**Efficient Selenium Scripting and Troubleshoot Scenarios**

In the previous module, we discussed the technical implications **while implementing logging in a framework**. We discussed [**log4j utility**](http://www.softwaretestinghelp.com/log4j-tutorial-selenium-tutorial-26/) at length. We discussed the basic components those constitute log4j from a usability perspective. With the Appenders and layouts, user is leveraged to choose the desired logging format/pattern and the data source/location.

In the current 27th module in this , we would shift our focus towards a few trivial **yet important topics** that would guide us troubleshoot some recurrent problems. We may or may not use them in daily scripting but they would be helpful in the long run.

We would **discuss some advance concepts wherein we would deal with mouse and keyboard events, accessing multiple links by implementing lists**. So why not let’s just start and briefly discuss these topics with the help of appropriate scenarios and code snippets.

### JavaScript Executors

While automating a test scenario, there are certain actions those become an inherent part of test scripts.

**These actions may be:**

* Clicking a button, hyperlink etc.
* Typing in a text box
* Scrolling Vertically or Horizontally until the desired object is brought into view
* And many more

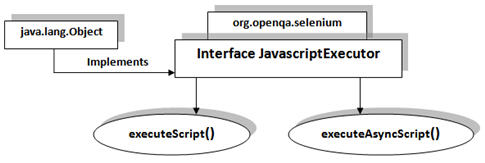
Now, it is evident from the earlier modules that the best way to automate such actions is by using Selenium commands.

**But what if the selenium commands don’t work?**

Yes, it is absolutely possible that the very basic and elementary Selenium Commands don’t work in certain situations.

That said, to be able to troubleshoot such situation, we shoulder JavaScript executors into the picture.

**What are JavaScript Executors?**

[](http://cdn2.softwaretestinghelp.com/wp-content/qa/uploads/2014/11/JavaScript-Executors1.jpg)

JavascriptExecutor interface is a part of org.openqa.selenium and implements java.lang.Object class. JavascriptExecutor presents the capabilities to execute JavaScript directly within the web-browser. To be able to execute the JavaScript, certain mechanisms in the form of methods along with a specific set of parameters are provided in its implementation.

**Methods**

**executeScript (String script, args)**

As the method name suggests, it executes the JavaScript within the current window, alert, frame etc (the window that the WebDriver instance is currently focusing on)

**executeAsyncScript (String script, args)**

As the method name suggests, it executes the JavaScript within the current window, alert, frame etc (the window that the WebDriver instance is currently focusing on)

The parameters and import statement are common to both the executor methods.

**Parameters**  
Script – the script to be executed  
Argument – the parameters that the script requires for its execution (if any)

**Import statement**  
To be able to use JavascriptExecutors in our test scripts, we need to import the package using the following syntax:

import org.openqa.selenium.JavascriptExecutor;

**Sample Code**

**#1) Clicking a web element**

|  |  |
| --- | --- |
| 1 | // Locating the web element using id |
| 2 | WebElement element = driver.findElement(By.id("id of the webelement")); | |

|  |  |
| --- | --- |
| 3 |  |
| 4 | // Instantiating JavascriptExecutor | |

|  |  |  |
| --- | --- | --- |
| 5 | JavascriptExecutor js = (JavascriptExecutor)driver; | |
| 6 |  |

|  |  |
| --- | --- |
| 7 | // Clicking the web element |
| 8 | js.executeScript("arguments[0].click();", element); | |

**#2) Typing in a Text Box**

|  |  |
| --- | --- |
| 1 | // Instantiating JavascriptExecutor |
| 2 | JavascriptExecutor js = (JavascriptExecutor)driver; | |

|  |  |
| --- | --- |
| 3 |  |
| 4 | // Typing the test data into Textbox | |

|  |  |
| --- | --- |
| 5 | js.executeScript("document.getElementById(‘id of the element’).value=’test data’;”); |

**#3) Scrolling down until the web element is in the view**

|  |  |  |
| --- | --- | --- |
| 1 | WebElement element=driver.findElement(By.xpath("//input[contains(@value,'Save')]")); | |
| 2 |  |

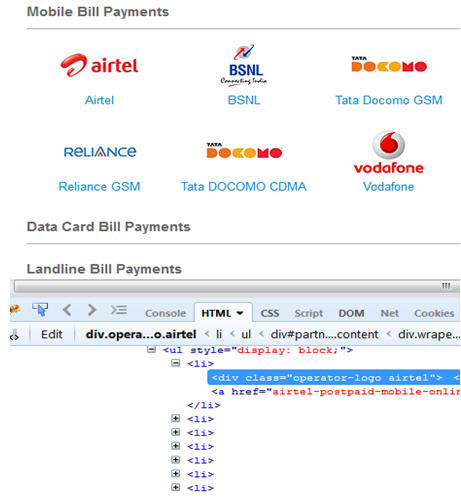
|  |  |  |
| --- | --- | --- |
| 3 | | // Instantiating the javascriptExecutor and scrolling into the view in the single test step |
| 4 | ((JavascriptExecutor)driver).executeScript("arguments[0].scrollIntoView(true);",element); | |

You may find various other ways of writing down the code for accessing JavascriptExecutors.

