Welcome To Selenium Training Program

Why is Testing Necessary?

- Testing is necessary because we all make mistakes.
- Some of the mistakes are unimportant, but some of them are expensive or dangerous.
- We need to check everything and anything we produce because things can always go wrong - Humans make mistakes all the time.

Why Automation Testing?

- **What is Automation Testing?**
  - Using a software(tool) to test a software(product/project).
- **Advantage of Automation Testing?**
  - Frequent regression testing
  - Rapid feedback to developers
  - Virtually unlimited iterations of test case execution
  - Support for Agile development methodologies
  - Disciplined documentation of test cases
  - Customized defect reporting
  - Finding defects missed by manual testing

Automation Tools

- HP QTP/UFT
- IBM RFT
- Borland Silk Test
- SmartBear Test Complete
- Selenium

Selenium

Selenium Vs QTP

- Advantages of Selenium over QTP

Selenium	QTP
Open source	Commercial
Can run tests across different browsers	Can only run tests in Internet Explorer, Firefox and Chrome.
Supports various operating systems	Works only in Windows
Supports mobile devices	Supports mobile devise using 3rd party software
Can execute tests while the browser is minimized	Needs to have the application under test to be visible on the desktop
Can execute tests in parallel.	Can execute tests in parallel by using Quality Center which is again a paid product.

Contd..

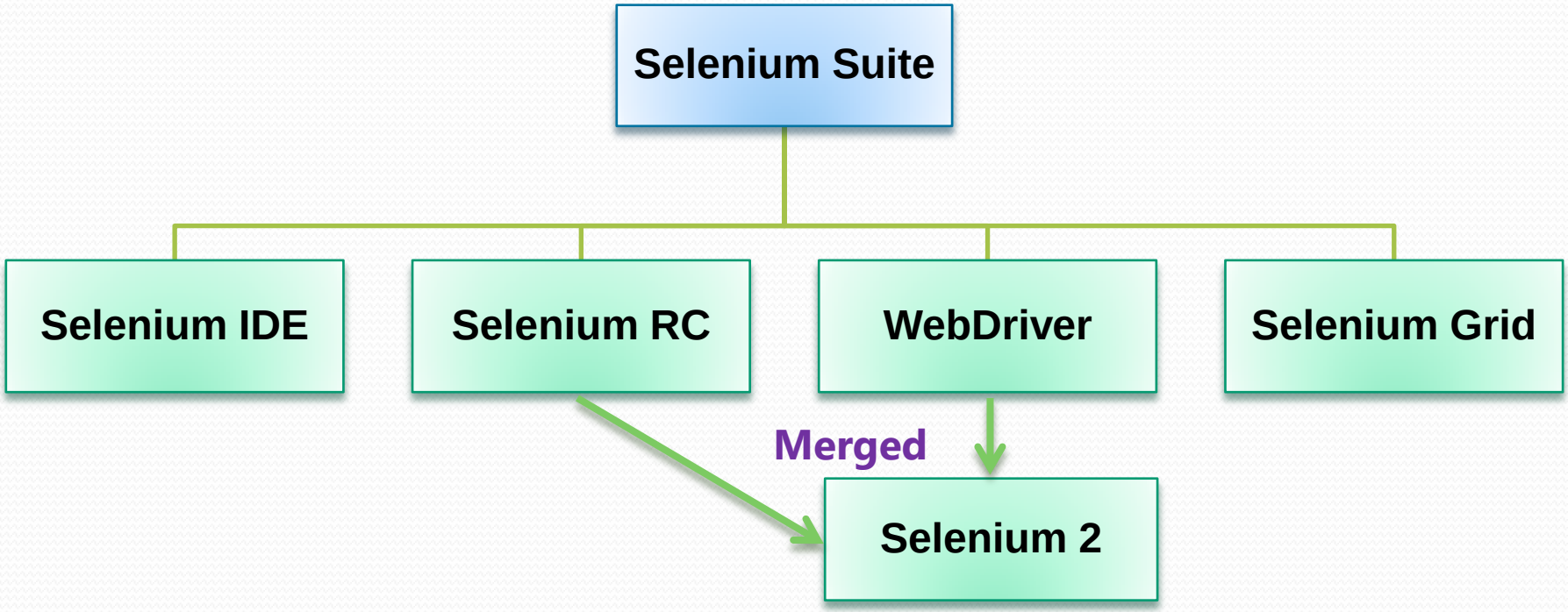
- Advantages of QTP over Selenium

QTP	Selenium
Can test both web and desktop applications	Can only test web applications
Comes with a built-in object repository	Has no built-in object repository
Automates faster than Selenium because it is a fully featured IDE.	Automates at a slower rate because it does not have a native IDE
Data-driven testing is easier to perform because it has built-in global and local data tables.	Data-driven testing is more cumbersome
Provides professional customer support	No official user support is being offered.
Has native capability to export test data into external formats	Has no native capability to export runtime data onto external formats
Test Reports are generated automatically	No native support to generate test /bug reports.

