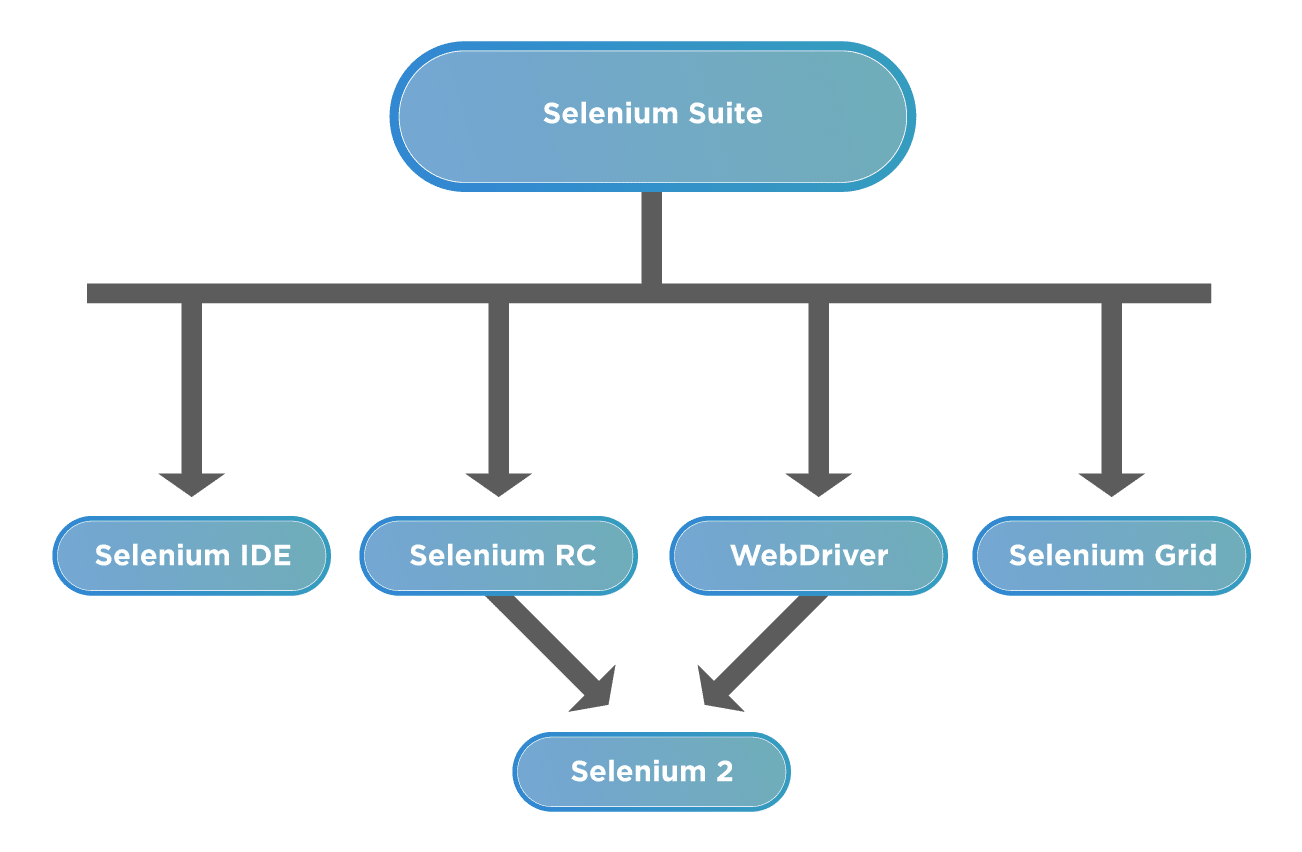
**SELENIUM NOTES**

**WebDriver Architecture**

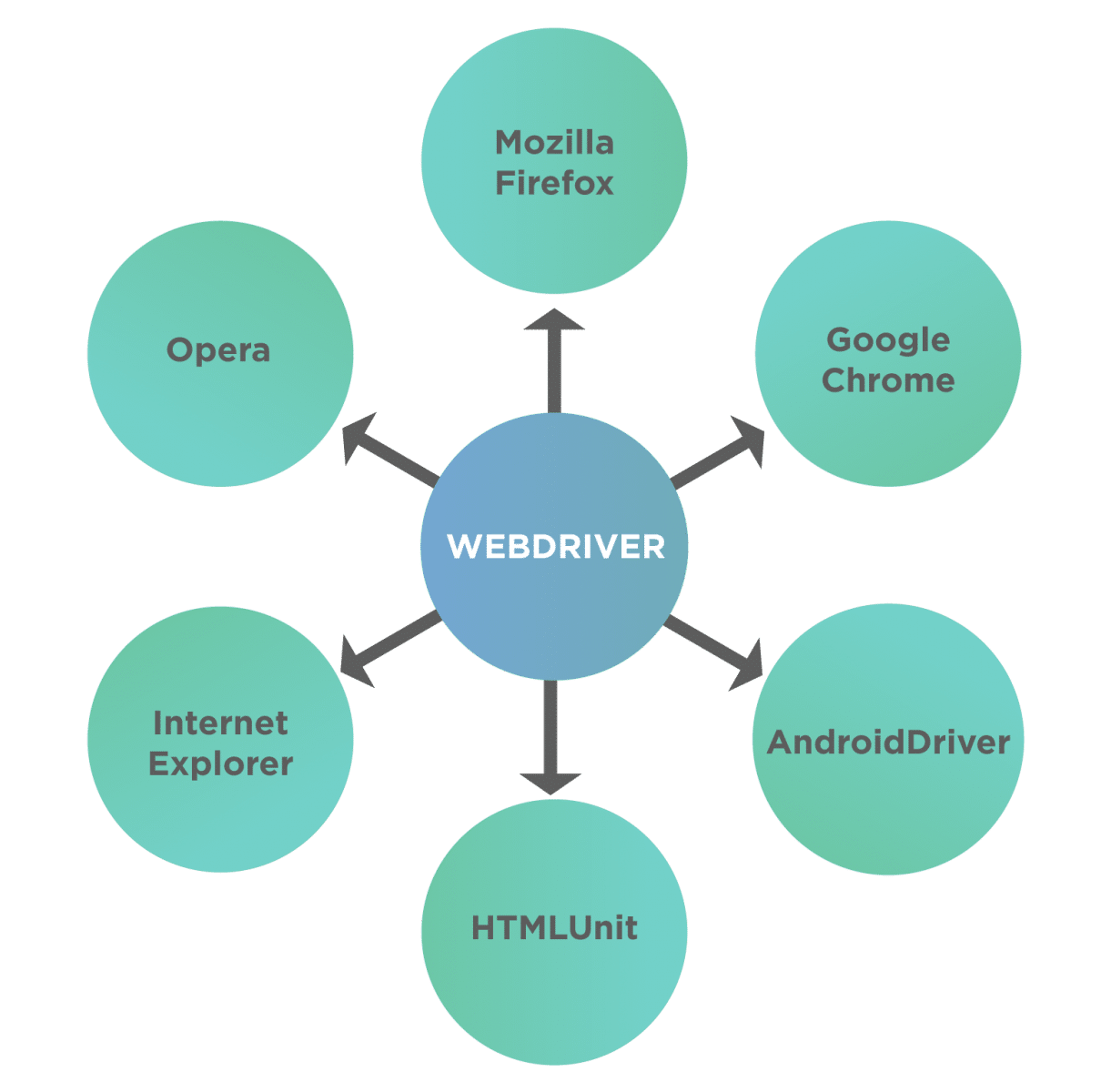
***Selenium WebDriver*** is a set of open-source [***APIs,***](https://en.wikipedia.org/wiki/Application_programming_interface) which provided the capabilities to interact with any of the modern web-browsers and then, in-turn to automate the user actions with that browser. It is an essential component of the ***Selenium*** family. As we know, Selenium is not an independent tool; rather, it is a collection of tools that make the Selenium suite, which was created when two projects ***Selenium RC and WebDriver were merged.***



