



# **Selenium Documentation**

*Release 1.0*

**Selenium Project**

October 07, 2009



# CONTENTS

<b>1</b>	<b>A Personal Note to the Reader</b>	<b>3</b>
<b>2</b>	<b>Introducing Selenium</b>	<b>5</b>
2.1	To Automate or Not to Automate? That is the Question! . . . . .	5
2.2	Test Automation for Web Applications . . . . .	5
2.3	Introducing Selenium . . . . .	6
2.4	Selenium Components . . . . .	6
2.5	Supported Browsers . . . . .	7
2.6	Flexibility and Extensibility . . . . .	7
2.7	About this Book . . . . .	8
2.8	The Documentation Team . . . . .	9
<b>3</b>	<b>Selenium Basics</b>	<b>11</b>
3.1	Getting Started – Choosing Your Selenium Tool . . . . .	11
3.2	Introducing Selenium Commands . . . . .	11
3.3	Test Suites . . . . .	13
3.4	Commonly Used Selenium Commands . . . . .	14
3.5	Summary . . . . .	14
<b>4</b>	<b>Selenium-IDE</b>	<b>15</b>
4.1	Introduction . . . . .	15
4.2	Installing the IDE . . . . .	15
4.3	Opening the IDE . . . . .	18
4.4	IDE Features . . . . .	18
4.5	Building Test Cases . . . . .	21
4.6	Running Test Cases . . . . .	24
4.7	Using Base URL to Run Test Cases in Different Domains . . . . .	24
4.8	Debugging . . . . .	25
4.9	Writing a Test Suite . . . . .	27
4.10	User Extensions . . . . .	28
4.11	Format . . . . .	29
4.12	Executing Selenium-IDE Tests on Different Browsers . . . . .	29
4.13	Troubleshooting . . . . .	29
<b>5</b>	<b>Selenese Selenium Commands</b>	<b>35</b>
5.1	Verifying Page Elements . . . . .	35
5.2	Locating Elements . . . . .	37
5.3	Matching Text Patterns . . . . .	42

5.4	The “AndWait” Commands . . . . .	44
5.5	The waitFor Commands in AJAX applications . . . . .	44
5.6	Sequence of Evaluation and Flow Control . . . . .	44
5.7	Store Commands and Selenium Variables . . . . .	45
5.8	JavaScript and Selenese Parameters . . . . .	46
5.9	<i>echo</i> - The Selenese Print Command . . . . .	46
5.10	Alerts, Popups, and Multiple Windows . . . . .	47
<b>6</b>	<b>Selenium-RC</b>	<b>49</b>
6.1	Introduction . . . . .	49
6.2	How Selenium-RC Works . . . . .	49
6.3	Installation . . . . .	51
6.4	From Selenese to a Program . . . . .	53
6.5	Programming Your Test . . . . .	57
6.6	Learning the API . . . . .	61
6.7	Reporting Results . . . . .	62
6.8	Adding Some Spice to Your Tests . . . . .	64
6.9	Selenium-RC Architecture . . . . .	67
6.10	Handling HTTPS and Security Popups . . . . .	70
6.11	Server Options . . . . .	71
6.12	Specifying the Path to a Specific Browser . . . . .	75
6.13	Troubleshooting . . . . .	75
<b>7</b>	<b>Test Design Considerations</b>	<b>83</b>
7.1	Introducing Test Design Options . . . . .	83
7.2	Web Page Content – Static vs. Dynamic Pages . . . . .	83
7.3	Location Strategies . . . . .	85
7.4	UI Mapping with Selenium . . . . .	85
7.5	Bitmap Comparison . . . . .	86
7.6	Recovery From Failure . . . . .	86
7.7	Types of Tests . . . . .	87
7.8	Solving Common Web-App Problems . . . . .	87
7.9	Interpreting Test Results . . . . .	87
7.10	Organizing Your Test Scripts . . . . .	87
7.11	Organizing Your Test Suites . . . . .	87
7.12	Example Test Suites . . . . .	89
<b>8</b>	<b>Selenium-Grid</b>	<b>91</b>
<b>9</b>	<b>User-Extensions</b>	<b>93</b>
9.1	Introduction . . . . .	93
9.2	Actions . . . . .	93
9.3	Accessors/Assertions . . . . .	93
9.4	Locator Strategies . . . . .	94
9.5	Using User-Extensions With Selenium-IDE . . . . .	95
9.6	Using User-Extensions With Selenium RC . . . . .	95
<b>10</b>	<b>.NET client driver configuration</b>	<b>99</b>
<b>11</b>	<b>Java Client Driver Configuration</b>	<b>103</b>
11.1	Configuring Selenium-RC With Eclipse . . . . .	103
11.2	Configuring Selenium-RC With IntelliJ . . . . .	119

<b>12 Python Client Driver Configuration</b>	<b>131</b>
<b>13 Locating Techniques</b>	<b>135</b>
13.1 Useful XPATH patterns . . . . .	135
13.2 Starting to use CSS instead of XPATH . . . . .	135



Contents:



# A PERSONAL NOTE TO THE READER

Hello, and welcome to Selenium! The Documentation Team would like to welcome you, and to thank you for being interested in Selenium.

We have worked very, very hard on this document. Why? We absolutely believe this is the best tool for web-application testing. We feel its extensibility and flexibility, along with its tight integration with the browser, is unmatched by available proprietary tools. We are very excited to promote Selenium and, hopefully, to expand its user community. In short, we really want to “get the word out” about Selenium.

We truly believe you will be similarly excited once you learn how Selenium approaches test automation. It’s quite different from other tools. Whether you are brand-new to Selenium, or have been using it for awhile, we believe this documentation will truly help to spread the knowledge around. Also, we have aimed to write so that those completely new to test automation will be able to use this document as a stepping stone. No doubt, experienced users and “newbies” will benefit from our Selenium User’s Guide.

Please realize that this document is a work in progress. There are planned areas we haven’t written yet. However, we have written the beginning chapters first so newcomers can get started more smoothly. We have also already added some valuable information that more experienced users will appreciate. This document will be a ‘live’ document on the SeleniumHQ website where frequent updates will occur as we complete the additional planned documentation.

Thanks very much for reading.

– the Selenium Documentation Team



# INTRODUCING SELENIUM

## 2.1 To Automate or Not to Automate? That is the Question!

Is automation always advantageous? When should one decide to automate test cases?

It is **not** always advantageous to automate test cases. There are times when manual testing may be more appropriate. For instance, if the application's user interface will change considerably in the near future, then any automation would need to be rewritten. Also, sometimes there simply is not enough time to build test automation. For the short term, manual testing may be more effective. If an application has a very tight deadline, there is currently no test automation available, and it's imperative that the testing get done within that time frame, then manual testing is the best solution.

However, automation has specific advantages for improving the long-term efficiency of a software team's testing processes. Test automation supports:

- Frequent regression testing
- Rapid feedback to developers during the development process
- Virtually unlimited iterations of test case execution
- Customized reporting of application defects
- Support for Agile and eXtreme development methodologies
- Disciplined documentation of test cases
- Finding defects missed by manual testing

## 2.2 Test Automation for Web Applications

Many, perhaps most, software applications today are written as web-based applications to be run in an Internet browser. The effectiveness of testing these applications varies widely among companies and organizations. In an era of continuously improving software processes, such as eXtreme programming (XP) and Agile, it can be argued that disciplined testing and quality assurance practices are still underdeveloped in many organizations. Software testing is often conducted manually. At times, this is effective; however there are alternatives to manual testing that many organizations are unaware of, or lack the skills to perform. Utilizing these alternatives would in most cases greatly improve the efficiency of their software development by adding efficiencies to their testing.

Test automation is often the answer. Test automation means using a tool to run repeatable tests against the target application whenever necessary.

There are many advantages to test automation. Most are related to the repeatability of the tests and the speed at which the tests can be executed. There are a number of commercial and open source tools available for assisting with the development of test automation. Selenium is possibly the most widely-used open source solution. This user's guide will assist both new and experienced Selenium users in learning effective techniques in building test automation for web applications.

This guide introduces Selenium, teaches its most widely used features, and provides useful advice in best practices accumulated from the Selenium community. Many examples are provided. Also, technical information on the internal structure of Selenium and recommended uses of Selenium are provided as contributed by a consortium of experienced Selenium users. It is our hope that this guide will get additional new users excited about using Selenium for test automation. We hope this guide will assist in "getting the word out" that quality assurance and software testing have many options beyond what is currently practiced. We hope this user's guide and Selenium itself provide a valuable aid to boosting the reader's efficiency in his or her software testing processes.

## 2.3 Introducing Selenium

Selenium is a robust set of tools that supports rapid development of test automation for web-based applications. Selenium provides a rich set of testing functions specifically geared to the needs of testing of a web application. These operations are highly flexible, allowing many options for locating UI elements and comparing expected test results against actual application behavior.

One of Selenium's key features is the support for executing one's tests on multiple browser platforms.

## 2.4 Selenium Components

Selenium is composed of three major tools. Each one has a specific role in aiding the development of web application test automation.

### 2.4.1 Selenium-IDE

Selenium-IDE is the Integrated Development Environment for building Selenium test cases. It operates as a Firefox add-on and provides an easy-to-use interface for developing and running individual test cases or entire test suites. Selenium-IDE has a recording feature, which will keep account of user actions as they are performed and store them as a reusable script to play back. It also has a context menu (right-click) integrated with the Firefox browser, which allows the user to pick from a list of assertions and verifications for the selected location. Selenium-IDE also offers full editing of test cases for more precision and control.

Although Selenium-IDE is a Firefox only add-on, tests created in it can also be run against other browsers by using Selenium-RC and specifying the name of the test suite on the command line.

### 2.4.2 Selenium-RC (Remote Control)

Selenium-RC allows the test automation developer to use a programming language for maximum flexibility and extensibility in developing test logic. For instance, if the application under test returns a result set, and if the automated test program needs to run tests on each element in the result set, the programming language's iteration support can be used to iterate through the result set, calling Selenium commands to run tests on each item.

Selenium-RC provides an API (Application Programming Interface) and library for each of its supported languages: HTML, Java, C#, Perl, PHP, Python, and Ruby. This ability to use Selenium-RC with a high-level programming language to develop test cases also allows the automated testing to be integrated with a project's automated build environment.

### 2.4.3 Selenium-Grid

Selenium-Grid allows the Selenium-RC solution to scale for large test suites or test suites that must be run in multiple environments. With Selenium-Grid multiple instances of Selenium-RC are running on various operating system and browser configurations, each of these when launching register with a hub. When tests are sent to the hub they are then redirected to an available Selenium-RC, which will launch the browser and run the test. This allows for running tests in parallel, with the entire test suite theoretically taking only as long to run as the longest individual test.

## 2.5 Supported Browsers

Browser	Selenium-IDE	Selenium-RC	Operating Systems
Firefox 3	1.0 Beta-1 & 1.0 Beta-2: Record and playback tests	Start browser, run tests	Windows, Linux, Mac
Firefox 2	1.0 Beta-1: Record and playback tests	Start browser, run tests	Windows, Linux, Mac
IE 8		Under development	Windows
IE 7	Test execution only via Selenium-RC*	Start browser, run tests	Windows
Safari 3	Test execution only via Selenium-RC	Start browser, run tests	Mac
Safari 2	Test execution only via Selenium-RC	Start browser, run tests	Mac
Opera 9	Test execution only via Selenium-RC	Start browser, run tests	Windows, Linux, Mac
Opera 8	Test execution only via Selenium-RC	Start browser, run tests	Windows, Linux, Mac
Google Chrome	Test execution only via Selenium-RC(Windows)	Start browser, run tests	Windows
Others	Test execution only via Selenium-RC	Partial support possible**	As applicable

\* Tests developed on Firefox via Selenium-IDE can be executed on any other supported browser via a simple Selenium-RC command line.

\*\* Selenium-RC server can start any executable, but depending on browser security settings, there may be technical limitations that would limit certain features.

## 2.6 Flexibility and Extensibility

You'll find that Selenium is highly flexible. There are multiple ways in which one can add functionality to Selenium's framework to customize test automation for one's specific testing needs. This is, perhaps, Selenium's strongest characteristic when compared with proprietary test automation tools and other open source solutions. Selenium-RC support for multiple programming and scripting languages allows the

test writer to build any logic they need into their automated testing and to use a preferred programming or scripting language of one's choice.

Selenium-IDE allows for the addition of user-defined “user-extensions” for creating additional commands customized to the user's needs. Also, it is possible to re-configure how the Selenium-IDE generates its Selenium-RC code. This allows users to customize the generated code to fit in with their own test frameworks. Finally, Selenium is an Open Source project where code can be modified and enhancements can be submitted for contribution.

## 2.7 About this Book

This reference documentation targets both new users of Selenium and those who have been using Selenium and are seeking additional knowledge. It introduces the novice to Selenium test automation. We do not assume the reader has experience in testing beyond the basics.

The experienced Selenium user will also find this reference valuable. It compiles in one place a set of useful Selenium techniques and best practices by drawing from the knowledge of multiple experienced Selenium QA professionals.

The remaining chapters of the reference present:

***Selenium Basics*** Introduces Selenium by describing how to select the Selenium component most appropriate for your testing tasks. Also provides a general description of Selenium commands and syntax. This section allows you to get a general feel for how Selenium approaches test automation and helps you decide where to begin.

***Selenium-IDE*** Teaches how to build test cases using the Selenium Integrated Development Environment. This chapter also describes useful techniques for making your scripts more readable when interpreting defects caught by your Selenium tests. We explain how your test script can be “exported” to the programming language of your choice. Finally, this section describes some configurations available for extending and customizing how the Selenium-IDE supports test case development.

***Selenium Commands*** Describes a subset of the most useful Selenium commands in detail. This chapter shows what types of actions, verifications and assertions can be made against a web application.

***Selenium-RC*** Explains how to develop an automated test program using the Selenium-RC API. Many examples are presented in both a programming language and a scripting language. Also, the installation and setup of Selenium-RC is covered here. The various modes, or configurations, that Selenium-RC supports are described, along with their trade-offs and limitations. Architecture diagrams are provided to help illustrate these points. A number of solutions to problems which are often difficult for the new user, are described in this chapter. This includes handling Security Certificates, https requests, pop-ups and the opening of new windows.

***Test Design Considerations*** Presents many useful techniques for using Selenium efficiently. This includes scripting techniques and programming techniques for use with Selenium-RC. We cover examples of source code showing how to report defects in the application under test. We also cover techniques commonly asked about in the user forums such as how to implement data-driven tests (tests where one can vary the data between different test passes).

***Selenium-Grid*** *This chapter is not yet developed.*

***User extensions*** Presents all the information required for easily extending Selenium.

## 2.8 The Documentation Team

### 2.8.1 The Original Authors

- Dave Hunt
- Paul Grandjean
- Santiago Suarez Ordonez
- Tarun Kumar

The original authors who kickstarted this document are listed in alphabetical order. Each of us contributed significantly by taking a leadership role in specific areas. Each chapter originally had a primary author who kicked off the initial writing, but in the end, each of us made significant contributions to each chapter throughout the project.

### 2.8.2 Current Authors

- Mary Ann May-Pumphrey
- Peter Newhook

In addition to the original team members who are still involved (May '09), Mary Ann, and Peter have recently made major contributions. Their reviewing and editorial contributions proved invaluable. Mary Ann is actively writing new subsections and has provided editorial assistance throughout the document. Peter has provided assistance with restructuring our most difficult chapter and has provided valuable advice on topics to include. Their enthusiasm and dedication has been incredibly helpful. We hope they continue to be involved.

### 2.8.3 Acknowledgements

A huge special thanks goes to Patrick Lightbody. As an administrator of the SeleniumHQ website, his support has been invaluable. Patrick has helped us understand the Selenium community—our audience. He also set us up with everything we needed on the SeleniumHQ website for developing and releasing this user's guide. His enthusiasm and encouragement definitely helped drive this project. Also thanks goes to Andras Hatvani for his advice on publishing solutions, and to Amit Kumar for participating in our discussions and for assisting with reviewing the document.

And of course, we must *recognize the Selenium Developers*. They have truly designed an amazing tool. Without the vision of the original designers, and the continued efforts of the current developers, we would not have such a great tool to pass on to you, the reader.



# SELENIUM BASICS

## 3.1 Getting Started – Choosing Your Selenium Tool

Most people get started with Selenium-IDE. This is what we recommend. It's an easy way to get familiar with Selenium commands quickly. You can develop your first script in just a few minutes. Selenium-IDE is also very easy to install. See the *chapter on Selenium-IDE* for specifics.

You may also run your scripts from the Selenium-IDE. It's simple to use and is recommended for less-technical users. The IDE allows developing and running tests without the need for programming skills as required by Selenium-RC. The Selenium-IDE can serve as an excellent way to train junior-level employees in test automation. If one has an understanding of how to conduct manual testing of a website they can easily transition to using the Selenium-IDE for both, running and developing tests.

Some testing tasks are too complex though for the Selenium-IDE. When programming logic is required Selenium-RC must be used. For example, any tests requiring iteration, such as testing each element of a variable length list requires running the script from a programming language. Selenium-IDE does not support iteration or condition statements.

Finally, Selenium-Core is another way of running tests. One can run test scripts from a web-browser using the HTML interface *TestRunner.html*. This is the original method for running Selenium commands. It has limitations though. Similar to Selenium-IDE, it does not support iteration.

Selenium-Core also cannot switch between http and https protocols. Since the development of Selenium-IDE and Selenium-RC, more are using these tools rather than Selenium-Core. At the time of writing (April 09) it is still available and may be convenient for some. However, the Selenium community is encouraging the use Selenium-IDE and RC and discouraging the use of Selenium-Core. Support for Selenium-Core is becoming less available and it may even be deprecated in a future release.

## 3.2 Introducing Selenium Commands

### 3.2.1 Selenium Commands – Selenese

Selenium provides a rich set of commands for fully testing your web-app in virtually any way you may imagine. The command set is often called *selenese*. These commands essentially create a testing language.

In selenese, one can test the existence of UI elements based on their HTML tags, test for specific content, test for broken links, input fields, selection list options, submitting forms, and table data among other things. In addition Selenium commands support testing of window size, mouse position, alerts, Ajax functionality, pop up windows, event handling, and many other web-application features. The Command Reference (available at SeleniumHQ.org) lists all the available commands.

A *command* is what tells Selenium what to do. Selenium commands come in three “flavors”: **Actions**, **Accessors** and **Assertions**.

- **Actions** are commands that generally manipulate the state of the application. They do things like “click this link” and “select that option”. If an Action fails, or has an error, the execution of the current test is stopped.

Many Actions can be called with the “AndWait” suffix, e.g. “clickAndWait”. This suffix tells Selenium that the action will cause the browser to make a call to the server, and that Selenium should wait for a new page to load.

- **Accessors** examine the state of the application and store the results in variables, e.g. “storeTitle”. They are also used to automatically generate Assertions.
- **Assertions** are like Accessors, but they verify that the state of the application conforms to what is expected. Examples include “make sure the page title is X” and “verify that this checkbox is checked”.

All Selenium Assertions can be used in 3 modes: “assert”, “verify”, and “waitFor”. For example, you can “assertText”, “verifyText” and “waitForText”. When an “assert” fails, the test is aborted. When a “verify” fails, the test will continue execution, logging the failure. This allows a single “assert” to ensure that the application is on the correct page, followed by a bunch of “verify” assertions to test form field values, labels, etc.

“waitFor” commands wait for some condition to become true (which can be useful for testing Ajax applications). They will succeed immediately if the condition is already true. However, they will fail and halt the test if the condition does not become true within the current timeout setting (see the setTimeout action below).

### 3.2.2 Script Syntax

Selenium commands are simple, they consist of the command and two parameters. For example:

verifyText	//div//a[2]	Login
------------	-------------	-------

The parameters are not always required. It depends on the command. In some cases both are required, in others one parameter is required, and still in others the command may take no parameters at all. Here are a couple more examples:

goBackAndWait		
verifyTextPresent		Welcome to My Home Page
type	id=phone	(555) 666-7066
type	id=address1	\${myVariableAddress}

The command reference describes the parameter requirements for each command.

Parameters vary, however they are typically

- a *locator* for identifying a UI element within a page.
- a *text pattern* for verifying or asserting expected page content
- a *text pattern* or a selenium variable for entering text in an input field or for selecting an option from an option list.

Locators, text patterns, selenium variables, and the commands themselves are described in considerable detail in the section on Selenium Commands.

Selenium scripts that will be run from Selenium-IDE may be stored in an HTML text file format. This consists of an HTML table with three columns. The first column is used to identify the Selenium command, the second is a target and the final column contains a value. The second and third columns may not require values depending on the chosen Selenium command, but they should be present. Each table row represents a new Selenium command. Here is an example of a test that opens a page, asserts the page title and then verifies some content on the page:

```
<table>
  <tr><td>open</td><td></td><td>/download/</td></tr>
  <tr><td>assertTitle</td><td></td><td>Downloads</td></tr>
  <tr><td>verifyText</td><td>//h2</td><td>Downloads</td></tr>
</table>
```

Rendered as a table in a browser this would look like the following:

open		/download/
assertTitle		Downloads
verifyText	//h2	Downloads

The Selenese HTML syntax can be used to write and run tests without requiring knowledge of a programming language. With a basic knowledge of selenese and Selenium-IDE you can quickly produce and run testcases.

### 3.3 Test Suites

A test suite is a collection of tests. Often one will run all the tests in a test suite as one continuous batch-job.

When using Selenium-IDE, test suites also can be defined using a simple HTML file. The syntax again is simple. An HTML table defines a list of tests where each row defines the filesystem path to each test. An example tells it all.

```
<html>
<head>
<title>Test Suite Function Tests - Priority 1</title>
</head>
<body>
<table>
  <tr><td><b>Suite Of Tests</b></td></tr>
  <tr><td><a href= ". /Login.html " >Login</a></td></tr>
  <tr><td><a href= ". /SearchValues.html " >Test Searching for Values</a></td></tr>
  <tr><td><a href= ". /SaveValues.html " >Test Save</a></td></tr>
</table>
</body>
</html>
```

A file similar to this would allow running the tests all at once, one after another, from the Selenium-IDE.

Test suites can also be maintained when using Selenium-RC. This is done via programming and can be done a number of ways. Commonly Junit is used to maintain a test suite if one is using Selenium-RC with Java. Additionally, if C# is the chosen language, Nunit could be employed. If using an interpreted language like Python with Selenium-RC than some simple programming would be involved in setting up a test suite. Since the whole reason for using Sel-RC is to make use of programming logic for your testing this usually isn't a problem.

## 3.4 Commonly Used Selenium Commands

To conclude our introduction of Selenium, we'll show you a few typical Selenium commands. These are probably the most commonly used commands for building test.

**open** opens a page using a URL.

**click/clickAndWait** performs a click operation, and optionally waits for a new page to load.

**verifyTitle/assertTitle** verifies an expected page title.

**verifyTextPresent** verifies expected text is somewhere on the page.

**verifyElementPresent** verifies an expected UI element, as defined by its HTML tag, is present on the page.

**verifyText** verifies expected text and its corresponding HTML tag are present on the page.

**verifyTable** verifies a table's expected contents.

**waitForPageToLoad** pauses execution until an expected new page loads. Called automatically when `clickAndWait` is used.

**waitForElementPresent** pauses execution until an expected UI element, as defined by its HTML tag, is present on the page.

## 3.5 Summary

Now that you've seen an introduction to Selenium, you're ready to start writing your first scripts. We recommend beginning with the Selenium IDE and its context-sensitive, right-click, menu. This will allow you to get familiar with the most common Selenium commands quickly, and you can have a simple script done in just a minute or two. Chapter 3 gets you started and then guides you through all the features of the Selenium-IDE.

# SELENIUM-IDE

## 4.1 Introduction

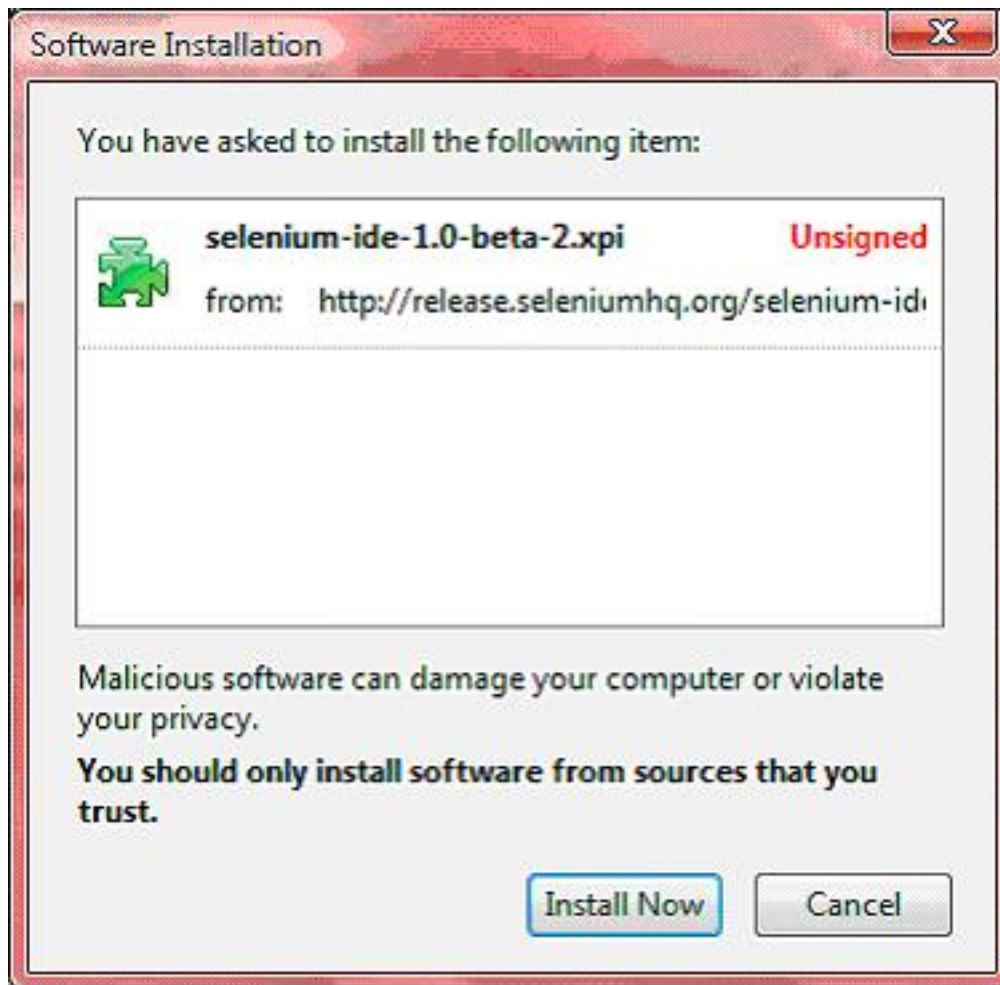
The Selenium-IDE (Integrated Development Environment) is the tool you use to develop your Selenium test cases. It's an easy-to-use Firefox plug-in and is generally the most efficient way to develop test cases. It also contains a context menu that allows you to first select a UI element from the browser's currently displayed page and then select from a list of Selenium commands with parameters pre-defined according to the context of the selected UI element. This is not only a time-saver, but also an excellent way of learning Selenium script syntax.

This chapter is all about the Selenium IDE and how to use it effectively.

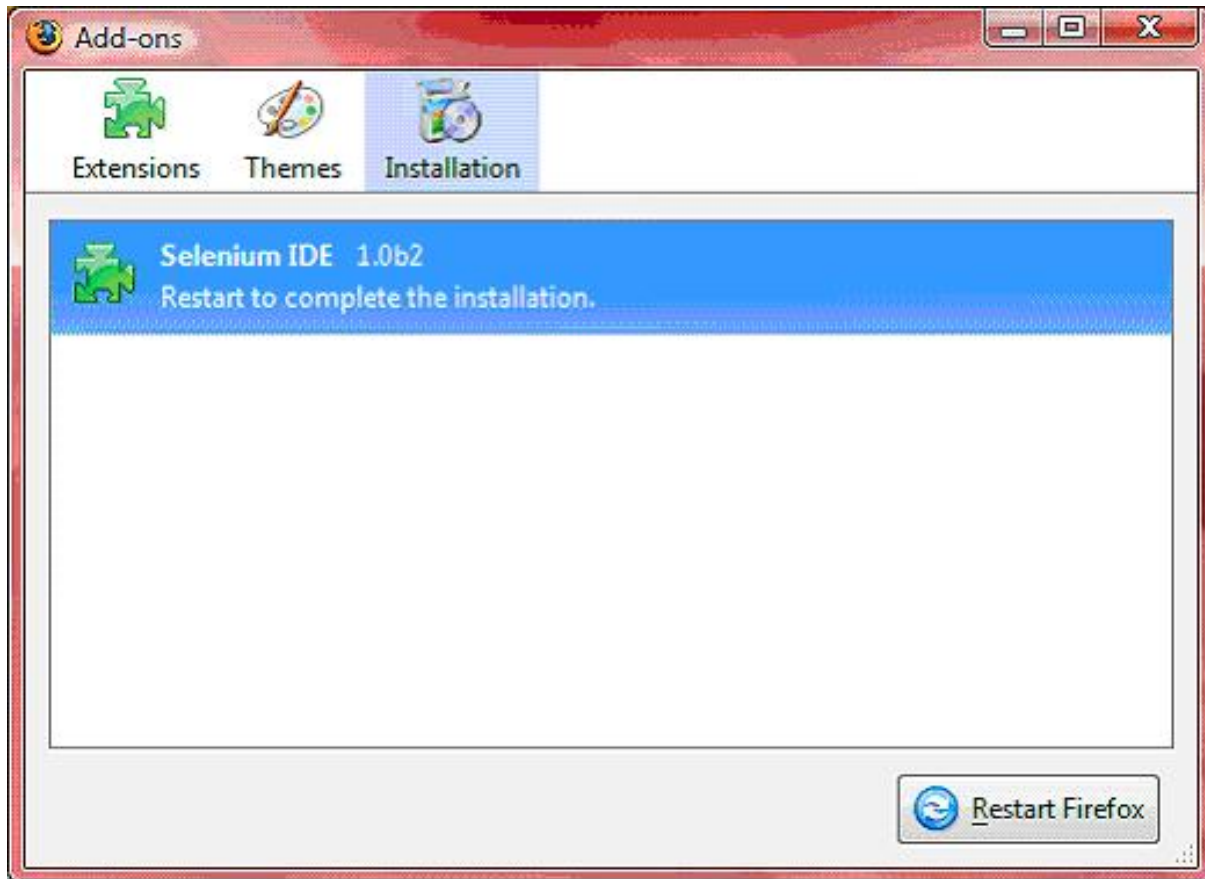
## 4.2 Installing the IDE

Using Firefox, first, download the IDE from the SeleniumHQ [downloads page](#)

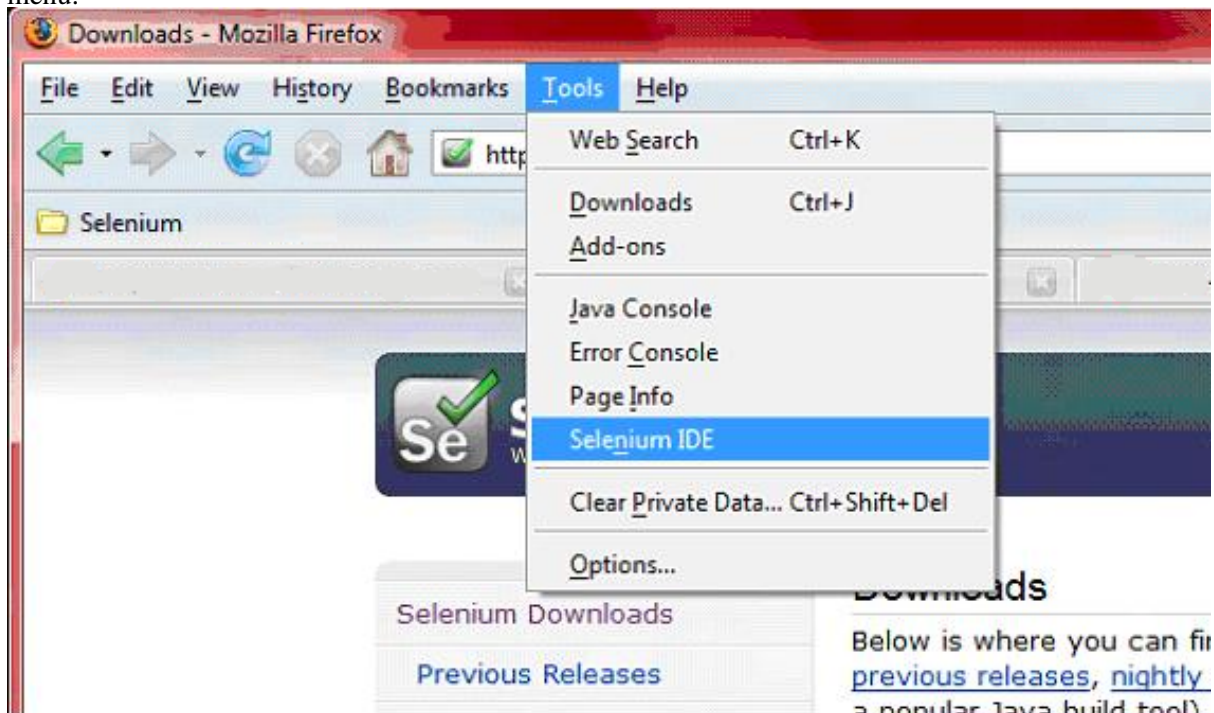
When downloading from Firefox, you'll be presented with the following window.



Select Install Now. The Firefox Add-ons window pops up, first showing a progress bar, and when the download is complete, displays the following.

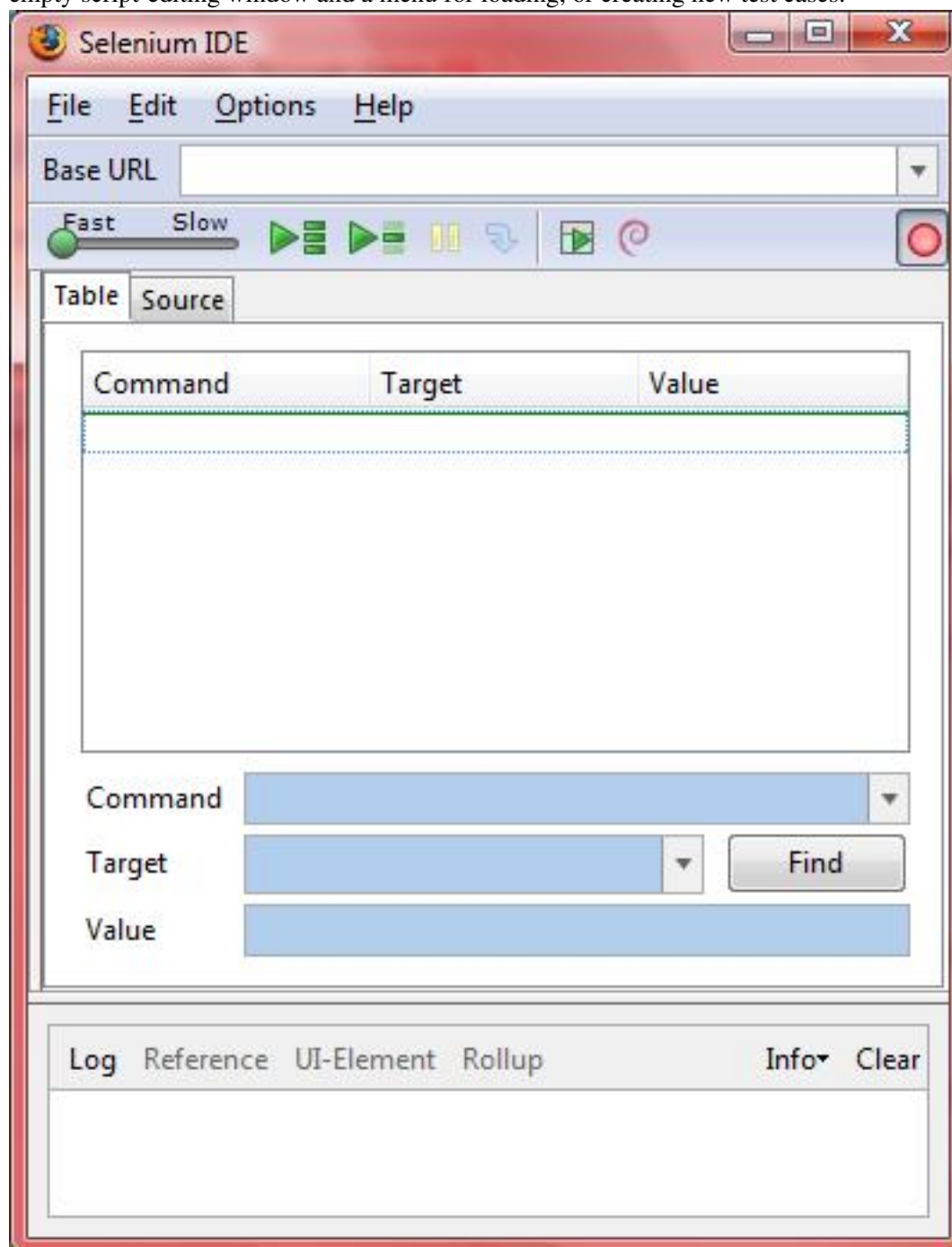


Restart Firefox. After Firefox reboots you will find the Selenium-IDE listed under the Firefox Tools menu.



## 4.3 Opening the IDE

To run the Selenium-IDE, simply select it from the Firefox Tools menu. It opens as follows with an empty script-editing window and a menu for loading, or creating new test cases.



## 4.4 IDE Features

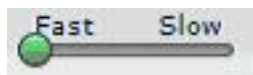
### 4.4.1 Menu Bar

The File menu allows you to create, open and save test case and test suite files. The Edit menu allows copy, paste, delete, undo and select all operations for editing the commands in your test case. The

Options menu allows the changing of settings. You can set the timeout value for certain commands, add user-defined user extensions to the base set of Selenium commands, and specify the format (language) used when saving your test cases. The Help menu is the standard Firefox Help menu; only one item on this menu—UI-Element Documentation—pertains to Selenium-IDE.

#### 4.4.2 Toolbar

The toolbar contains buttons for controlling the execution of your test cases, including a step feature for debugging your test cases. The right-most button, the one with the red-dot, is the record button.



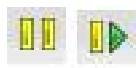
Speed Control: controls how fast your test case runs.



Run All: Runs the entire test suite when a test suite with multiple test cases is loaded.



Run: Runs the currently selected test. When only a single test is loaded this button and the Run All button have the same effect.



Pause/Resume: Allows stopping and re-starting of a running test case.



Step: Allows one to “step” through a test case by running it one command at a time. Use for debugging test cases.



TestRunner Mode: Allows you to run the test case in a browser loaded with the Selenium-Core TestRunner. The TestRunner is not commonly used now and is likely to be deprecated. This button is for evaluating test cases for backwards compatibility with the TestRunner. Most users will probably not need this button.



Apply Rollup Rules: This advanced feature allows repetitive sequences of Selenium commands to be grouped into a single action. Detailed documentation on rollup rules can be found in the UI-Element Documentation on the Help menu.



Record: Records the user’s browser actions.

#### 4.4.3 Test Case Pane

Your script is displayed in the test case pane. It has two tabs, one for displaying the command and their parameters in a readable “table” format.

Command	Target	Value
open	/Doc?id=dgtm5v7...	
selectWindow	name=null	
click	//a[@onclick="_h...	

The Source tab displays the test case in the native format in which the file will be stored. By default, this is HTML although it can be changed to a programming language such as Java or C#, or a scripting language like Python. See the Options menu for details. The Source view also allows one to edit the test case in its raw form, including copy, cut and paste operations.