Introduction to Selenium

- Selenium is an open-source(free) automation tool for Functional & Regression Testing.
- Selenium is useful to automate web applications.
- Selenium supports many browsers and operating systems.
- Selenium can be controlled by many programming languages and testing frameworks.
- Selenium is a suite of tools; those are:
  - **Selenium IDE**
  - **Selenium 1 (Selenium RC)**
  - **Selenium 2 (WebDriver)**
  - **Selenium Grid**

Contd..


Selenium Suite
Selenium IDE
Selenium RC
WebDriver
Selenium Grid
Merged
Selenium 2

History of The Selenium Project

Selenium Core

- Developed by **Jason Huggins in 2004**.
- It is a JavaScript Library that would automatically control the browser's actions.
- He named this program as the "**JavaScriptTestRunner**."
- He made JavaScriptRunner open-source which was later re-named as **Selenium Core**.

Selenium RC

- **Paul Hammant** created a server that will act as an HTTP proxy to "trick" the browser into believing that Selenium Core and the web application being tested come from the same domain.
- This system known as the **Selenium Remote Control** or **Selenium 1**.

Contd..

- **WebDriver**
- In 2006 **Simon Stewart** from Google created the **WebDriver**.
- In 2008, Selenium Team merged WebDriver and Selenium RC to form a more powerful tool called **Selenium 2**.

- **Selenium IDE**
- **Shinya Kasatani** of Japan created **Selenium IDE**.
- It is a Firefox extension that can automate the browser through a record-and-playback feature to increase the speed in creating test cases.
- He donated Selenium IDE to the Selenium Project in 2006.

- **Selenium Grid**
- Selenium Grid was developed by **Patrick Lightbody** to address the need of minimizing test execution times as much as possible.

Browser and Environment Support

	Selenium IDE	Selenium RC	WebDriver
Browser Support	Mozilla Firefox	Mozilla Firefox Internet Explorer Google Chrome Safari Opera Others	Internet Explorer versions 6 to 9, both 32 and 64-bit Firefox 3.0, 3.5, 3.6, 4.0, 5.0, 6, 7 and above (current version is 20.0.1) Google Chrome 12.0.712.0 and above (current version is 26.0.1410.64) Opera 11.5 and above (current version is 12.15) Android – 2.3 and above for phones and tablets (devices & emulators) iOS 3+ for phones (devices & emulators) and 3.2+ for tablets (devices & emulators) HtmlUnit 2.9 and above (current version is 2.11)
Operating System	Windows Mac OS X Linux	Windows Mac OS X Linux Solaris	All operating systems where the browsers above can run.

SELENIUM IDE

Introduction Selenium IDE

- Selenium IDE is a framework in the Selenium suite.
- It is a **Firefox plugin**.
- It has a recording feature, which records user actions as they are performed and then exports them as a reusable script in one of many programming languages.
- Selenium IDE is simply intended as a rapid prototyping tool.
- Selenium IDE doesn’t provide iteration or conditional statements for test scripts.
- To create more advanced test cases, you need to use either Selenium RC or WebDriver.

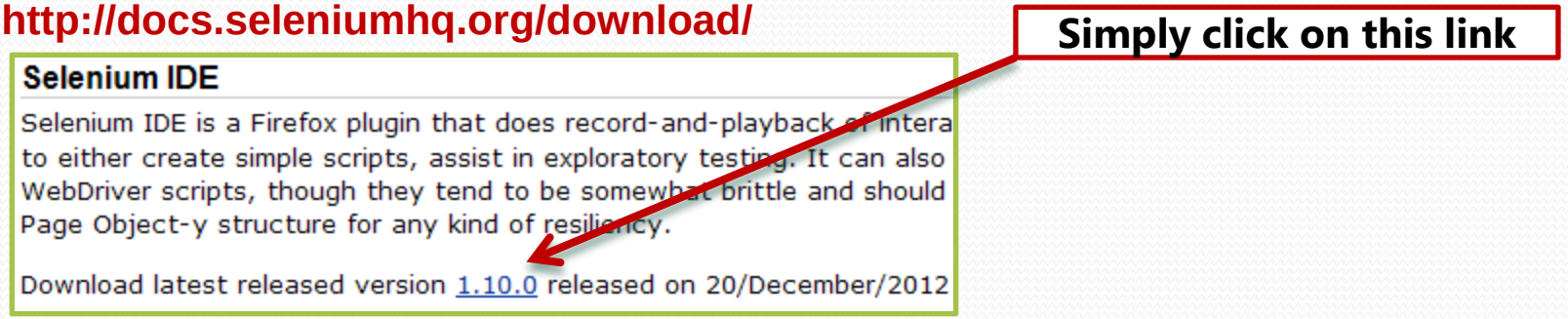
Contd..