***Why Selenium WebDriver is popular?***

Apart from the above-mentioned capabilities, *WebDriver*, being part of the Selenium family, also encompassed some of the unique characteristics, which adds to its popularity as a web automation tool. A few of those characteristics are:

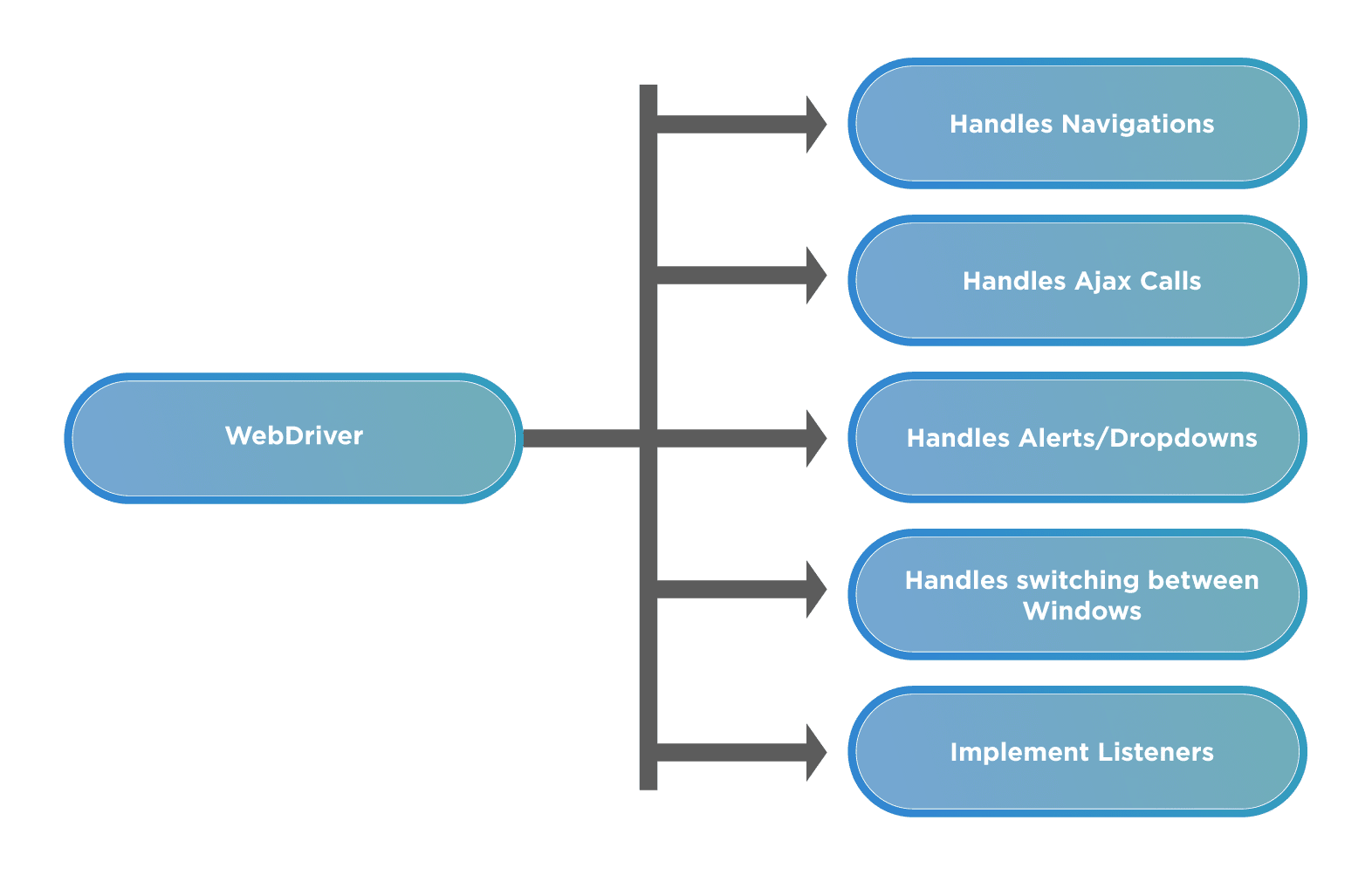
* ***Multi-Browser Compatibility****- One of the prime reasons for the popularity of Selenium and WebDriver is its cross-browser support using the same piece of code. It gives the ability to run a specific piece of code that mimics a real-world user using a browser's native support to hit direct API calls without the need for any middleware software or device. The below shows a sample list of browsers supported:*