### Accessing multiple elements in a List

At times, we may come across elements of same type like multiple hyperlinks, images etc arranged in an ordered or unordered list. Thus, it makes absolute sense to deal with such elements by a single piece of code and this can be done using WebElement List. Refer the screenshot below to understand the elements I am talking about.

------------

[](http://cdn2.softwaretestinghelp.com/wp-content/qa/uploads/2014/11/Selenium-scripting-tips-1.jpg)

In the above image, we see that the various service providers belong to an unordered list. Thus, verification of click ability and visibility of these elements can be done by a single piece of code by using a list of elements.

**Import statement**  
To be able to use WebElement list in our test scripts, we need to import the package using the following syntax:

import java.util.List;

**Sample Code**

|  |  |
| --- | --- |
| 1 | // Storing the list |
| 2 | List <WebElement> serviceProviderLinks = driver.findElements(By.xpath("//div[@id='ServiceProvider']//ul//li")); | |

|  |  |
| --- | --- |
| 3 |  |
| 4 | // Fetching the size of the list | |

|  |  |  |
| --- | --- | --- |
| 5 | int listSize = serviceProviderLinks.size(); | |
| 6 | for (int i=0; i<listSize; i++) |

|  |  |  |
| --- | --- | --- |
| 7 | { | |
| 8 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | // Clicking on each service provider link | | |
| 10 | | serviceProviderLinks.get(i).click(); |

|  |  |
| --- | --- |
| 11 |  |
| 12 | // Navigating back to the previous page that stores link to service providers | |

|  |  |  |
| --- | --- | --- |
| 13 | driver.navigate().back(); | |
| 14 | } |

There are various requirements under which the lists can be used to verify the elements with suitable implementation changes.

### Handling keyboard and mouse events

**Handling Keyboard Events**

As also said earlier, there are n numbers of ways to deal with the same problem statement in different contexts.

Thus, at times a necessity arises to deal with a problem by changing the conventional dealing strategy with a more advance strategy. I have witnessed cases where I could not deal with alerts and pop up etc. by selenium commands thus I had to opt for different java utilities to deal with it using keyboard strokes and mouse events.

Robot class is one such option to perform keyboard events and mouse events.

Let us understand the concept with the help of a scenario and its implementation.

**Scenario:**

Let us gather a situation where an unnecessary pop up appears on the screen which cannot be accepted or dismissed using Alert Interface, thus the only wise option we are left with is to close down the window using shortcut keys – “Alt + space bar + C”. Let us see how we close the pop up using Robot Class.

Before, initiating the implementation, we should import the necessary package to be able to use Robot class within our test script.

**Import Statement**

import java.awt.Robot;

**Sample Code**

|  |  |  |
| --- | --- | --- |
| 1 | // Instantiating Robot class | |
| 2 | Robot rb =new Robot(); |

|  |  |
| --- | --- |
| 3 |  |
| 4 | // Calling KeyPress event | |

|  |  |
| --- | --- |
| 5 | rb.keyPress(KeyEvent.VK\_ALT); |
| 6 | rb.keyPress(KeyEvent.VK\_SPACE); | |

|  |  |  |
| --- | --- | --- |
| 7 | rb.keyPress(KeyEvent.VK\_C); | |
| 8 |  |

|  |  |  |
| --- | --- | --- |
| 9 | // Calling KeyRelease event | |
| 10 | | rb.keyRelease(KeyEvent.VK\_C); | |

|  |  |  |
| --- | --- | --- |
| 11 | rb.keyRelease(KeyEvent.VK\_SPACE); | |
| 12 | rb.keyRelease(KeyEvent.VK\_ALT); |

Robot class can also be used to handle mouse events but let us here look at the selenium’s capabilities to handle mouse events.

**Handling Mouse Events**

WebDriver offers a wide range of interaction utilities that the user can exploit to automate mouse and keyboard events. Action Interface is one such utility which simulates the single user interactions.

Thus, we would witness Action Interface to mouse hover on a drop down which then opens a list of options in the next scenario.

**Scenario:**

1. Mouse Hover on the dropdown
2. Click on one of the items in the list options

**Import Statement**

import org.openqa.selenium.interactions.Actions;

**Sample Code**

|  |  |
| --- | --- |
| 1 | // Instantiating Action Interface |
| 2 | Actions actions=<strong>new</strong> Actions(driver); | |

|  |  |
| --- | --- |
| 3 |  |
| 4 | // howering on the dropdown | |

|  |  |  |
| --- | --- | --- |
| 5 | actions.moveToElement(driver.findElement(By.<em>id</em>("id of the dropdown"))).perform(); | |
| 6 |  |

|  |  |
| --- | --- |
| 7 | // Clicking on one of the items in the list options |
| 8 | WebElement subLinkOption=driver.findElement(By.id("id of the sub link")); | |

|  |  |
| --- | --- |
| 9 | subLinkOption.click(); |

### Conclusion

In this module, we discussed some advance topics related to efficient scripting and to troubleshoot scenarios where the user is required to handle mouse and keyboard events. We also discussed how to store more than one web element in a list. I hope you would be able to troubleshoot these impediments if encountered.