PROS	CONS
Very easy to use and install.	Available only in Firefox.
No programming experience is required, though knowledge of HTML and DOM are needed.	Designed only to create prototypes of tests.
Can export tests to formats usable in Selenium RC and WebDriver.	No support for iteration and conditional operations.
Has built-in help and test results reporting module.	Test execution is slow compared to that of Selenium RC and WebDriver.
Provides support for extensions.	

Installing Selenium IDE

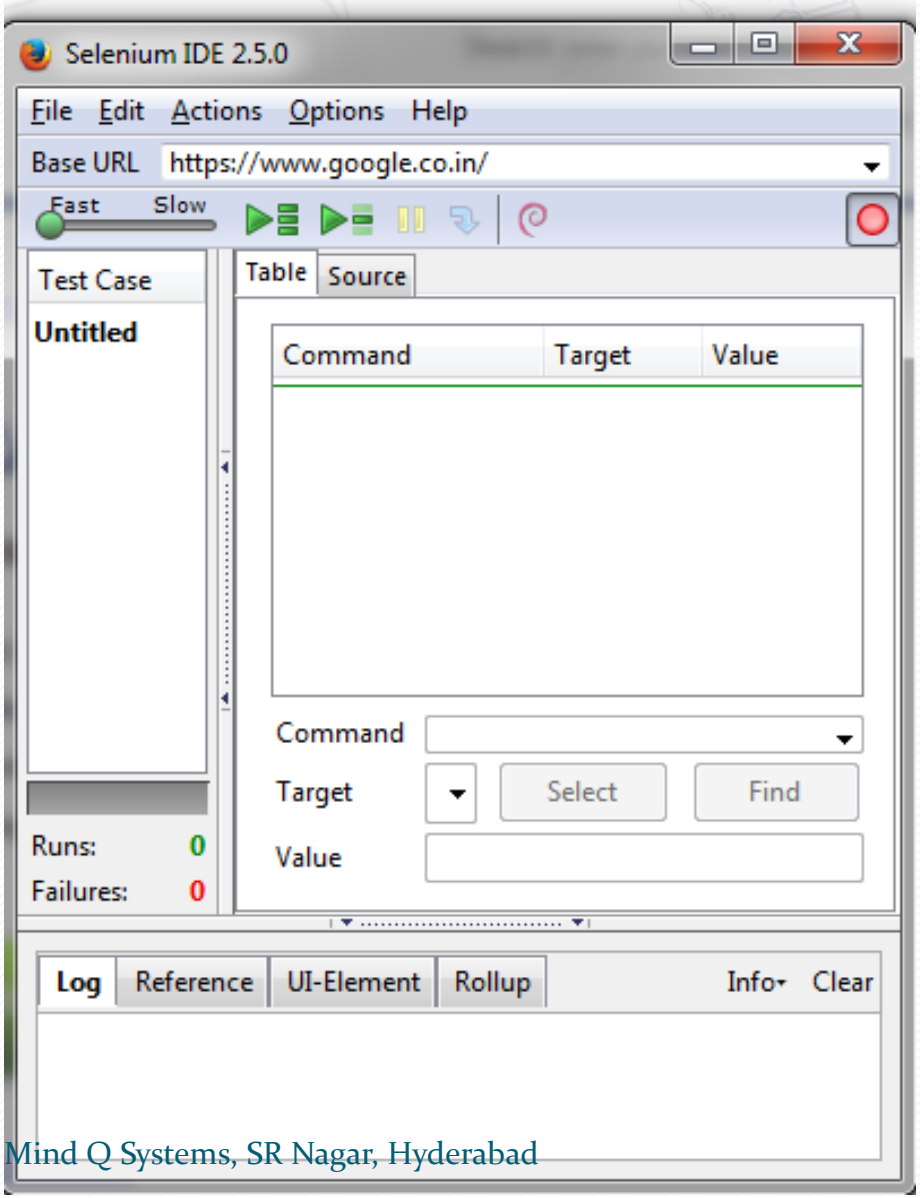
- **Prerequisite:** Mozilla Firefox and Internet Connection.
- Open Firefox and download the IDE from the following SeleniumHQ downloads page:

  http://docs.seleniumhq.org/download/

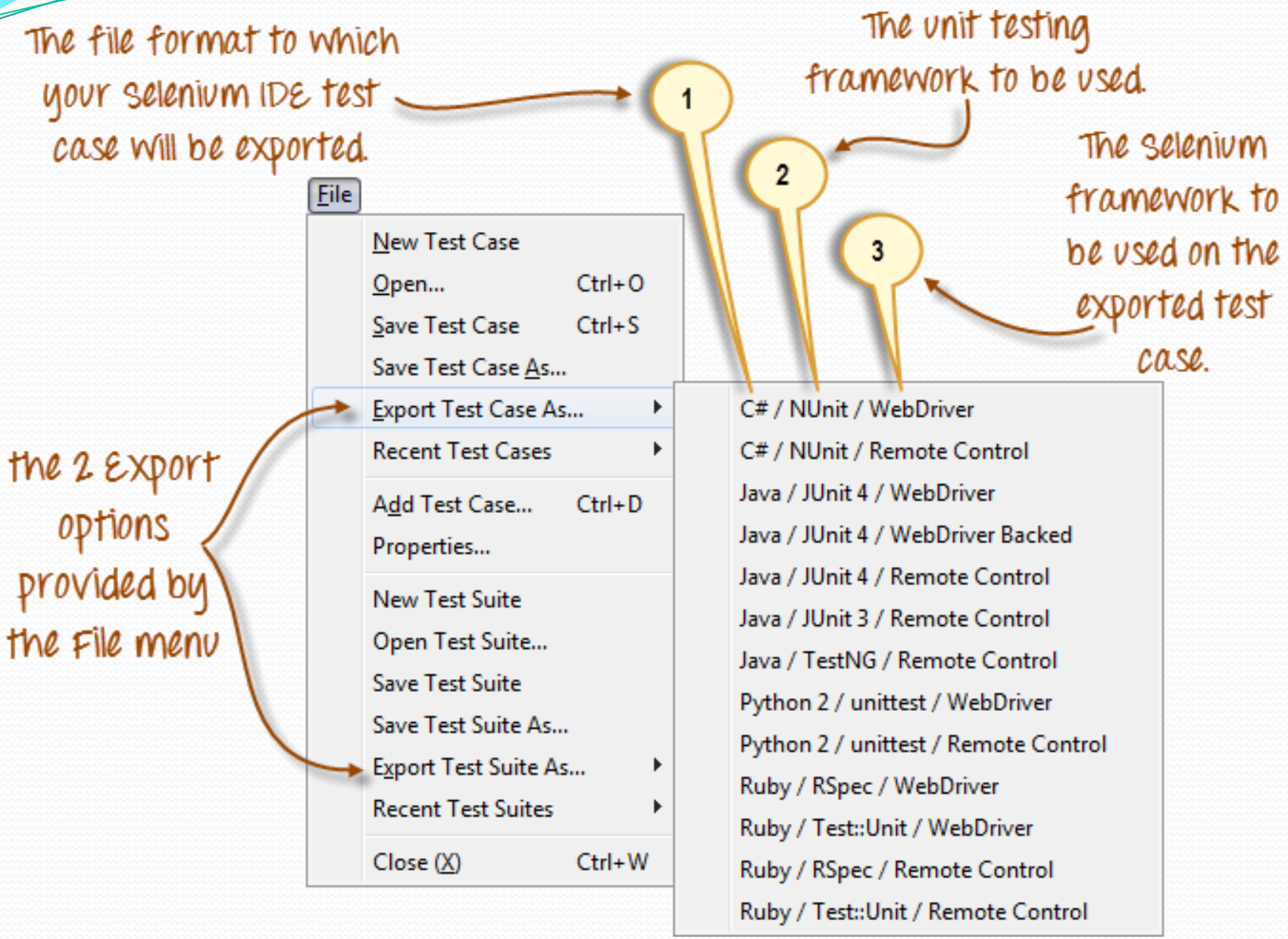


- When a security notification pop-up displayed, click on **"Allow"**
- Wait until Firefox completes the download and then click "**Install Now**."
- Wait until installation is completed. In the pop-up window, click "**Restart Now**."
- After Firefox has restarted, **launch Selenium IDE** using either of two ways:
  - ✓ By short-cut key**: Ctrl+Alt+S**
  - ✓ By clicking on the **Firefox menu button** > **Web Developer** > **Selenium IDE**

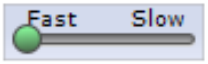
Contd.. (Selenium IDE window)



Contd.. (File > Export …)



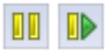
Toolbar



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 you to "step" through a test case by running it one command at a time. Use for debugging test cases.

Apply Rollup Rules: This advanced feature allows repetitive sequences of Selenium commands to be grouped into a single action.

