-ajax/
ajax/examples/loadingpanel/explicitshowhide/defaultcs.aspx";

    WebDriver driver; WebDriverWait wait;

    @BeforeClass
    public void setUp() {
        driver=new FirefoxDriver(); driver.manage().window().maximize(); driver.navigate().to(URL);
    }

    @Test
    public void test_AjaxExample() {

        /*Wait for grid to appear*/
    }
}
```

```

By container = By.cssSelector(".demo-container"); wait = new WebDriverWait(driver, 5);
wait.until(ExpectedConditions.presenceOfElementLocated(container));

/*Get the text before performing an ajax call*/ WebElement noDatesTextElement =
driver.findElement(By.xpath("//div[@class='RadAjaxPanel']/span"));
String textBeforeAjaxCall = noDatesTextElement.getText().trim();

/*Click on the date*/ driver.findElement(By.linkText("1")).click();
/*Wait for loader to disappear */ By loader = By.className("raDiv");
wait.until(ExpectedConditions.invisibilityOfElementLocated(loader));

/*Get the text after ajax call*/ WebElement selectedDatesTextElement =
driver.findElement(By.xpath("//div[@class='RadAjaxPanel']/span"));
wait.until(ExpectedConditions.visibilityOf(selectedDatesTextElement)); String textAfterAjaxCall =
selectedDatesTextElement.getText().trim();

/*Verify both texts before ajax call and after ajax call text.*/
Assert.assertNotEquals(textBeforeAjaxCall, textAfterAjaxCall);

String expectedTextAfterAjaxCall = "Friday, January 01, 2016";
/*verify expected text with text updated after ajax call*/ Assert.assertEquals(textAfterAjaxCall,
expectedTextAfterAjaxCall);

}
}

```