The Command, Target, and Value entry fields display the currently selected command along with its parameters. These are entry fields where you can modify the currently selected command. The first parameter specified for a command in the Reference tab of the bottom pane always goes in the Target field. If a second parameter is specified by the Reference tab, it always goes in the Value field.

Command	<input type="text" value="selectWindow"/>	<input type="button" value="Find"/>
Target	<input type="text" value="name=null"/>	<input type="button" value="Find"/>
Value	<input type="text"/>	

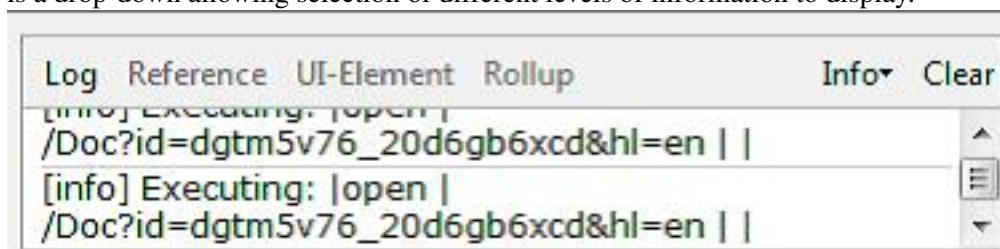
If you start typing in the Command field, a drop-down list will be populated based on the first characters you type; you can then select your desired command from the drop-down.

#### 4.4.4 Log/Reference/UI-Element/Rollup Pane

The bottom pane is used for four different functions—Log, Reference, UI-Element, and Rollup—depending on which tab is selected.

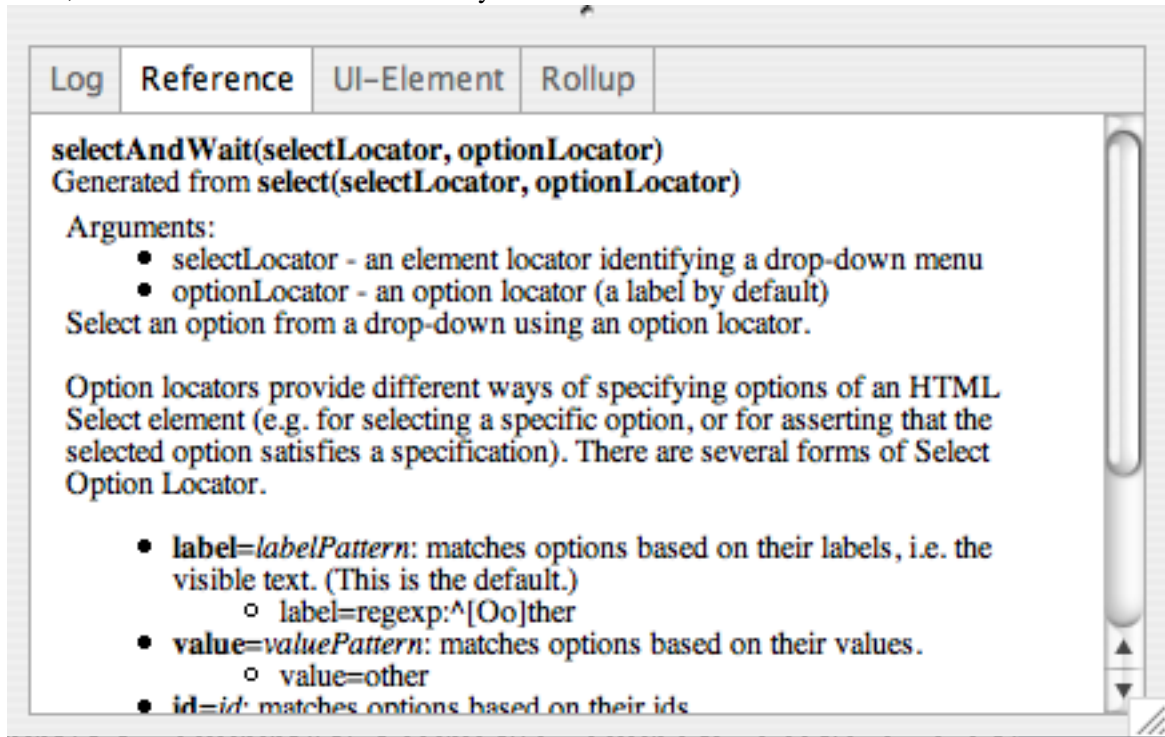
##### Log

When you run your test case, error messages and information messages showing the progress are displayed in this pane automatically, even if you do not first select the Log tab. These messages are often useful for test case debugging. Notice the Clear button for clearing the Log. Also notice the Info button is a drop-down allowing selection of different levels of information to display.



## Reference

The Reference tab is the default selection whenever you are entering or modifying Selenese commands and parameters in Table mode. In Table mode, the Reference pane will display documentation on the current command. When entering or modifying commands, whether from Table or Source mode, it is critically important to ensure that the parameters specified in the Target and Value fields match those specified in the parameter list specified in the Reference pane. The number of parameters provided must match the number specified, the order of parameters provided must match the order specified, and the type of parameters provided must match the type specified. If there is a mismatch in any of these three areas, the command will not run correctly.



While the Reference tab is invaluable as a quick reference, it is still often necessary to consult the Selenium [Reference](#) document.

## UI-Element and Rollup

Detailed information on these two panes (which cover advanced features) can be found in the UI-Element Documentation on the Help menu of Selenium-IDE.

## 4.5 Building Test Cases

There are three primary methods for developing test cases. Frequently, a test developer will require all three techniques.

### 4.5.1 Recording

Many first-time users begin by recording a test case from their interactions with a website. When Selenium-IDE is first opened, the record button is ON by default.

**Note:** This can be set to OFF as a default with an available user extension.

During recording, Selenium-IDE will automatically insert commands into your test case based on your actions. Typically, this will include:

- clicking a link - *click* or *clickAndWait* commands
- entering values - *type* command
- selecting options from a drop-down listbox - *select* command
- clicking checkboxes or radio buttons - *click* command

Here are some “gotchas” to be aware of:

- The *type* command may require clicking on some other area of the web page for it to record.
- Following a link usually records a *click* command. You will often need to change this to *clickAndWait* to ensure your test case pauses until the new page is completely loaded. Otherwise, your test case will continue running commands before the page has loaded all its UI elements. This will cause unexpected test case failures.

### 4.5.2 Adding Verifications and Asserts With the Context Menu

Your test cases will also need to check the properties of a web-page. This requires *assert* and *verify* commands. We won't describe the specifics of these commands here; that is in the chapter on “*Selenese*” *Selenium Commands*. Here we'll simply describe how to add them to your test case.

With Selenium-IDE recording, go to the browser displaying your test application and right click anywhere on the page. You will see a context menu showing *verify* and/or *assert* commands.

The first time you use Selenium, there may only be one Selenium command listed. As you use the IDE however, you will find additional commands will quickly be added to this menu. Selenium-IDE will attempt to predict what command, along with the parameters, you will need for a selected UI element on the current web-page.

Let's see how this works. Open a web-page of your choosing and select a block of text on the page. A paragraph or a heading will work fine. Now, right-click the selected text. The context menu should give you a *verifyTextPresent* command and the suggested parameter should be the text itself.

Also, notice the Show All Available Commands menu option. This shows many, many more commands, again, along with suggested parameters, for testing your currently selected UI element.

Try a few more UI elements. Try right-clicking an image, or a user control like a button or a checkbox. You may need to use Show All Available Commands to see options other than *verifyTextPresent*. Once you select these other options, the more commonly used ones will show up on the primary context menu. For example, selecting *verifyElementPresent* for an image should later cause that command to be available on the primary context menu the next time you select an image and right-click.

Again, these commands will be explained in detail in the chapter on Selenium commands. For now though, feel free to use the IDE to record and select commands into a test case and then run it. You can learn a lot about the Selenium commands simply by experimenting though the IDE.

### 4.5.3 Editing

#### Insert Command

##### Table View

Select the point in your test case where you want to insert the command. Right-click and select Insert Command. Now use the command editing text fields to enter your new command and its parameters.

##### Source View

Select the point in your test case where you want to insert the command, and enter the HTML tags needed to create a 3-column row containing the Command, first parameter (if one is required by the Command), and second parameter (again, if one is required). Be sure to save your test before switching back to Table view.

#### Insert Comment

Comments may be added to make your test case more readable. These comments are ignored when the test case is run.

In order to add vertical white space (one or more blank lines) in your tests, you must create empty comments. An empty command will cause an error during execution.

##### Table View

Select the point in your test case where you want to insert the comment. Right-click and select Insert Comment. Now use the Command field to enter the comment. Your comment will appear in purple font.

##### Source View

Select the point in your test case where you want to insert the comment. Add an HTML-style comment, i.e., `<!-- your comment here -->`.

#### Edit a Command or Comment

##### Table View

Simply select the line to be changed and edit it using the Command, Target, and Value fields.

##### Source View

Since Source view provides the equivalent of a WYSIWYG editor, simply modify which line you wish—command, parameter, or comment.

### 4.5.4 Opening and Saving a Test Case

The File=>Open, Save and Save As menu commands behave similarly to opening and saving files in most other programs. When you open an existing test case, Selenium-IDE displays its Selenium commands in the test case pane.

Test suite files can also be opened and saved via the File menu. However, such operations have their own menu entries near the bottom; the Open, Save, and Save As items are only for files.

**Note:** At the time of this writing, there's a bug, where at times, when the IDE is first opened and then you select File=>Open, nothing happens. If you see this, close down the IDE and restart it (you don't need to close the browser itself). This will fix the problem.

## 4.6 Running Test Cases

The IDE allows many options for running your test case. You can run a test case all at once, stop and start it, run it one line at a time, run a single command you are currently developing, and you can do a batch run of an entire test suite. Execution of test cases is very flexible in the IDE.

**Run a Test Case** Click the Run button to run the currently displayed test case.

**Run a Test Suite** Click the Run All button to run all the test cases in the currently loaded test suite.

**Stop and Start** The Pause button can be used to stop the test case while it is running. The icon of this button then changes to indicate the Resume button. To continue click Resume.

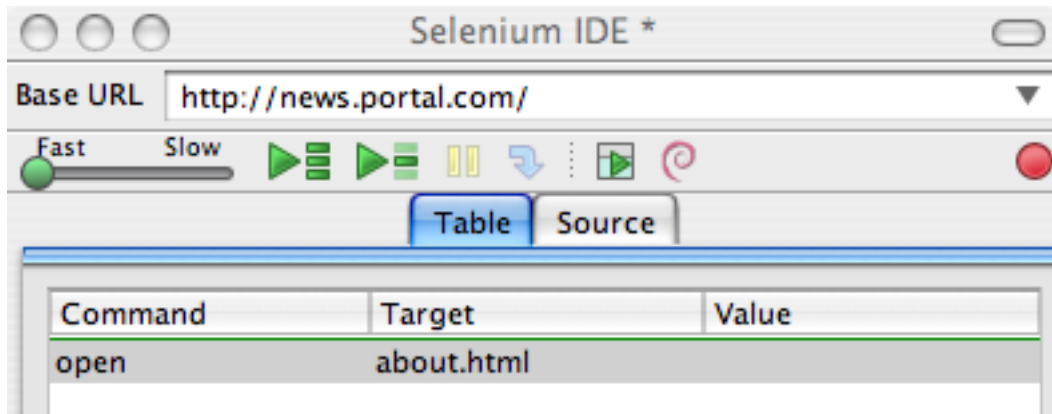
**Stop in the Middle** You can set a breakpoint in the test case to cause it to stop on a particular command. This is useful for debugging your test case. To set a breakpoint, select a command, right-click, and from the context menu select Toggle Breakpoint.

**Start from the Middle** You can tell the IDE to begin running from a specific command in the middle of the test case. This also is used for debugging. To set a startpoint, select a command, right-click, and from the context menu select Set/Clear Start Point.

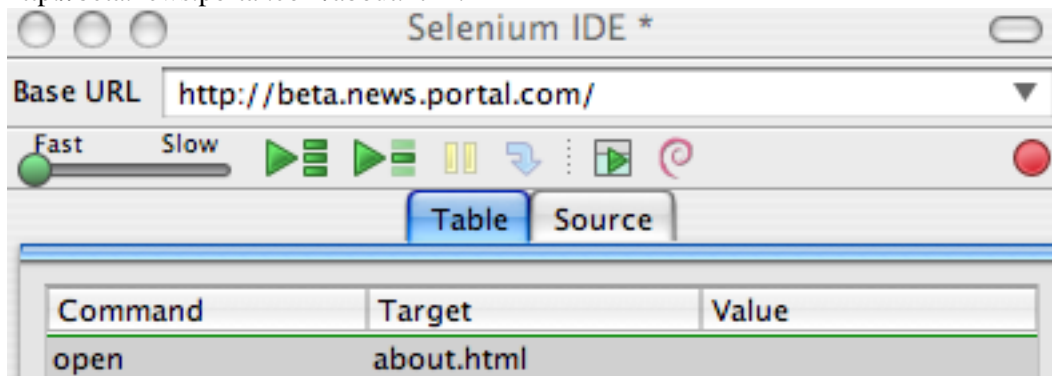
**Run Any Single Command** Double-click any single command to run it by itself. This is useful when writing a single command. It lets you immediately test a command you are constructing, when you are not sure if it is correct. You can double-click it to see if it runs correctly. This is also available from the context menu.

## 4.7 Using Base URL to Run Test Cases in Different Domains

The *Base URL* field at the top of the Selenium-IDE window is very useful for allowing test cases to be run across different domains. Suppose that a site named `http://news.portal.com` had an in-house beta site named `http://beta.news.portal.com`. Any test cases for these sites that begin with an *open* statement should specify a *relative URL* as the argument to *open* rather than an *absolute URL* (one starting with a protocol such as `http:` or `https:`). Selenium-IDE will then create an absolute URL by appending the *open* command's argument onto the end of the value of Base URL. For example, the test case below would be run against `http://news.portal.com/about.html`:



This same test case with a modified Base URL setting would be run against `http://beta.news.portal.com/about.html`:



## 4.8 Debugging

Debugging means finding and fixing errors in your test case. This is a normal part of test case development.

We won't teach debugging here as most new users to Selenium will already have some basic experience with debugging. If this is new to you, we recommend you ask one of the developers in your organization.

### 4.8.1 Breakpoints and Startpoints

The Sel-IDE supports the setting of breakpoints and the ability to start and stop the running of a test case, from any point within the test case. That is, one can run up to a specific command in the middle of the test case and inspect how the test case behaves at that point. To do this, set a breakpoint on the command just before the one to be examined.

To set a breakpoint, select a command, right-click, and from the context menu select *Toggle Breakpoint*. Then click the Run button to run your test case from the beginning up to the breakpoint.

It is also sometimes useful to run a test case from somewhere in the middle to the end of the test case or up to a breakpoint that follows the starting point. For example, suppose your test case first logs into the website and then performs a series of tests and you are trying to debug one of those tests. However, you only need to login once, but you need to keep rerunning your tests as you are developing them. You can login once, then run your test case from a startpoint placed after the login portion of your test case. That will prevent you from having to manually logout each time you rerun your test case.

To set a startpoint, select a command, right-click, and from the context menu select *Set/Clear Start Point*.

Then click the Run button to execute the test case beginning at that startpoint.

### 4.8.2 Stepping Through a Testcase

To execute a test case one command at a time (“step through” it), follow these steps:

1. Start the test case running with the Run button from the toolbar.



1. Immediately pause the executing test case with the Pause button.



1. Repeatedly select the Step button.



### 4.8.3 Find Button

The Find button is used to see which UI element on the currently displayed webpage (in the browser) is used in the currently selected Selenium command. This is useful when building a locator for a command’s first parameter (see the section on *locators* in the Selenium Commands chapter). It can be used with any command that must identify a UI element on a webpage, i.e. *click*, *clickAndWait*, *type*, and certain *assert* and *verify* commands, among others.

From Table view, select any command that has a locator parameter. Click the Find button. Now look on the webpage displayed in the Firefox browser. There should be a bright green rectangle enclosing the element specified by the locator parameter.

### 4.8.4 Page Source for Debugging

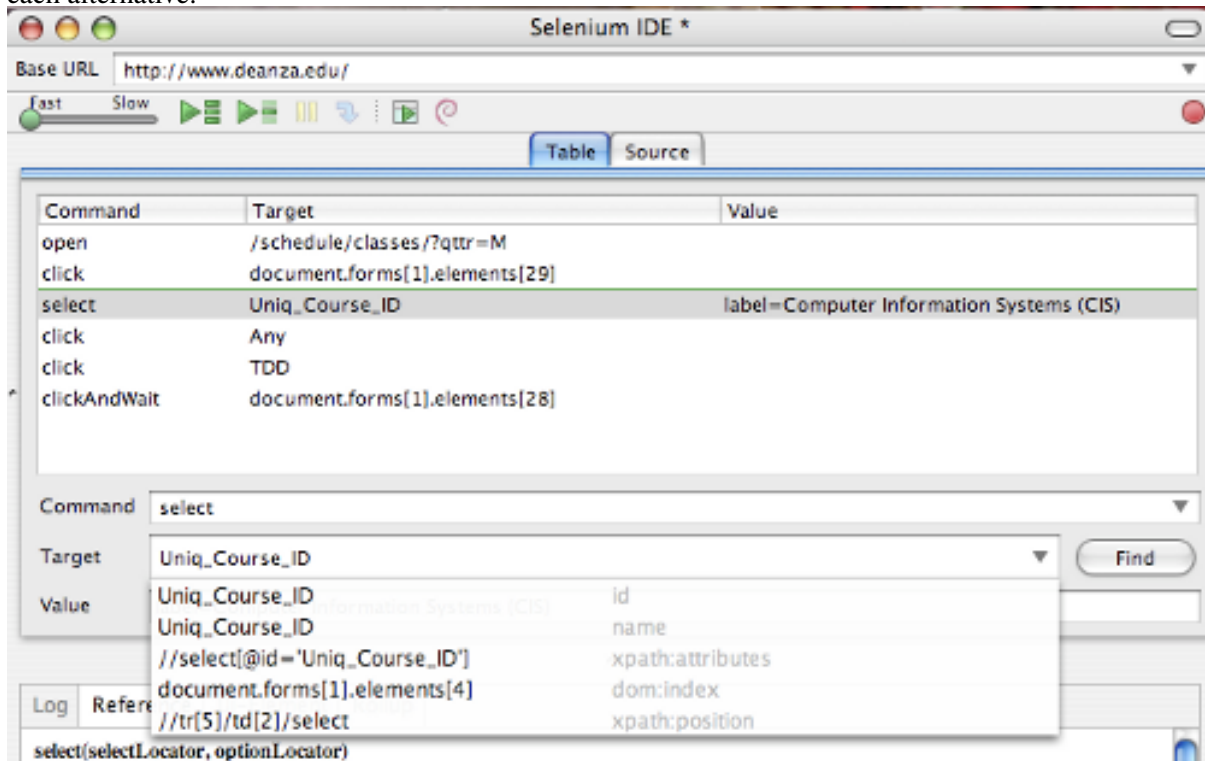
Often, when debugging a test case, you simply must look at the page source (the HTML for the webpage you’re trying to test) to determine a problem. Firefox makes this easy. Simply, right-click the webpage and select Page Source. The HTML opens in a separate window. Use its Search feature (Edit=>Find) to search for a keyword to find the HTML for the UI element you’re trying to test.

Alternatively, select just that portion of the webpage for which you want to see the source. Then right-click the webpage and select View Selection Source. In this case, the separate HTML window will contain just a small amount of source, with highlighting on the portion representing your selection.

### 4.8.5 Locator Assistance

Whenever Selenium-IDE records a locator-type argument, it stores additional information which allows the user to view other possible locator-type arguments that could be used instead. This feature can be very useful for learning more about locators, and is often needed to help one build a different type of locator than the type that was recorded.

This locator assistance is presented on the Selenium-IDE window as a drop-down list accessible at the right end of the Target field (only when the Target field contains a recorded locator-type argument). Below is a snapshot showing the contents of this drop-down for one command. Note that the first column of the drop-down provides alternative locators, whereas the second column indicates the type of each alternative.



## 4.9 Writing a Test Suite

A test suite is a collection of test cases which is displayed in the leftmost pane in the IDE. The test suite pane can be manually opened or closed via selecting a small dot halfway down the right edge of the pane (which is the left edge of the entire Selenium-IDE window if the pane is closed).

The test suite pane will be automatically opened when an existing test suite is opened *or* when the user selects the New Test Case item from the File menu. In the latter case, the new test case will appear immediately below the previous test case.

Selenium-IDE does not yet support loading pre-existing test cases into a test suite. Users who want to create or modify a test suite by adding pre-existing test cases must manually edit a test suite file.

A test suite file is an HTML file containing a one-column table. Each cell of each row in the `<tbody>` section contains a link to a test case. The example below is of a test suite containing four test cases:

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" >
  <title>Sample Selenium Test Suite</title>
</head>
<body>
  <table cellpadding="1" cellspacing="1" border="1" >
    <thead>
      <tr><td>Test Cases for De Anza A-Z Directory Links</td></tr>
    </thead>
```

```
<tbody>
  <tr><td><a href= ". /a.html " >A Links</a></td></tr>
  <tr><td><a href= ". /b.html " >B Links</a></td></tr>
  <tr><td><a href= ". /c.html " >C Links</a></td></tr>
  <tr><td><a href= ". /d.html " >D Links</a></td></tr>
</tbody>
</table>
</body>
</html>
```

**Note:** Test case files should not have to be co-located with the test suite file that invokes them. And on Mac OS and Linux systems, that is indeed the case. However, at the time of this writing, a bug prevents Windows users from being able to place the test cases elsewhere than with the test suite that invokes them.

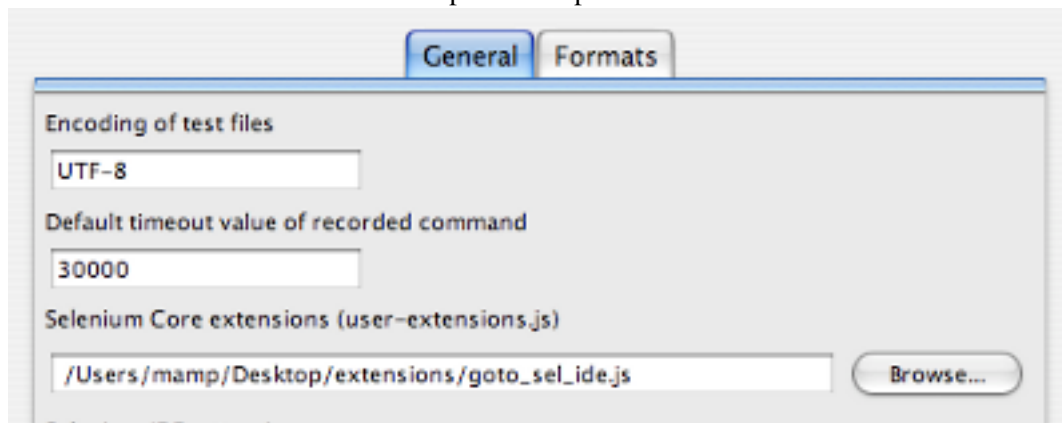
## 4.10 User Extensions

User extensions are JavaScript files that allow one to create his or her own customizations and features to add additional functionality. Often this is in the form of customized commands although this extensibility is not limited to additional commands.

There are a number of useful [extensions](#) created by users.

Perhaps the most popular of all Selenium-IDE extensions is one which provides flow control in the form of while loops and primitive conditionals. This extension is the [goto\\_sel\\_ide.js](#). For an example of how to use the functionality provided by this extension, look at the [page](#) created by its author.

To install this extension, put the pathname to its location on your computer in the **Selenium Core extensions** field of Selenium-IDE's Options=>Options=>General tab.



After selecting the **OK** button, you must close and reopen Selenium-IDE in order for the extensions file to be read. Any change you make to an extension will also require you to close and reopen Selenium-IDE.

Information on writing your own extensions can be found near the bottom of the Selenium [Reference](#) document.

## 4.11 Format

Format, under the Options menu, allows you to select a language for saving and displaying the test case. The default is HTML.

If you will be using Selenium-RC to run your test cases, this feature is used to translate your test case into a programming language. Select the language, i.e. Java, PHP, you will be using with Selenium-RC for developing your test programs. Then simply save the test case using File=>Save. Your test case will be translated into a series of functions in the language you choose. Essentially, program code supporting your test is generated for you by Selenium-IDE.

Also, note that if the generated code does not suit your needs, you can alter it by editing a configuration file which defines the generation process. Each supported language has configuration settings which are editable. This is under the Options=>Options=>Format tab.

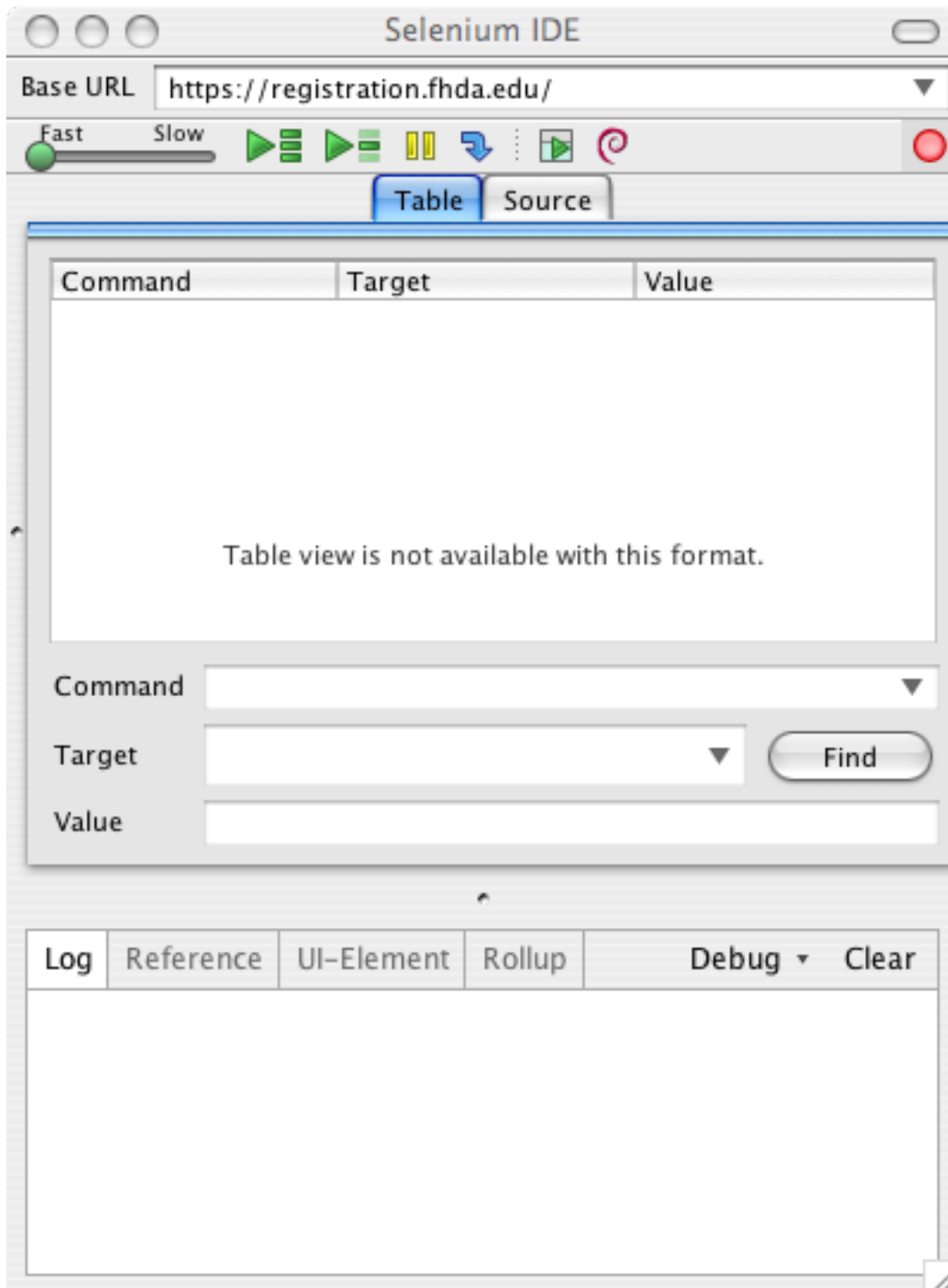
**Note:** At the time of this writing, this feature is not yet supported by the Selenium developers. However the author has altered the C# format in a limited manner and it has worked well.

## 4.12 Executing Selenium-IDE Tests on Different Browsers

While Selenium-IDE can only run tests against Firefox, tests developed with Selenium-IDE can be run against other browsers, using a simple command-line interface that invokes the Selenium-RC server. This topic is covered in the *Run Selenese tests* section on Selenium-RC chapter. The *-htmlSuite* command-line option is the particular feature of interest.

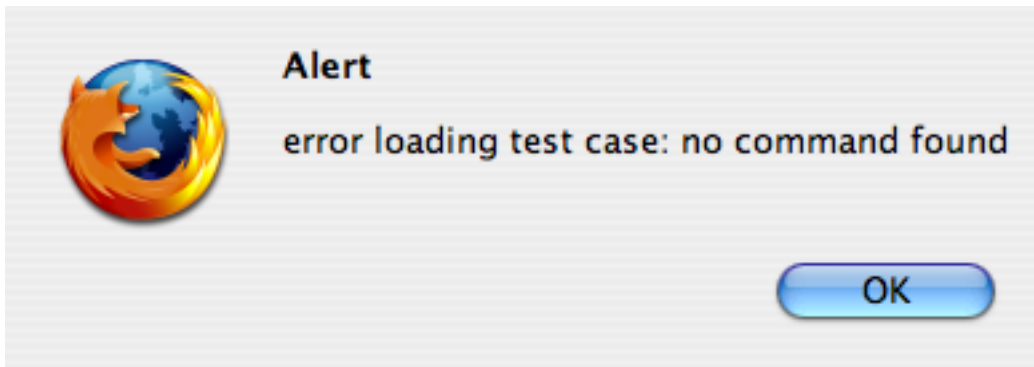
## 4.13 Troubleshooting

Below is a list of image/explanation pairs which describe frequent sources of problems with Selenium-IDE:

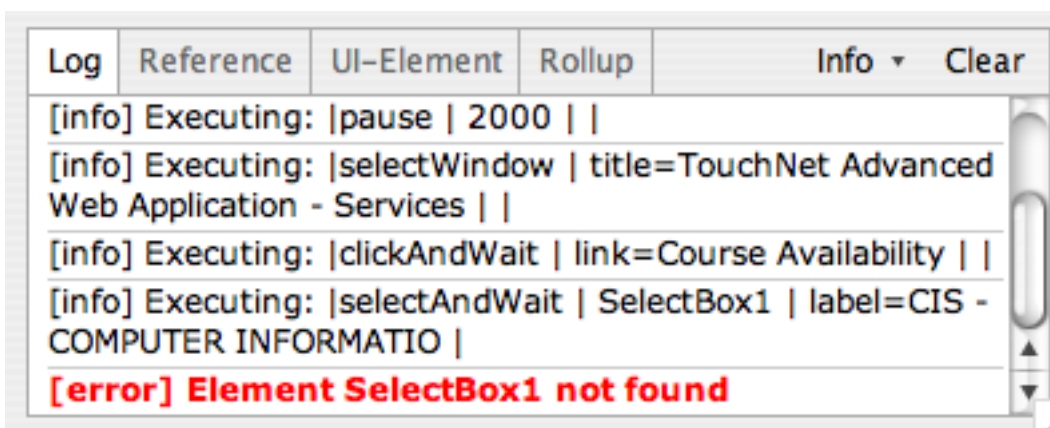


This problem occurs occasionally when Selenium IDE is first brought up. The solution is to close and reopen Selenium IDE. The bug has been filed as [SIDE-230](#).

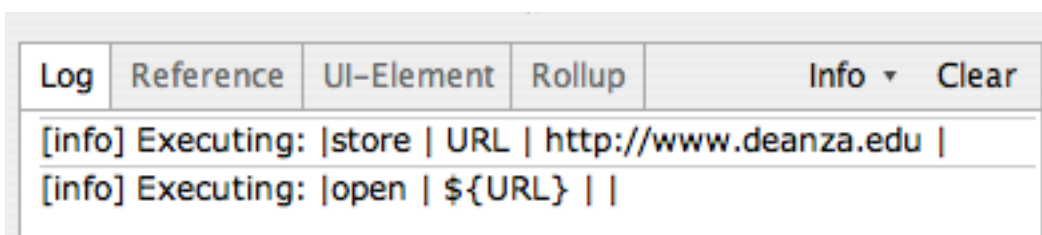
---



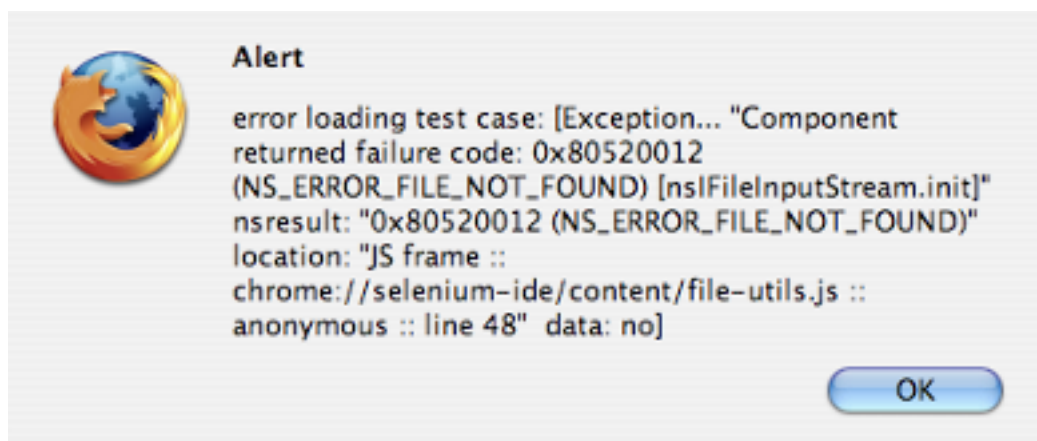
You've used **File=>Open** to try to open a test suite file. Use **File=>Open Test Suite** instead.



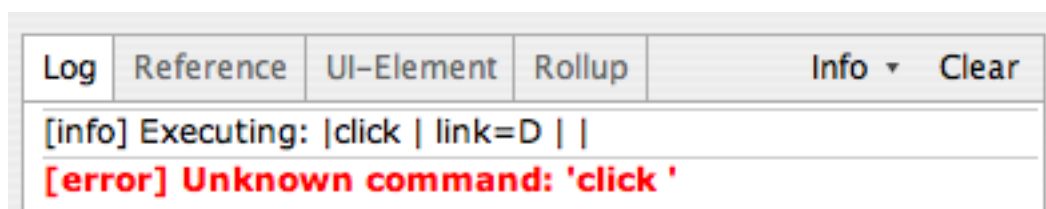
This type of **error** may indicate a timing problem, i.e., the element specified by a locator in your command wasn't fully loaded when the command was executed. Try putting a **pause 5000** before the command to determine whether the problem is indeed related to timing. If so, investigate using an appropriate **waitFor\*** or **\*AndWait** command immediately before the failing command.



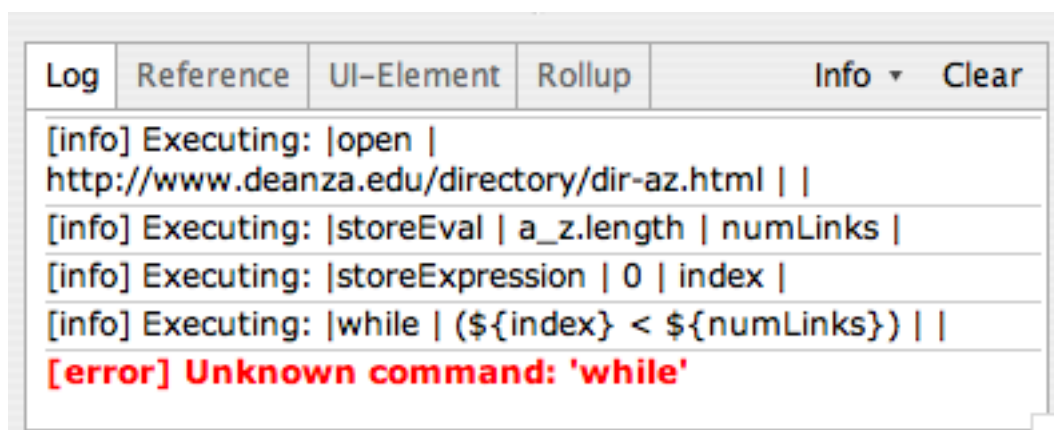
Whenever your attempt to use variable substitution fails as is the case for the **open** command above, it indicates that you haven't actually created the variable whose value you're trying to access. This is sometimes due to putting the variable in the **Value** field when it should be in the **Target** field or vice versa. In the example above, the two parameters for the **store** command have been erroneously placed in the reverse order of what is required. For any Selenese command, the first required parameter must go in the **Target** field, and the second required parameter (if one exists) must go in the **Value** field.



One of the test cases in your test suite cannot be found. Make sure that the test case is indeed located where the test suite indicates it is located. Also, make sure that your actual test case files have the .html extension both in their filenames, and in the test suite file where they are referenced.



Selenium-IDE is very *space-sensitive*! An extra space before or after a command will cause it to be unrecognizable.



Your extension file's contents have not been read by Selenium-IDE. Be sure you have specified the proper pathname to the extensions file via **Options=>Options=>General** in the **Selenium Core extensions** field. Also, Selenium-IDE must be restarted after any change to either an extensions file *or* to the contents of the **Selenium Core extensions** field.

The screenshot shows the Selenium IDE interface. At the top, there is a table with three columns: Command, Target, and Value. The table contains three rows: 'open' with target '/directory/dir-az.html', 'clickAndWait' with target 'link=Business and Computer Systems ...', and 'verifyTitle' with target 'Business/Computer Systems Division'. Below the table, there is a form for editing the selected command. The 'Command' dropdown is set to 'verifyTitle', the 'Target' text box contains 'Business/Computer Systems Division', and the 'Value' text box is empty. A 'Find' button is next to the target text box. At the bottom, there is a log window with columns for Log, Reference, UI-Element, Rollup, and Info. The log shows three info messages for the 'open', 'clickAndWait', and 'verifyTitle' commands. The final message is an error: **[error] Actual value 'Business/Computer Systems Division' did not match 'Business/Computer Systems Division'**. The error message shows that the actual value has two spaces between 'System' and 'Division', while the expected value has only one space.

This type of error message makes it appear that Selenium-IDE has generated a failure where there is none. However, Selenium-IDE is correct that the actual value does not match the value specified in such test cases. The problem is that the log file error messages collapse a series of two or more spaces into a single space, which is confusing. In the example above, note that the parameter for **verifyTitle** has two spaces between the words “System” and “Division.” The page’s actual title has only one space between these words. Thus, Selenium-IDE is correct to generate an error.



# SELENESE SELENIUM COMMANDS

Selenium commands, often called *selenese*, are the set of commands that run your tests. A sequence of these commands is a *test script*. Here we explain those commands in detail, and we present the many choices you have in testing your web application when using Selenium.

## 5.1 Verifying Page Elements

Verifying *UI elements* on a web page is probably the most common feature of your automated tests. Selenese allows multiple ways of checking for UI elements. It is important that you understand these different methods because these methods define what you are actually testing.

For example, will you test that...

1. an element is present somewhere on the page?
2. specific text is somewhere on the page?
3. specific text is at a specific location on the page?

For example, if you are testing a text heading, the text and its position at the top of the page are probably relevant for your test. If, however, you are testing for the existence of an image on the home page, and the web designers frequently change the specific image file along with its position on the page, then you only want to test that *an image* (as opposed to the specific image file) exists *somewhere on the page*.

### 5.1.1 Assertion or Verification?

Choosing between **assert** and **verify** comes down to convenience and management of failures. There's very little point checking that the first paragraph on the page is the correct one if your test has already failed when checking that the browser is displaying the expected page. If you're not on the correct page, you'll probably want to abort your test case so that you can investigate the cause and fix the issue(s) promptly. On the other hand, you may want to check many attributes of a page without aborting the test case on the first failure as this will allow you to review all failures on the page and take the appropriate action. Effectively an **assert** will fail the test and abort the current test case, whereas a **verify** will fail the test and continue to run the test case.

The best use of this feature is to logically group your test commands, and start each group with an **assert** followed by one or more **verify** test commands. An example follows:

open	/download/	
assertTitle	Downloads	
verifyText	//h2	Downloads
assertTable	1.2.1	Selenium IDE
verifyTable	1.2.2	June 3, 2008
verifyTable	1.2.3	1.0 beta 2

The above example first opens a page and then **asserts** that the correct page is loaded by comparing the title with the expected value. Only if this passes will the following command run and **verify** that the text is present in the expected location. The test case then **asserts** the first column in the second row of the first table contains the expected value, and only if this passed will the remaining cells in that row be **verified**.

### 5.1.2 verifyTextPresent

The command `verifyTextPresent` is used to verify *specific text exists somewhere on the page*. It takes a single argument—the text pattern to be verified. For example:

<code>verifyTextPresent</code>	<code>Marketing Analysis</code>	
--------------------------------	---------------------------------	--

This would cause Selenium to search for, and verify, that the text string “Marketing Analysis” appears somewhere on the page currently being tested. Use `verifyTextPresent` when you are interested in only the text itself being present on the page. Do not use this when you also need to test where the text occurs on the page.

### 5.1.3 verifyElementPresent

Use this command when you must test for the presence of a specific UI element, rather than its content. This verification does not check the text, only the HTML tag. One common use is to check for the presence of an image.

<code>verifyElementPresent</code>	<code>//div/p/img</code>	
-----------------------------------	--------------------------	--

This command verifies that an image, specified by the existence of an `<img>` HTML tag, is present on the page, and that it follows a `<div>` tag and a `<p>` tag. The first (and only) parameter is a *locator* for telling the Selenese command how to find the element. Locators are explained in the next section.

`verifyElementPresent` can be used to check the existence of any HTML tag within the page. One can check the existence of links, paragraphs, divisions `<div>`, etc. Here are a few more examples.

<code>verifyElementPresent</code>	<code>//div/p</code>	
<code>verifyElementPresent</code>	<code>//div/a</code>	
<code>verifyElementPresent</code>	<code>id=Login</code>	
<code>verifyElementPresent</code>	<code>link=Go to Marketing Research</code>	
<code>verifyElementPresent</code>	<code>//a[2]</code>	
<code>verifyElementPresent</code>	<code>//head/title</code>	

These examples illustrate the variety of ways a UI element may be tested. Again, locators are explained in the next section.

### 5.1.4 verifyText

Use `verifyText` when both the text and its UI element must be tested. `verifyText` must use a locator. If one chooses an *XPath* or *DOM* locator, one can verify that specific text appears at a specific location on the page relative to other UI components on the page.

verifyText	//table/tr/td/div/p	This is my text and it occurs right after the div inside the table.
------------	---------------------	---

## 5.2 Locating Elements

For many Selenium commands, a target is required. This target identifies an element in the content of the web application, and consists of the location strategy followed by the location in the format `locatorType=location`. The locator type can be omitted in many cases. The various locator types are explained below with examples for each.

### 5.2.1 Default Locators

You can choose to omit the locator type in the following situations:

- Locators starting with “document” will use the DOM locator strategy. See [Locating by DOM](#)
- Locators starting with “//” will use the XPath locator strategy. See [Locating by XPath](#).
- Locators that start with anything other than the above or a valid locator type will default to using the identifier locator strategy. See [Locating by Identifier](#).

### 5.2.2 Locating by Identifier

This is probably the most common method of locating elements and is the catch-all default when no recognised locator type is used. With this strategy, the first element with the `id` attribute value matching the location will be used. If no element has a matching `id` attribute, then the first element with an `name` attribute matching the location will be used.

For instance, your page source could have `id` and `name` attributes as follows:

```

1 <html>
2 <body>
3   <form id= "loginForm" >
4     <input name= "username" type= "text" />
5     <input name= "password" type= "password" />
6     <input name= "continue" type= "submit" value= "Login" />
7   </form>
8 </body>
9 </html>
```

The following locator strategies would return the elements from the HTML snippet above indicated by line number:

- `identifier=loginForm (3)`
- `identifier=username (4)`
- `identifier=continue (5)`
- `continue (5)`

Since the `identifier` type of locator is the default, the `identifier=` in the first three examples above is not necessary.

### 5.2.3 Locating by Id

This type of locator is more limited than the identifier locator type, but also more explicit. Use this when you know an element's id attribute.