Record: Records the user's browser actions.

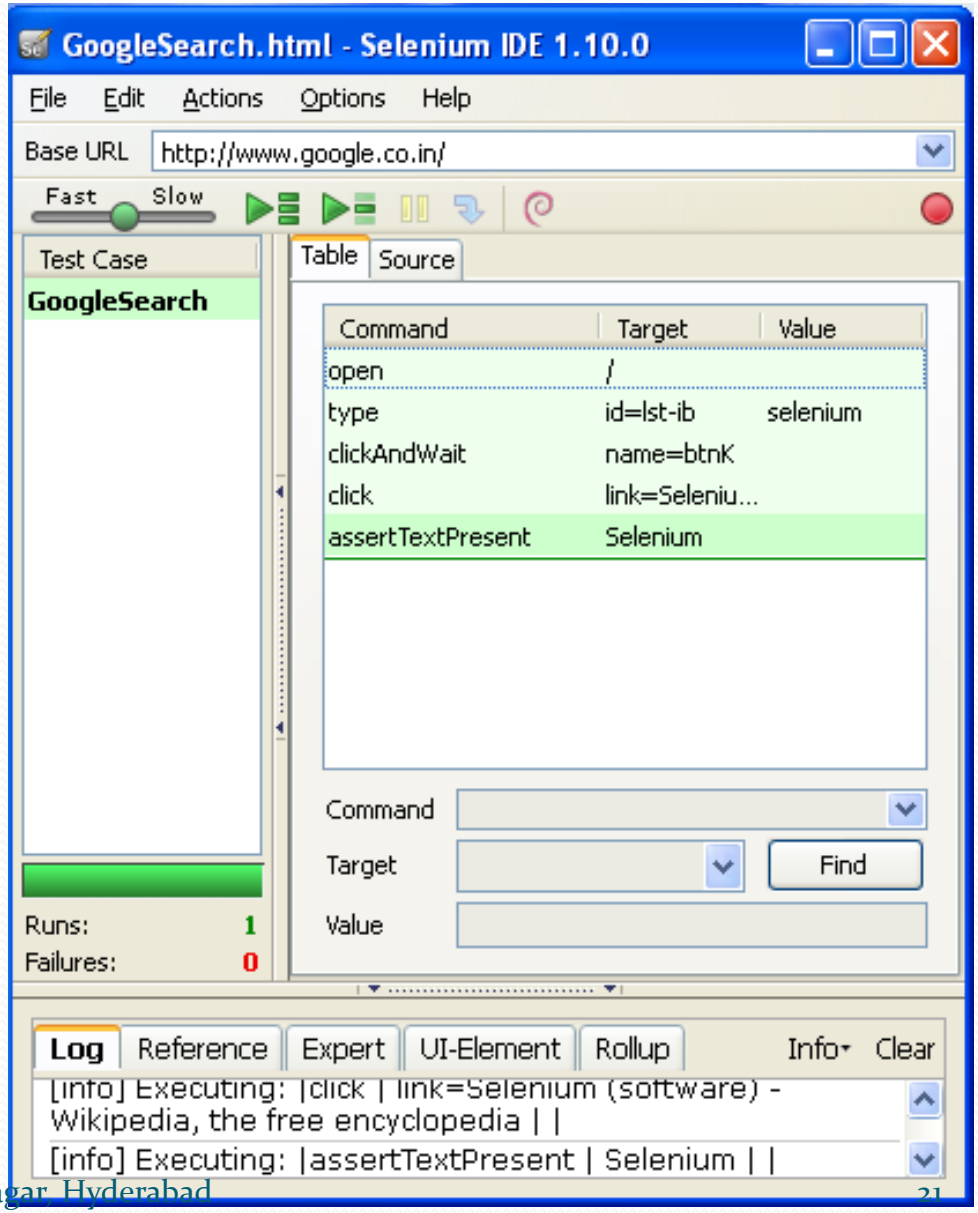
Recording

- Launch Firefox and Selenium IDE.
- Type the value for our Base URL: **http://www.google.co.in/**
- Toggle the Record button on (if it is not yet toggled on by default).
- Type “Selenium” in the text box and click on “Google Search” button
- Click on **“*Selenium* (software) - Wikipedia, the free encyclopedia”** link
- Select text “Selenium”, right click and select **“assertTextPresent”**
- Save the test as **“GoogleSearch”** and you notice that the file was saved as HTML.
- The following script will be generated for the above operations:

Command	Target	Value
open	/	
type	id=lst-ib	selenium
clickAndWait	name=btnK	
click	link=Seleniu...	
assertTextPresent	Selenium	

Playback

- Select the execution mode: **Fast/Slow**
- Click on Playback(Play entire test suite/Play current test case)
- Selenium IDE should be able to replicate everything flawlessly.