* ***Multi-Language Support****- Not all testers are well versed with a particular language. Since Selenium provides support for many languages so a tester can use any language out of the supported languages and then use WebDriver for automation. This gives the freedom to write code in the language people are comfortable with.*
* ***Faster Execution****- Unlike Selenium RC, WebDriver doesn't depend on a middleware server to communicate with the browser. WebDriver directs communications with browsers using a defined protocol (JSON Wire), which enables it to communicate faster than most Selenium tools. Also since JSON Wire itself uses JSON, which is very lightweight, so the amount of data transfer per call is very minimum. The below figure shows clearly how the WebDriver communicates with the Browser:*



* ***Locating Web Elements****- In order to perform actions like Click, Type, Drag, and Drop we first need to identify on which web element (like button, checkbox, drop-down, textarea) we need to perform an action. To facilitate this, WebDriver has provided methods to identify web elements using various HTML attributes - like id, name, class, CSS, tag name, XPath, linktext etc.*
* ***Handling dynamic web elements****- There are times when there are web elements on-page, which change with every reload of the page. Since the HTML attributes change, it becomes a challenge to identify these elements. Selenium provides multiple methods to handle these situations -*
  + ***Absolute Xpath****- this contains the complete XML path of the element in question.*
  + ***Contains( )****- using these functional elements can be searched with partial or full text and can be used to handle dynamic elements.*
  + ***Starts-With( )****- this function is based on finding elements using starting text of the attribute under question.*
* ***Handling Waiting for Elements****- Not all the pages have the same structure. Some are lightweight, while some have a considerable amount of data handling or AJAX calls. Many times the web elements take some time to load. To account for this WebDriver has provided multiple waiting mechanisms that can be used to pause the script execution for a required amount of time based on certain conditions and then continue once the condition is full-filled. The following figure shows a sample list that shows the capabilities of WebDriver which helps in handling the dynamic behavior of web pages.*