```
1 <html>
2 <body>
3   <form id= "loginForm" >
4     <input name= "username" type= "text" />
5     <input name= "password" type= "password" />
6     <input name= "continue" type= "submit" value= "Login" />
7     <input name= "continue" type= "button" value= "Clear" />
8   </form>
9 </body>
10 </html>
```

- id=loginForm(3)

### 5.2.4 Locating by Name

The name locator type will locate the first element with a matching name attribute. If multiple elements have the same value for a name attribute, then you can use filters to further refine your location strategy. The default filter type is value (matching the value attribute).

```
1 <html>
2 <body>
3   <form id= "loginForm" >
4     <input name= "username" type= "text" />
5     <input name= "password" type= "password" />
6     <input name= "continue" type= "submit" value= "Login" />
7     <input name= "continue" type= "button" value= "Clear" />
8   </form>
9 </body>
10 </html>
```

- name=username(4)
- name=continue value=Clear(7)
- name=continue Clear(7)
- name=continue type=button(7)

**Note:** Unlike some types of XPath and DOM locators, the three types of locators above allow Selenium to test a UI element independent of its location on the page. So if the page structure and organization is altered, the test will still pass. One may or may not want to also test whether the page structure changes. In the case where web designers frequently alter the page, but its functionality must be regression tested, testing via id and name attributes, or really via any HTML property, becomes very important.

## 5.2.5 Locating by XPath

XPath is the language used for locating nodes in an XML document. As HTML can be an implementation of XML (XHTML), Selenium users can leverage this powerful language to target elements in their web applications. XPath extends beyond (as well as supporting) the simple methods of locating by id or name attributes, and opens up all sorts of new possibilities such as locating the third checkbox on the page.

One of the main reasons for using XPath is when you don't have a suitable id or name attribute for the element you wish to locate. You can use XPath to either locate the element in absolute terms (not advised), or relative to an element that does have an id or name attribute. XPath locators can also be used to specify elements via attributes other than id and name.

Absolute XPaths contain the location of all elements from the root (html) and as a result are likely to fail with only the slightest adjustment to the application. By finding a nearby element with an id or name attribute (ideally a parent element) you can locate your target element based on the relationship. This is much less likely to change and can make your tests more robust.

Since only `xpath` locators start with `"/"`, it is not necessary to include the `xpath=` label when specifying an XPath locator.

```

1 <html>
2 <body>
3 <form id= "loginForm" >
4 <input name= "username" type= "text" />
5 <input name= "password" type= "password" />
6 <input name= "continue" type= "submit" value= "Login" />
7 <input name= "continue" type= "button" value= "Clear" />
8 </form>
9 </body>
10 </html>

```

- `xpath=/html/body/form[1]` (3) - *Absolute path (would break if the HTML was changed only slightly)*
- `//form[1]` (3) - *First form element in the HTML*
- `xpath=//form[@id='loginForm']` (3) - *The form element with @id of 'loginForm'*
- `xpath=//form[input/@name='username']` (4) - *First form element with an input child element with @name of 'username'*
- `//input[@name='username']` (4) - *First input element with @name of 'username'*
- `//form[@id='loginForm']/input[1]` (4) - *First input child element of the form element with @id of 'loginForm'*
- `//input[@name='continue'][@type='button']` (7) - *Input with @name 'continue' and @type of 'button'*
- `//form[@id='loginForm']/input[4]` (7) - *Fourth input child element of the form element with @id of 'loginForm'*

These examples cover some basics, but in order to learn more, the following references are recommended:

- [W3Schools XPath Tutorial](#)
- [W3C XPath Recommendation](#)
- [XPath Tutorial](#) - with interactive examples.

There are also a couple of very useful Firefox Add-ons that can assist in discovering the XPath of an element:

- [XPath Checker](#) - suggests XPath and can be used to test XPath results.
- [Firebug](#) - XPath suggestions are just one of the many powerful features of this very useful add-on.

### 5.2.6 Locating Hyperlinks by Link Text

This is a simple method of locating a hyperlink in your web page by using the text of the link. If two links with the same text are present, then the first match will be used.

```
1 <html>
2 <body>
3 <p>Are you sure you want to do this?</p>
4 <a href= "continue.html" >Continue</a>
5 <a href= "cancel.html" >Cancel</a>
6 </body>
7 </html>
```

- `link=Continue` (4)
- `link=Cancel` (5)

### 5.2.7 Locating by DOM

The Document Object Model represents an HTML document and can be accessed using JavaScript. This location strategy takes JavaScript that evaluates to an element on the page, which can be simply the element's location using the hierarchical dotted notation.

Since only `dom` locators start with "document", it is not necessary to include the `dom=` label when specifying a `dom` locator.

```
1 <html>
2 <body>
3 <form id= "loginForm" >
4 <input name= "username" type= "text" />
5 <input name= "password" type= "password" />
6 <input name= "continue" type= "submit" value= "Login" />
7 <input name= "continue" type= "button" value= "Clear" />
8 </form>
9 </body>
10 </html>
```

- `dom=document.getElementById('loginForm')` (3)

- `dom=document.forms['loginForm']` (3)
- `dom=document.forms[0]` (3)
- `document.forms[0].username` (4)
- `document.forms[0].elements['username']` (4)
- `document.forms[0].elements[0]` (4)
- `document.forms[0].elements[3]` (7)

You can use Selenium itself as well as other sites and extensions to explore the DOM of your web application. A good reference exists on [W3Schools](#).

### 5.2.8 Locating by CSS

CSS (Cascading Style Sheets) is a language for describing the rendering of HTML and XML documents. CSS uses Selectors for binding style properties to elements in the document. These Selectors can be used by Selenium as another locating strategy.

```

1 <html>
2 <body>
3 <form id= "loginForm" >
4 <input class= "required" name= "username" type= "text" />
5 <input class= "required passfield" name= "password" type= "password" />
6 <input name= "continue" type= "submit" value= "Login" />
7 <input name= "continue" type= "button" value= "Clear" />
8 </form>
9 </body>
10 </html>

```

- `css=form#loginForm` (3)
- `css=input[name="username"]` (4)
- `css=input.required[type="text"]` (4)
- `css=input.passfield` (5)
- `css=#loginForm input[type="button"]` (4)
- `css=#loginForm input:nth-child(2)` (5)

For more information about CSS Selectors, the best place to go is [the W3C publication](#). You'll find additional references there.

**Note:** Most experienced Selenium users recommend CSS as their locating strategy of choice as it's considerably faster than XPath and can find the most complicated objects in an intrinsic HTML document.

## 5.3 Matching Text Patterns

Like locators, *patterns* are a type of parameter frequently required by Selenese commands. Examples of commands which require patterns are **verifyTextPresent**, **verifyTitle**, **verifyAlert**, **assertConfirmation**, **verifyText**, and **verifyPrompt**. And as has been mentioned above, link locators can utilize a pattern. Patterns allow one to *describe*, via the use of special characters, what text is expected rather than having to specify that text exactly.

There are three types of patterns: *globbing*, *regular expressions*, and *exact*.

### 5.3.1 Globbing Patterns

Most people are familiar with globbing as it is utilized in filename expansion at a DOS or Unix/Linux command line such as `ls *.c`. In this case, globbing is used to display all the files ending with a `.c` extension that exist in the current directory. Globbing is fairly limited. Only two special characters are supported in the Selenium implementation:

`*` which translates to “match anything,” i.e., nothing, a single character, or many characters.

`[]` (*character class*) which translates to “match any single character found inside the square brackets.” A dash (hyphen) can be used as a shorthand to specify a range of characters (which are contiguous in the ASCII character set). A few examples will make the functionality of a character class clear:

`[aeiou]` matches any lowercase vowel

`[0-9]` matches any digit

`[a-zA-Z0-9]` matches any alphanumeric character

In most other contexts, globbing includes a third special character, the `?`. However, Selenium globbing patterns only support the asterisk and character class.

To specify a globbing pattern parameter for a Selenese command, one can prefix the pattern with a **glob:** label. However, because globbing patterns are the default, one can also omit the label and specify just the pattern itself.

Below is an example of two commands that use globbing patterns. The actual link text on the page being tested was “Film/Television Department”; by using a pattern rather than the exact text, the **click** command will work even if the link text is changed to “Film & Television Department” or “Film and Television Department”. The glob pattern’s asterisk will match “anything or nothing” between the word “Film” and the word “Television”.

click	link=glob:Film*Television Department
verifyTitle	glob:*Film*Television*

The actual title of the page reached by clicking on the link was “De Anza Film And Television Department - Menu”. By using a pattern rather than the exact text, the `verifyTitle` will pass as long as the two words “Film” and “Television” appear (in that order) anywhere in the page’s title. For example, if the page’s owner should shorten the title to just “Film & Television Department,” the test would still pass. Using a pattern for both a link and a simple test that the link worked (such as the `verifyTitle` above does) can greatly reduce the maintenance for such test cases.

### 5.3.2 Regular Expression Patterns

*Regular expression* patterns are the most powerful of the three types of patterns that Selenese supports. Regular expressions are also supported by most high-level programming languages, many text editors,

and a host of tools, including the Linux/Unix command-line utilities **grep**, **sed**, and **awk**. In Selenese, regular expression patterns allow a user to perform many tasks that would be very difficult otherwise. For example, suppose your test needed to ensure that a particular table cell contained nothing but a number. `regexp: [0-9]+` is a simple pattern that will match a decimal number of any length.

Whereas Selenese globbing patterns support only the `*` and `[ ]` (character class) features, Selenese regular expression patterns offer the same wide array of special characters that exist in JavaScript. Below are a subset of those special characters:

PATTERN	MATCH
<code>.</code>	any single character
<code>[ ]</code>	character class: any single character that appears inside the brackets
<code>*</code>	quantifier: 0 or more of the preceding character (or group)
<code>+</code>	quantifier: 1 or more of the preceding character (or group)
<code>?</code>	quantifier: 0 or 1 of the preceding character (or group)
<code>{1,5}</code>	quantifier: 1 through 5 of the preceding character (or group)
<code> </code>	alternation: the character/group on the left or the character/group on the right
<code>()</code>	grouping: often used with alternation and/or quantifier

Regular expression patterns in Selenese need to be prefixed with either `regexp:` or `regexpi:.` The former is case-sensitive; the latter is case-insensitive.

A few examples will help clarify how regular expression patterns can be used with Selenese commands. The first one uses what is probably the most commonly used regular expression pattern—`.*` (“dot star”). This two-character sequence can be translated as “0 or more occurrences of any character” or more simply, “anything or nothing.” It is the equivalent of the one-character globbing pattern `*` (a single asterisk).

<code>click</code>	<code>link=regexp:Film.*Television Department</code>	
<code>verifyTitle</code>	<code>regexp:. *Film.*Television.*</code>	

The example above is functionally equivalent to the earlier example that used globbing patterns for this same test. The only differences are the prefix (**regexp:** instead of **glob:**) and the “anything or nothing” pattern (`.*` instead of just `*`).

The more complex example below tests that the Yahoo! Weather page for Anchorage, Alaska contains info on the sunrise time:

<code>open</code>	<a href="http://weather.yahoo.com/forecast/USAK0012.html">http://weather.yahoo.com/forecast/USAK0012.html</a>	
<code>verifyTextPresent</code>	<code>regexp:Sunrise: *[0-9]{1,2}:[0-9]{2} [ap]m</code>	

Let’s examine the regular expression above one part at a time:

<code>Sunrise: *</code>	The string <b>Sunrise:</b> followed by 0 or more spaces
<code>[0-9]{1,2}</code>	1 or 2 digits (for the hour of the day)
<code>:</code>	The character <code>:</code> (no special characters involved)
<code>[0-9]{2}</code>	2 digits (for the minutes) followed by a space
<code>[ap]m</code>	“a” or “p” followed by “m” (am or pm)

### 5.3.3 Exact Patterns

The **exact** type of Selenium pattern is of marginal usefulness. It uses no special characters at all. So, if one needed to look for an actual asterisk character (which is special for both globbing and regular expression patterns), the **exact** pattern would be one way to do that. For example, if one wanted to select an item labeled “Real `*`” from a dropdown, the following code might work or it might not. The asterisk in the `glob:Real *` pattern will match anything or nothing. So, if there was an earlier select option labeled “Real Numbers,” it would be the option selected rather than the “Real `*`” option.

select	//select	glob:Real *
--------	----------	-------------

In order to ensure that the “Real \*” item would be selected, the `exact :` prefix could be used to create an **exact** pattern as shown below:

select	//select	exact:Real *
--------	----------	--------------

But the same effect could be achieved via escaping the asterisk in a regular expression pattern:

select	//select	regexp:Real \*
--------	----------	----------------

It’s rather unlikely that most testers will ever need to look for an asterisk or a set of square brackets with characters inside them (the character class for globbing patterns). Thus, globbing patterns and regular expression patterns are sufficient for the vast majority of us.

## 5.4 The “AndWait” Commands

The difference between a command and its *AndWait* alternative is that the regular command (e.g. *click*) will do the action and continue with the following command as fast as it can, while the *AndWait* alternative (e.g. *clickAndWait*) tells Selenium to **wait** for the page to load after the action has been done.

The *AndWait* alternative is always used when the action causes the browser to navigate to another page or reload the present one.

Be aware, if you use an *AndWait* command for an action that does not trigger a navigation/refresh, your test will fail. This happens because Selenium will reach the *AndWait*’s timeout without seeing any navigation or refresh being made, causing Selenium to raise a timeout exception.

## 5.5 The waitFor Commands in AJAX applications

In AJAX driven web applications; data is retrieved from server without refreshing the page. Using *andWait* commands will not work as the page is not actually refreshed. Pausing the test execution for certain period of time is also not a good approach as web element might appear later or earlier than the stipulated period depending on the system’s responsiveness, load or other uncontrolled factors of the moment; leading to test failures. The best approach would be to wait for the needed element in a dynamic period and then continue the execution as soon as element is found.

This is done using *waitFor* commands, as *waitForElementPresent* or *waitForVisible*, which wait dynamically, checking for the desired condition every second and stop as soon as the condition is met.

## 5.6 Sequence of Evaluation and Flow Control

When a script runs, it simply runs in sequence, one command after another.

Selenese, by itself, does not support condition statements (if-else, etc.) or iteration (for, while, etc.). Many useful tests can be conducted without flow control. However, for a functional test of dynamic content, possibly involving multiple pages, programming logic is often needed.

When flow control is needed, there are three options:

1. Run the script using Selenium-RC and a client library such as Java or PHP to utilize the programming language’s flow control features.
2. Run a small JavaScript snippet from within the script using the `storeEval` command.

3. Install the `goto_sel_ide.js` extension.

Most testers will export the test script into a programming language file that uses the Selenium-RC API (see the Selenium-IDE chapter). However, some organizations prefer to run their scripts from Selenium-IDE whenever possible (such as when they have many junior-level people running tests for them, or when programming skills are lacking). If this is your case, consider a JavaScript snippet or the `goto_sel_ide.js` extension.

## 5.7 Store Commands and Selenium Variables

One can use Selenium variables to store constants at the beginning of a script. Also, when combined with a data-driven test design (discussed in a later section), Selenium variables can be used to store values passed to your test program from the command-line, from another program, or from a file.

The plain `store` command is the most basic of the many store commands and can be used to simply store a constant value in a selenium variable. It takes two parameters, the text value to be stored and a selenium variable. Use the standard variable naming conventions of only alphanumeric characters when choosing a name for your variable.

```
store | paul@mysite.org | userName
```

Later in your script, you'll want to use the stored value of your variable. To access the value of a variable, enclose the variable in curly brackets (`{ }`) and precede it with a dollar sign like this.

```
verifyText | //div/p | ${userName}
```

A common use of variables is for storing input for an input field.

```
type | id=login | ${userName}
```

Selenium variables can be used in either the first or second parameter and are interpreted by Selenium prior to any other operations performed by the command. A Selenium variable may also be used within a locator expression.

An equivalent store command exists for each verify and assert command. Here are a couple more commonly used store commands.

### 5.7.1 storeElementPresent

This corresponds to `verifyElementPresent`. It simply stores a boolean value—"true" or "false"—depending on whether the UI element is found.

### 5.7.2 storeText

`StoreText` corresponds to `verifyText`. It uses a locator to identify specific page text. The text, if found, is stored in the variable. `StoreText` can be used to extract text from the page being tested.

### 5.7.3 storeEval

This command takes a script as its first parameter. Embedding JavaScript within Selenese is covered in the next section. `StoreEval` allows the test to store the result of running the script in a variable.

## 5.8 JavaScript and Selenese Parameters

JavaScript can be used with two types of Selenese parameters—**script** and non-script (usually expressions). In most cases, you'll want to access and/or manipulate a test case variable inside the JavaScript snippet used as a Selenese parameter. All variables created in your test case are stored in a JavaScript *associative array*. An associative array has string indexes rather than sequential numeric indexes. The associative array containing your test case's variables is named **storedVars**. Whenever you wish to access or manipulate a variable within a JavaScript snippet, you must refer to it as **storedVars['yourVariableName']**.

### 5.8.1 JavaScript Usage with Script Parameters

Several Selenese commands specify a **script** parameter including **assertEval**, **verifyEval**, **storeEval**, and **waitForEval**. These parameters require no special syntax. A Selenium-IDE user would simply place a snippet of JavaScript code into the appropriate field, normally the **Target** field (because a **script** parameter is normally the first or only parameter).

The example below illustrates how a JavaScript snippet can be used to perform a simple numerical calculation:

store	10	hits
storeXPathCount	//blockquote	blockquotes
storeEval	storedVars['hits']-storedVars['blockquotes']	paragraphs

This next example illustrates how a JavaScript snippet can include calls to methods, in this case the JavaScript String object's `toUpperCase` method and `toLowerCase` method.

store	Edith Wharton	name
storeEval	storedVars['name'].toUpperCase()	uc
storeEval	storedVars['name'].toLowerCase()	lc

### 5.8.2 JavaScript Usage with Non-Script Parameters

JavaScript can also be used to help generate values for parameters, even when the parameter is not specified to be of type **script**. However, in this case, special syntax is required—the JavaScript snippet must be enclosed inside curly braces and preceded by the label `javascript`, as in `javascript {*yourCodeHere*}`. Below is an example in which the `type` command's second parameter value is generated via JavaScript code using this special syntax:

store	league of nations	searchString
type	q	javascript{storedVars['searchString'].toUpperCase()}

## 5.9 echo - The Selenese Print Command

Selenese has a simple command that allows you to print text to your test's output. This is useful for providing informational progress notes in your test which display on the console as your test is running. These notes also can be used to provide context within your test result reports, which can be useful for finding where a defect exists on a page in the event your test finds a problem. Finally, echo statements can be used to print the contents of Selenium variables.

echo	Testing page footer now.	
echo	Username is \${userName}	

## 5.10 Alerts, Popups, and Multiple Windows

*This section is not yet developed.*



# SELENIUM-RC

## 6.1 Introduction

Selenium-RC is the solution for tests that need more than simple browser actions and linear execution. Selenium-RC uses the full power of programming languages to create more complex tests like reading and writing files, querying a database, emailing test results.

You'll want to use Selenium-RC whenever your test requires logic not supported by Selenium-IDE. What logic could this be? For example, Selenium-IDE does not directly support:

- condition statements
- iteration
- logging and reporting of test results
- error handling, particularly unexpected errors
- database testing
- test case grouping
- re-execution of failed tests
- test case dependency
- screenshot capture of test failures

Although these tasks are not supported by Selenium directly, all of them can be achieved by using programming techniques with a language-specific Selenium-RC client library.

In the Adding Some Spice to Your Tests section, you'll find examples that demonstrate the advantages of using a programming language for your tests.

## 6.2 How Selenium-RC Works

First, we will describe how the components of Selenium-RC operate and the role each plays in running your test scripts.

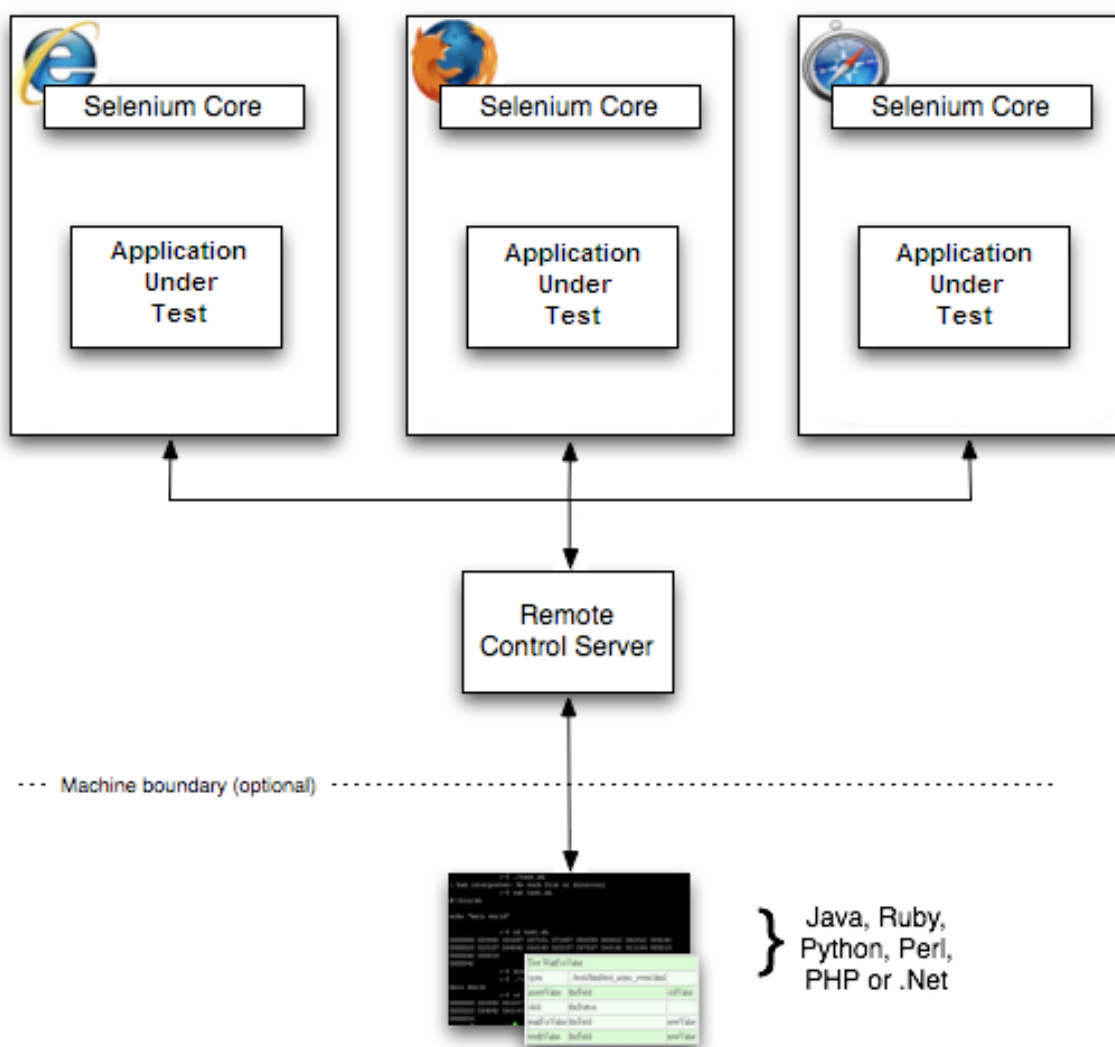
## 6.2.1 RC Components

Selenium-RC components are:

- The Selenium Server which launches and kills browsers, interprets and runs the Selenese commands passed from the test program, and acts as an *HTTP proxy*, intercepting and verifying HTTP messages passed between the browser and the AUT.
- Client libraries which provide the interface between each programming language and the Selenium-RC Server.

Here is a simplified architecture diagram....

Windows, Linux, or Mac (as appropriate)...



The diagram shows the client libraries communicate with the Server passing each Selenium command for execution. Then the server passes the Selenium command to the browser using Selenium-Core JavaScript commands. The browser, using its JavaScript interpreter, executes the Selenium command. This runs the Selenese action or verification you specified in your test script.

## 6.2.2 Selenium Server

Selenium Server receives Selenium commands from your test program, interprets them, and reports back to your program the results of running those tests.

The RC server bundles Selenium Core and automatically injects it into the browser. This occurs when your test program opens the browser (using a client library API function). Selenium-Core is a JavaScript program, actually a set of JavaScript functions which interprets and executes Selenese commands using the browser's built-in JavaScript interpreter.

The Server receives the Selenese commands from your test program using simple HTTP GET/POST requests. This means you can use any programming language that can send HTTP requests to automate Selenium tests on the browser.

## 6.2.3 Client Libraries

The client libraries provide the programming support that allows you to run Selenium commands from a program of your own design. There is a different client library for each supported language. A Selenium client library provides a programming interface (API), i.e., a set of functions, which run Selenium commands from your own program. Within each interface, there is a programming function that supports each Selenese command.

The client library takes a Selenese command and passes it to the Selenium Server for processing a specific action or test against the application under test (AUT). The client library also receives the result of that command and passes it back to your program. Your program can receive the result and store it into a program variable and reporting it as a success or failure, or possibly taking corrective action if it was an unexpected error.

So to create a test program, you simply write a program that runs a set of Selenium commands using a client library API. And, optionally, if you already have a Selenese test script created in the Selenium-IDE, you can *generate the Selenium-RC code*. The Selenium-IDE can translate (using its Export menu item) its Selenium commands into a client-driver's API function calls. See the Selenium-IDE chapter for specifics on exporting RC code from Selenium-IDE.

## 6.3 Installation

After downloading the Selenium-RC zip file from the [downloads page](#), you'll notice it has several sub-folders. These folders have all the components you need for using Selenium-RC with the programming language of your choice.

Once you've chosen a language to work with, you simply need to:

- Install the Selenium-RC Server.
- Set up a programming project using a language specific client driver.

### 6.3.1 Installing Selenium Server

The Selenium-RC server is simply a Java *jar* file (*selenium-server.jar*), which doesn't require any special installation. Just downloading the zip file and extracting the server in the desired directory is sufficient.

### 6.3.2 Running Selenium Server

Before starting any tests you must start the server. Go to the directory where Selenium-RC's server is located and run the following from a command-line console.

```
java -jar selenium-server.jar
```

This can be simplified by creating a batch or shell executable file (.bat on Windows and .sh on Linux) containing the command above. Then make a shortcut to that executable on your desktop and simply double-click the icon to start the server.

For the server to run you'll need Java installed and the PATH environment variable correctly configured to run it from the console. You can check that you have Java correctly installed by running the following on a console:

```
java -version
```

If you get a version number (which needs to be 1.5 or later), you're ready to start using Selenium-RC.

### 6.3.3 Using the Java Client Driver

- Download Selenium-RC from the SeleniumHQ [downloads page](#).
- Extract the file *selenium-java-client-driver.jar*.
- Open your desired Java IDE (Eclipse, NetBeans, IntelliJ, Netweaver, etc.)
- Create a new project.
- Add the selenium-java-client-driver.jar files to your project as references.
- Add to your project classpath the file *selenium-java-client-driver.jar*.
- From Selenium-IDE, export a script to a Java file and include it in your Java. project, or write your Selenium test in Java using the selenium-java-client API. The API is presented later in this chapter. You can either use JUnit, or TestNg to run your test, or you can write your own simple main() program. These concepts are explained later in this section.
- Run Selenium server from the console.
- Execute your test from the Java IDE or from the command-line.

For details on Java test project configuration, see the Appendix sections *Configuring Selenium-RC With Eclipse* and *Configuring Selenium-RC With IntelliJ*.

### 6.3.4 Using the Python Client Driver

- Download Selenium-RC from the SeleniumHQ [downloads page](#)
- Extract the file *selenium.py*
- Either write your Selenium test in Python or export a script from Selenium-IDE to a python file.
- Add to your test's path the file *selenium.py*

- Run Selenium server from the console
- Execute your test from a console or your Python IDE

For details on Python client driver configuration, see the appendix *Python Client Driver Configuration*.

### 6.3.5 Using the .NET Client Driver

- Download Selenium-RC from the SeleniumHQ [downloads page](#)
- Extract the folder
- Download and install [NUnit](#) ( Note: You can use NUnit as your test engine. If you're not familiar yet with NUnit, you can also write a simple main() function to run your tests; however NUnit is very useful as a test engine.)
- Open your desired .Net IDE (Visual Studio, SharpDevelop, MonoDevelop)
- Create a class library (.dll)
- Add references to the following DLLs: nmock.dll, nunit.core.dll, nunit. framework.dll, ThoughtWorks.Selenium.Core.dll, ThoughtWorks.Selenium. IntegrationTests.dll and ThoughtWorks.Selenium.UnitTests.dll
- Write your Selenium test in a .Net language (C#, VB.Net), or export a script from Selenium-IDE to a C# file and copy this code into the class file you just created.
- Write your own simple main() program or you can include NUnit in your project for running your test. These concepts are explained later in this chapter.
- Run Selenium server from console
- Run your test either from the IDE, from the NUnit GUI or from the command line

For specific details on .NET client driver configuration with Visual Studio, see the appendix *.NET client driver configuration*.

## 6.4 From Selenese to a Program

The primary task for using Selenium-RC is to convert your Selenese into a programming language. In this section, we provide several different language-specific examples.

### 6.4.1 Sample Test Script

Let's start with an example Selenese test script. Imagine recording the following test with Selenium-

IDE.	open	/	selenium rc
	type	q	
	clickAndWait	btnG	
	assertTextPresent	Results * for selenium rc	

Note: This example would work with the Google search page <http://www.google.com>

## 6.4.2 Selene as Programming Code

Here is the test script exported (via Selenium-IDE) to each of the supported programming languages. If you have at least basic knowledge of an object-oriented programming language, you will understand how Selenium runs Selene commands by reading one of these examples. To see an example in a specific language, select one of these buttons.

In C#:

```
using System;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading;
using NUnit.Framework;
using Selenium;

namespace SeleniumTests
{
    [TestFixture]
    public class NewTest
    {
        private ISelenium selenium;
        private StringBuilder verificationErrors;

        [SetUp]
        public void SetupTest ()
        {
            selenium = new DefaultSelenium("localhost", 4444, "*firefox", "http://www");
            selenium.Start();
            verificationErrors = new StringBuilder();
        }

        [TearDown]
        public void TeardownTest ()
        {
            try
            {
                selenium.Stop();
            }
            catch (Exception)
            {
                // Ignore errors if unable to close the browser
            }
            Assert.AreEqual("", verificationErrors.ToString());
        }

        [Test]
        public void TheNewTest ()
        {
            selenium.Open("/");
            selenium.Type("q", "selenium rc");
            selenium.Click("btnG");
            selenium.WaitForPageToLoad("30000");
            Assert.IsTrue(selenium.IsTextPresent("Results * for selenium rc"));
        }
    }
}
```

**In Java:**

```

package com.example.tests;

import com.thoughtworks.selenium.*;
import java.util.regex.Pattern;

public class NewTest extends SeleneseTestCase {
    public void setUp() throws Exception {
        setUp( "http://www.google.com/", "*firefox" );
    }
    public void testNew() throws Exception {
        selenium.open( "/" );
        selenium.type( "q", "selenium rc" );
        selenium.click( "btnG" );
        selenium.waitForPageToLoad( "30000" );
        assertTrue( selenium.isTextPresent( "Results * for selenium rc" ) );
    }
}

```

**In Perl:**

```

use strict;
use warnings;
use Time::HiRes qw( sleep );
use Test::WWW::Selenium;
use Test::More "no_plan" ;
use Test::Exception;

my $sel = Test::WWW::Selenium->new( host => "localhost",
                                   port => 4444,
                                   browser => "*firefox",
                                   browser_url => "http://www.google.com/" );

$sel->open_ok( "/" );
$sel->type_ok( "q", "selenium rc" );
$sel->click_ok( "btnG" );
$sel->wait_for_page_to_load_ok( "30000" );
$sel->is_text_present_ok( "Results * for selenium rc" );

```

**In PHP:**

```

<?php

require_once 'PHPUnit/Extensions/SeleniumTestCase.php' ;

class Example extends PHPUnit_Extensions_SeleniumTestCase
{
    function setUp()
    {
        $this->setBrowser( " *firefox " );
        $this->setBrowserUrl( " http://www.google.com/ " );
    }
}

```

```
function testMyTestCase()
{
  $this->open( " / " );
  $this->type( " q ", " selenium rc " );
  $this->click( " btnG " );
  $this->waitForPageToLoad( " 30000 " );
  $this->assertTrue( $this->isTextPresent( " Results * for selenium rc " ) );
}
}
?>
```

**in Python:**

```
from selenium import selenium
import unittest, time, re

class NewTest(unittest.TestCase):
    def setUp(self):
        self.verificatiOnErrors = []
        self.selenium = selenium( " localhost ", 4444, " *firefox ",
            " http://www.google.com/ " )
        self.selenium.start()

    def test_new(self):
        sel = self.selenium
        sel.open( " / " )
        sel.type( " q ", " selenium rc " )
        sel.click( " btnG " )
        sel.wait_for_page_to_load( " 30000 " )
        self.failUnless( sel.is_text_present( " Results * for selenium rc " ) )

    def tearDown(self):
        self.selenium.stop()
        self.assertEqual( [], self.verificatiOnErrors)
```

**in Ruby:**

```
require " selenium "
require " test/unit "

class NewTest < Test::Unit::TestCase
  def setup
    @verification_errors = []
    if $selenium
      @selenium = $selenium
    else
      @selenium = Selenium::SeleniumDriver.new( " localhost ", 4444, " *firefox ", " h
      @selenium.start
    end
    @selenium.set_context( " test_new " )
  end

  def teardown
    @selenium.stop unless $selenium
```

```

    assert_equal [], @verification_errors
end

def test_new
  @selenium.open " / "
  @selenium.type " q ", " selenium rc "
  @selenium.click " btnG "
  @selenium.wait_for_page_to_load " 30000 "
  assert @selenium.is_text_present( " Results * for selenium rc " )
end
end

```

In the next section we'll explain how to build a test program using the generated code.

## 6.5 Programming Your Test

Now we'll illustrate how to program your own tests using examples in each of the supported programming languages. There are essentially two tasks. \* Generate your script into a programming language from Selenium-IDE, optionally modifying the result. \* And two, write a very simple main program that executes the generated code. Optionally, you can adopt a test engine platform like JUnit or TestNG for Java, or NUnit for .NET if you are using one of those languages.

Here, we show language-specific examples. The language-specific APIs tend to differ from one to another, so you'll find a separate explanation for each.

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

### 6.5.1 Java

For Java, people use either Junit or TestNG as the test engine. Some development environments like Eclipse have direct support for these via plug-ins. This makes it even easier. Teaching JUnit or TestNG is beyond the scope of this document however materials may be found online and there are publications available. If you are already a “java-shop” chances are your developers will already have some experience with one of these test frameworks.

You will probably want to rename the test class from “NewTest” to something of your own choosing. Also, you will need to change the browser-open parameters in the statement:

```
selenium = new DefaultSelenium("localhost", 4444, "*iehta", "http://www.google.com/");
```

The Selenium-IDE generated code will look like this. This example has comments added manually for additional clarity.