Create Test Case Manually

- Manually create a test case for the following steps:

1. Open Google.
2. Type **“Selenium”** in the text box and click on “Google Search” button
3. Click on **“*Selenium* (software) - Wikipedia, the free encyclopedia”** link
4. Insert an assert and verify statements.

Firebug

- Firebug is an open source tool; And It is a Firefox add-on.
- Firebug is a web development tool that facilitates the debugging, editing, and monitoring of any website's CSS, HTML, DOM and JavaScript.
- We use firebug to inspect the HTML elements of the web application under test.
- It will provide us the name of the element that our Selenese command would act upon.

❑ **Installing Firebug**

- Open Firefox and navigate to Firebug's download page: **https://getfirebug.com/downloads/**
- Click the **"download/Add to Firefox"** button.
- Note: Please check your Firefox version and then add the compatible firebug version.

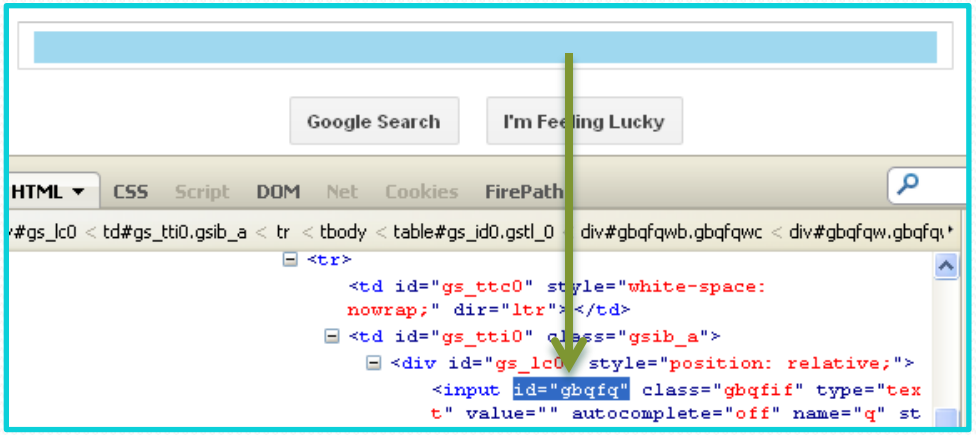
Locators

- **Element Locators** tells Selenium which HTML element a command refers to.
- Many commands require an Element Locator as the "**target**" attribute.
- The different types of locators are:
  - **ID**
  - **Name**
  - **Link**
  - **CSS Selector**
  - **DOM (Document Object Model)**
  - **XPath**

Locating by ID

- Use this when you know an element's id attribute.
- This is the most common way of locating elements since ID's are supposed to be unique for each element.

  Target Format: id=*id of the element*



- **Pros:** Each id is supposed to be unique so no chance of matching several elements
- **Cons:** Works well only on elements with fixed ids and not generated ones

Locating by Name

- Same as Id strategy except that we use the **“name=”** prefix instead.

  Target Format: name=*name of the element*

- **When multiple elements have the same name then we can use Filters.**

- Filters are additional attributes used to distinguish elements with the same name.

  Target Format: name=*name_of_the_element filter=value_of_filter*

  Value: The default filter type is **value** (matching the value attribute).

  Example:

  <input name="continue" type="button" value="Clear" />

  name=continue value=Clear or name=continue Clear or name=continue type=button

Locating by Link Text

- This strategy is intended to select links only and selects the anchor element containing the specified text.
- We access the link by prefixing our target with “link=” and then followed by the hyperlink text.

  Target Format: link=*link_text*

  Example: link = Maps

```
<a id="gb_8" class="gbzt" href="http://maps.google.co.in/maps?hl=en&
tab=wl" onclick="gbar.qs(this);gbar.logger.il(1,{t:8});">
    <span class="gbtb2"></span>
    <span class="gbts">Maps</span>
</a>
```

- **Pros:** Will only select anchor elements, Useful when testing navigation
- **Cons:** You have to know the text of the link before

Locating by CSS Selector

- A “selector” is the instruction in a CSS rule, setting that tells the browser what elements to ‘select’ for styling.
- The CSS locator strategy uses CSS selectors to find the elements in the page.
- It is the most common locating strategy of advanced Selenium users because it can access even those elements that have no ID or name.
- When using this strategy, we always prefix the target box with “**css=**”.

❑ **Locating by Tag and ID**

Syntax: css=*tag#id*

Example: css=input#gbqfq or css=#gbqfq

Contd..