```

package com.example.tests;
// We specify the package of our test

```

```
import com.thoughtworks.selenium.*;
// This is the driver's import. You'll use this for instantiating a
// browser and making it do what you need.

import java.util.regex.Pattern;
// Selenium-IDE add the Pattern module because it's sometimes used for
// regex validations. You can remove the module if it's not used in your
// script.

public class NewTest extends SeleneseTestCase {
// We create our Selenium test case

    public void setUp() throws Exception {
        setUp( "http://www.google.com/", "*firefox" );
        // We instantiate and start the browser
    }

    public void testNew() throws Exception {
        selenium.open( "/" );
        selenium.type( "q", "selenium rc" );
        selenium.click( "btnG" );
        selenium.waitForPageToLoad( "30000" );
        assertTrue( selenium.isTextPresent( "Results * for selenium rc" ) );
        // These are the real test steps
    }
}
```

### 6.5.2 C#

The .NET Client Driver works with Microsoft.NET. It can be used with any .NET testing framework like NUnit or the Visual Studio 2005 Team System.

Selenium-IDE assumes you will use NUnit as your testing framework. You can see this in the generated code below. It includes the *using* statement for NUnit along with corresponding NUnit attributes identifying the role for each member function of the test class.

You will probably have to rename the test class from “NewTest” to something of your own choosing. Also, you will need to change the browser-open parameters in the statement:

```
selenium = new DefaultSelenium("localhost", 4444, "*iehta", "http://www.google.com/");
```

The generated code will look similar to this.

```
using System;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading;
using NUnit.Framework;
using Selenium;

namespace SeleniumTests

{
    [TestFixture]
```

```
public class NewTest
{
    private ISelenium selenium;

    private StringBuilder verificationErrors;

    [SetUp]
    public void SetupTest ()
    {
        selenium = new DefaultSelenium( "localhost", 4444, "*iehta",
            "http://www.google.com/" );

        selenium.Start ();

        verificationErrors = new StringBuilder ();
    }

    [TearDown]
    public void TeardownTest ()
    {
        try
        {
            selenium.Stop ();
        }

        catch (Exception)
        {
            // Ignore errors if unable to close the browser
        }

        Assert.AreEqual( "", verificationErrors.ToString());
    }

    [Test]
    public void TheNewTest ()
    {
        // Open Google search engine.
        selenium.Open( "http://www.google.com/" );

        // Assert Title of page.
        Assert.AreEqual( "Google", selenium.GetTitle());

        // Provide search term as "Selenium OpenQA"
        selenium.Type( "q", "Selenium OpenQA" );

        // Read the keyed search term and assert it.
        Assert.AreEqual( "Selenium OpenQA", selenium.GetValue( "q" ));

        // Click on Search button.
        selenium.Click( "btnG" );

        // Wait for page to load.
        selenium.WaitForPageToLoad( "5000" );
    }
}
```

```
// Assert that "www.openqa.org" is available in search results.
Assert.IsTrue(selenium.IsTextPresent( "www.openqa.org" ));

// Assert that page title is - "Selenium OpenQA - Google Search"
Assert.AreEqual( "Selenium OpenQA - Google Search",
                selenium.GetTitle());
    }
}
}
```

You can allow NUnit to manage the execution of your tests. Or alternatively, you can write a simple main() program that instantiates the test object and runs each of the three methods, SetupTest(), NewTest(), and TeardownTest() in turn.

### 6.5.3 Python

Pyunit is the test framework to use for Python. To learn pyunit refer to its *official documentation* <<http://docs.python.org/library/unittest.html>>.

The basic test structure is:

```
from selenium import selenium
# This is the driver's import. You'll use this class for instantiating a
# browser and making it do what you need.

import unittest, time, re
# This are the basic imports added by Selenium-IDE by default.
# You can remove the modules if they are not used in your script.

class NewTest(unittest.TestCase):
# We create our unittest test case

    def setUp(self):
        self.errors = []
        # This is an empty array where we will store any verification errors
        # we find in our tests

        self.selenium = selenium( " localhost ", 4444, " *firefox ",
                                " http://www.google.com/ " )
        self.selenium.start()
        # We instantiate and start the browser

    def test_new(self):
        # This is the test code. Here you should put the actions you need
        # the browser to do during your test.

        sel = self.selenium
        # We assign the browser to the variable "sel" (just to save us from
        # typing "self.selenium" each time we want to call the browser).

        sel.open( " / " )
        sel.type( " q ", " selenium rc " )
        sel.click( " btnG " )
        sel.wait_for_page_to_load( " 30000 " )
```

```

self.failUnless(sel.is_text_present(" Results * for selenium rc "))
# These are the real test steps

def tearDown(self):
    self.selenium.stop()
    # we close the browser (I'd recommend you to comment this line while
    # you are creating and debugging your tests)

    self.assertEqual([], self.verifyErrors)
    # And make the test fail if we found that any verification errors
    # were found

```

### 6.5.4 Perl, PHP, Ruby

The members of the documentation team have not used Sel-RC with Perl, PHP or Ruby. If you are using Selenium-RC with either of these two languages please contact the Documentation Team (see the chapter on contributing). We would love to include some examples from you and your experiences support Perl and PHP users.

## 6.6 Learning the API

The Selenium-RC API uses naming conventions that, assuming you understand Selenese, much of the interface will be self-explanatory. Here, however, we explain the most critical and possibly less obvious, aspects of the API.

### 6.6.1 Starting the Browser

**In C#:**

```

selenium = new DefaultSelenium("localhost", 4444, "*firefox", "http://www.google.com/");
selenium.Start();

```

**In Java:**

```

setUp("http://www.google.com/", "*firefox");

```

**In Perl:**

```

my $sel = Test::WWW::Selenium->new( host => "localhost",
                                   port => 4444,
                                   browser => "*firefox",
                                   browser_url => "http://www.google.com/" );

```

**In PHP:**

```

$this->setBrowser("*firefox");
$this->setBrowserUrl("http://www.google.com/");

```

**In Python:**

```
self.selenium = selenium( " localhost " , 4444 , " *firefox " ,  
                          " http://www.google.com/ " )  
self.selenium.start()
```

### In Ruby:

```
if $selenium  
  @selenium = $selenium  
else  
  @selenium = Selenium::SeleniumDriver.new( " localhost " , 4444 , " *firefox " , " http:  
  @selenium.start
```

Each of these examples opens the browser and represents that browser by assigning a “browser instance” to a program variable. This browser variable is then used to call methods from the browser. These methods execute the Selenium commands, i.e. like *open* or *type* or the *verify* commands.

The parameters required when creating the browser instance are:

**host** Specifies the IP address of the computer where the server is located. Usually, this is the same machine as where the client is running, so in this case *localhost* is passed. In some clients this is an optional parameter.

**port** Specifies the TCP/IP socket where the server is listening waiting for the client to establish a connection. This also is optional in some client drivers.

**browser** The browser in which you want to run the tests. This is a required parameter.

**url** The base url of the application under test. This is required by all the client libs and is integral information for starting up the browser-proxy-AUT communication.

Note that some of the client libraries require the browser to be started explicitly by calling its *start()* method.

## 6.6.2 Running Commands

Once you have the browser initialized and assigned to a variable (generally named “selenium”) you can make it run Selenese commands by calling the respective methods from the browser variable. For example, to call the *type* method of the selenium object:

```
selenium.type( " field-id " , " string to type " )
```

In the background the browser will actually perform a *type* operation, essentially identical to a user typing input into the browser, by using the locator and the string you specified during the method call.

## 6.7 Reporting Results

Selenium-RC does not have its own mechanism for reporting results. Rather, it allows you to build your reporting customized to your needs using features of your chosen programming language. That’s great, but what if you simply want something quick that’s already done for you? Often an existing library or test framework will exist that can meet your needs faster than developing your own test reporting code.

### 6.7.1 Test Framework Reporting Tools

Test frameworks are available for many programming languages. These, along with their primary function of providing a flexible test engine for executing your tests, include library code for reporting results. For example, Java has two commonly used test frameworks, JUnit and TestNG. .NET also has its own, NUnit.

We won't teach the frameworks themselves here; that's beyond the scope of this user guide. We will simply introduce the framework features that relate to Selenium along with some techniques you can apply. There are good books available on these test frameworks however along with information on the internet.

### 6.7.2 Test Report Libraries

Also available are third-party libraries specifically created for reporting test results in your chosen programming language. These often support a variety of formats such as HTML or PDF.

### 6.7.3 What's The Best Approach?

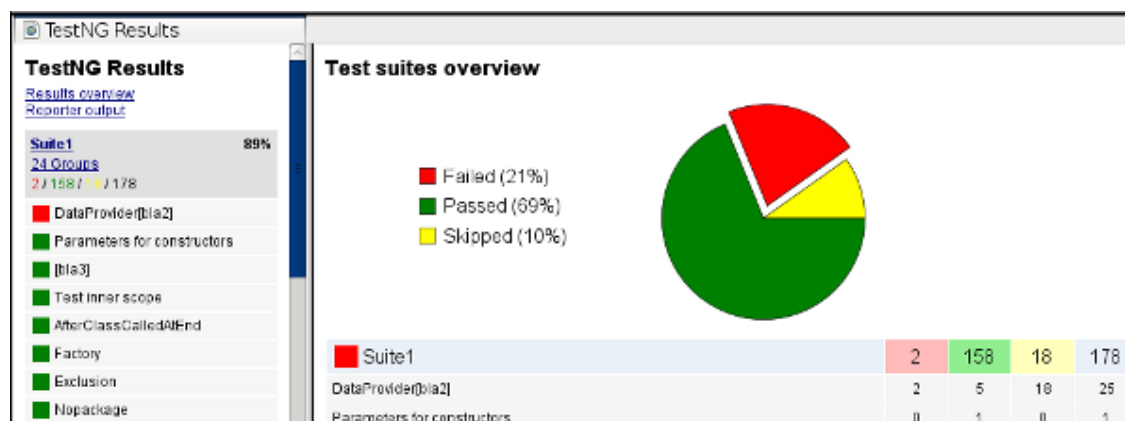
Most people new to the testing frameworks will begin with the framework's built-in reporting features. From there most will examine any available libraries as that's less time consuming than developing your own. As you begin to use Selenium no doubt you will start putting in your own "print statements" for reporting progress. That may gradually lead to you developing your own reporting, possibly in parallel to using a library or test framework. Regardless, after the initial, but short, learning curve you will naturally develop what works best for your own situation.

### 6.7.4 Test Reporting Examples

To illustrate, we'll direct you to some specific tools in some of the other languages supported by Selenium. The ones listed here are commonly used and have been used extensively (and therefore recommended) by the authors of this guide.

#### Test Reports in Java

- If Selenium Test cases are developed using JUnit then JUnit Report can be used to generate test reports. Refer to [JUnit Report](#) for specifics.
- If Selenium Test cases are developed using TestNG then no external task is required to generate test reports. The TestNG framework generates an HTML report which lists details of tests. See [TestNG Report](#) for more.
- ReportNG is a HTML reporting plug-in for the TestNG framework. It is intended as a replacement for the default TestNG HTML report. ReportNG provides a simple, colour-coded view of the test results. See [ReportNG](#) for more.
- Also, for a very nice summary report try using TestNG-xslt. A TestNG-xslt Report looks like this.



See [TestNG-xslt](#) for more.

## Logging the Selenese Commands

- Logging Selenium can be used to generate a report of all the Selenese commands in your test along with the success or failure of each. Logging Selenium extends the Java client driver to add this Selenese logging ability. Please refer to [Logging Selenium](#).

## Test Reports for Python

- When using Python Client Driver then `HTMLTestRunner` can be used to generate a Test Report. See [HTMLTestRunner](#).

## Test Reports for Ruby

- If RSpec framework is used for writing Selenium Test Cases in Ruby then its HTML report can be used to generate test report. Refer to [RSpec Report](#) for more.

**Note:** If you are interested in a language independent log of what's going on, take a look at [Selenium Server Logging](#)

## 6.8 Adding Some Spice to Your Tests

Now we'll get to the whole reason for using Selenium-RC, adding programming logic to your tests. It's the same as for any program. Program flow is controlled using condition statements and iteration. In addition you can report progress information using I/O. In this section we'll show some examples of how programming language constructs can be combined with Selenium to solve common testing problems.

You will find as you transition from the simple tests of the existence of page elements to tests of dynamic functionality involving multiple web-pages and varying data that you will require programming logic for verifying expected results. Basically, the Selenium-IDE does not support iteration and standard condition statements. You can do some conditions by embedding javascript in Selenese parameters, however iteration is impossible, and most conditions will be much easier in a programming language. In addition, you may need exception-handling for error recovery. For these reasons and others, we have written this section to illustrate the use of common programming techniques to give you greater 'verification power' in your automated testing.

The examples in this section are written in Java, although the code is simple and can be easily adapted to the other supported languages. If you have some basic knowledge of an object-oriented programming language you shouldn't have difficulty understanding this section.

### 6.8.1 Iteration

Iteration is one of the most common things people need to do in their tests. For example, you may want to execute a search multiple times. Or, perhaps for verifying your test results you need to process a "result set" returned from a database.

Using the same Google search example we used earlier, let's check the Selenium search results. This test could use the Selenese:

open	/	
type	q	selenium rc
clickAndWait	btnG	
assertTextPresent	Results * for selenium rc	
type	q	selenium ide
clickAndWait	btnG	
assertTextPresent	Results * for selenium ide	
type	q	selenium grid
clickAndWait	btnG	
assertTextPresent	Results * for selenium grid	

The code has been repeated to run the same steps 3 times. But multiple copies of the same code is not good program practice because it's more work to maintain. By using a programming language, we can iterate over the search results for a more flexible and maintainable solution.

#### In C#:

```
// Collection of String values.
String[] arr = { "ide", "rc", "grid" };

// Execute loop for each String in array 'arr'.
foreach (String s in arr) {
    sel.open( "/" );
    sel.type( "q", "selenium " +s);
    sel.click( "btnG" );
    sel.waitForPageToLoad( "30000" );
    assertTrue( "Expected text: " +s+ " is missing on page."
, sel.isTextPresent( "Results * for selenium " + s));
}
```

### 6.8.2 Condition Statements

To illustrate using conditions in tests we'll start with an example. A common problem encountered while running Selenium tests occurs when an expected element is not available on page. For example, when running the following line:

```
selenium.type( "q", "selenium " +s);
```

If element 'q' is not on the page then an exception is thrown:

```
com.thoughtworks.selenium.SeleniumException: ERROR: Element q not found
```

This can cause your test to abort. For some tests that's what you want. But often that is not desirable as your test script has many other subsequent tests to perform.

A better approach is to first validate if the element is really present and then take alternatives when it is not. Let's look at this using Java.

```
// If element is available on page then perform type operation.
if(selenium.isElementPresent("q")) {
    selenium.type("q", "Selenium rc");
} else {
    System.out.printf("Element: " +q+ " is not available on page.")
}
```

The advantage of this approach is to continue with test execution even if some UI elements are not available on page.

### 6.8.3 Executing Javascript from Your Test

Javascript comes very handy in exercising application which is not directly supported by selenium. **getEval** method of selenium API can be used to execute java script from selenium RC.

Consider an application having check boxes with no static identifiers. In this case one could evaluate js from selenium RC to get ids of all check boxes and then exercise them.

```
public static String[] getAllCheckboxIds () {
    String script = "var inputId = new Array();" ;// Create array in java script.
    script += "var cnt = 0;" ; // Counter for check box ids.
    script += "var inputFields = new Array();" ; // Create array in java script.
    script += "inputFields = window.document.getElementsByTagName('input');" ; // Collect
    script += "for(var i=0; i<inputFields.length; i++) {" ; // Loop through the collect
    script += "if(inputFields[i].id !=null " +
        "&& inputFields[i].id !='undefined' " +
        "&& inputFields[i].getAttribute('type') == 'checkbox') {" ; // If input field is of
    script += "inputId[cnt]=inputFields[i].id ;" + // Save check box id to inputId array
        "cnt++;" + // increment the counter.
        "}" + // end of if.
        "}" ; // end of for.
    script += "inputId.toString();" ;// Convert array in to string.
    String[] checkboxIds = selenium.getEval(script).split(","); // Split the string.
    return checkboxIds;
}
```

To count number of images on a page:

```
selenium.getEval("window.document.images.length;");
```

Remember to use window object in case of dom expressions as by default selenium window is referred and not the test window.

## 6.9 Selenium-RC Architecture

**Note:** This topic tries to explain the technical implementation behind Selenium-RC. It's not fundamental for a Selenium user to know this, but could be useful for understanding some of the problems you can find in the future.

To understand in detail how Selenium-RC Server works and why it uses proxy injection and heightened privilege modes you must first understand the same origin policy.

### 6.9.1 The Same Origin Policy

The main restriction that Selenium's has faced is the Same Origin Policy. This security restriction is applied by every browser in the market and its objective is to ensure that a site's content will never be accessible by a script from other site. The Same Origin Policy dictates that any code loaded within the browser can only operate within that website's domain. It cannot perform functions on another website. So for example, if the browser loads javascript code when it loads `www.mysite.com`, it cannot run that loaded code against `www.mysite2.com`—even if that's another of your sites. If this were possible, a script placed on any website you open, would be able to read information on your bank account if you had the account page opened on other tab. This is called XSS (Cross-site Scripting).

To work within this policy, Selenium-Core (and its JavaScript commands that make all the magic happen) must be placed in the same origin as the Application Under Test (same URL).

Historically, Selenium-Core was limited by this problem since it was implemented in Javascript. Selenium-RC is not, however, restricted by the Same Origin Policy. Its use of the Selenium Server as a proxy avoids this problem. It, essentially, tells the browser that the browser is working on a single “spoofed” website that the Server provides.

**Note:** You can find additional information about this topic on Wikipedia pages about [Same Origin Policy](#) and [XSS](#).

### 6.9.2 Proxy Injection

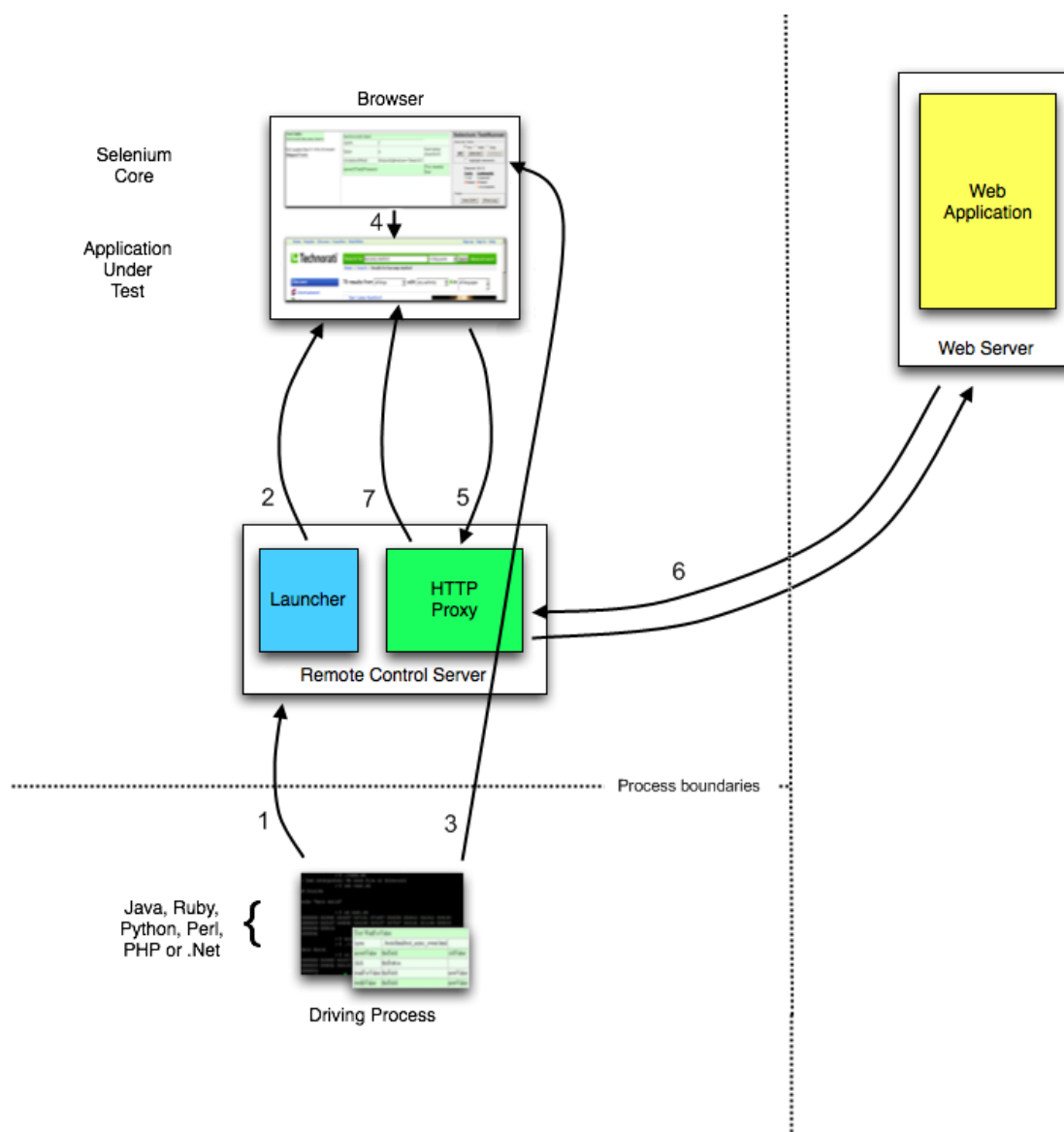
The first method Selenium used to avoid the The Same Origin Policy was Proxy Injection. In Proxy Injection Mode, the Selenium Server acts as a client-configured <sup>1</sup> **HTTP proxy** <sup>2</sup>, that sits between the browser and the Application Under Test. It then masks the AUT under a fictional URL (embedding Selenium-Core and the set of tests and delivering them as if they were coming from the same origin).

Here is an architectural diagram.

---

<sup>1</sup> The proxy is a third person in the middle that passes the ball between the two parts. It acts as a “web server” that delivers the AUT to the browser. Being a proxy, gives the capability of “lying” about the AUT's real URL.

<sup>2</sup> The browser is launched with a configuration profile that has set `localhost:4444` as the HTTP proxy, this is why any HTTP request that the browser does will pass through Selenium server and the response will pass through it and not from the real server.



As a test suite starts in your favorite language, the following happens:

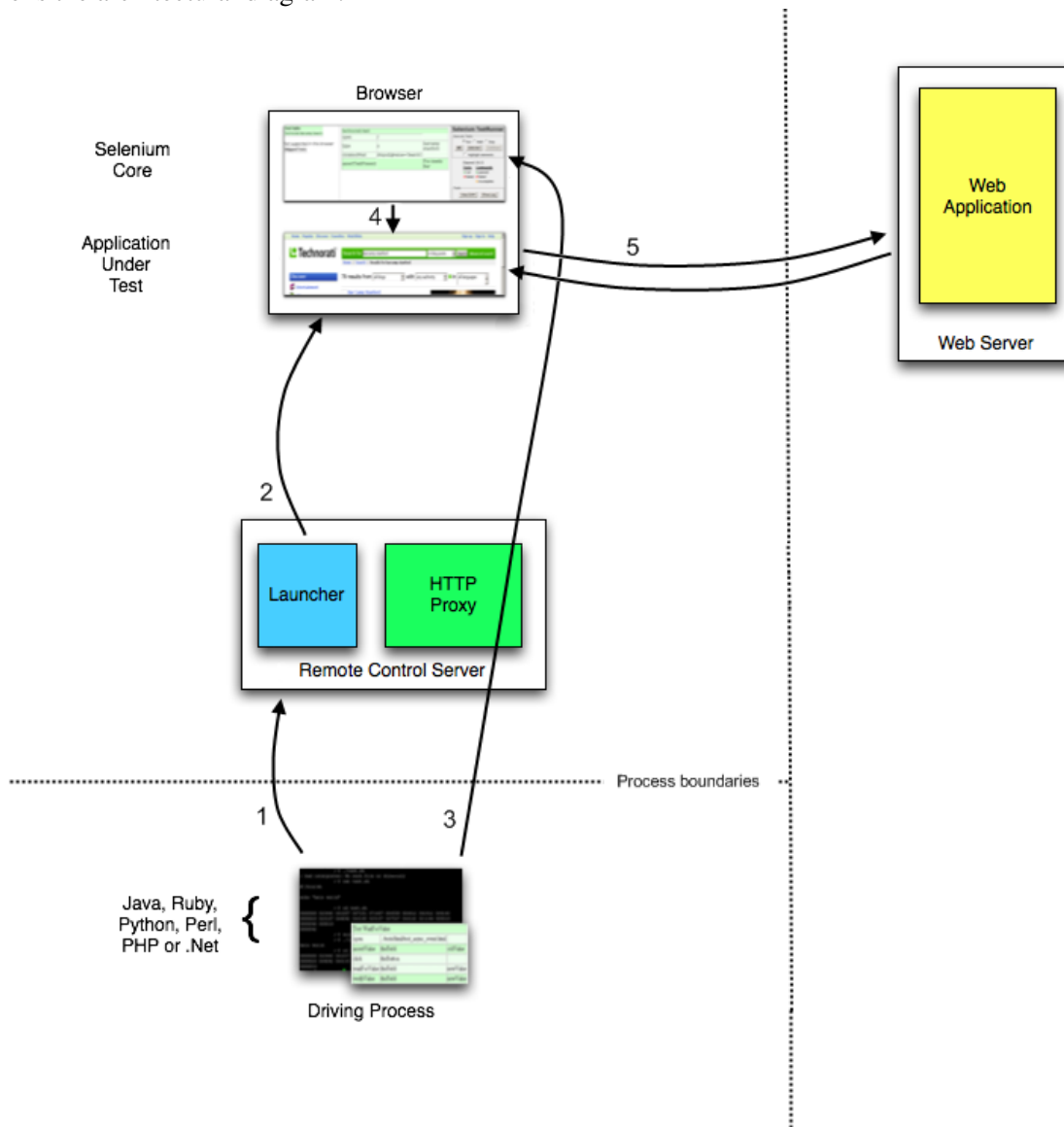
1. The client/driver establishes a connection with the selenium-RC server.
2. Selenium-RC server launches a browser (or reuses an old one) with an URL that injects Selenium-Core's javascript into the browser-loaded web page.
3. The client-driver passes a Selenese command to the server.
4. The Server interprets the command and then triggers the corresponding javascript execution to execute that command within the browser.
5. Selenium-Core instructs the browser to act on that first instruction, typically opening a page of the AUT.
6. The browser receives the open request and asks for the website's content to the Selenium-RC server (set as the HTTP proxy for the browser to use).
7. Selenium-RC server communicates with the Web server asking for the page and once it receives it, it sends the page to the browser masking the origin to look like the page comes from the same server as Selenium-Core (this allows Selenium-Core to comply with the Same Origin Policy).

8. The browser receives the web page and renders it in the frame/window reserved for it.

### 6.9.3 Heightened Privileges Browsers

This workflow on this method is very similar to Proxy Injection but the main difference is that the browsers are launched in a special mode called *Heightened Privileges*, which allows websites to do things that are not commonly permitted (as doing XSS, or filling file upload inputs and pretty useful stuff for Selenium). By using these browser modes, Selenium Core is able to directly open the AUT and read/interact with its content without having to pass the whole AUT through the Selenium-RC server.

Here is the architectural diagram.



As a test suite starts in your favorite language, the following happens:

1. The client/driver establishes a connection with the selenium-RC server.
2. Selenium-RC server launches a browser (or reuses an old one) with an URL that will load Selenium-Core in the web page.
3. Selenium-Core gets the first instruction from the client/driver (via another HTTP request made to the Selenium-RC Server).

4. Selenium-Core acts on that first instruction, typically opening a page of the AUT.
5. The browser receives the open request and asks the Web Server for the page. Once the browser receives the web page, renders it in the frame/window reserved for it.

## 6.10 Handling HTTPS and Security Popups

Many applications switch from using HTTP to HTTPS when they need to send encrypted information such as passwords or credit card information. This is common with many of today's web applications. Selenium-RC supports this.

To ensure the HTTPS site is genuine, the browser will need a security certificate. Otherwise, when the browser accesses the AUT using HTTPS, it will assume that application is not 'trusted'. When this occurs the browser displays security popups, and these popups cannot be closed using Selenium-RC.

When dealing with HTTPS in a Selenium-RC test, you must use a run mode that supports this and handles the security certificate for you. You specify the run mode when your test program initializes Selenium.

In Selenium-RC 1.0 beta 2 and later use `*firefox` or `*iexplore` for the run mode. In earlier versions, including Selenium-RC 1.0 beta 1, use `*chrome` or `*iehta`, for the run mode. Using these run modes, you will not need to install any special security certificates; Selenium-RC will handle it for you.

In version 1.0 the run modes `*firefox` or `*iexplore` are recommended. However, there are additional run modes of `*iexploreproxy` and `*firefoxproxy`. These are provided only for backwards compatibility only, and should not be used unless required by legacy test programs. Their use will present limitations with security certificate handling and with the running of multiple windows if your application opens additional browser windows.

In earlier versions of Selenium-RC, `*chrome` or `*iehta` were the run modes that supported HTTPS and the handling of security popups. These were considered 'experimental modes although they became quite stable and many used them. If you are using Selenium 1.0 you do not need, and should not use, these older run modes.

### 6.10.1 Security Certificates Explained

Normally, your browser will trust the application you are testing by installing a security certificate which you already own. You can check this in your browser's options or internet properties (if you don't know your AUT's security certificate ask your system administrator). When Selenium loads your browser it injects code to intercept messages between the browser and the server. The browser now thinks untrusted software is trying to look like your application. It responds by alerting you with popup messages.

To get around this, Selenium-RC, (again when using a run mode that support this) will install its own security certificate, temporarily, to your client machine in a place where the browser can access it. This tricks the browser into thinking it's accessing a site different from your AUT and effectively suppresses the popups.

Another method used with earlier versions of Selenium was to install the Cybervillians security certificate provided with your Selenium installation. Most users should no longer need to do this however, if you are running Selenium-RC in proxy injection mode, you may need to explicitly install this security certificate.

## 6.11 Server Options

When the server is launched, command line options can be used to change the default server behaviour. Recall, the server is started by running the following.

```
$ java -jar selenium-server.jar
```

To see the list of options, run the server with the `-h` option.

```
$ java -jar selenium-server.jar -h
```

You'll see a list of all the options you can use with the server and a brief description of each. The provided descriptions will not always be enough, so we've provided explanations for some of the more important options.

### 6.11.1 Proxy Configuration

If one is behind HTTP proxy which requires authentication then one should set up `http.proxyHost`, `http.proxyPort`, `http.proxyUser` and `http.proxyPassword` like this -

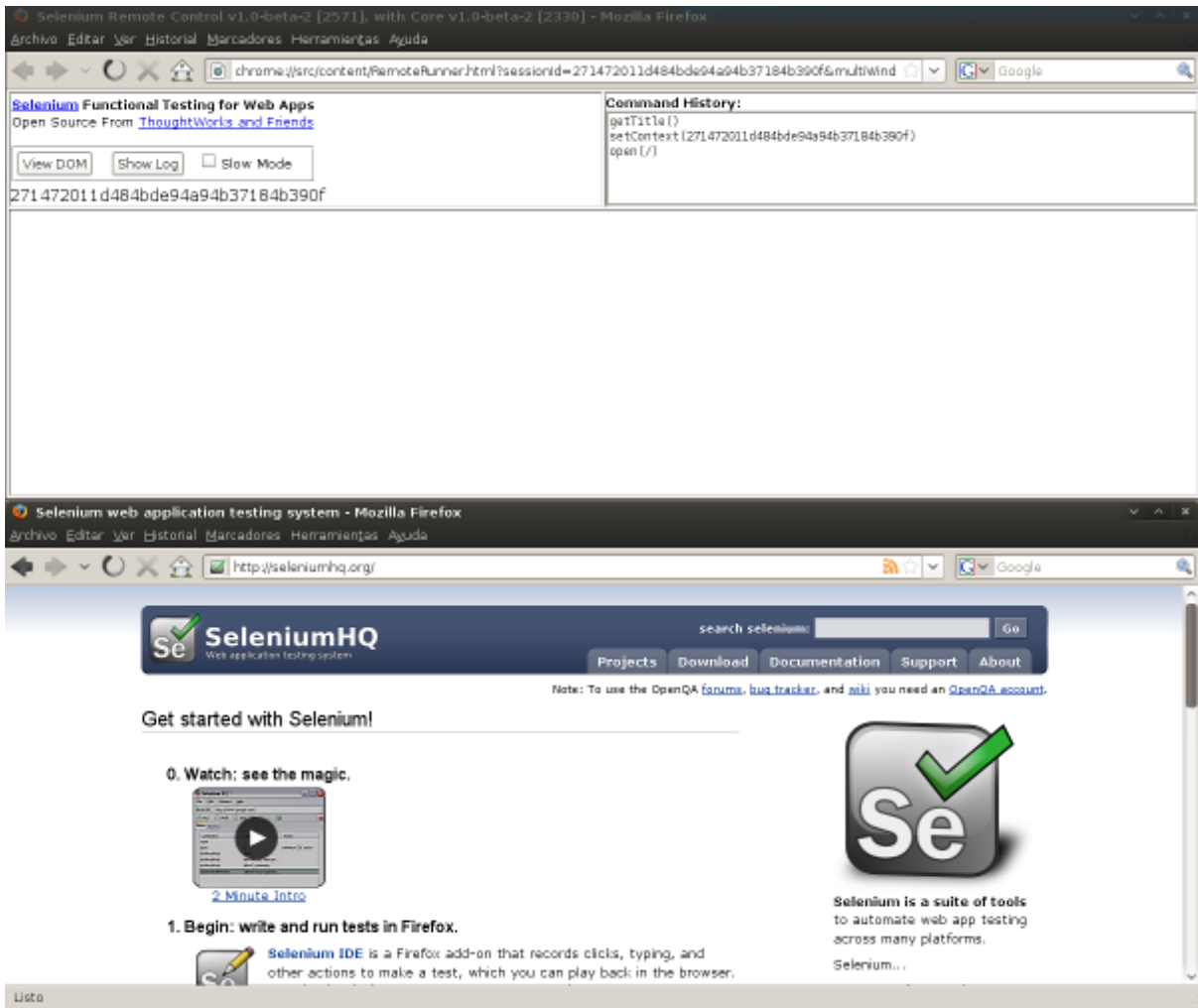
```
$ java -jar selenium-server.jar -Dhttp.proxyHost=proxy.com -Dhttp.proxyPort=8080 -Dhttp.
```

### 6.11.2 Multi-Window Mode

If you are using Selenium 1.0 you can probably skip this section, since multiwindow mode is the default behavior. However, prior to version 1.0, Selenium by default ran the application under test in a sub frame as shown here.



Some applications didn't run correctly in a sub frame, and needed to be loaded into the top frame of the window. The multi-window mode option allowed the AUT to run in a separate window rather than in the default frame where it could then have the top frame it required.



For older versions of Selenium you must specify multiwindow mode explicitly with the following option:

```
-multiwindow
```

In Selenium-RC 1.0, if you want to run your test within a single frame (i.e. using the standard for earlier Selenium versions) you can state this to the Selenium Server using the option

```
-singlewindow
```

### 6.11.3 Specifying the Firefox Profile

Firefox will not run two instances simultaneously unless you specify a separate profile for each instance. Selenium-RC 1.0 and later runs in a separate profile automatically, so if you are using Selenium 1.0, you can probably skip this section. However, if you're using an older version of Selenium or if you need to use a specific profile for your tests (such as adding an https certificate or having some addons installed), you will need to explicitly specify the profile.

First, to create a separate Firefox profile, follow this procedure. Open the Windows Start menu, select "Run", then type and enter one of the following:

```
firefox.exe -profilemanager
```

```
firefox.exe -P
```

Create the new profile using the dialog. The when you run Selenium Server, tell it to use this new Firefox profile with the server command-line option `-firefoxProfileTemplate` and specify the path to the profile using it's filename and directory path.

```
-firefoxProfileTemplate "path to the profile"
```

**Warning:** Be sure to put your profile in a new folder separate from the default!!! The Firefox profile manager tool will delete all files in a folder if you delete a profile, regardless of whether they are profile files or not.

More information about Firefox profiles can be found in [Mozilla's Knowledge Base](#)

### 6.11.4 Run Selenese Directly Within the Server Using `-htmlSuite`

You can run Selenese html files directly within the Selenium Server by passing the html file to the server's command line. For instance:

```
java -jar selenium-server.jar -htmlSuite "*firefox" "http://www.google.com" "c:\absolute"
```

This will automatically launch your HTML suite, run all the tests and save a nice HTML report with the results.

**Note:** When using this option, the server will start the tests and wait for a specified number of seconds for the test to complete; if the test doesn't complete within that amount of time, the command will exit with a non-zero exit code and no results file will be generated.

This command line is very long so be careful when you type it. Note this requires you to pass in an HTML Selenese suite, not a single test. Also be aware the `-htmlSuite` option is incompatible with `-interactive` You cannot run both at the same time.

### 6.11.5 Selenium Server Logging

#### Server-Side Logs

When launching selenium server the `-log` option can be used to record valuable debugging information reported by the Selenium Server to a text file.

```
java -jar selenium-server.jar -log selenium.log
```

This log file is more verbose than the standard console logs (it includes DEBUG level logging messages). The log file also includes the logger name, and the ID number of the thread that logged the message. For example:

```
20:44:25 DEBUG [12] org.openqa.selenium.server.SeleniumDriverResourceHandler -  
Browser 465828/:top frame1 posted START NEW
```

The message format is

```
TIMESTAMP (HH:mm:ss) LEVEL [THREAD] LOGGER - MESSAGE
```

This message may be multiline.

## Browser-Side Logs

JavaScript on the browser side (Selenium Core) also logs important messages; in many cases, these can be more useful to the end-user than the regular Selenium Server logs. To access browser-side logs, pass the **-browserSideLog** argument to the Selenium Server.

```
java -jar selenium-server.jar -browserSideLog
```

**-browserSideLog** can be combined with the **-log** argument, to log browserSideLogs (as well as all other DEBUG level logging messages) to a file.

## 6.12 Specifying the Path to a Specific Browser

You can specify to Selenium-RC a path to a specific browser. This is useful if you have different versions of the same browser, and you wish to use a specific one. Also, this is used to allow your tests to run against a browser not directly supported by Selenium-RC. When specifying the run mode, use the *\*custom* specifier followed by the full path to the browser's executable:

```
*custom <path to browser>
```

## 6.13 Troubleshooting

When first getting started with Selenium-RC there's a few potential problems that are commonly encountered. We present them along with their solutions here.

### 6.13.1 Unable to Connect to Server

When your test program cannot connect to the Selenium Server, an exception will be thrown in your test program. It should display this message or a similar one:

```
"Unable to connect to remote server....Inner Exception Message: No
connection could be made because the target machine actively refused it...."
```

```
(using .NET and XP Service Pack 2)
```

If you see a message like this, be sure you started the Selenium Server. If you did, then there is some problem with the connectivity between the two components. This should not normally happen when your operating system has typical networking and TCP/IP settings. If you continue to have trouble, try a different computer.

You can also use common networking tools like *ping*, *telnet*, *ipconfig/ifconfig* (on windows), etc to ensure you first have a valid network connection. Also, if you're trying to connect to the Selenium

Server on a remote machine try running it locally first and verifying you can get a connection using “localhost” as your connection parameter.

### 6.13.2 Unable to Load the Browser

Ok, not a very friendly error, sorry, but if the Selenium Server cannot load the browser you will probably see this error.

```
(500) Internal Server Error
```

This error seems to occur when Selenium-RC cannot load the browser.

```
500 Internal Server Error
```

(using .NET and XP Service Pack 2)

This could be caused by

- Firefox (prior to Selenium 1.0) cannot start because the browser is already open and you did not specify a separate profile. See the section on Firefox profiles under Server Options.
- The run mode you’re using doesn’t match any browser on your machine. Check the parameters you passed to Selenium when you program opens the browser.
- You specified the path to the browser explicitly (using “\*custom”—see above) but the path is incorrect. Check to be sure the path is correct. Also check the forums to be sure there are no known issues with your browser and the “\*custom” parameters.

### 6.13.3 Selenium Cannot Find the AUT

If your test program starts the browser successfully, but the browser doesn’t display the website you’re testing, the most likely cause is your test program is not using the correct URL.

This can easily happen. When you use Selenium-IDE to export you script, it inserts a dummy URL. You must manually change the URL to the correct one for your application to be tested.

### 6.13.4 Firefox Refused Shutdown While Preparing a Profile

This most often occurs when you run your Selenium-RC test program against Firefox, but you already have a Firefox browser session running and, you didn’t specify a separate profile when you started the Selenium Server. The error from the test program looks like this:

```
Error: java.lang.RuntimeException: Firefox refused shutdown while  
preparing a profile
```

Here’s the complete error msg from the server:

```
16:20:03.919 INFO - Preparing Firefox profile...  
16:20:27.822 WARN - GET /selenium-server/driver/?cmd=getNewBrowserSession&l=*fir  
efox&2=http%3a%2f%2fsage-webappl.qa.idc.com HTTP/1.1  
java.lang.RuntimeException: Firefox refused shutdown while preparing a profile  
    at org.openqa.selenium.server.browserlaunchers.FirefoxCustomProfileLaunc
```

```
her.waitForFullProfileToBeCreated(FirefoxCustomProfileLauncher.java:277)
.....
Caused by: org.openqa.selenium.server.browserlaunchers.FirefoxCustomProfileLaunc
her$FileLockRemainedException: Lock file still present! C:\DOCUME~1\jsvec\LOCALS
~1\Temp\customProfileDir203138\parent.lock
```

To resolve this, see the section on [Specifying the Firefox Profile](#)

### 6.13.5 Versioning Problems

Make sure your version of Selenium supports the version of your browser. For example, Selenium-RC 0.92 does not support Firefox 3. At times you may be lucky (I was). But don't forget to check which browser versions are supported by the version of Selenium you are using. When in doubt, use the latest release version of Selenium with the most widely used version of your browser.

### 6.13.6 Error message: “(Unsupported major.minor version 49.0)” while starting server

This error says you're not using a correct version of Java. The Selenium Server requires Java 1.5 or higher.

To check double-check your java version, run this from the command line.

```
java -version
```

You should see a message showing the Java version.

```
java version "1.5.0_07"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_07-b03)
Java HotSpot(TM) Client VM (build 1.5.0_07-b03, mixed mode)
```

If you see a lower version number, you may need to update the JRE, or you may simply need to add it to your PATH environment variable.

### 6.13.7 404 error when running the `getNewBrowserSession` command

If you're getting a 404 error while attempting to open a page on “<http://www.google.com/selenium-server/>“, then it must be because the Selenium Server was not correctly configured as a proxy. The “selenium-server” directory doesn't exist on google.com; it only appears to exist when the proxy is properly configured. Proxy Configuration highly depends on how the browser is launched with `*firefox`, `*iexplore`, `*opera`, or `*custom`.

- `*iexplore`: If the browser is launched using `*iexplore`, you could be having a problem with Internet Explorer's proxy settings. Selenium Server attempts to configure the global proxy settings in the Internet Options Control Panel. You must make sure that those are correctly configured when Selenium Server launches the browser. Try looking at your Internet Options control panel. Click on the “Connections” tab and click on “LAN Settings”.
  - If you need to use a proxy to access the application you want to test, you'll need to start Selenium Server with “`-Dhttp.proxyHost`”; see the [Proxy Configuration](#) for more details.

- You may also try configuring your proxy manually and then launching the browser with `*custom`, or with `*iehta` browser launcher.
- `*custom`: When using `*custom` you must configure the proxy correctly (manually), otherwise you'll get a 404 error. Double-check that you've configured your proxy settings correctly. To check whether you've configured the proxy correctly is to attempt to intentionally configure the browser incorrectly. Try configuring the browser to use the wrong proxy server hostname, or the wrong port. If you had successfully configured the browser's proxy settings incorrectly, then the browser will be unable to connect to the Internet, which is one way to make sure that one is adjusting the relevant settings.
- For other browsers (`*firefox`, `*opera`) we automatically hard-code the proxy for you, and so there are no known issues with this functionality. If you're encountering 404 errors and have followed this user guide carefully post your results to user forums for some help from the user community.

### 6.13.8 Why am I getting a permission denied error?

The most common reason for this error is that your session is attempting to violate the same-origin policy by crossing domain boundaries (e.g., accesses a page from `http://domain1` and then accesses a page from `http://domain2`) or switching protocols (moving from `http://domainX` to `https://domainX`). For this to be solved, try using the 'Heightened Privileges Browsers' if you're working with the Proxy Injection browsers. This is covered in some detail in the tutorial. Make sure you read the section about the The Same Origin Policy, Proxy Injection carefully.

If the previous situation was not your case, it can also occur when JavaScript attempts to look at objects which are not yet available (before the page has completely loaded), or tries to look at objects which are no longer available (after the page has started to be unloaded). This is most typically encountered with AJAX pages which are working with sections of a page or subframes that load and/or reload independently of the larger page. For this type of problem, it is common that the error is intermittent. Often it is impossible to reproduce the problem with a debugger because the trouble stems from race conditions which are not reproducible when the debugger's overhead is added to the system. Try first adding a static pause to make sure this is the situation and then moving on to the `waitFor` kind of commands: *waitFor commands in Selenese Chapter*

### 6.13.9 Running Tests with Different Browser Configurations

Normally Selenium-RC automatically configures the browser, but if you launch the browser using the `"*custom"` run mode, you can force Selenium RC to launch the browser as-is, without using an automatic configuration. (Note that this is also the way one launches other browsers that Selenium RC doesn't yet explicitly support.)

For example, you can launch Firefox with a custom configuration like this:

```
cmd=getNewBrowserSession&l=*custom c:\Program Files\Mozilla Firefox\firefox.exe&2=htt
```

Note that when launching the browser this way, you must manually configure the browser to use the Selenium Server as a proxy. Normally this just means opening your browser preferences and specifying "localhost:4444" as an HTTP proxy, but instructions for this can differ radically from browser to browser. Consult your browser's documentation for details.

Beware that Mozilla browsers can be a little fussy about how they start and stop. One may need to set the `MOZ_NO_REMOTE` environment variable to make Mozilla browsers behave a little more predictably.

Unix users should avoid launching the browser using a shell script; it's generally better to use the binary executable (e.g. `firefox-bin`) directly.

### 6.13.10 How to Block Popup Windows?

There are several kinds of "Popups" that you can get during a Selenium test. You may not be able to close these popups by running selenium commands if they are initiated by the browser as opposed to your AUT. Therefore, you'll need to know how to manage these. Each type needs to be addressed differently.

- HTTP basic authentication dialogs: These dialogs prompt for a username/password to login to the site. To login to a site that requires HTTP basic authentication, use a username and password in the URL, as described in [RFC 1738](#), like this: `open("http://myusername:myuserpassword@myexample.com/blah/blah/blah")`.
- SSL certificate warnings: Selenium RC automatically attempts to spoof SSL certificates when it is enabled as a proxy; see more on this in the section on HTTPS. If your browser is configured correctly, you should never see SSL certificate warnings, but you may need to configure your browser to trust our dangerous "CyberVillains" SSL certificate authority. Again, refer to the HTTPS section for how to do this.
- modal JavaScript alert/confirmation/prompt dialogs: Selenium tries to conceal those dialogs from you (by replacing `window.alert`, `window.confirm` and `window.prompt`) so they won't stop the execution of your page. If you're actually seeing an alert pop-up, it's probably because it fired during the page load process, which is usually too early for us to protect the page. Selenese contains commands for asserting or verifying alert and confirmation popups. See the sections on these topics in Chapter 4. (Note at this time of writing we haven't written those sections, but intend to do so very soon).

### 6.13.11 On Linux, why isn't my Firefox browser session closing?

On Unix/Linux you must invoke "firefox-bin" directly, so make sure that executable is on the path. If executing Firefox through a shell script, when it comes time to kill the browser Selenium RC will kill the shell script, leaving the browser running. You can specify the path to `firefox-bin` directly, like this.

```
cmd=getNewBrowserSession&1=*firefox /usr/local/firefox/firefox-bin&2=http://www.google.c
```

### 6.13.12 Firefox \*chrome doesn't work with custom profile

Check Firefox profile folder -> `prefs.js` -> `user_pref("browser.startup.page", 0)`; Comment this line like this: `//user_pref("browser.startup.page", 0);` and try again.

### 6.13.13 How can I avoid using complex xpath expressions to my test?

If the elements in HTML (button, table, label, etc) have element IDs, then one can reliably retrieve all elements without ever resorting to xpath. These element IDs should be explicitly created by the application. But non-descriptive element ID (i.e. `id_147`) tends to cause two problems: first, each time the application is deployed, different element ids could be generated. Second, a non-specific element id

makes it hard for automation testers to keep track of and determine which element ids are required for testing.

You might consider trying the [UI-Element](#) extension in this situation.

### 6.13.14 Is it ok to load a custom pop-up as the parent page is loading (i.e., before the parent page's javascript window.onload() function runs)?

No. Selenium relies on interceptors to determine window names as they are being loaded. These interceptors work best in catching new windows if the windows are loaded AFTER the onload() function. Selenium may not recognize windows loaded before the onload function.

### 6.13.15 Problems With Verify Commands

If you export your tests from Selenium-IDE, you may find yourself getting empty verify strings from your tests (depending on the programming language used).

*Note: This section is not yet developed.*

### 6.13.16 Safari and MultiWindow Mode

If you are using Safari 4 to run your tests and you only get Selenium's window open while the second window with your AUT isn't, make sure that the "Block Pop-Up Windows" preference is deactivated.

This can also be a problem when you use `openWindow()`

More info: [http://jira.openqa.org/browse/SEL-639#action\\_17735](http://jira.openqa.org/browse/SEL-639#action_17735)

### 6.13.17 Firefox on Linux

On Unix/Linux, versions of Selenium before 1.0 needed to invoke "firefox-bin" directly, so if you are using a previous version, make sure that the real executable is on the path.

On most Linux distributions, the real `firefox-bin` is located on:

```
/usr/lib/firefox-x.x.x/
```

Where the x.x.x is the version number you currently have. So, to add that path to the user's path, you will have to add the following to your `.bashrc` file:

```
export PATH="$PATH:/usr/lib/firefox-x.x.x/"
```

If necessary, you can specify the path to `firefox-bin` directly in your test, like this:

```
" *firefox /usr/lib/firefox-x.x.x/firefox-bin "
```

### 6.13.18 IE and Style Attributes

If you are running your tests on Internet Explorer and you are trying to locate elements using their `style` attribute, you're definitely in trouble. Probably a locator like this:

```
//td[@style="background-color:yellow"]
```

Would perfectly work in Firefox, Opera or Safari but it won't work on IE. That's because the keys in *@style* are interpreted as uppercase once the page is parsed by IE. So, even if the source code is in lowercase, you should use:

```
//td[@style="BACKGROUND-COLOR:yellow"]
```

This is a problem if your test is intended to work on multiple browsers, but you can easily code your test to detect the situation and try the alternative locator that only works in IE.

### **6.13.19 Where should I go if I have questions about Selenium RC that aren't answered in this FAQ?**

Try our [user forums](#)



# TEST DESIGN CONSIDERATIONS

*NOTE: This chapter is not finished. We have some content here already though. We decided not to hold back on information just because a chapter was not ready.*

## 7.1 Introducing Test Design Options

*This section is not yet developed.*

## 7.2 Web Page Content – Static vs. Dynamic Pages

*This section is not yet developed.*

This topic is explained as - Object identification for Static content and Object identification for Dynamic contents. The examples described here are specific to Java and must have equivalent in the language of your choice.

### 7.2.1 Object Identification for Static HTML Objects

*This section has not been reviewed or edited.*

Static HTML Objects might look as:

```
<a class="button" id="adminHomeForm" onclick="return oamSubmitForm('adminHomeForm', 'a
```

This is HTML snippet for a button and its id is “adminHomeForm”. This id remains constant within the all occurrences of page. Hence to click this button you just have to use the following selenium command:

```
selenium.click("adminHomeForm");
```

### 7.2.2 Object identification with Dynamic HTML Objects

*This section has not been reviewed or edited.*

Dynamic HTML of an object might look as:

```
<input type="checkbox" value="true" id="addForm:_id74:_id75:0:_id79:0:checkBox" name=
```

This is HTML snippet for a check box. Its id and name (addForm:\_id74:\_id75:0:\_id79:0:checkBox) both are same and both are dynamic (they will change the next time you open the application). In this case normal object identification would look like:

```
selenium.click("addForm:_id74:_id75:0:_id79:0:checkBox);
```

Given the dynamic nature of id this approach would not work. The best way is to capture this id dynamically from the website itself. It can be done as:

```
String[] checkBoxIds = selenium.getAllFields(); // Collect all input ids on page.
if(!GenericValidator.IsBlankOrNull(checkBoxIds[i])) // If collected id is not null.
{
    // If the id starts with addForm
    if(checkBoxIds[i].indexOf("addForm") > -1) {
        selenium.check(checkBoxIds[i]);
    }
}
```

This approach will work only if there is one field whose id has got the text 'addForm' appended to it.

Consider one more example of a Dynamic object. A page with two links having the same name (one which appears on page) and same html name. Now if href is used to click the link, it would always be clicking on first element. Click on second element link can be achieved as following:

```
// Flag for second appearance of link.
boolean isSecondInstanceLink = false;

// Desired link.
String editInfo = null;

// Collect all links.
String[] links = selenium.getAllLinks();

// Loop through collected links.
for(String linkID: links) {

    // If retrieved link is not null
    if(!GenericValidator.IsBlankOrNull(linkID)) {

        // Find the inner HTML of link.
        String editTermSectionInfo = selenium.getEval("window.document.getElementById('

// If retrieved link is expected link.
if(editTermSectionInfo.equalsIgnoreCase("expectedlink")) {

    // If it is second appearance of link then save the link id and break the loop
    if(isSecondInstanceLink) {
        editInfo = linkID;
        break;
    }

    // Set the second appearance of Autumn term link to true as
    isSecondInstanceLink = true;
}
}
}
```

```
// Click on link.
selenium.click(editInfo);
```

## 7.3 Location Strategies

*This section is not yet developed.*

## 7.4 UI Mapping with Selenium

*This section has not been reviewed or edited.*

UI Map is a repository for all Objects of test scripts. Advantages of using UI Maps are:

- Having centralized location for UI objects instead of having them scattered through out the script.
- Centralized location of objects makes maintenance of Tests easy.
- Arcane HTML ids and names can be given comprehensible names and it increases readability of scripts.

Consider following example (in java) of selenium tests for a website:

```
public void testNew() throws Exception {
    selenium.open( "http://www.test.com" );
    selenium.type( "loginForm:tbUsername", "xxxxxxxx" );
    selenium.click( "loginForm:btnLogin" );
    selenium.click( "adminHomeForm:_activitynew" );
    selenium.waitForPageToLoad( "30000" );
    selenium.click( "addEditEventForm:_idcancel" );
    selenium.waitForPageToLoad( "30000" );
    selenium.click( "adminHomeForm:_activityold" );
    selenium.waitForPageToLoad( "30000" );
}
```

There is hardly any thing comprehensible from script. Even the regular users of application would not be able to figure out as to what script does. A better script would have been:

```
public void testNew() throws Exception {
    selenium.open( "http://www.test.com" );
    selenium.type( admin.username, "xxxxxxxx" );
    selenium.click( admin.loginbutton );
    selenium.click( admin.events.createnewevent );
    selenium.waitForPageToLoad( "30000" );
    selenium.click( admin.events.cancel );
    selenium.waitForPageToLoad( "30000" );
    selenium.click( admin.events.viewoldevents );
    selenium.waitForPageToLoad( "30000" );
}
```

Though again there are no comments provided in the script but it is more comprehensible because of the keywords used in scripts. (please beware that UI Map is not replacement of comments) So a more comprehensible script would look as following -

```
public void testNew() throws Exception {  
  
    // Open app url.  
    selenium.open( "http://www.test.com" );  
  
    // Provide admin username.  
    selenium.type( admin.username, "xxxxxxxx" );  
  
    // Click on Login button.  
    selenium.click( admin.loginbutton );  
  
    // Click on Create New Event button.  
    selenium.click( admin.events.createnewevent );  
    selenium.waitForPageToLoad( "30000" );  
  
    // Click on Cancel button.  
    selenium.click( admin.events.cancel );  
    selenium.waitForPageToLoad( "30000" );  
  
    // Click on View Old Events button.  
    selenium.click( admin.events.viewoldevents );  
    selenium.waitForPageToLoad( "30000" );  
}
```

Herein whole idea is to have centralized location for objects and using comprehensible names for objects. To achieve this properties files can be used in java. Properties file contains key/value pairs, where in key and value both are String values.

Consider a property file *prop.properties* which has got definition of HTML object used above

```
admin.username = loginForm:tbUsername  
admin.loginbutton = loginForm:btnLogin  
admin.events.createnewevent = adminHomeForm:_activitynew  
admin.events.cancel = addEditEventForm:_idcancel  
admin.events.viewoldevents = adminHomeForm:_activityold
```

Hence still our objects refer to html objects but we have introduced a layer of abstraction between test script and UI elements. Values can be read from properties file and used in Test Class to implement UI Map. For more on Properties files follow this [URL](#).

## 7.5 Bitmap Comparison

*This section has not been developed yet.*

## 7.6 Recovery From Failure

*This section has not been developed yet.*

## 7.7 Types of Tests

*This section has not been developed yet.*

- Page Rendering Tests
- Forms Tests
- Specific Function Tests - For example, if a .jsp is called from used to retrieve data based on parameter input.
- User Scenario Test - A Multiple Page Functional Test

## 7.8 Solving Common Web-App Problems

*This section has not been developed yet.*

- Assert vs Verify (or possibly put this under script development)
- Judgement calls, when to `verifyTextPresent`, `verifyElementPresent`, or `verifyText`.
- Handling Login/Logout State
- Processing a Result Set

## 7.9 Interpreting Test Results

*This section has not been developed yet.*

## 7.10 Organizing Your Test Scripts

*This section has not been developed yet.*

## 7.11 Organizing Your Test Suites

*This section has not been developed yet.*

### 7.11.1 Data Driven Testing

*This section needs an introduction and it has not been completed yet.*

**In Python:**

```
# Collection of String values
source = open("input_file.txt", "r")
values = source.readlines()
source.close()
# Execute For loop for each String in the values array
for search in values:
```

```
sel.open( " / " )
sel.type( " q ", search)
sel.click( " btnG " )
sel.waitForPageToLoad( " 30000 " )
self.failUnless( sel.is_text_present( " Results * for " + search))
```

Why would we want a separate file with data in it for our tests? One important method of testing concerns running the same test repetetively with differnt data values. This is called *Data Driven Testing* and is a very common testing task. Test automation tools, Selenium included, generally handle this as it's often a common reason for building test automation to support manual testing methods.

The Python script above opens a text file. This file contains a different search string on each line. The code then saves this in an array of strings, and at last, it's iterating over the strings array and doing the search and assert on each.

This is a very basic example of what you can do, but the idea is to show you things that can easily be done with either a programming or scripting language when they're difficult or even impossible to do using Selenium-IDE.

Refer to [Selnium RC wiki](#) for examples on reading data from spread sheet or using data provider capabilities of TestNG with java client driver.

### 7.11.2 Error Handling

*Note: This section is not yet developed.*

A quick note though—recognize that your programming language's exception- handling support can be used for error handling and recovery.

### 7.11.3 Database Validations

Since you can also do database queries from your favorite programming language, assuming you have database support functions, why not using them for some data validations/retrieval on the Application Under Test?

Consider example of Registration process where in registered email address is to be retrieved from database. Specific cases of establishing DB connection and retrieving data from DB would be:

#### **In Java:**

```
// Load Microsoft SQL Server JDBC driver.
Class.forName( "com.microsoft.sqlserver.jdbc.SQLServerDriver" );

// Prepare connection url.
String url = "jdbc:sqlserver://192.168.1.180:1433;DatabaseName=TEST_DB" ;

// Get connection to DB.
public static Connection con =
DriverManager.getConnection(url, "username", "password" );

// Create statement object which would be used in writing DDL and DML
// SQL statement.
public static Statement stmt = con.createStatement();
```

```
// Send SQL SELECT statements to the database via the Statement.executeQuery
// method which returns the requested information as rows of data in a
// ResultSet object.

ResultSet result = stmt.executeQuery
( "select top 1 email_address from user_register_table" );

// Fetch value of "email_address" from "result" object.
String emailaddress = result.getString( "email_address" );

// Use the fetched value to login to application.
selenium.type( "userid", emailaddress);
```

This is very simple example of data retrieval from DB in Java. A more complex test could be to validate that inactive users are not able to login to application. This wouldn't take too much work from what you've already seen.

## 7.12 Example Test Suites

*This section has not been developed yet.*



# SELENIUM-GRID

Please refer to the Selenium Grid website

[http://selenium-grid.seleniumhq.org/how\\_it\\_works.html](http://selenium-grid.seleniumhq.org/how_it_works.html)

*This section is not yet developed. If there is a member of the community who is experienced in Selenium-Grid, and would like to contribute, please contact the Documentation Team. We would love to have you contribute.*



# USER-EXTENSIONS

*NOTE: This section is close to completion, but it has not been reviewed and edited.*

## 9.1 Introduction

It can be quite simple to extend Selenium, adding your own actions, assertions and locator-strategies. This is done with JavaScript by adding methods to the Selenium object prototype, and the PageBot object prototype. On startup, Selenium will automatically look through methods on these prototypes, using name patterns to recognize which ones are actions, assertions and locators. The following examples try to give an indication of how Selenium can be extended with JavaScript.

## 9.2 Actions

All methods on the Selenium prototype beginning with “do” are added as actions. For each action foo there is also an action fooAndWait registered. An action method can take up to two parameters, which will be passed the second and third column values in the test. Example: Add a “typeRepeated” action to Selenium, which types the text twice into a text box.

```
Selenium.prototype.doTypeRepeated = function(locator, text) {
  // All locator-strategies are automatically handled by "findElement"
  var element = this.page().findElement(locator);

  // Create the text to type
  var valueToType = text + text;

  // Replace the element text with the new text
  this.page().replaceText(element, valueToType);
};
```

## 9.3 Accessors/Assertions

All getFoo and isFoo methods on the Selenium prototype are added as accessors (storeFoo). For each accessor there is an assertFoo, verifyFoo and waitForFoo registered. An assert method can take up to 2 parameters, which will be passed the second and third column values in the test. You can also define your own assertions literally as simple “assert” methods, which will also auto-generate “verify” and “waitFor” commands. Example: Add a valueRepeated assertion, that makes sure that the element

value consists of the supplied text repeated. The 2 commands that would be available in tests would be `assertValueRepeated` and `verifyValueRepeated`.

```
Selenium.prototype.assertValueRepeated = function(locator, text) {
  // All locator-strategies are automatically handled by "findElement"
  var element = this.page().findElement(locator);

  // Create the text to verify
  var expectedValue = text + text;

  // Get the actual element value
  var actualValue = element.value;

  // Make sure the actual value matches the expected
  Assert.matches(expectedValue, actualValue);
};
```

### 9.3.1 Automatic availability of `storeFoo`, `assertFoo`, `assertNotFoo`, `waitForFoo` and `waitForNotFoo` for every `getFoo`

All `getFoo` and `isFoo` methods on the Selenium prototype automatically result in the availability of `storeFoo`, `assertFoo`, `assertNotFoo`, `verifyFoo`, `verifyNotFoo`, `waitForFoo`, and `waitForNotFoo` commands. Example, if you add a `getTextLength()` method, the following commands will automatically be available: `storeTextLength`, `assertTextLength`, `assertNotTextLength`, `verifyTextLength`, `verifyNotTextLength`, `waitForTextLength`, and `waitForNotTextLength` commands.

```
Selenium.prototype.getTextLength = function(locator, text) {
  return this.getText(locator).length;
};
```

Also note that the `assertValueRepeated` method described above could have been implemented using `isValueRepeated`, with the added benefit of also automatically getting `assertNotValueRepeated`, `storeValueRepeated`, `waitForValueRepeated` and `waitForNotValueRepeated`.

## 9.4 Locator Strategies

All `locateElementByFoo` methods on the PageBot prototype are added as locator-strategies. A locator strategy takes 2 parameters, the first being the locator string (minus the prefix), and the second being the document in which to search. Example: Add a “`valuerepeated=`” locator, that finds the first element a value attribute equal to the the supplied value repeated.

```
// The "inDocument" is a the document you are searching.
PageBot.prototype.locateElementByValueRepeated = function(text, inDocument) {
  // Create the text to search for
  var expectedValue = text + text;

  // Loop through all elements, looking for ones that have
  // a value === our expected value
  var allElements = inDocument.getElementsByTagName("*");
  for (var i = 0; i < allElements.length; i++) {
    var testElement = allElements[i];
```

```

        if (testElement.value && testElement.value === expectedValue) {
            return testElement;
        }
    }
    return null;
};

```

## 9.5 Using User-Extensions With Selenium-IDE

User-extensions are very easy to use with the selenium IDE.

1. Create your user extension and save it as user-extensions.js. While this name isn't technically necessary, it's good practice to keep things consistent.
2. Open Firefox and open Selenium-IDE.
3. Click on Tools, Options
4. In Selenium Core Extensions click on Browse and find the user-extensions. js file. Click on OK.
5. Your user-extension will not yet be loaded, you must close and restart Selenium-IDE.
6. In your empty test, create a new command, your user-extension should now be an options in the Commands dropdown.

## 9.6 Using User-Extensions With Selenium RC

If you Google “Selenium RC user-extension” ten times you will find ten different approaches to using this feature. Below, is the official Selenium suggested approach.

### 9.6.1 Example

C#

1. Place your user extension in the same directory as your Selenium Server.
2. If you are using client code generated by the Selenium-IDE you will need to make a couple small edits. First, you will need to create an `HttpCommandProcessor` object with class scope (outside the `SetupTest` method, just below `private StringBuilder verificationErrors;`)

```
HttpCommandProcessor proc;
```

1. Next, instantiate that `HttpCommandProcessor` object as you would the `DefaultSelenium` object. This can be done in the test setup.

```
proc = new HttpCommandProcessor("localhost", 4444, "*iexplore", "http://google.ca/");
```

1. Instantiate the `DefaultSelenium` object using the `HttpCommandProcessor` object you created.

```
selenium = new DefaultSelenium(proc);
```

1. Within your test code, execute your user-extension by calling it with the `DoCommand()` method of `HttpCommandProcessor`. This method takes two arguments: a string to identify the user-extension method you want to use and string array to pass arguments. Notice that the first letter of your function is lower case, regardless of the capitalization in your user-extension. Selenium automatically does this to keep common JavaScript naming conventions. Because JavaScript is case sensitive, your test will fail if you begin this command with a capital. `inputParams` is the array of arguments you want to pass to the JavaScript user-extension. In this case there is only one string in the array because there is only one parameter for our user extension, but a longer array will map each index to the corresponding user-extension parameter. Remember that user extensions designed for Selenium-IDE will only take two arguments.

```
string[] inputParams = { "Hello World" };  
proc.DoCommand( "alertWrapper" , inputParams);
```

1. Start the test server using the `-userExtensions` argument and pass in your `user-extensions.js` file.

```
java -jar selenium-server.jar -userExtensions user-extensions.js
```

```
using System;  
using System.Text;  
using System.Text.RegularExpressions;  
using System.Threading;  
using NUnit.Framework;  
using Selenium;  
  
namespace SeleniumTests  
{  
    [TestFixture]  
    public class NewTest  
    {  
  
        private ISelenium selenium;  
        private StringBuilder verificationErrors;  
        private HttpCommandProcessor proc;  
  
        [SetUp]  
        public void SetupTest()  
        {  
            proc = new HttpCommandProcessor( "localhost" , 4444, "*iexpl  
            selenium = new DefaultSelenium(proc);  
            //selenium = new DefaultSelenium("localhost", 4444, "*iexpl  
            selenium.Start();  
            verificationErrors = new StringBuilder();  
  
        }  
    }  
}
```

```
[TearDown]
public void TeardownTest()
{
    try
    {
        selenium.Stop();
    }
    catch (Exception)
    {
        // Ignore errors if unable to close the browser
    }
    Assert.AreEqual("", verificationErrors.ToString());
}