- **Locating by Tag and Class**

  Syntax: css=*tag*.class

  Example: css=input.gbqfif or css=.gbqfif

- **Locating by Tag and Attribute**

  Syntax: css=*tag*[attribute=value]

  Example: css=input[id=gbqfq]

- **Locating by tag, class, and attribute**

  Syntax: css=*tag.class*[*attribute=value*]

  Example: css=input.radio[tabindex=2] or css=.radio[tabindex=2]

- **Locating by inner text**

  Syntax: css=*tag:contains("inner text")*

  Example: css=span:contains("Maps")

Locating by DOM

- DOM is a model in which the HTML document or Web page contains objects (elements, links, etc.).
- There are four basic ways to locate an element through DOM:
  - **getElementById**
  - **getElementsByName**
  - **dom:name**
  - **dom:index**

Contd..

- **getElementById**

  Syntax: document.getElementById(“*id of the element*”)

  Example: document.getElementById("gbqfq")

- **getElementsByName**
  - It will get a collection of elements whose names are all the same.
  - Each element is indexed with a number starting from 0 just like an array

  Syntax:

  document.getElementsByName(“*name*”)[*index*]

  Example:

  document.getElementsByName("radtripType")[1]

Contd..

- **dom:name**

- This method will only apply if the element you are accessing is contained within a named form.

  Syntax: document.forms[“*name of the form*”].elements[“*name of the element*”]

  Example: document.forms["flightsSearchForm"].elements["search_btn"]

- **dom:index**

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

  Syntax: document.forms[*index of the form*].elements[*index of the element*]

  Example: document.forms[0].elements[4] **or**

  document.forms["form1"].elements["txtId"]

Locating by XPath

- XPath is a language used for locating nodes in an XML document.
- Since HTML can be thought of as an implementation of XML, we can also use XPath in locating HTML elements.
- **Advantage:**
  - It can access almost any element, even those without class, name, or id attributes.
- **Disadvantage:**
  - Too many different rules and considerations.
  - Relies on browser's XPath implementation which is not always complete (especially on IE) hence not recommended for cross-browser testing.

XPath Locators

```
<html>
 <body>
  <form id="loginForm">
   <input class="required" name="username" type="text" />
   <input class="required passfield" name="password" type="password" />
   <input name="continue" type="submit" value="Login" />
   <input name="continue" type="button" value="Clear" />
  </form>
 </body>
 </html>
```

Contd..

- Xpath = /html/body/form[1] (3) - *Absolute path*
- Xpath = //form[1] (3) - *First form element in the HTML*
- Xpath = //form[@id='loginForm'] (3) - *The form element with attribute named 'id' and the value 'loginForm'*
- Xpath = //form[input/@name='username'] (3) - *First form element with an input child element with attribute named 'name' and the value 'username'*
- Xpath = //input[@name='username'] (4) - *First input element with attribute named 'name' and the value 'username'*
- Xpath = //form[@id='loginForm']/input[1] (4) - *First input child element of the form element with attribute named 'id' and the value 'loginForm'*
- Xpath = //input[@name='continue'][@type='button'] (7) - *Input with attribute named 'name' and the value 'continue' and attribute named 'type' and the value 'button'*
- Xpath = //form[@id='loginForm']/input[4] (7) - *Fourth input child element of the form element with attribute named 'id' and value 'loginForm'*

What locator strategy shall I USE?

- Locators are classified into two categories:
- **Structure-based locators:** locators that rely on the structure of the page to find elements.
  - Xpath, DOM, CSS
- **Attributes-based locators:** locators that relies on the attributes of the elements to locate them.
  - Id, Name, Link, CSS
- Most people choose CSS because it is the most flexible and gives a good balance between using structure and attributes to find the elements.

Selenium “Selenese” Commands

- A **command** tells Selenium what to do.
- Selenese commands can have up to a maximum of two parameters: **target** and **value**.
- Parameters are not required all the time.
- Selenium commands come in three 'flavors':
  - **Actions**, **Accessors** and **Assertions**.
- Each command call is one line in the test table of the form:

Command	Target	Value

Actions

- An action command is a command that directly interacts with the page elements.