[Test]
public void TheNewTest()
{
    selenium.Open("/");
    string[] inputParams = { "Hello World", };
    proc.DoCommand("alertWrapper", inputParams);
}
}
```

End

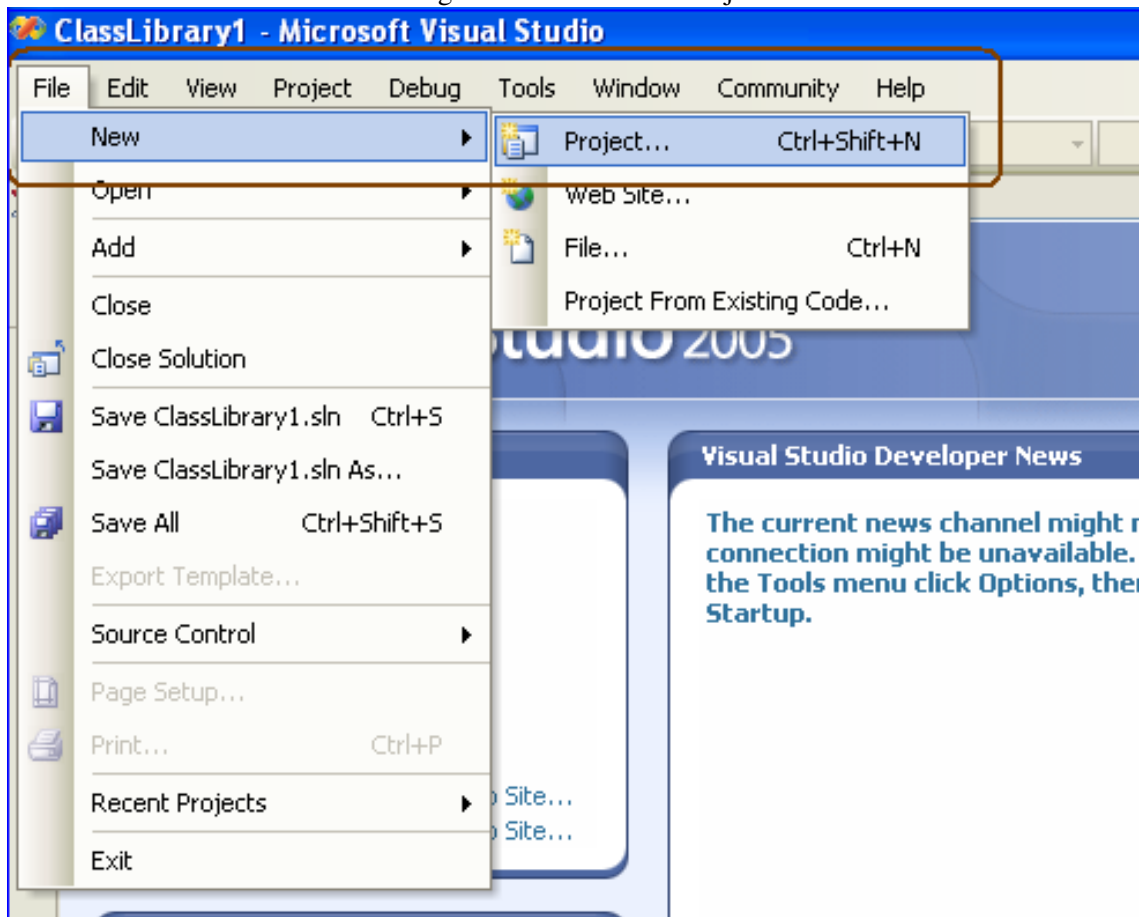
Appendixes:



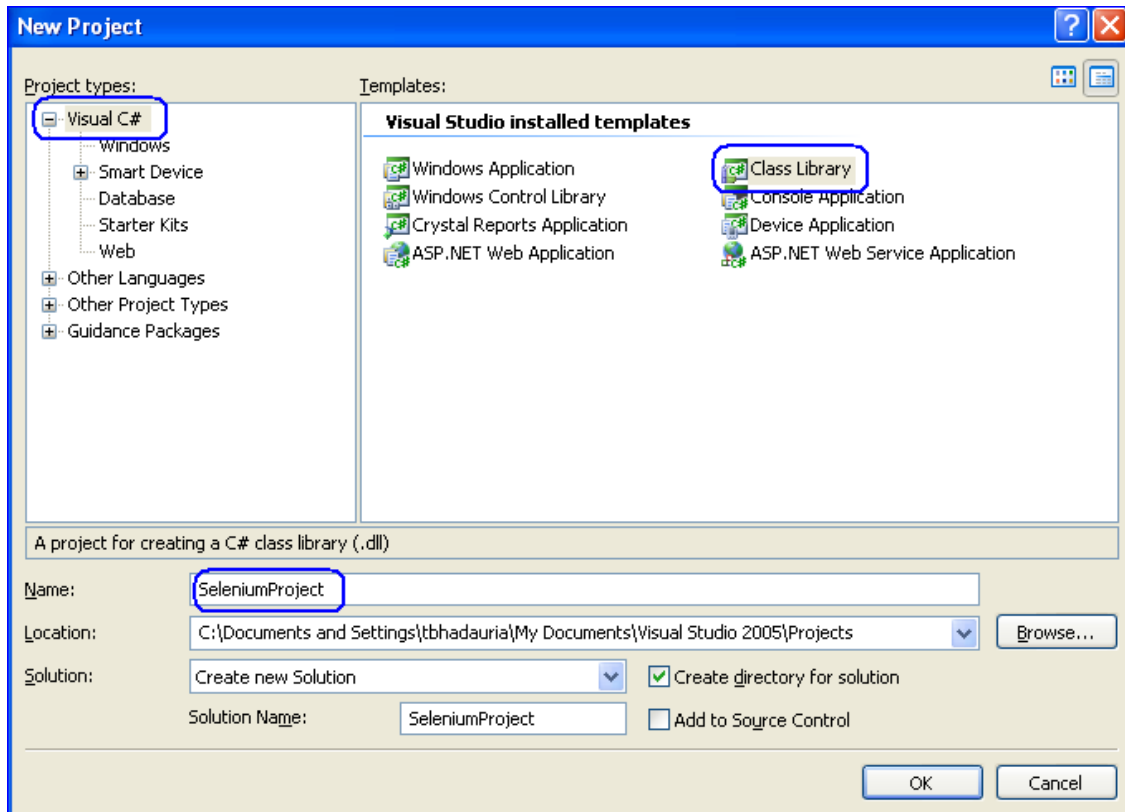
# .NET CLIENT DRIVER CONFIGURATION

.NET client Driver can be used with Microsoft Visual Studio. To Configure it with Visual do as Following.

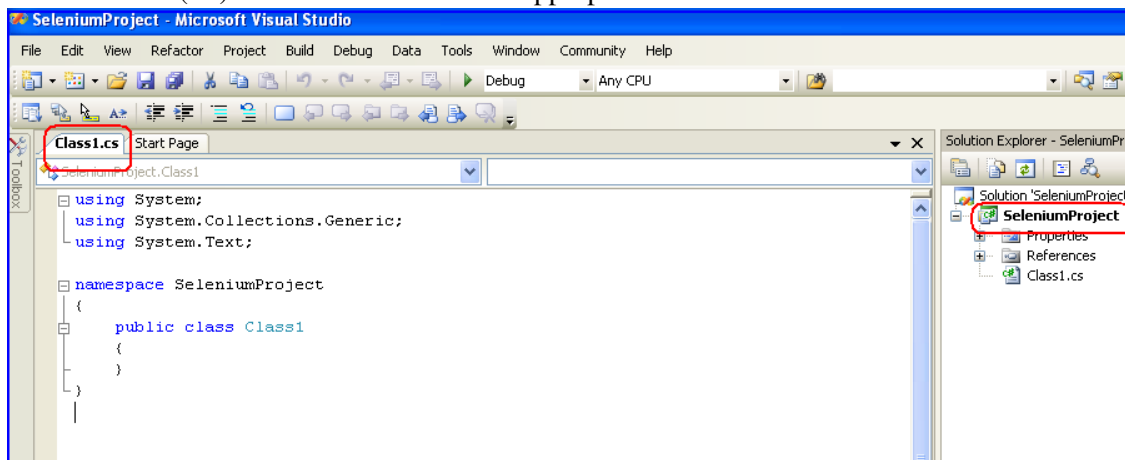
- Launch Visual Studio and navigate to File > New > Project.



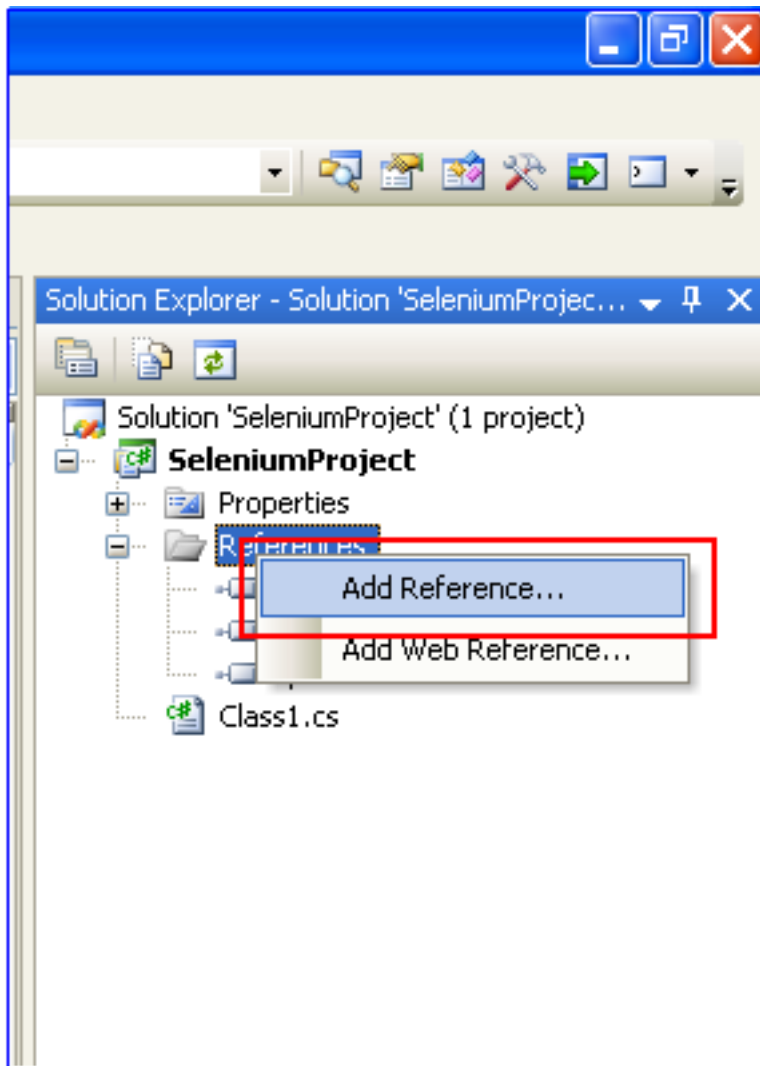
- Select Visual C# > Class Library > Name your project > Click on OK button.



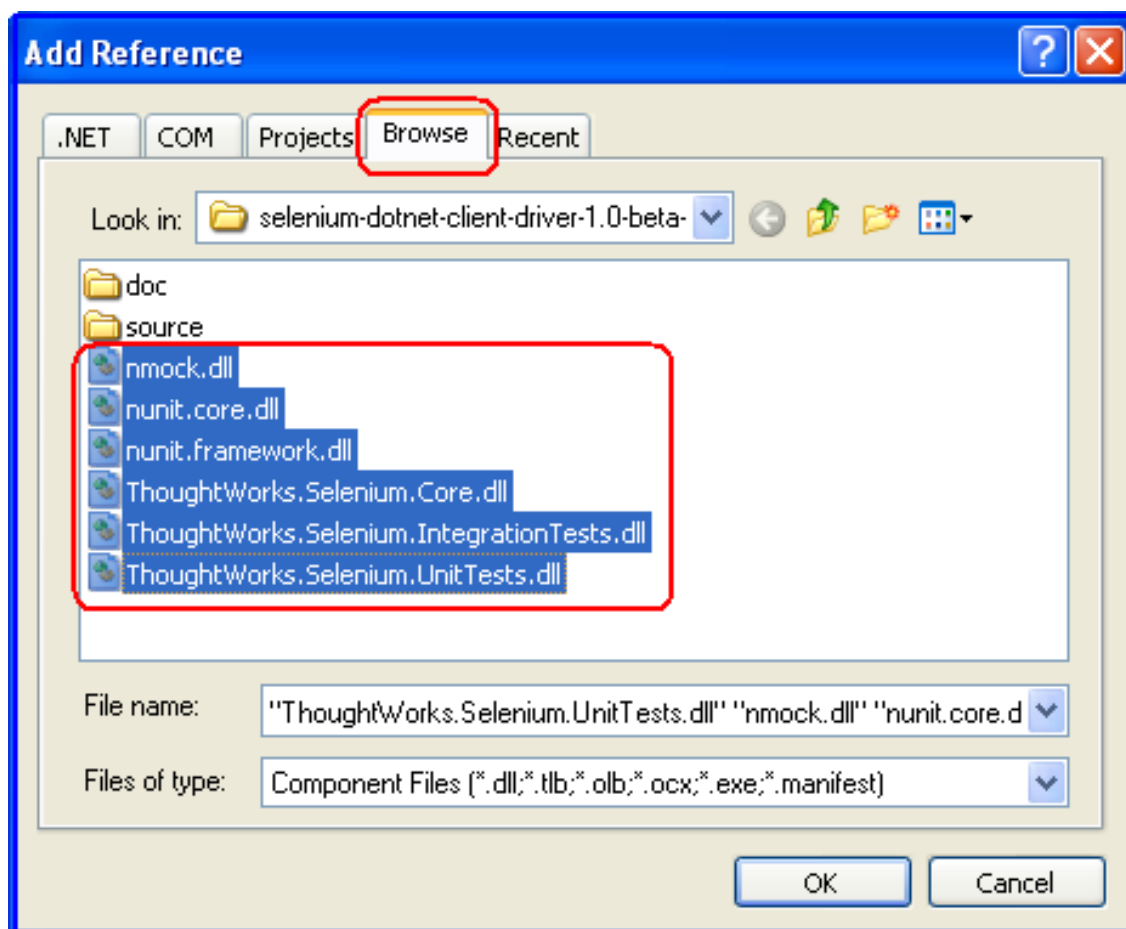
- A Class (.cs) is created. Rename it as appropriate.



- Under right hand pane of Solution Explorer right click on References > Add References.



- Select following dll files - nmock.dll, nunit.core.dll, nunit.framework.dll, ThoughtWorks.Selenium.Core.dll, ThoughtWorks.Selenium.IntegrationTests.dll, ThoughtWorks.Selenium.UnitTests.dll and click on Ok button



With This Visual Studio is ready for Selenium Test Cases.

# JAVA CLIENT DRIVER CONFIGURATION

In General configuration of Selenium-RC with any java IDE would have following steps:

- Download Selenium-RC from the SeleniumHQ [downloads page](#)
- Start any java IDE
- Create new project
- Add to your project classpath selenium-java-client-driver.jar
- Record your test from Selenium-IDE and translate it to java code (Selenium IDE has automatic translation feature to generate tests in variety of languages)
- Run selenium server from console
- Run your test in the IDE

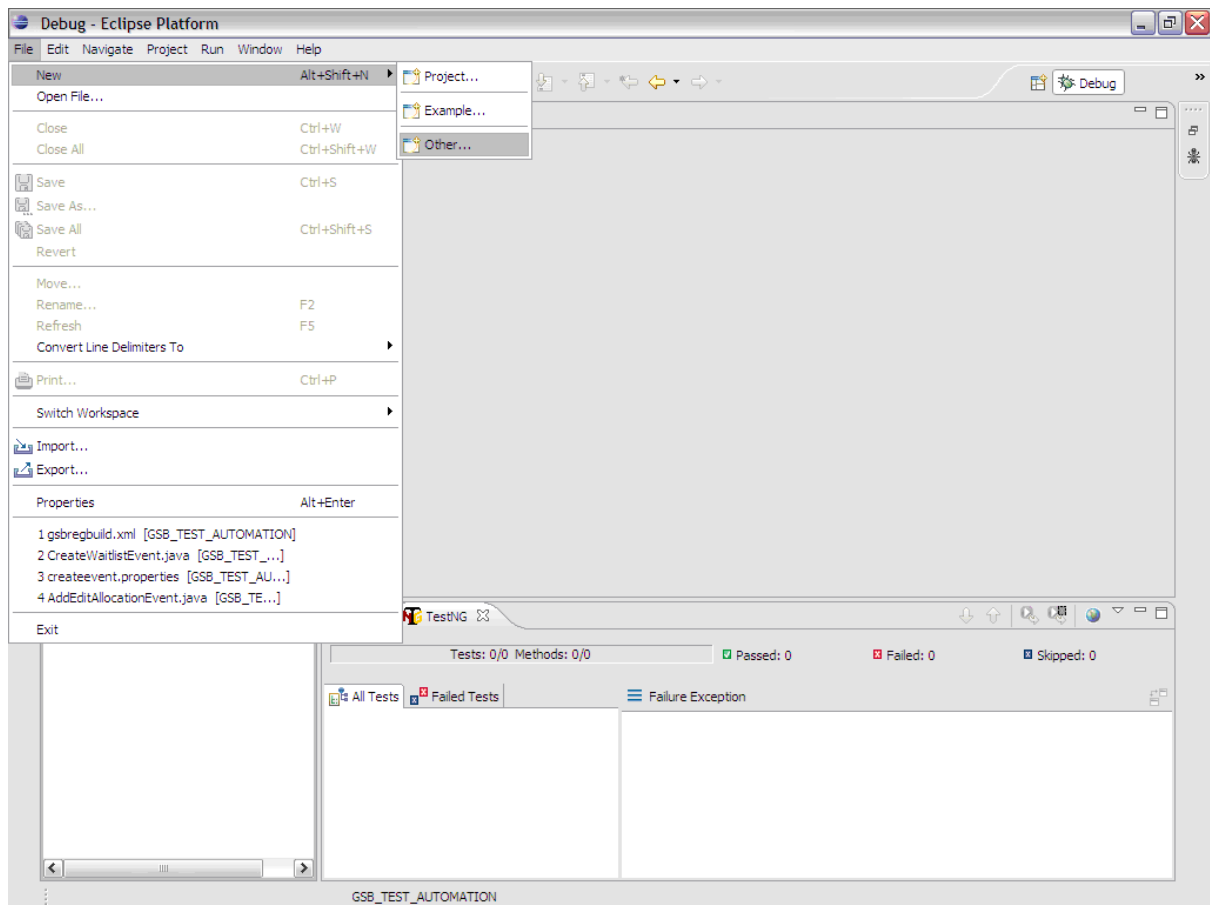
These points have been delineated below with reference to Eclipse and IntelliJ:

## 11.1 Configuring Selenium-RC With Eclipse

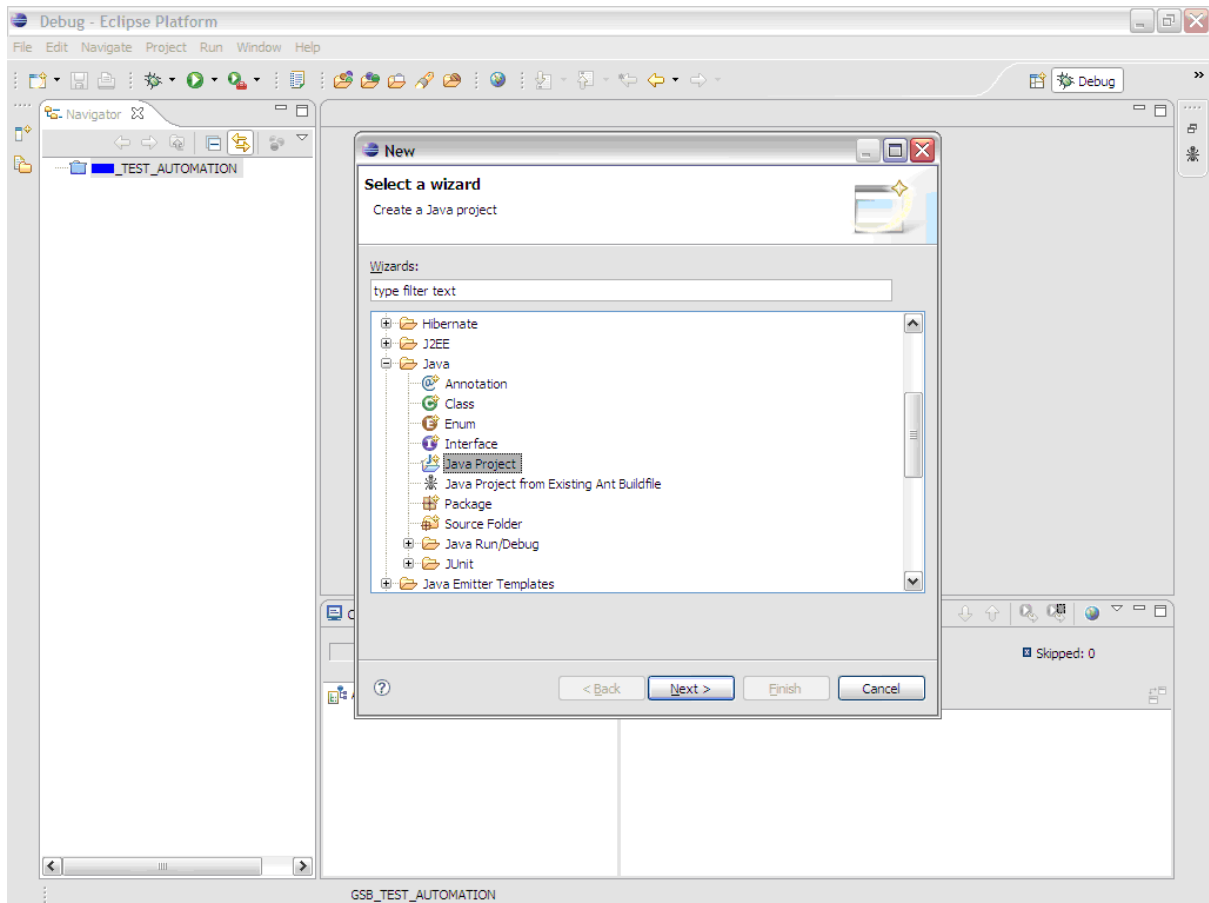
**Eclipse** is a multi-language software development platform comprising an IDE and a plug-in system to extend it. It is written primarily in Java and is used to develop applications in this language and, by means of the various plug-ins, in other languages as well as C/C++, Cobol, Python, Perl, PHP and more.

Following lines describes configuration of Selenium-RC with Eclipse - Version: 3.3.0. (Europa Release). It should not be too different for higher versions of Eclipse

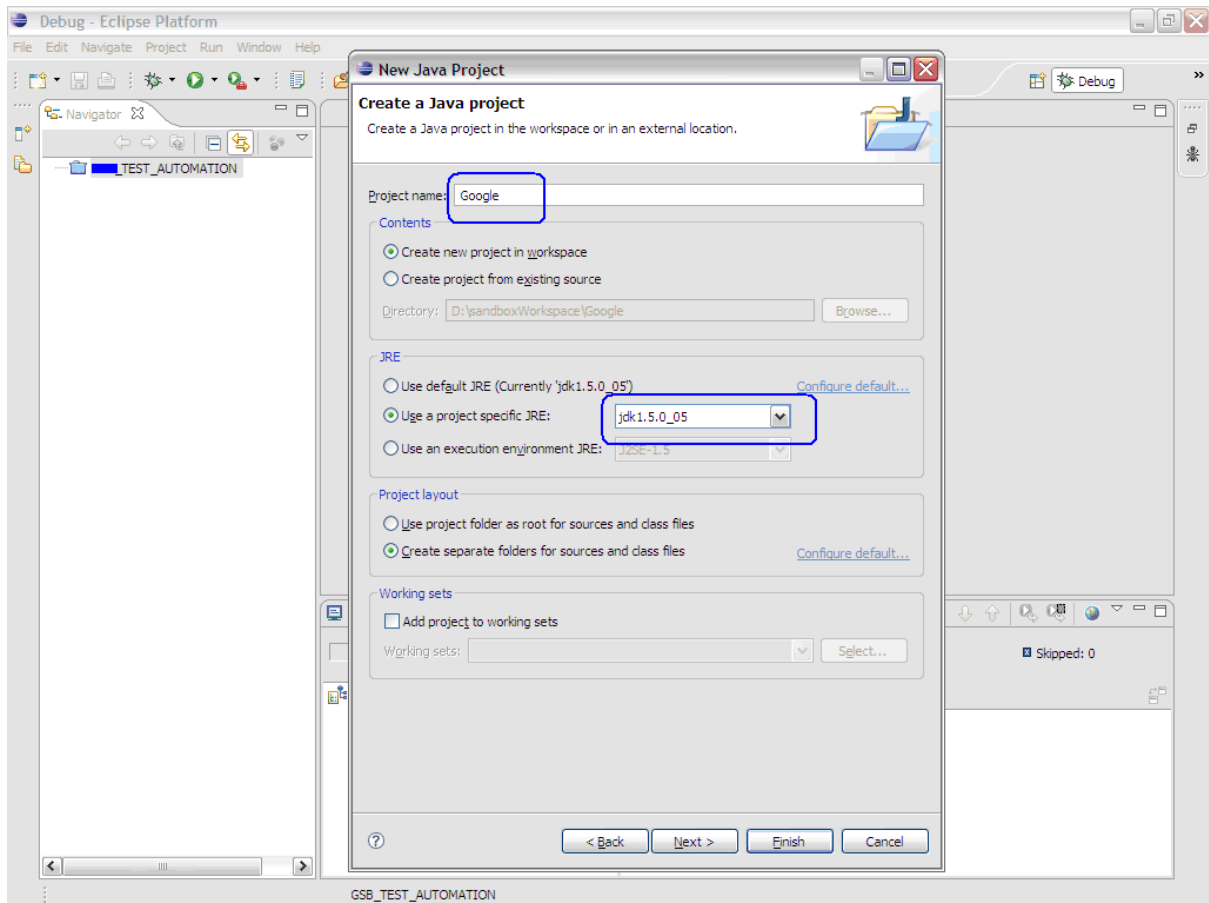
- Launch Eclipse.
- Select File > New > Other.



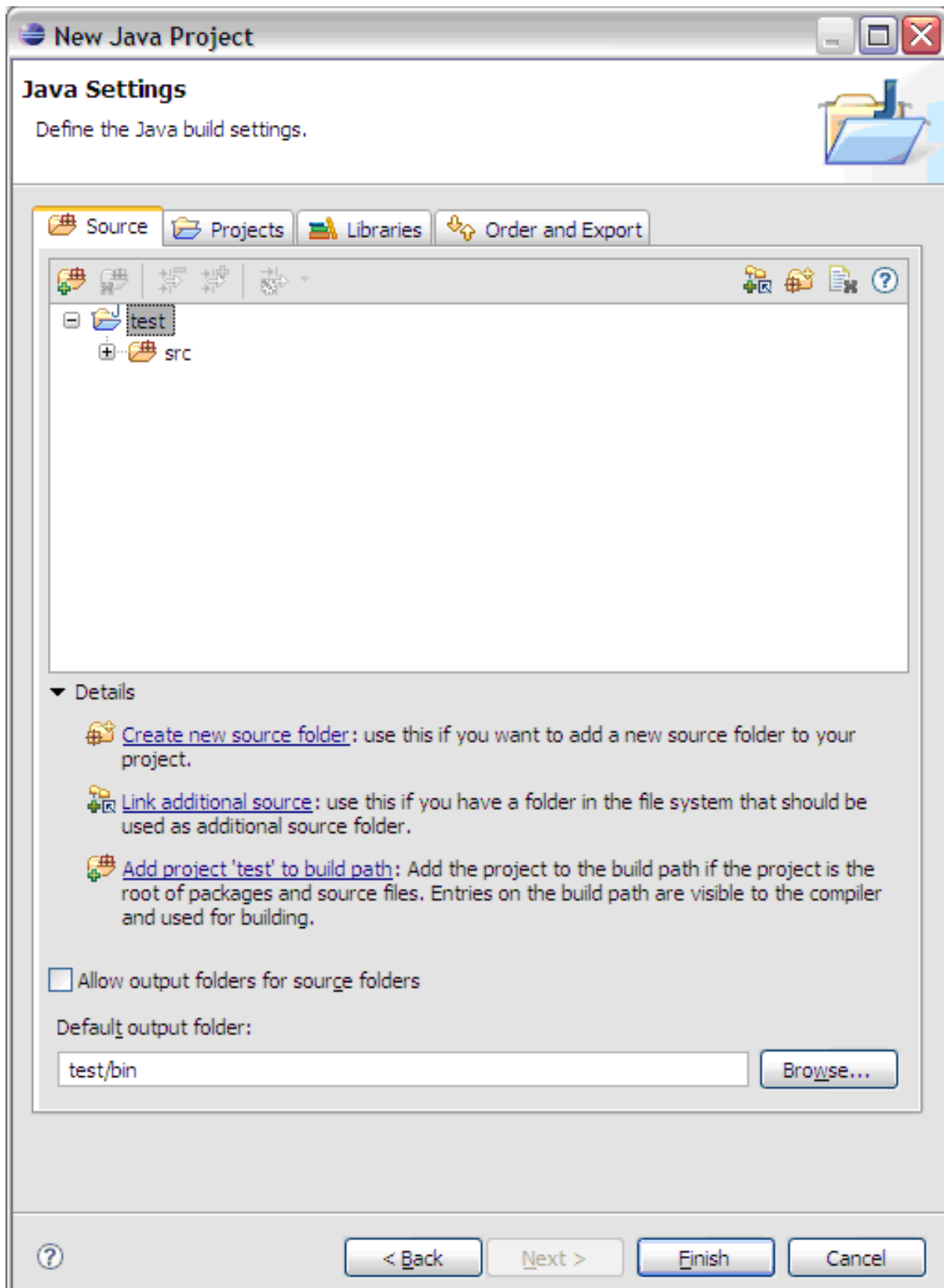
- Java > Java Project > Next



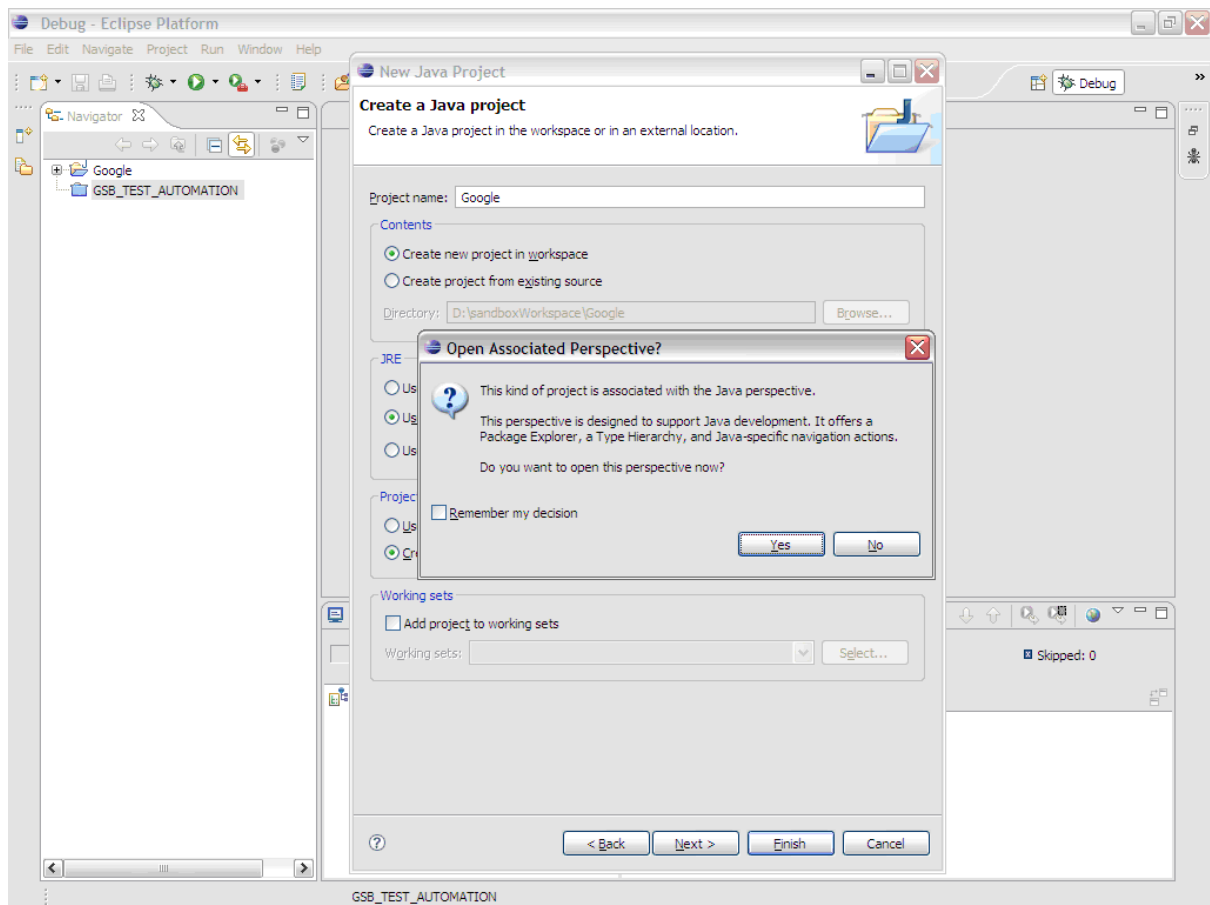
- Provide Name to your project, Select JDK in 'Use a project Specific JRE' option (JDK 1.5 selected in this example) > click Next



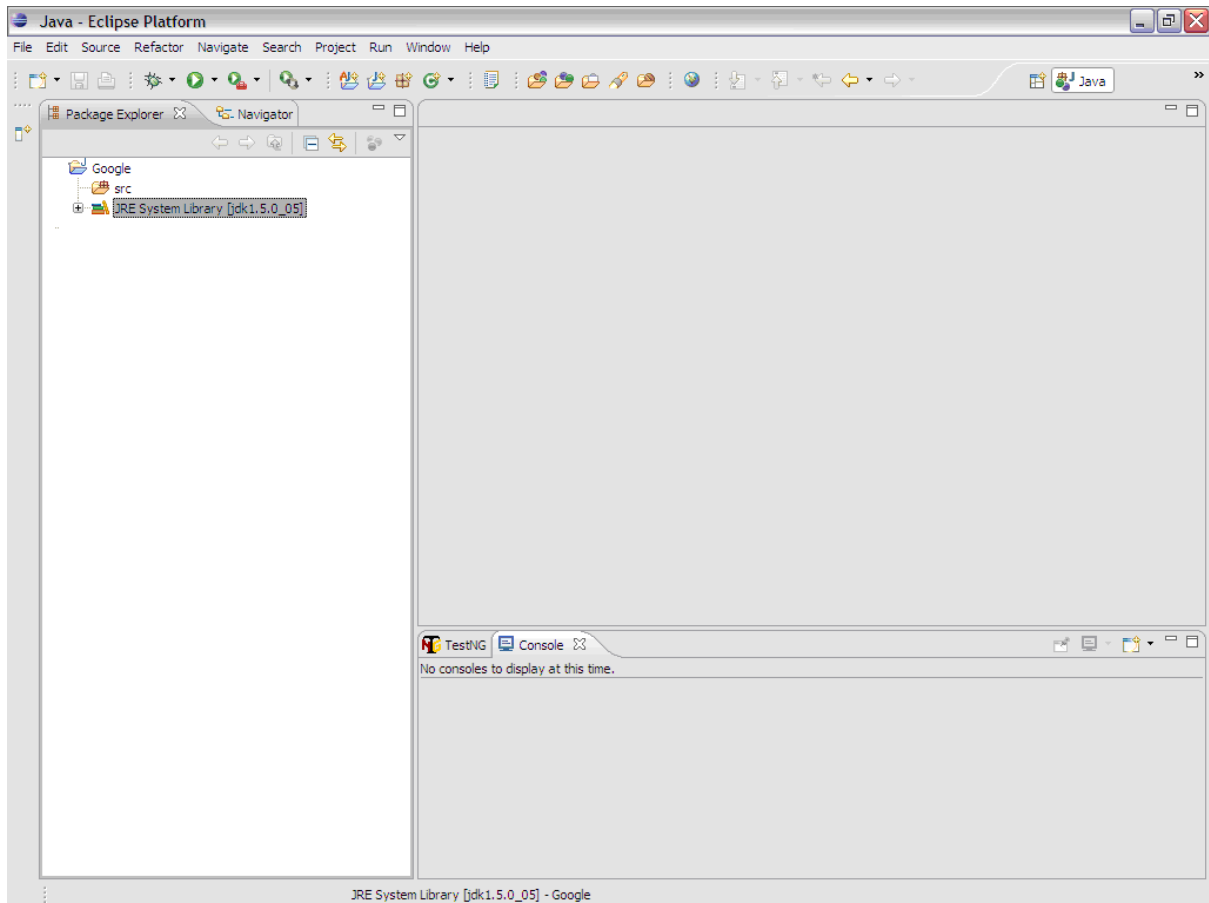
- Keep 'JAVA Settings' intact in next window. Project specific libraries can be added here. (This described in detail in later part of document.)



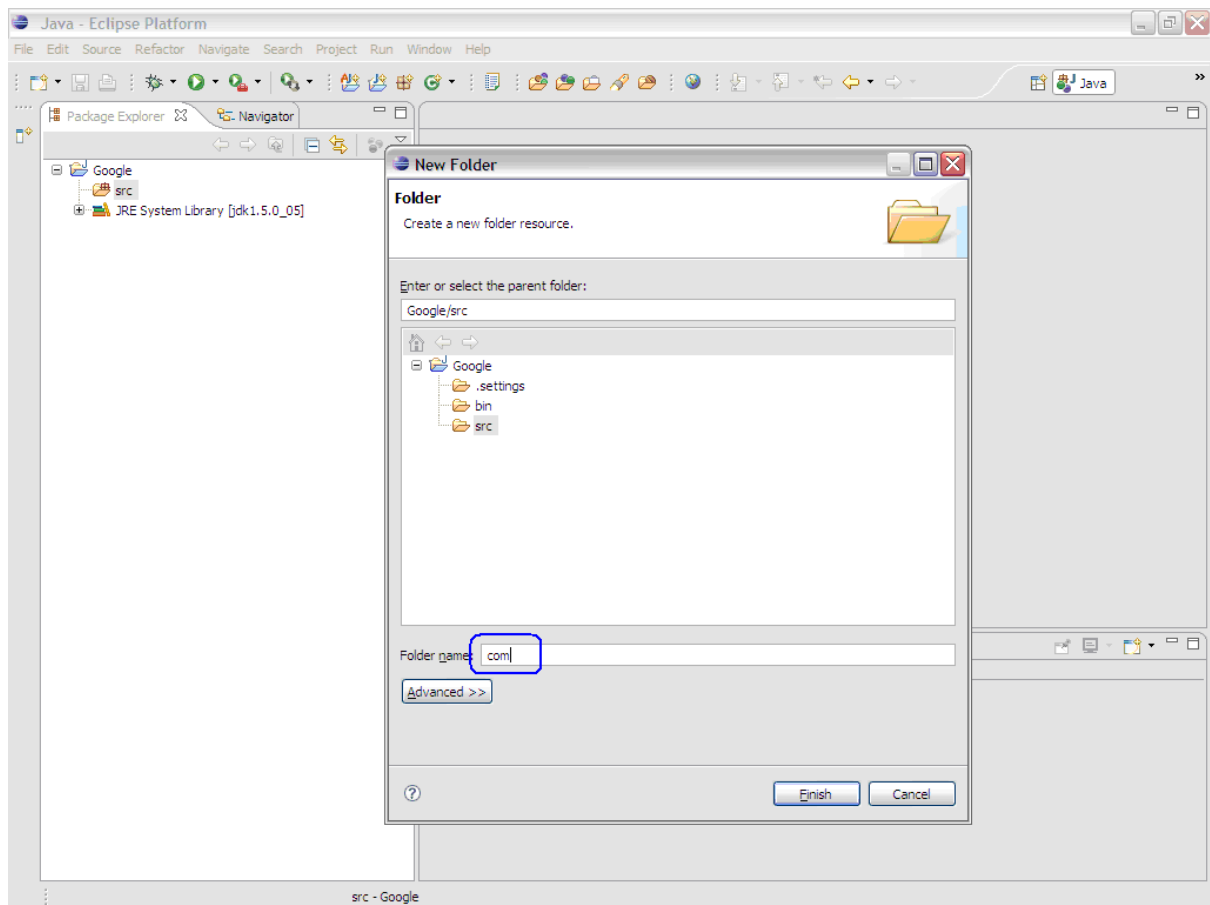
- Click Finish > Click on Yes in Open Associated Perspective pop up window.



This would create Project Google in Package Explorer/Navigator pane.

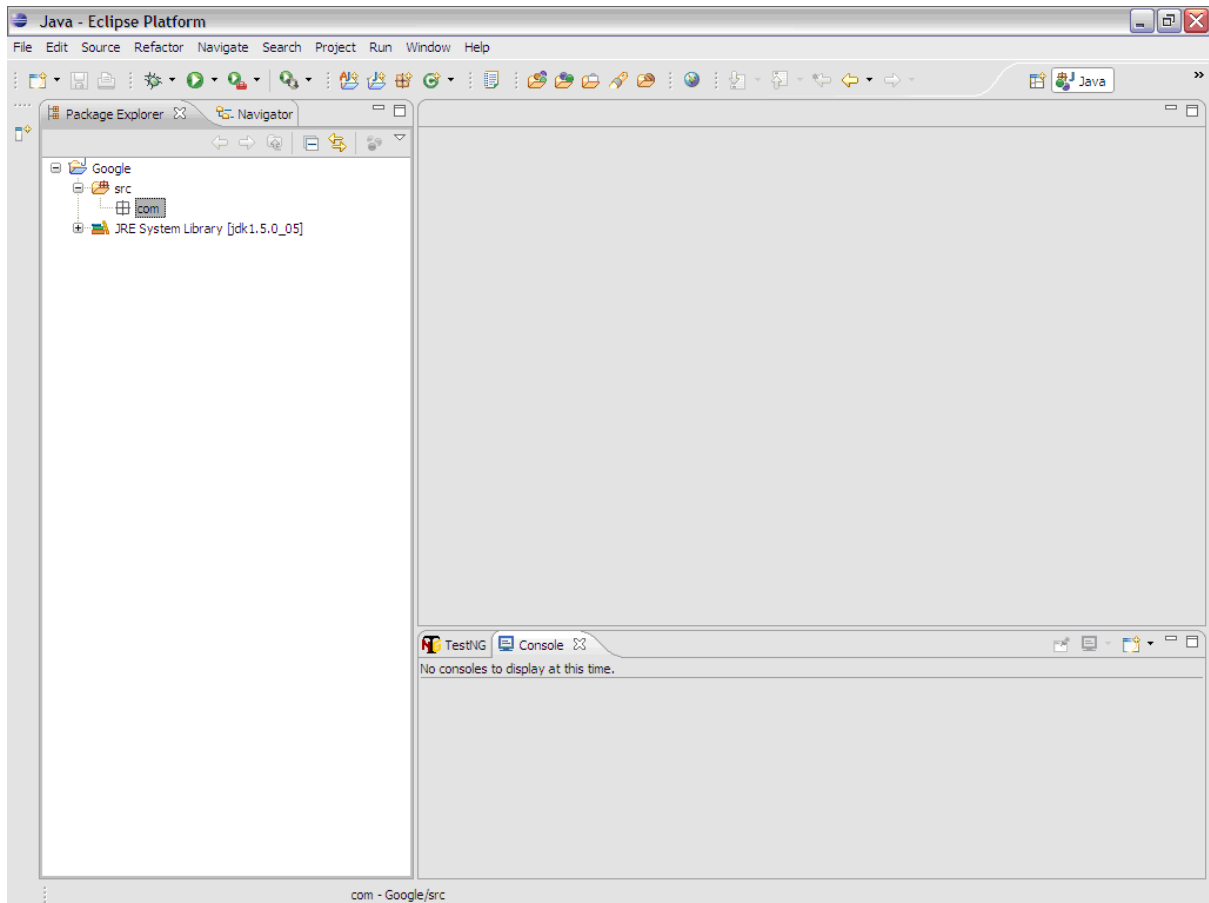


- Right click on src folder and click on New > Folder

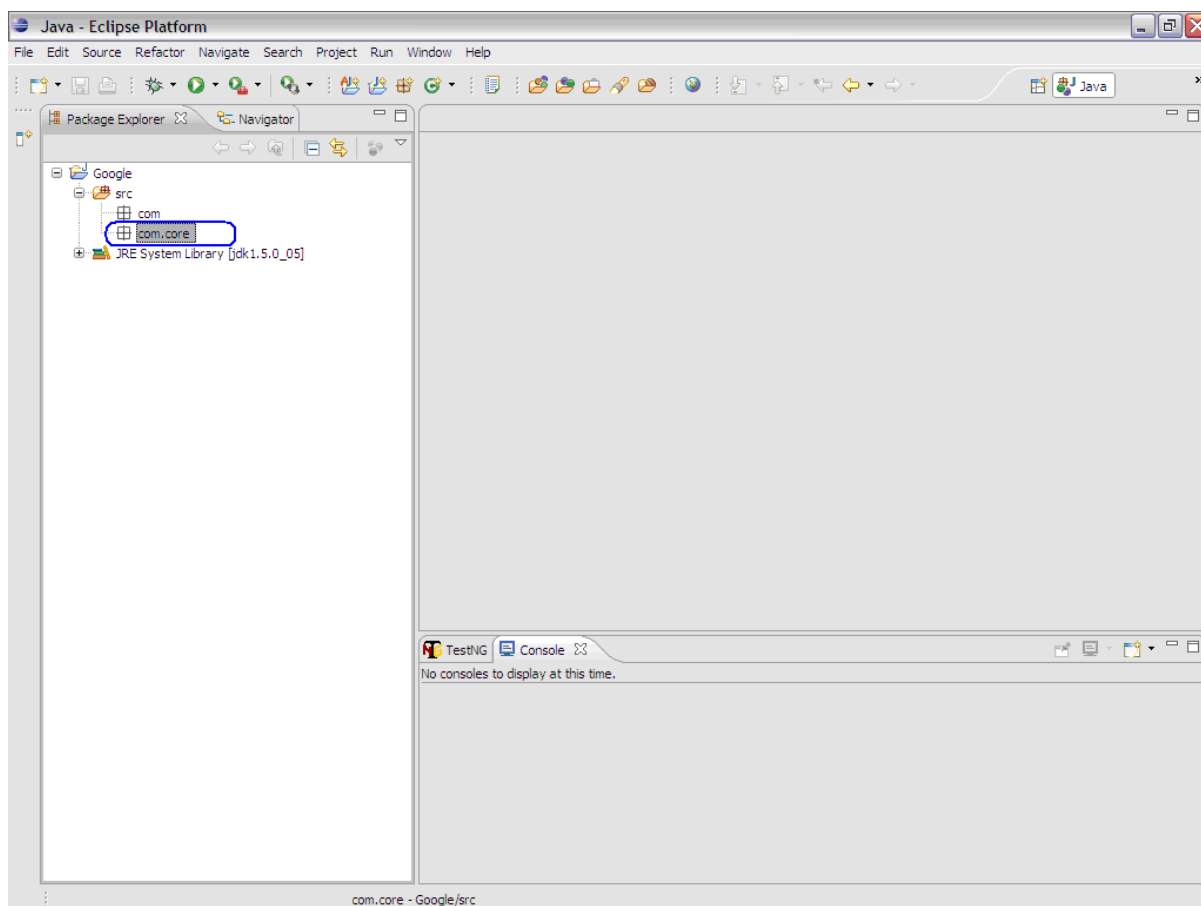


Name this folder as com and click on Finish button.

- This should get com package insider src folder.



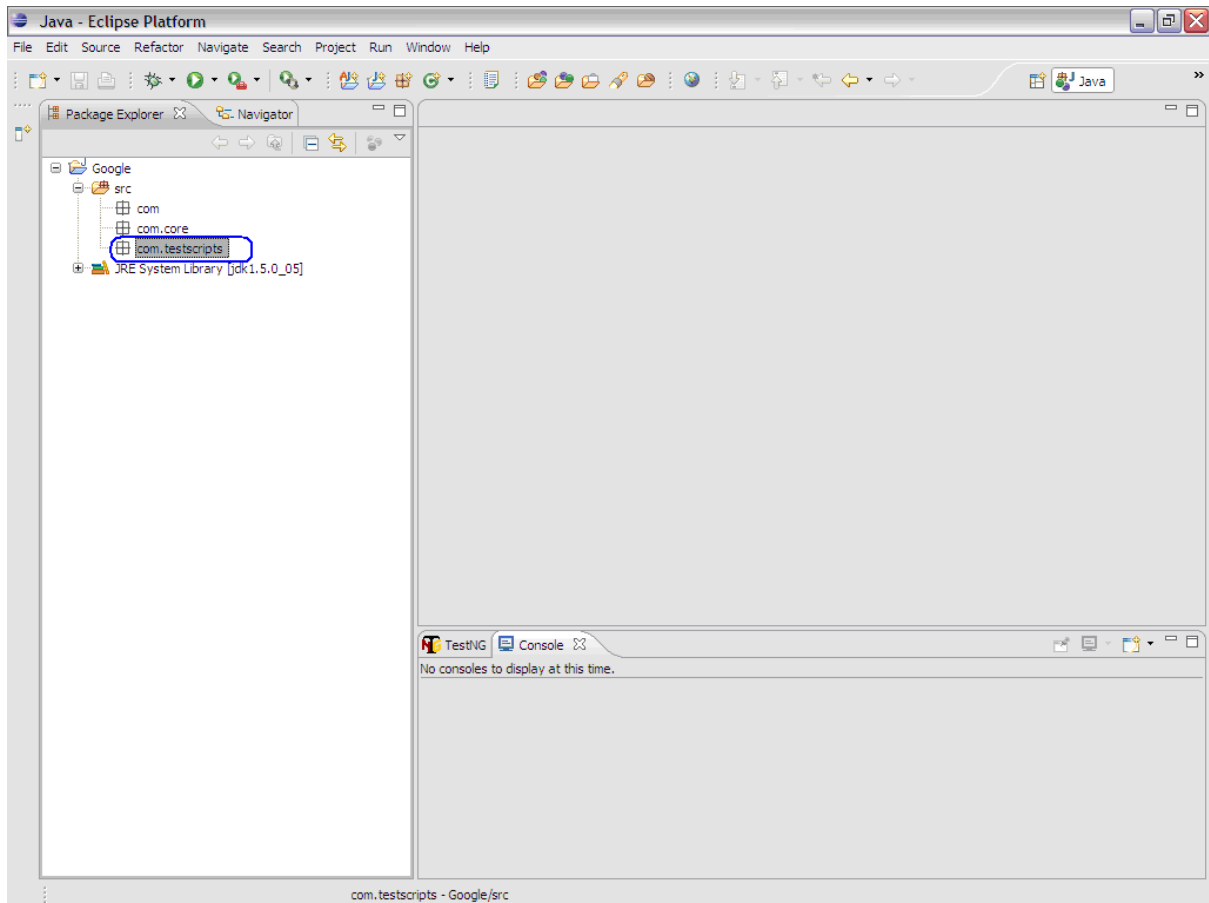
- Following the same steps create *core* folder inside *com*



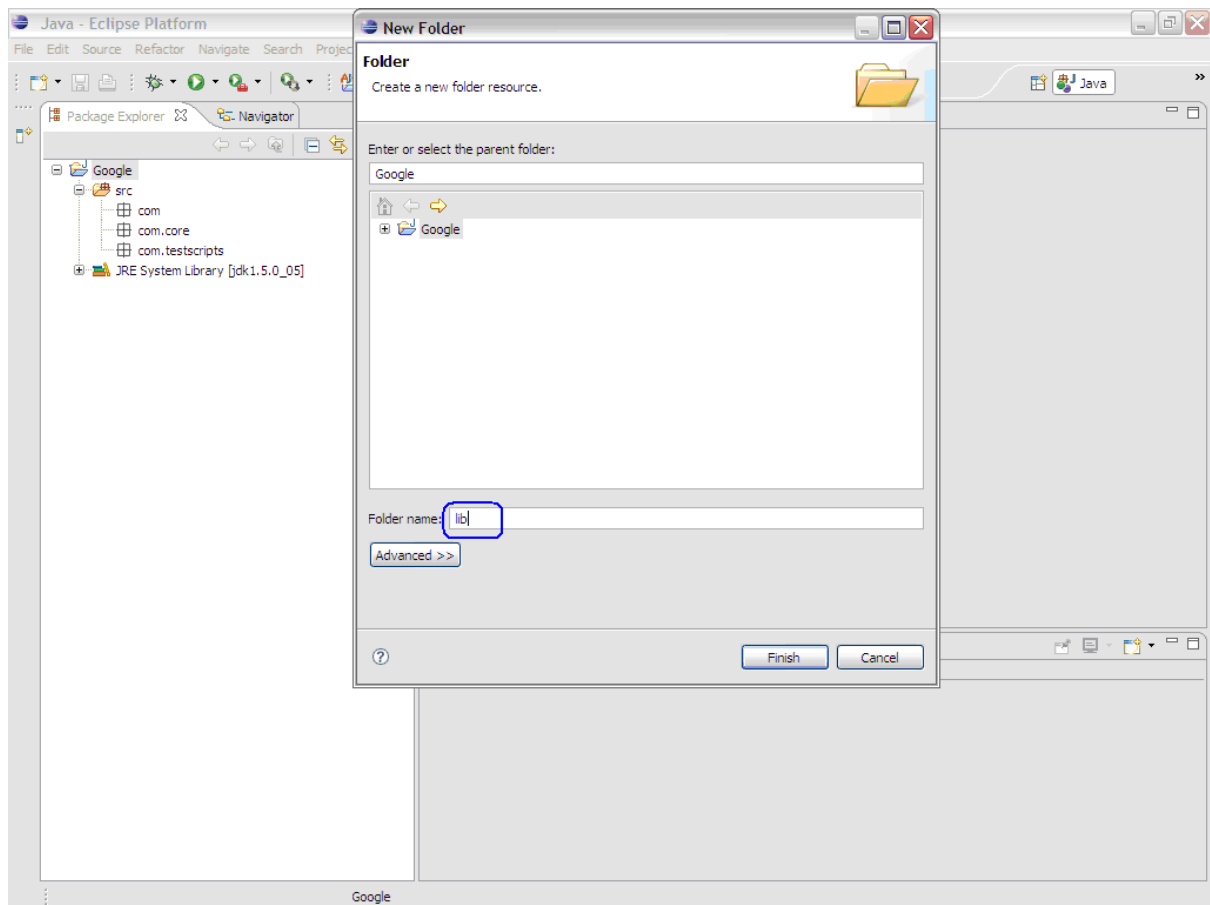
SelfTestCase class can be kept inside *core* package.

Create one more package inside *src* folder named *testscripts*. This is a place holder for test scripts.

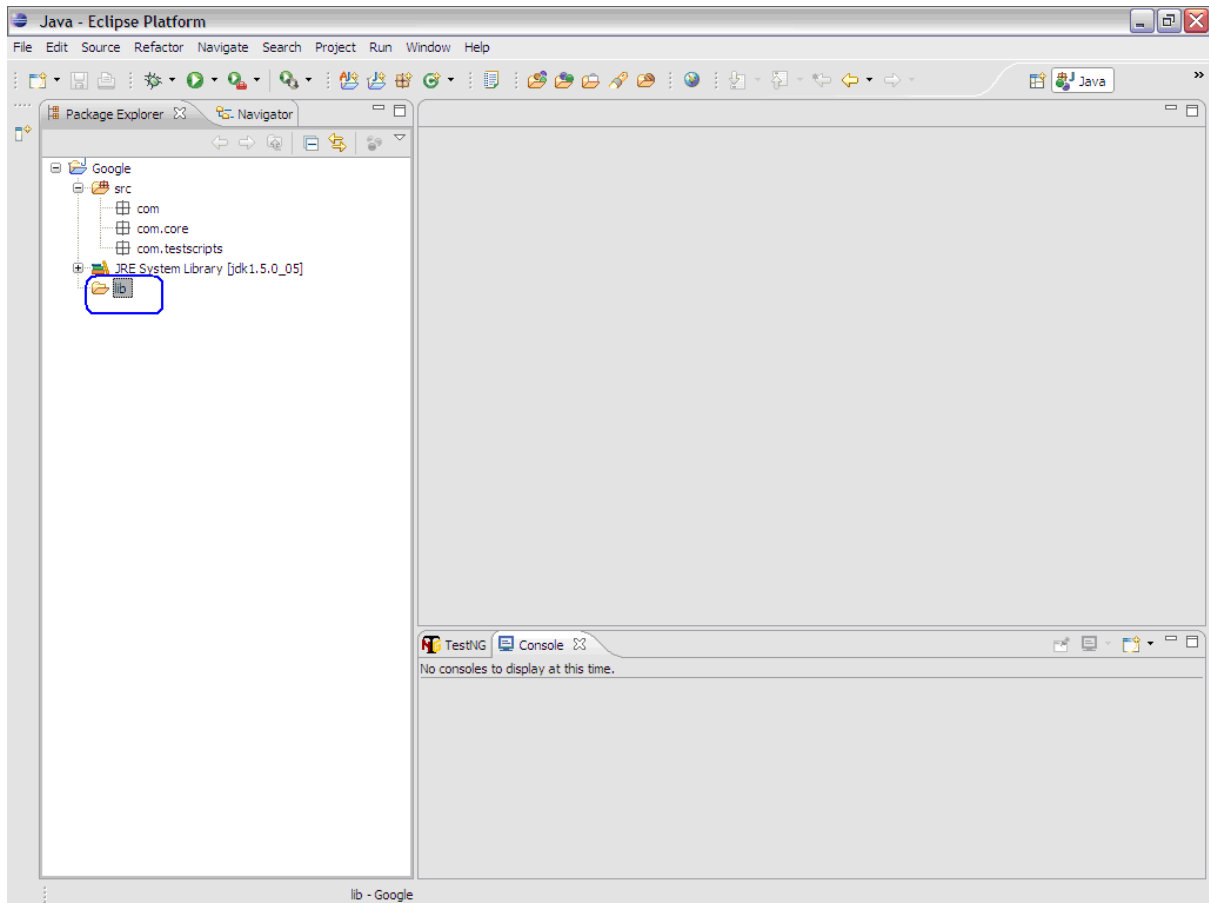