  Example: “click” and “type” commands

- If an Action fails, or has an error, the execution of the current test is stopped.
- Many Actions can be called with "AndWait" suffix,

  Example "clickAndWait"

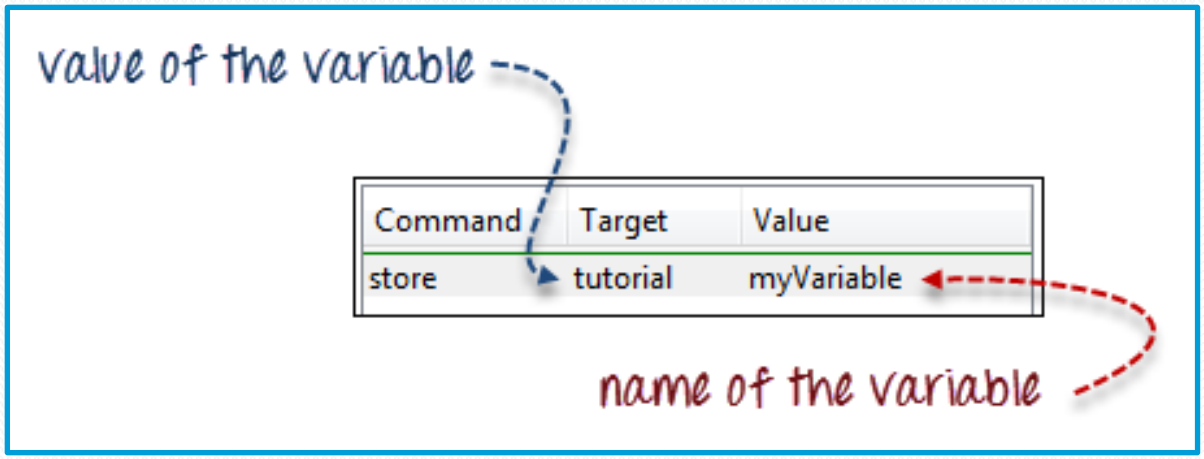
type	id=lst-ib	selenium
clickAndWait	name=btnK	
click	link=Seleniu...	

Accessors

- These are the commands that allows you to store values to a variable.

  Example: **“storeTitle”, storeText**

storeTitle	title



Assertions

- These are the commands that verify the state of the application and conforms what is expected.
- Assertions can be used in 3 modes: **"assert", "verify",** and **"waitFor"**.

  For example: **"assertText", "verifyText"** and **"waitForText"**.
- When an **"assert"** fails, the test execution is stopped.
- When a **"verify"** fails, the test execution will continue but logs the failure.
- **"waitFor"** commands wait for some condition to become true.

assertTextPresent	Selenium
verifyTextPresent	Selenium
waitForAllLinks	

Selenium Common Commands

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

Wait commands

- **The "AndWait" Commands**

- 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.
- If you use an *AndWait* command for an action that does not trigger a navigation/refresh, your test will fail.

Contd..

- **The waitFor Commands**

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

echo - The Selenese Print Command

- Selenese has a simple command that allows you to print text to your test's output.
- echo statements can be used to print the contents of Selenium variables.

Command	Target	Value
echo	Testing page footer now.	
echo	Username is	

Store Commands and Selenium Variables

- You can use Selenium variables to store constants at the beginning of a script.
- 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.

Command	Target	Value
store	Hyderabad	myPlace

- To access the variable, simply enclose it in a ${ … } symbol.
- **Example:**

Command	Target	Value
type	Id=userName	${myPlace}

Contd..

- **StoreElementPresent -** stores a Boolean value–"true" or "false"–depending on whether the UI element is found.

Command	Target	Value
StoreElementPresent	Id=userName	myPlace
echo	${myPlace}	

- **storeText -** used to store the inner text of an element into a variable.

Command	Target	Value
storeText	css=h1	textVar
echo	${textVar}	

Alerts, Popups, and Multiple Windows

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

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

Confirmations

- Confirmations are pop-ups that give you an OK and a CANCEL button, as opposed to alerts which give you only the OK button.
- The commands you can use in handling confirmations are:
  - **assertConfirmation/assertNotConfirmation**
  - **assertConfirmationPresent/assertConfirmationNotPresent**
  - **storeConfirmation**
  - **storeConfirmationPresent**
  - **verifyConfirmation/verifyNotConfirmation**
  - **verifyConfirmationPresent/verifyConfirmationNotPresent**
  - **chooseOkOnNextConfirmation/chooseOkOnNextConfirmationAndWait**
  - **chooseCancelOnNextConfirmation**

Multiple Windows

- We use the **selectWindow** command in switching between windows.
- Setting selectWindow's target to "null" will automatically select the parent window.

Command	Target	Value
open	/	
click	id=netbanking	
click	id=loginsubmit	
waitForPopUp	HDFC Bank NetBanking - Internet Banking Services by HDFC Bank	30000
selectWindow	HDFC Bank NetBanking - Internet Banking Services by HDFC Bank	
click	//*[@id='wrapper']/div[6]/a/img	
selectWindow	null	
storeTitle	MyHomePage	
echo	${MyHomePage}	
click	id=prepaidcard	

Thank you