*Please notice this is about the organization of project and it entirely depends on individual's choice / organization's standards. Test scripts package can further be segregated depending upon the project requirements.*



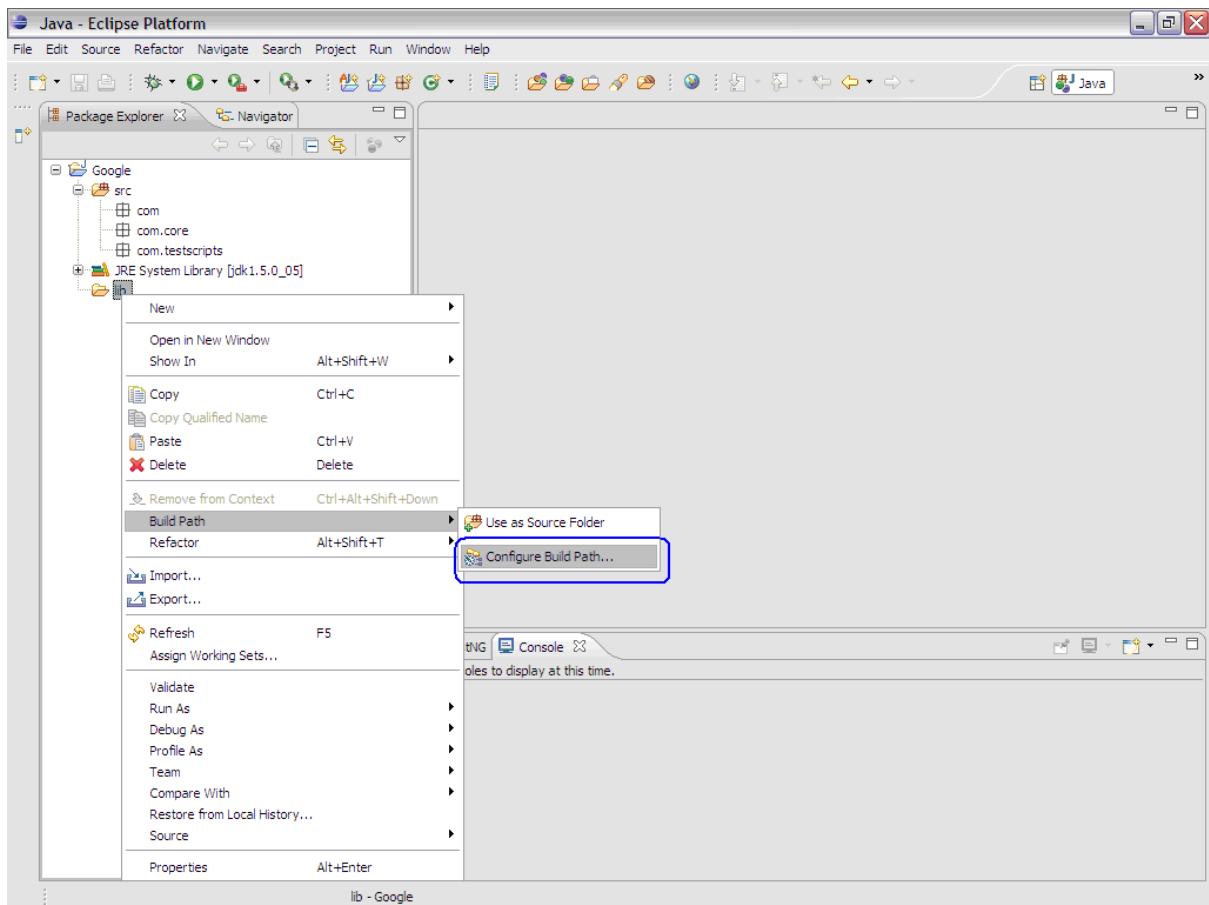
- Create a folder called lib inside project Google. Right click on Project name > New > Folder. This is a place holder for jar files to project (i.e. Selenium client driver, selenium server etc)



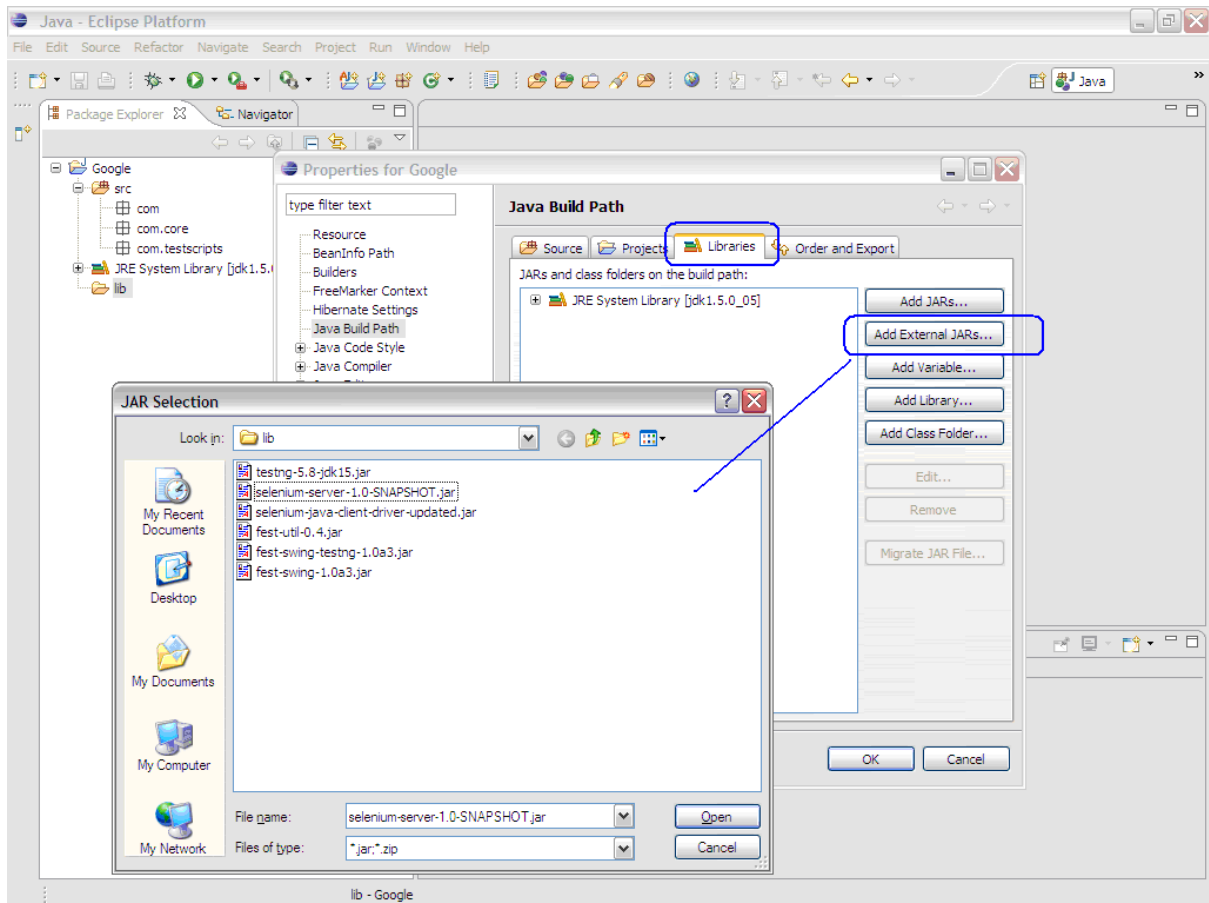
This would create lib folder in Project directory.



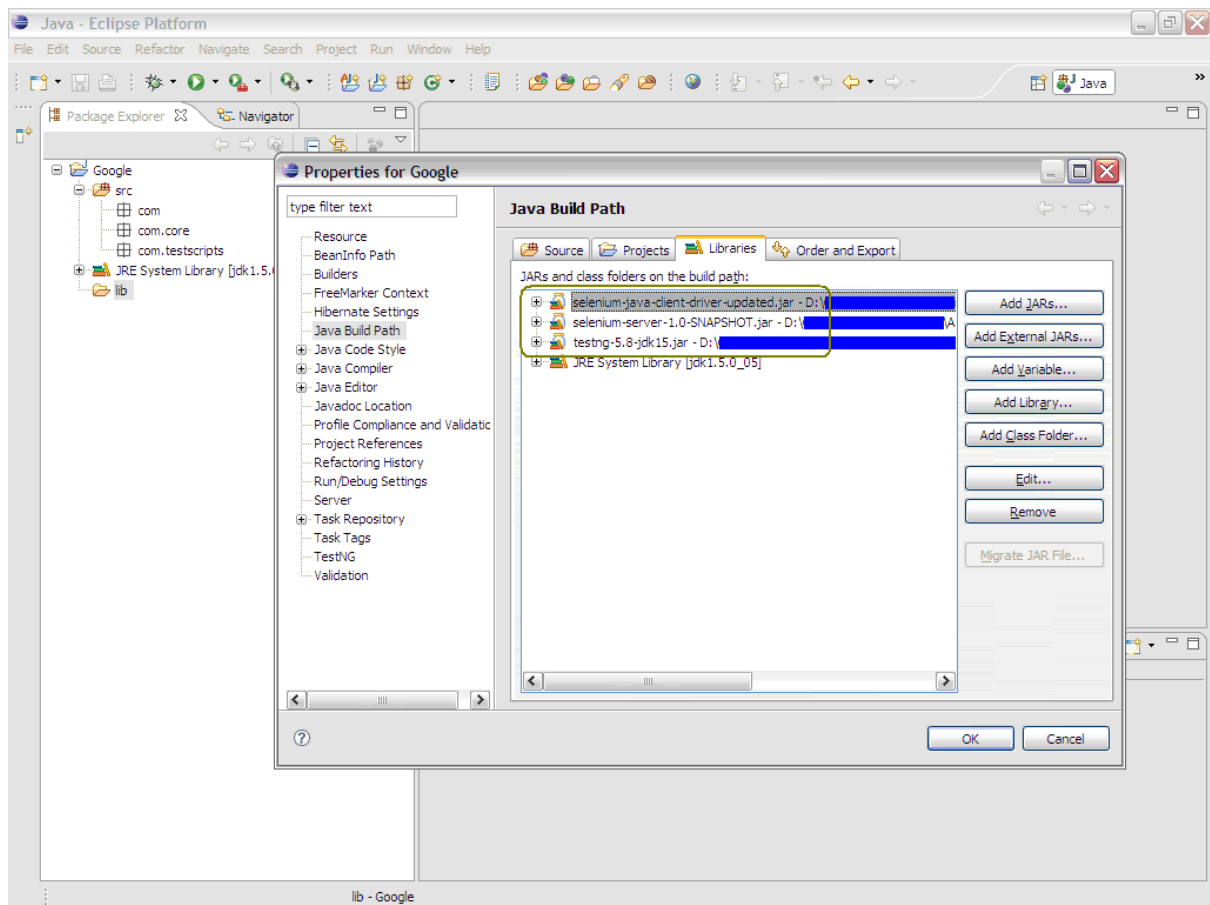
- Right click on *lib* folder > Build Path > Configure build Path



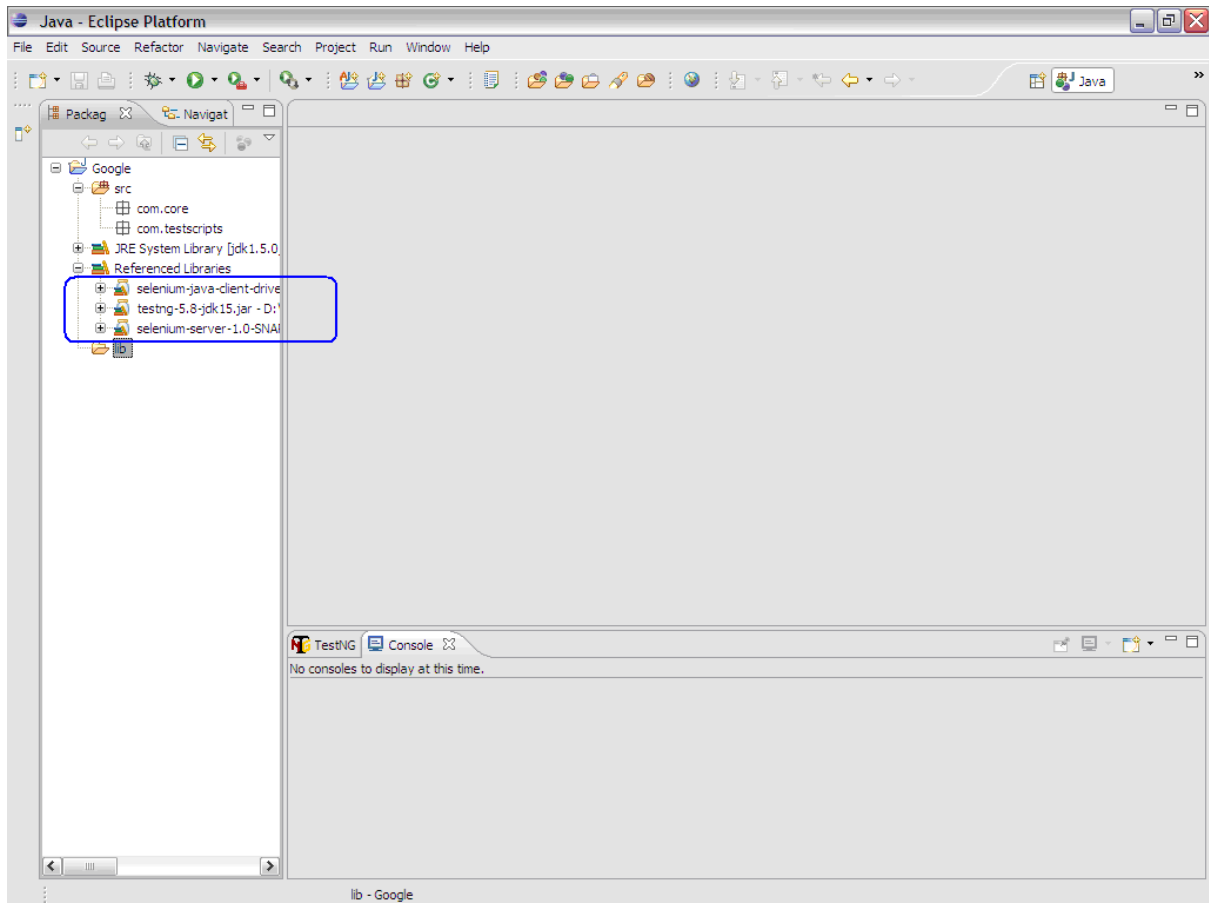
- Under Library tab click on Add External Jars to navigate to directory where jar files are saved. Select the jar files which are to be added and click on Open button.



After having added jar files click on OK button.



Added libraries would appear in Package Explorer as following:

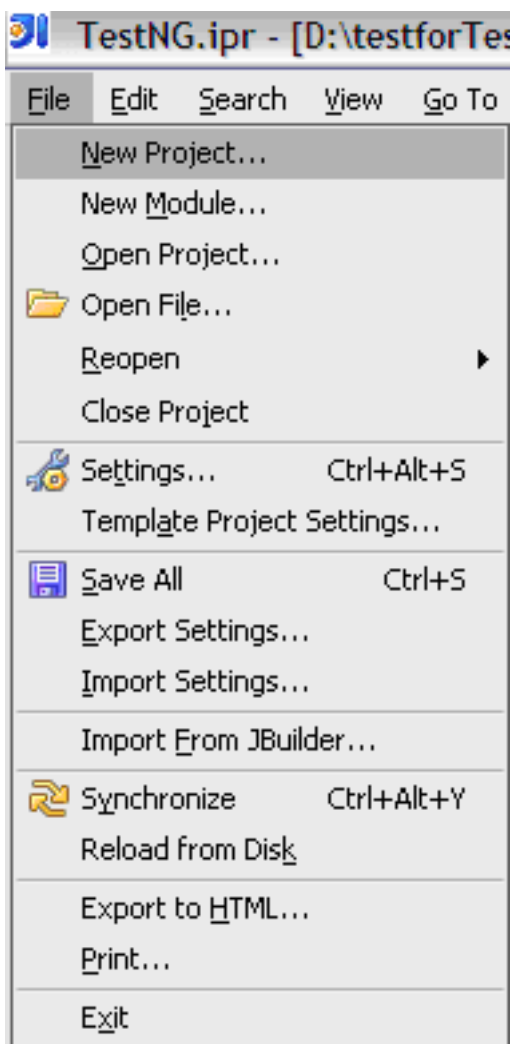


## 11.2 Configuring Selenium-RC With IntelliJ

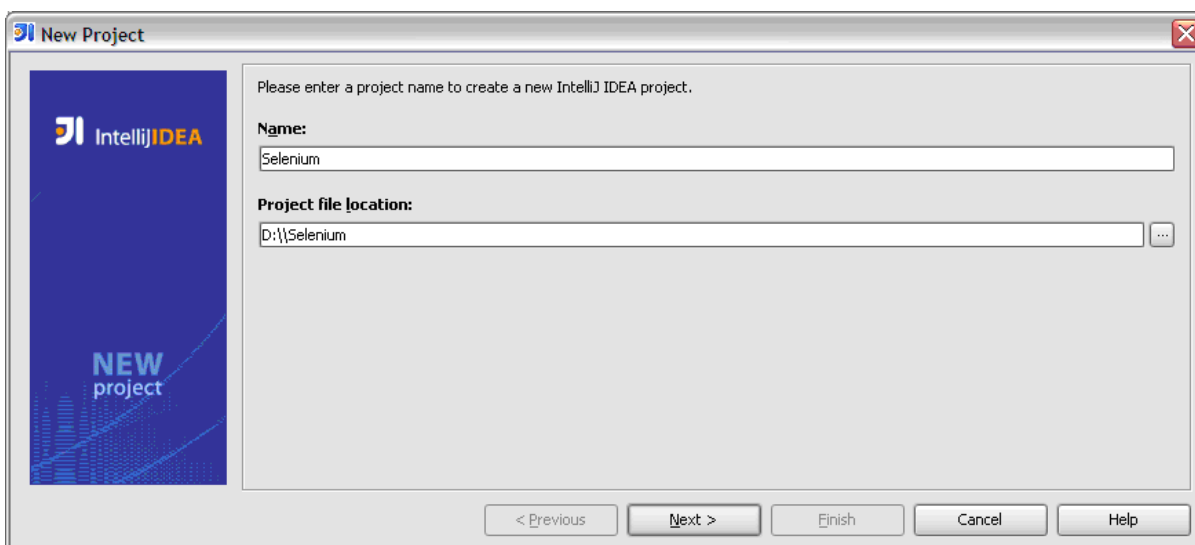
**IntelliJ IDEA** is a commercial Java IDE by the company JetBrains. IntelliJ provides a set of integrated refactoring tools that allow programmers to quickly redesign their code. IntelliJ IDEA provides close integration with popular open source development tools such as CVS, Subversion, Apache Ant and JUnit.

Following lines describes configuration of Selenium-RC with IntelliJ 6.0 It should not be very different for higher version of IntelliJ.

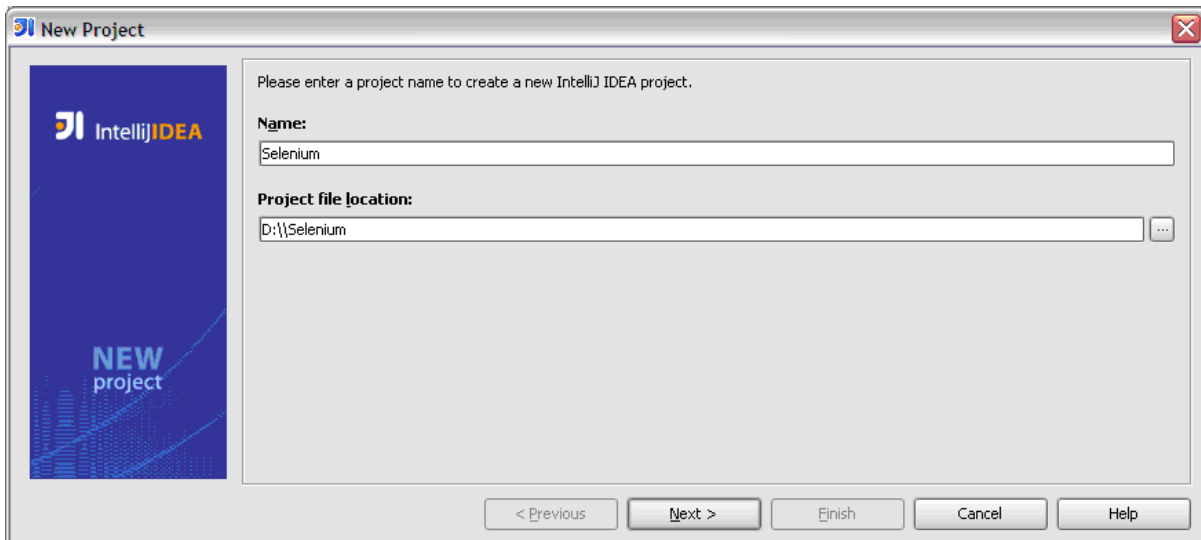
- Open a New Project in IntelliJ IDEA.



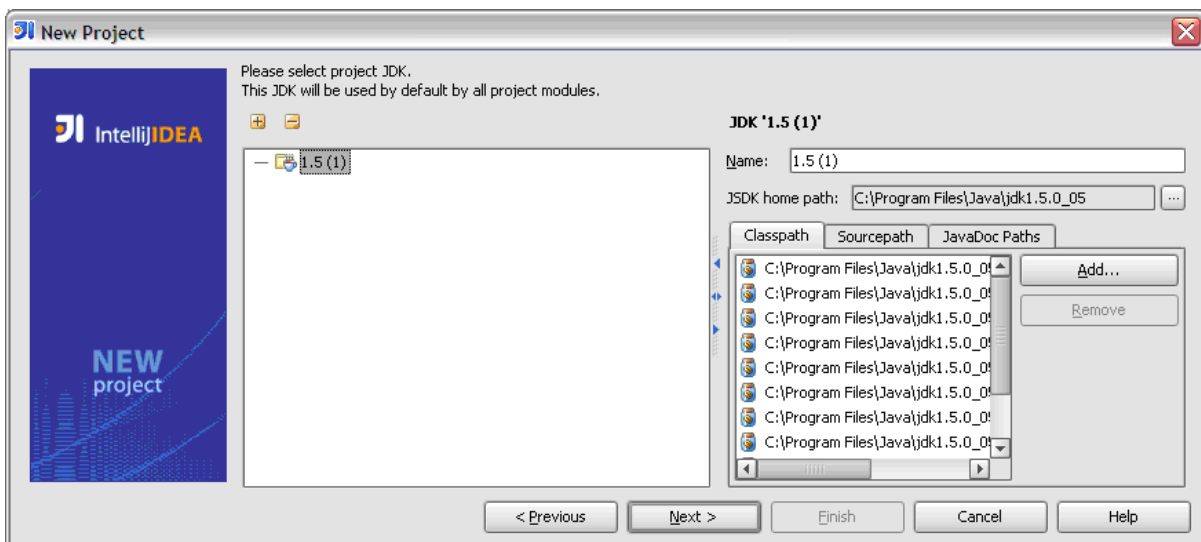
- Provide name and location to Project.



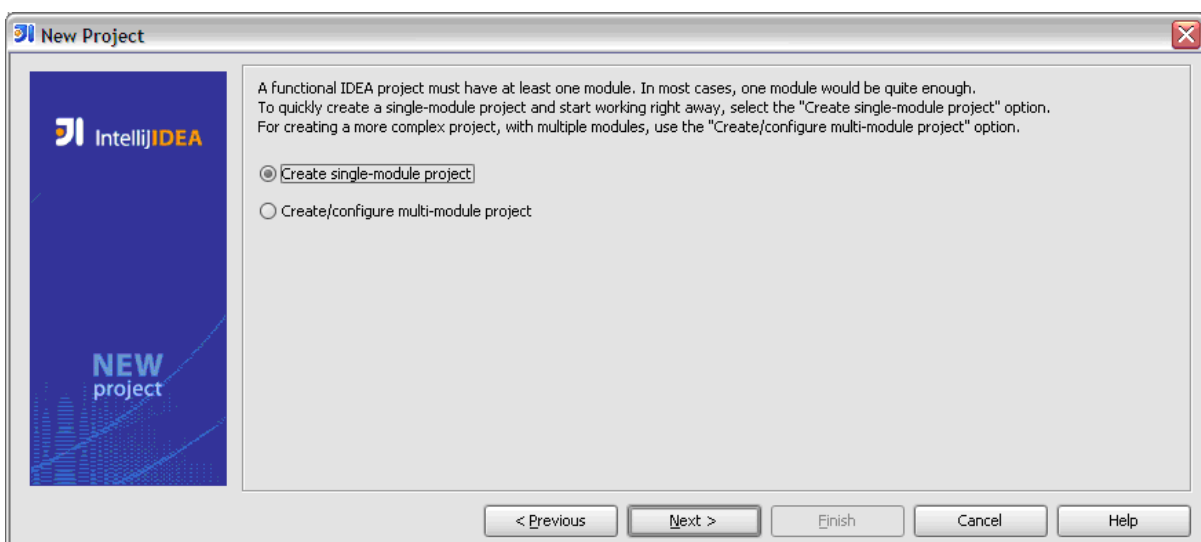
- Click Next and provide compiler output path.



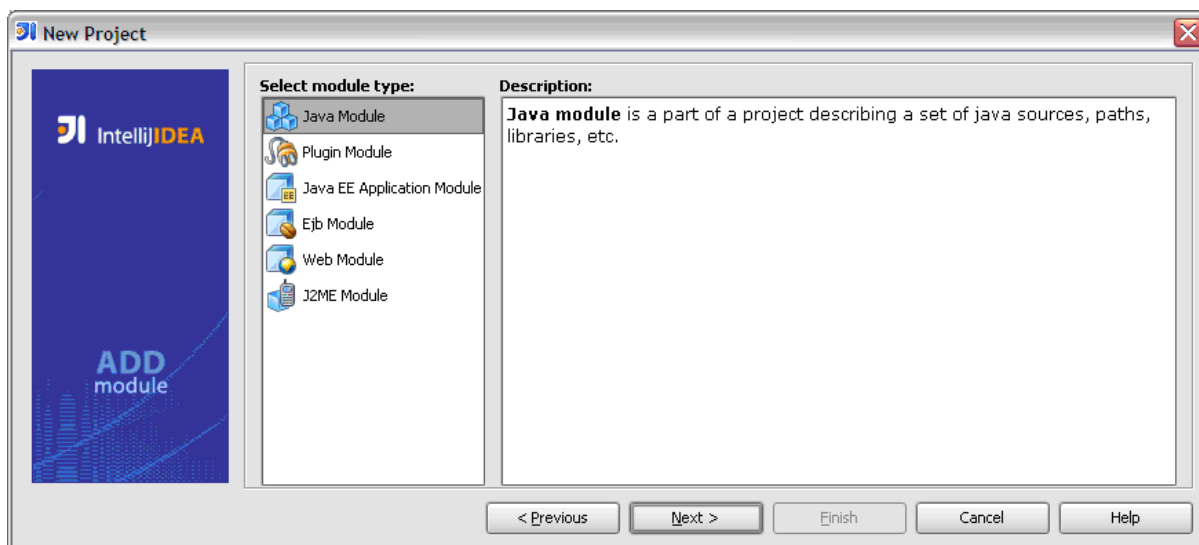
- Click Next and select the JDK to be used.



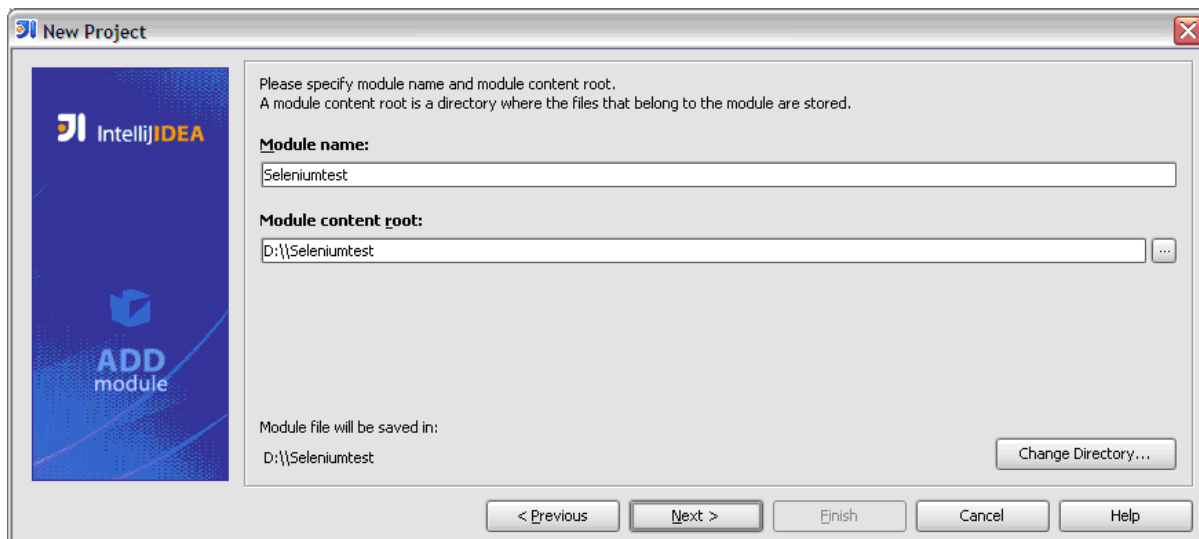
- Click Next and select Single Module Project.



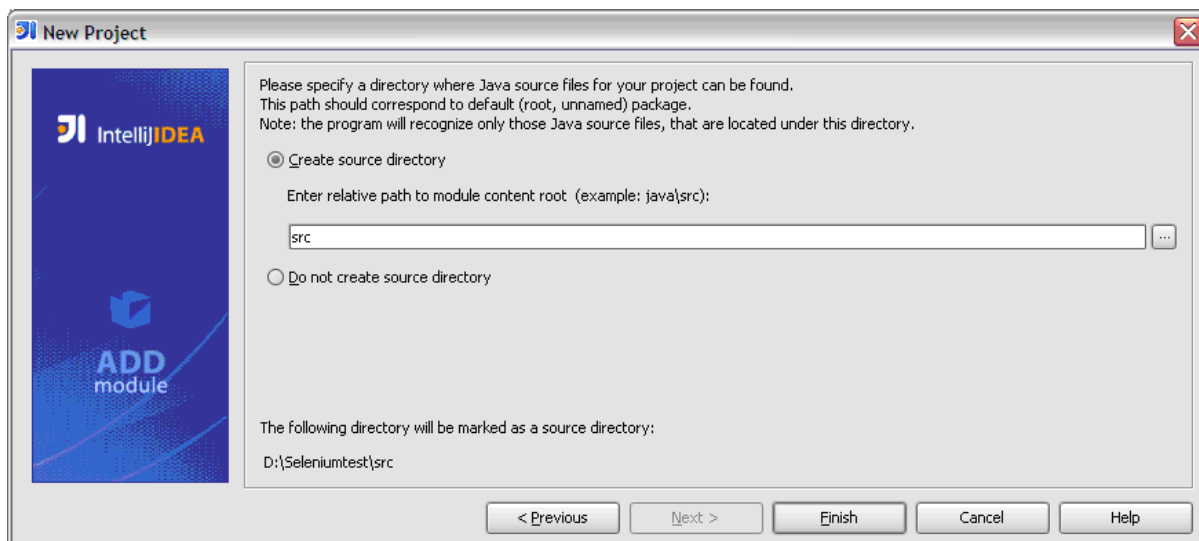
- Click Next and select Java module.



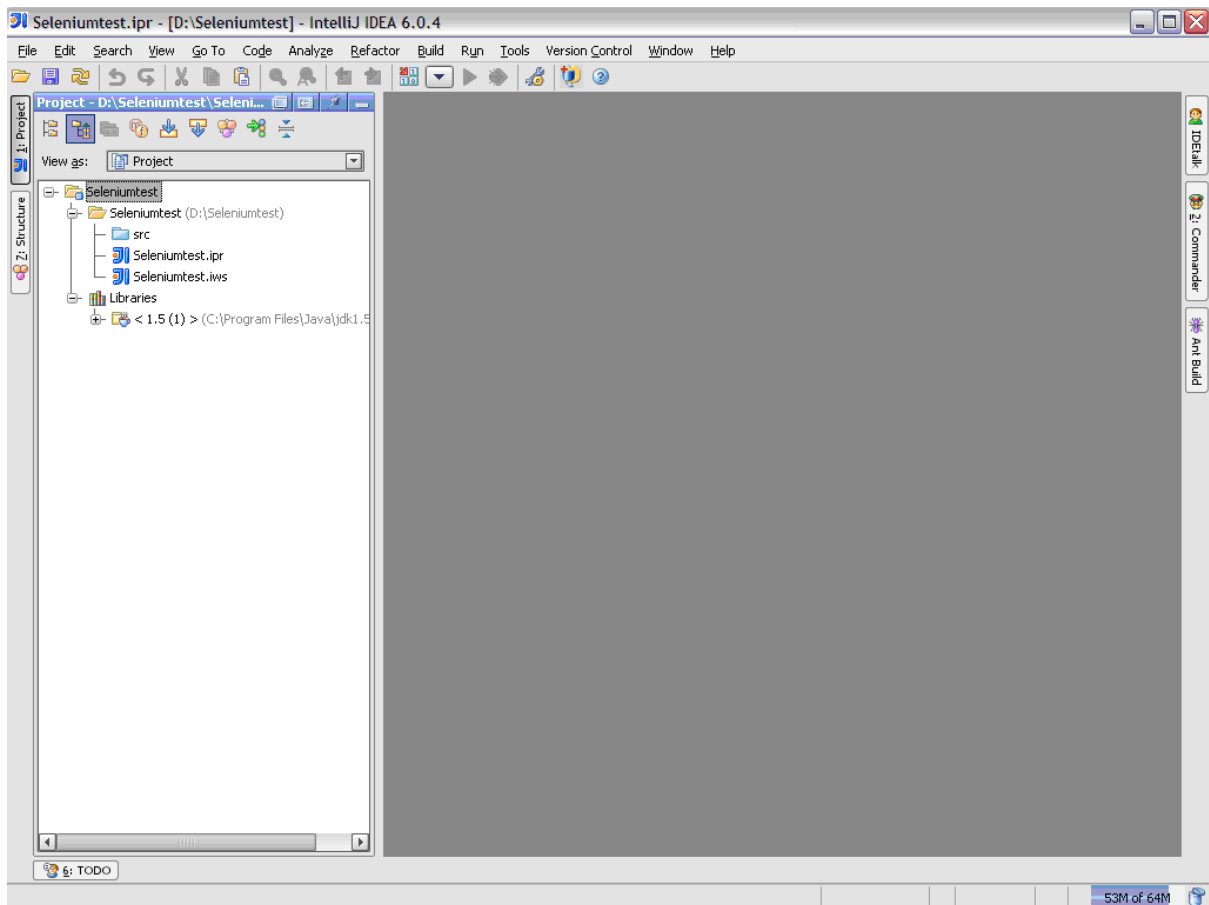
- Click Next and provide Module name and Module content root.



- Click Next and select Source directory.

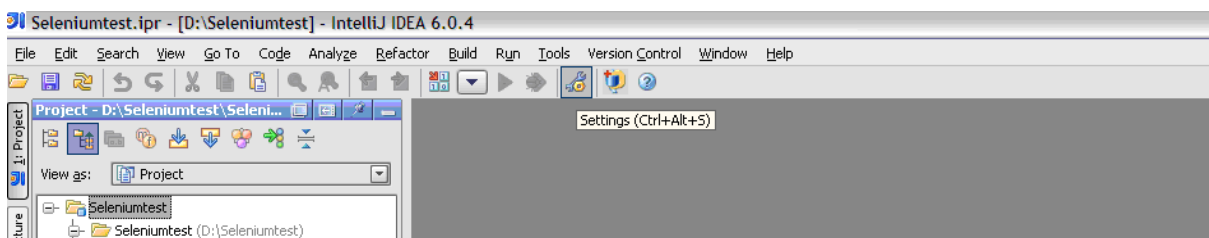


- At last click Finish. This will launch the Project Pan.

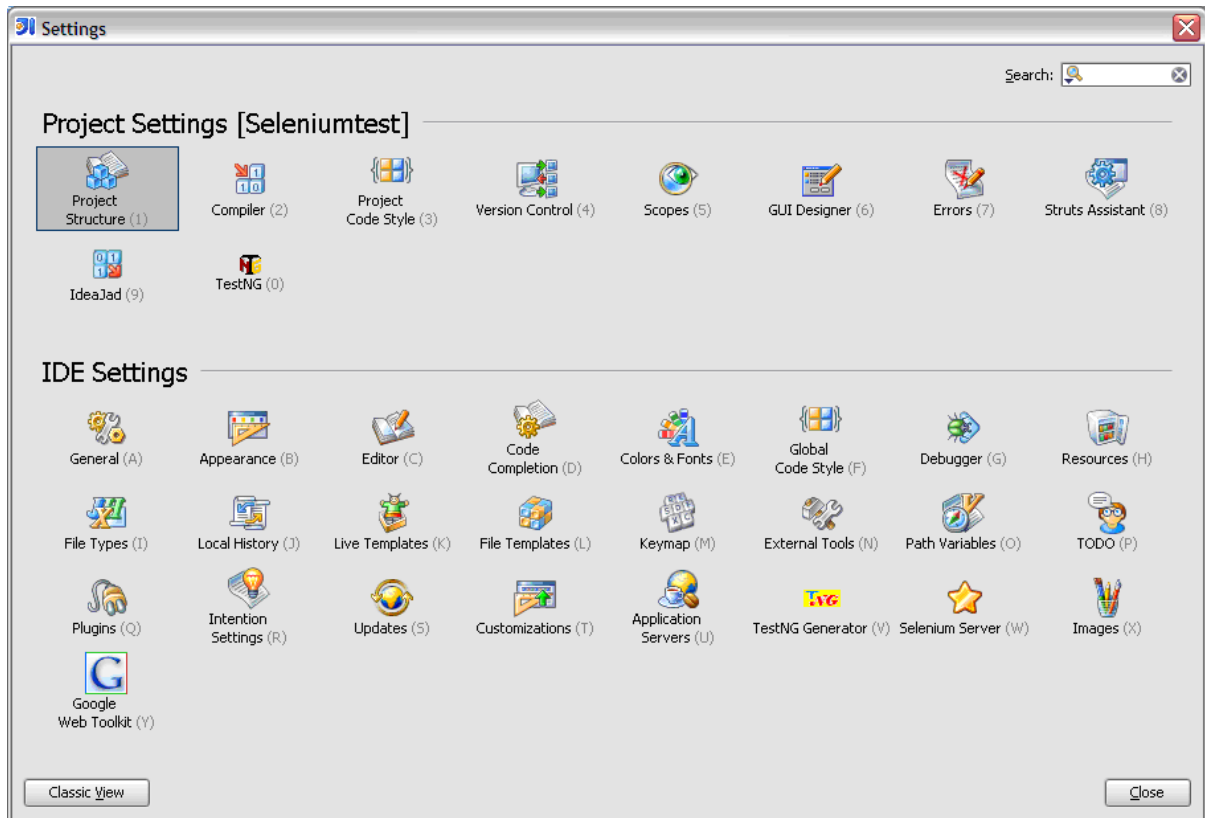


### Adding Libraries to Project:

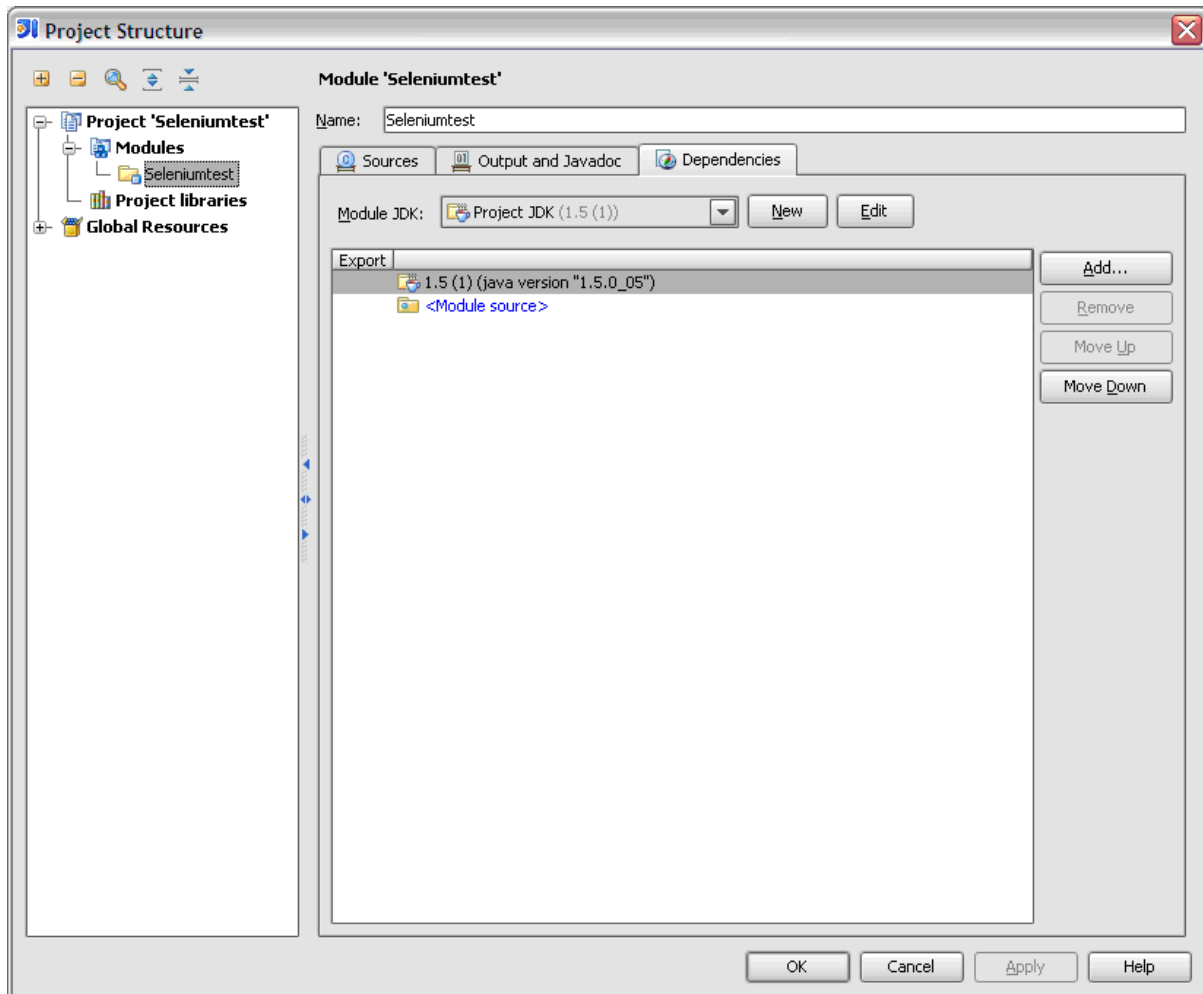
- Click on *Settings* button in the Project Tool bar.



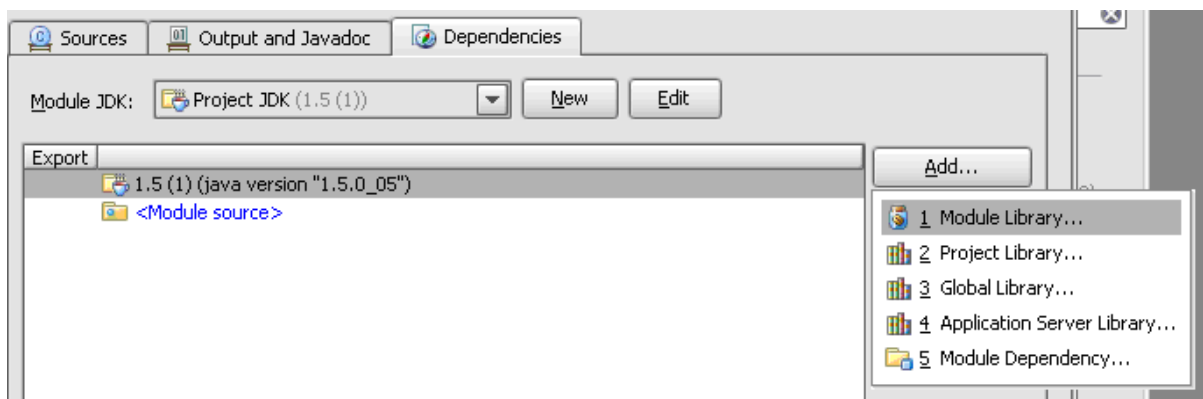
- Click on *Project Structure* in Settings pan.



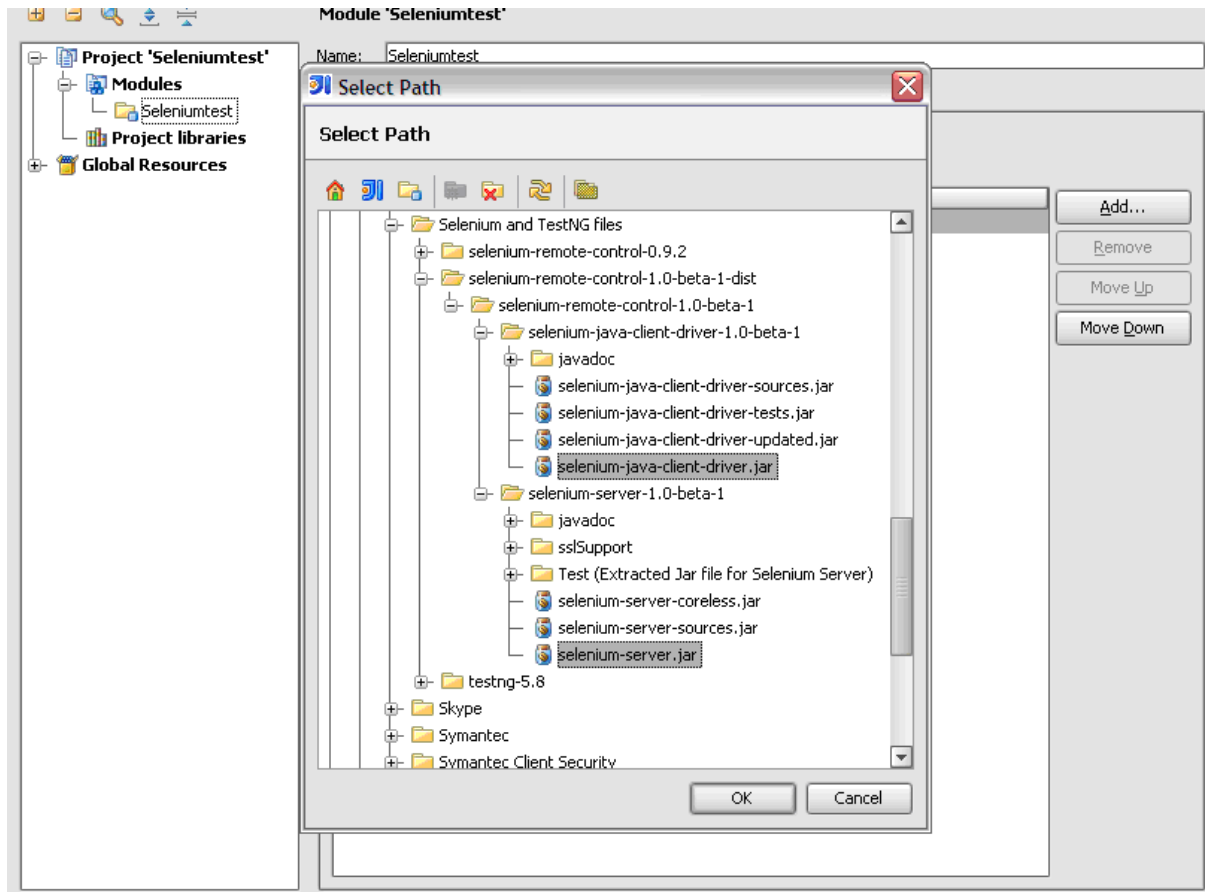
- Select *Module* in Project Structure and browse to *Dependencies* tab.



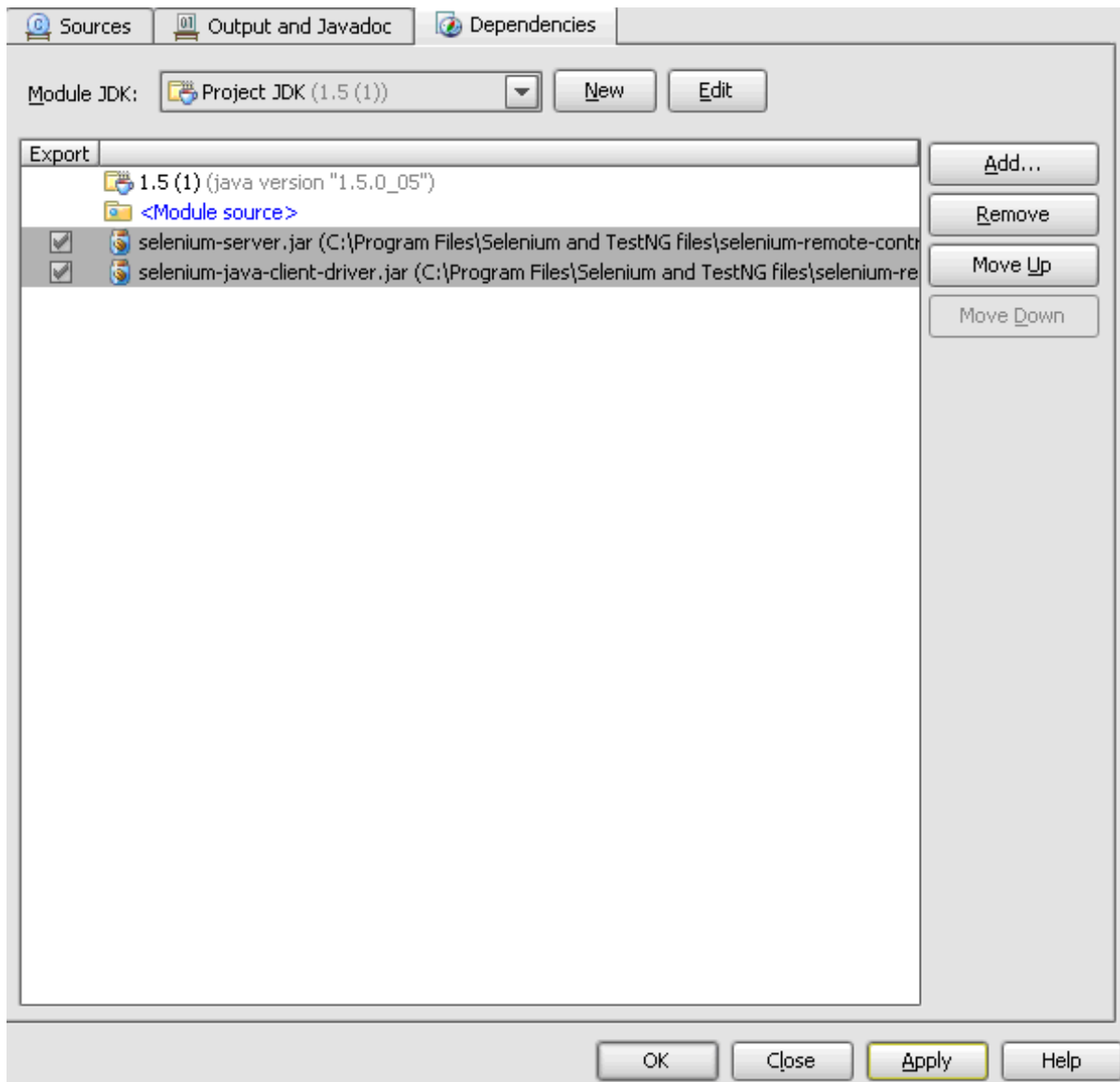
- Click on Add button followed by click on Module Library.



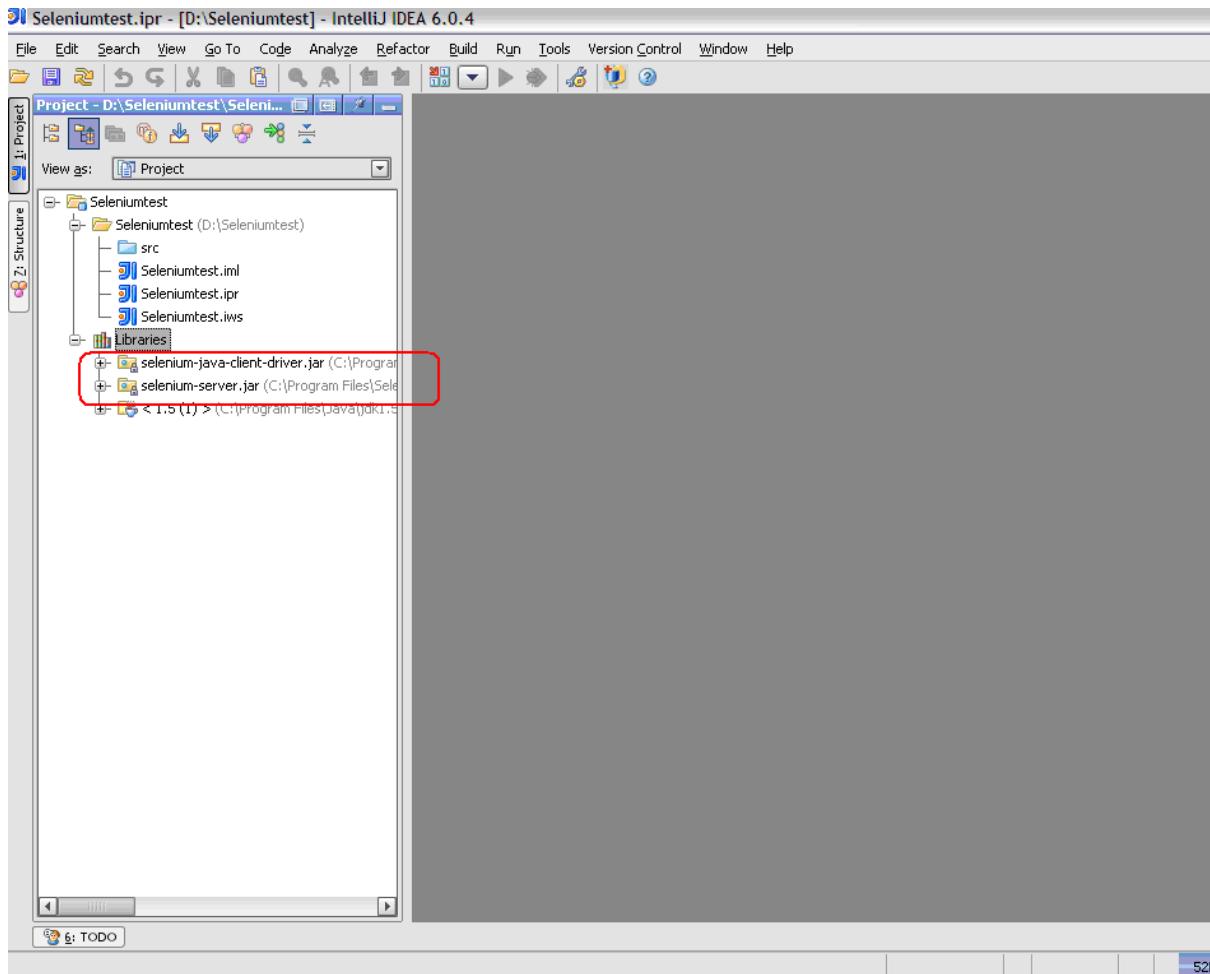
- Browse to the Selenium directory and select selenium-java-client-driver.jar and selenium-server.jar. (Multiple Jars can be selected by holding down the control key.)



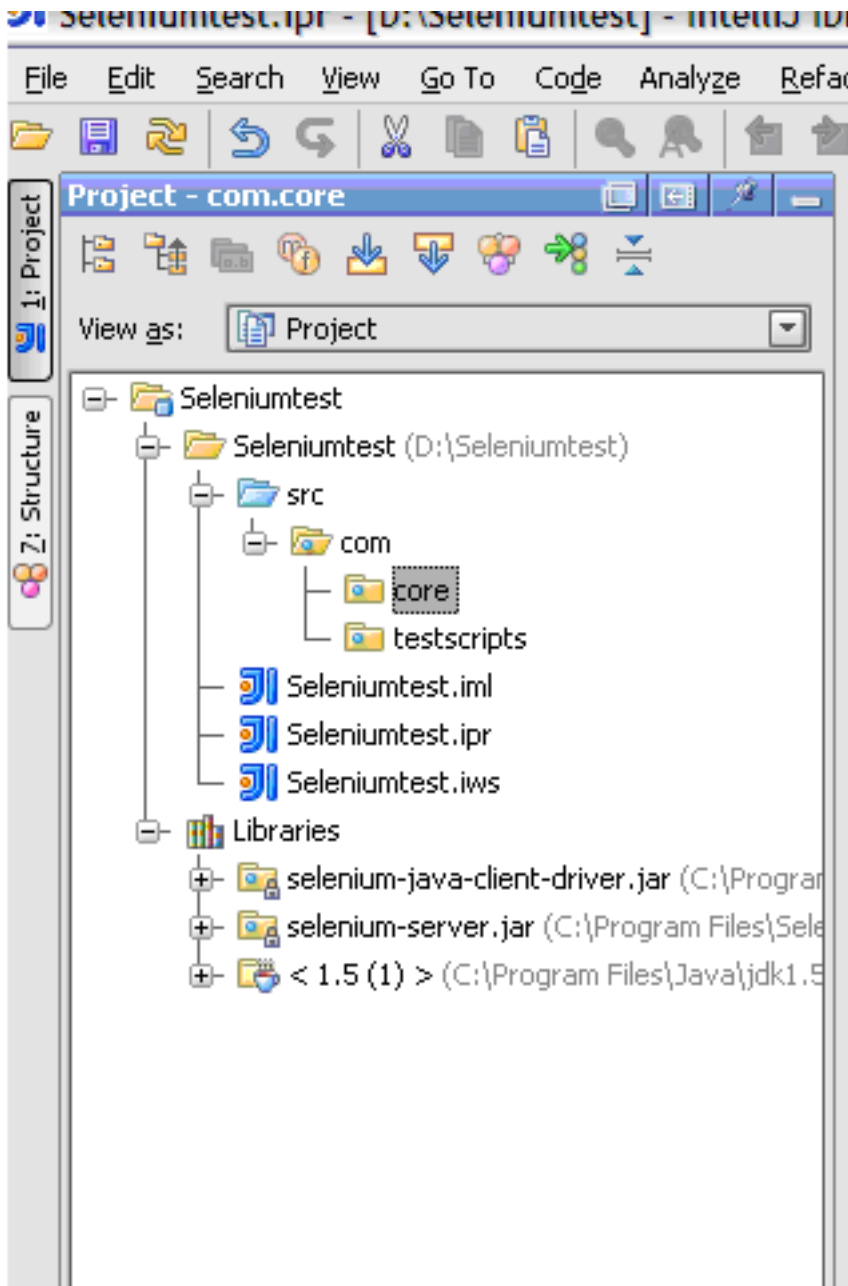
- Select both jar files in project pan and click on *Apply* button.



- Now click ok on Project Structure followed by click on Close on Project Settings pan. Added jars would appear in project Library as following.

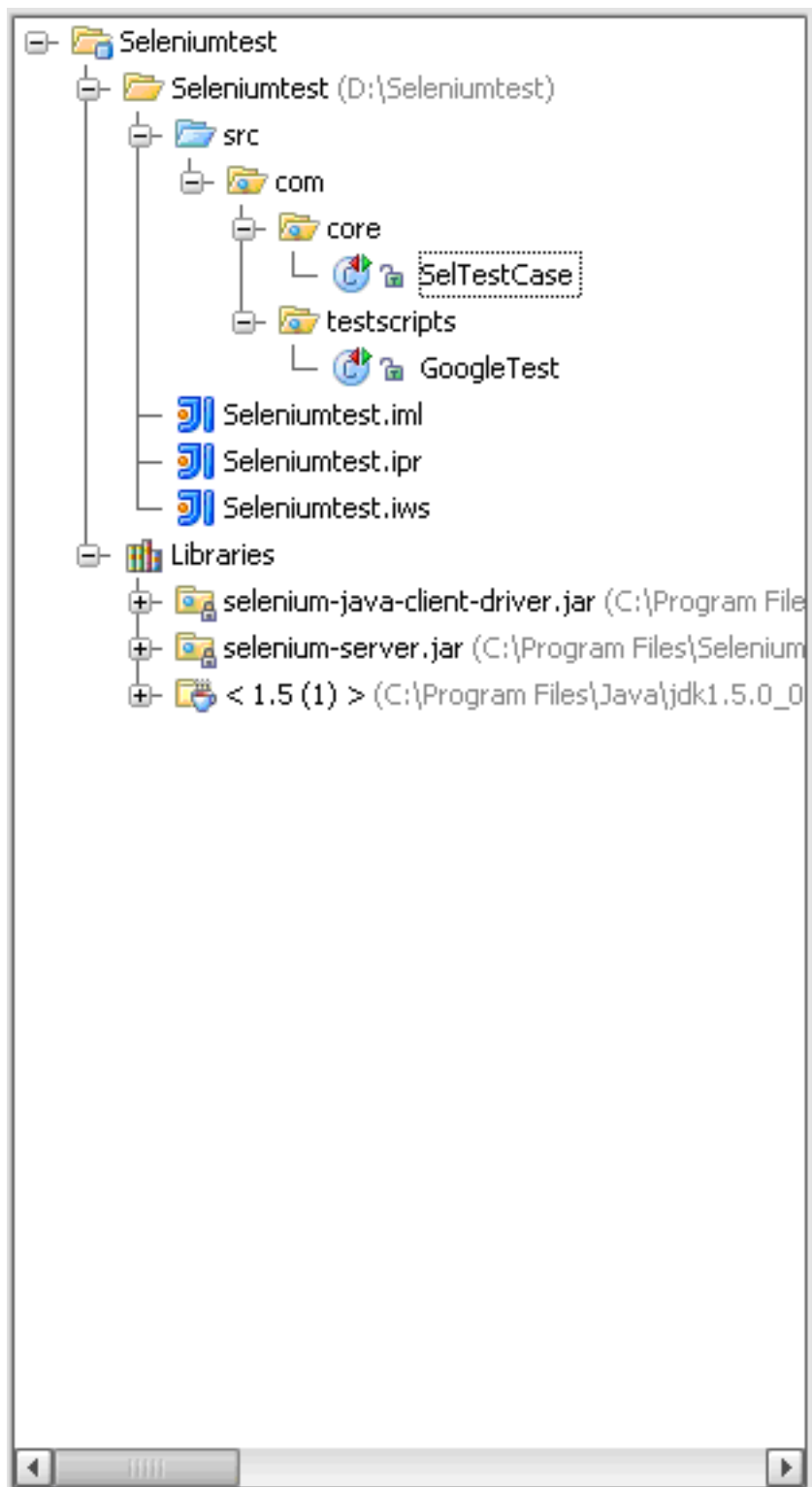


- Create the directory structure in src folder as following.



**Note:** This is not hard and fast convention and might vary from project to project.

- Herein *core* contains the `SelTestCase` class which is used to create Selenium object and fire up the browser. *testscripts* package contains the test classes which extend the `SelTestCase` class. Hence extended structure would look as following.



# PYTHON CLIENT DRIVER CONFIGURATION

- Download Selenium-RC from the SeleniumHQ [downloads page](#)
- Extract the file *selenium.py*
- Either write your Selenium test in Python or export a script from Selenium-IDE to a python file.
- Add to your test's path the file *selenium.py*
- Run Selenium server from the console
- Execute your test from a console or your Python IDE

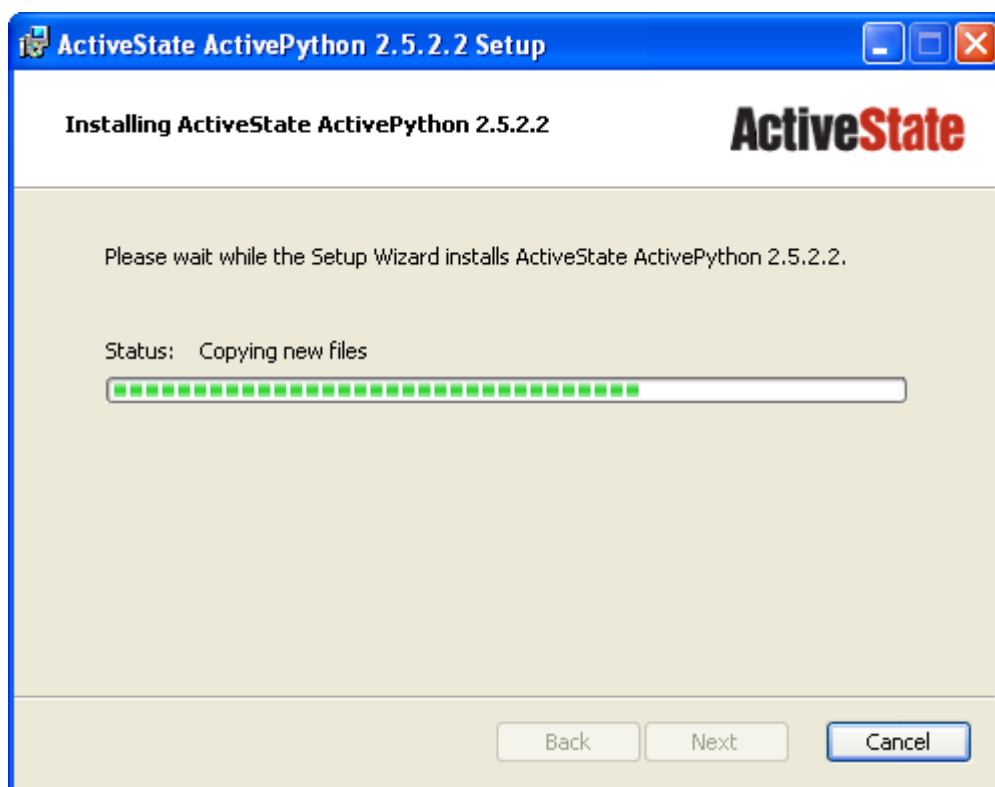
The following steps describe the basic installation procedure. After following this, the user can start using the desired IDE, (even write tests in a text processor and run them from command line!) without any extra work (at least on the Selenium side).

- Installing Python

**Note:** This will cover python installation on Windows and Mac only, as in most linux distributions python is already pre-installed by default.

- Windows

1. Download Active python's installer from ActiveState's official site:  
<http://activestate.com/Products/activepython/index.mhtml>
2. Run the installer downloaded (ActivePython-x.x.x.x-win32-x86.msi)



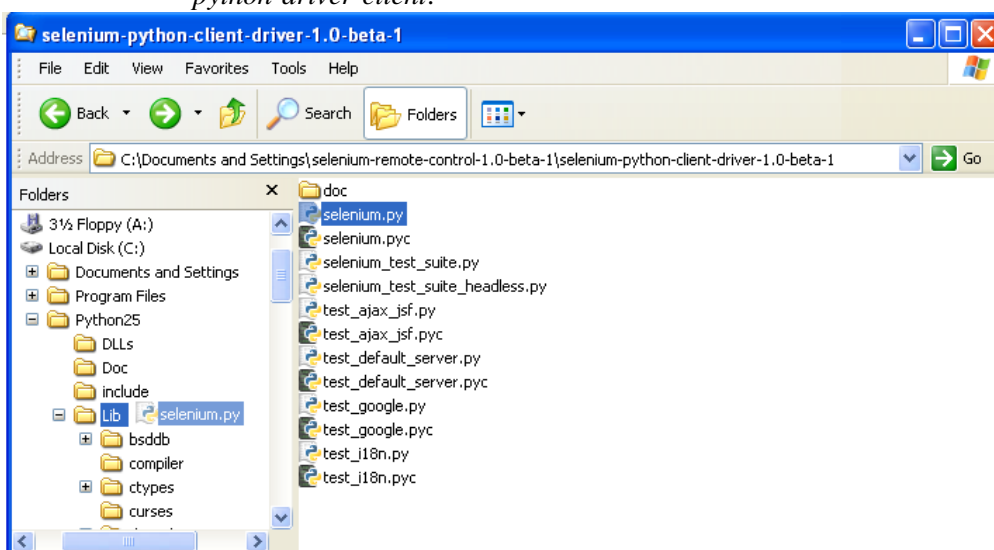
- Mac

The latest Mac OS X version (Leopard at this time) comes with Python pre-installed. To install an extra Python, get a universal binary at <http://www.pythonmac.org/> (packages for Python 2.5.x).

You will get a .dmg file that you can mount. It contains a .pkg file that you can launch.



- Installing the Selenium driver client for python
  1. Download the last version of Selenium Remote Control from the [downloads page](#)
  2. Extract the content of the downloaded zip file
  3. Copy the module with the Selenium's driver for Python (selenium.py) in the folder *C:/Python25/Lib* (this will allow you to import it directly in any script you write). You will find the module in the extracted folder, it's located inside *selenium-python-driver-client*.



Congratulations, you're done! Now any python script that you create can import selenium and start interacting with the browsers.



# LOCATING TECHNIQUES

## 13.1 Useful XPATH patterns

### 13.1.1 text

Not yet written - locate elements based on the text content of the node.

### 13.1.2 starts-with

Many sites use dynamic values for element's id attributes, which can make them difficult to locate. One simple solution is to use XPath functions and base the location on what you do know about the element. For example, if your dynamic ids have the format `<input id="text-12345" />` where 12345 is a dynamic number you could use the following XPath: `//input[starts-with(@id, 'text-')]`

### 13.1.3 contains

If an element can be located by a value that could be surrounded by other text, the contains function can be used. To demonstrate, the element `<span class="top heading bold">` can be located based on the 'heading' class without having to couple it with the 'top' and 'bold' classes using the following XPath: `//span[contains(@class, 'heading')]`. Incidentally, this would be much neater (and probably faster) using the CSS locator strategy `css=span.heading`

### 13.1.4 siblings

Not yet written - locate elements based on their siblings. Useful for forms and tables.

## 13.2 Starting to use CSS instead of XPATH

### 13.2.1 Locating elements based on class

In order to locate an element based on associated class in XPath you must consider that the element could have multiple classes and defined in any order, however with CSS locators this is much simpler (and faster).

- XPath: `//div[contains(@class, 'article-heading')]`

- CSS: `css=div.article-heading`