***What are the drawbacks of Selenium WebDriver?***

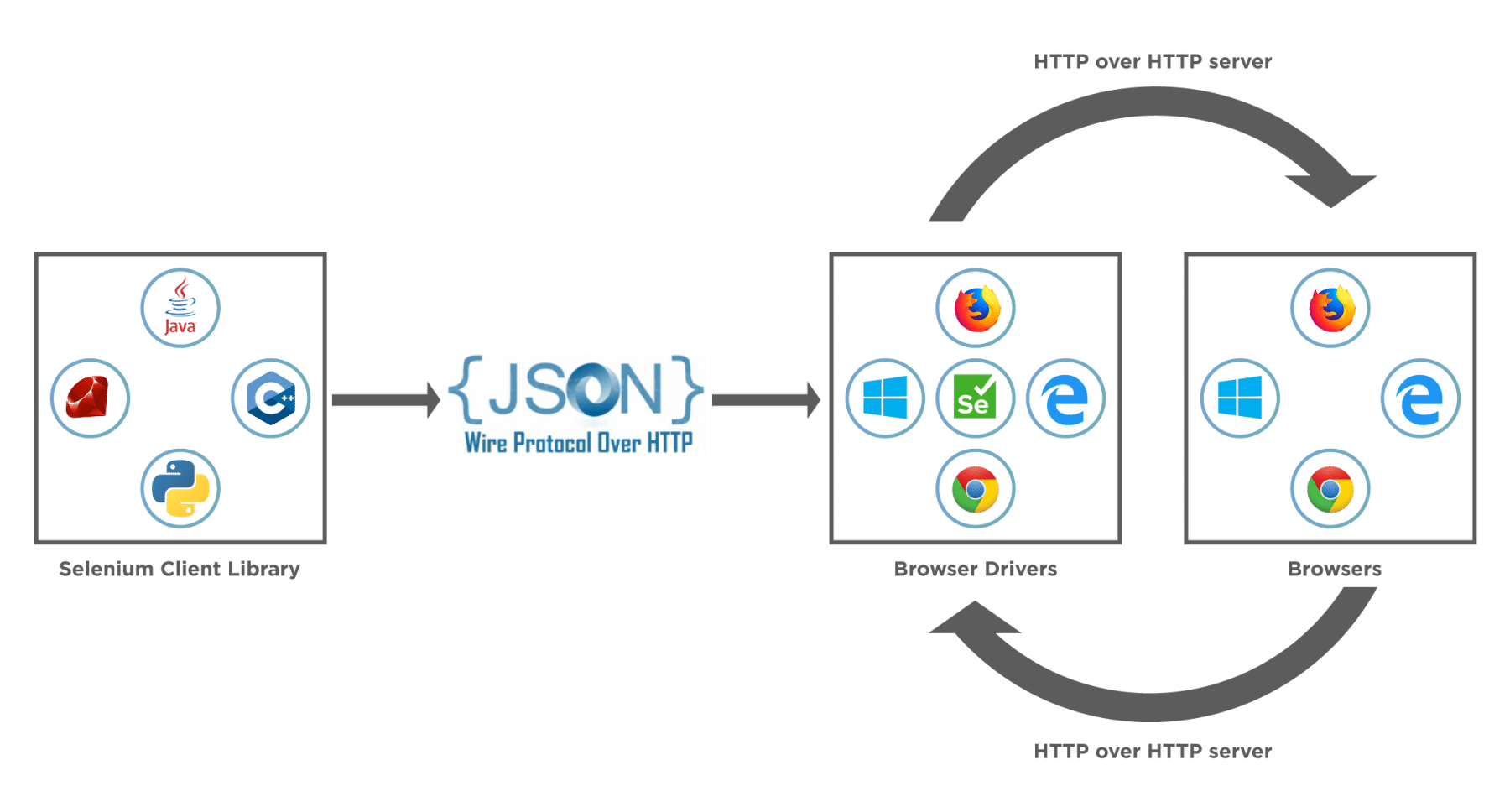
Although Selenium works a long way out in solving the \*UI \*and functional automation of web applications, it is not without its drawbacks. Let's look at some of the shortcomings/drawbacks:

* ***Requires Programming Knowledge and Expertise****- Since WebDriver allows you to automate the user actions using code written in a certain programming language, anyone who wants to use it should have a basic understanding of how coding in the language works. People who do not have an understanding of coding in a programming language will find it hard to use WebDriver.*
* ***No Support for Desktop Applications.****- Selenium ecosystem, including WebDriver, was built for the automation of web-applications. As such if you are looking to automate windows based applications, you will not be able to do so.*
* ***No Customer Support****- Selenium ecosystem, including WebDriver is completely open-source, which means it is driven by individuals and not by any company. Because of this, there is not a dedicated support team to look into your issues. If a person is stuck somewhere, there are many communities, forums which a person can rely on, but that's about it.*
* ***No Built In Object Repository****- Paid tools like UFT/QTP provide a centralized location to store objects/elements, called the Object Repository. This ability is not provided by default in Selenium WebDriver. This can be overcome using approaches like the Page Object Model, but it requires considerable coding skills and expertise.*
* ***Lack of built-in reporting****- Selenium WebDriver can help you run your automation tests but to provide a reporting capability, you would need to integrate it with a testing framework like Junit, TestNG, PyTest, Allure, etc.*
* ***Managing Browser-Selenium Dependencies****- Since Selenium has to rely on compatibility between the browser drivers and the actual browser itself, at many times due to incompatibility or bugs in either the browser driver or browser, functionality breaks, and users have to rely on community support to get it fixed.*

## Understanding of Selenium WebDriver Architecture

Being a part of the overall component system, we deduce that the ***Selenium WebDriver*** is not a standalone testing tool.  It comprises various components that are required to run tests. These are the architectural components of Selenium.

So first let's take a look at this image below

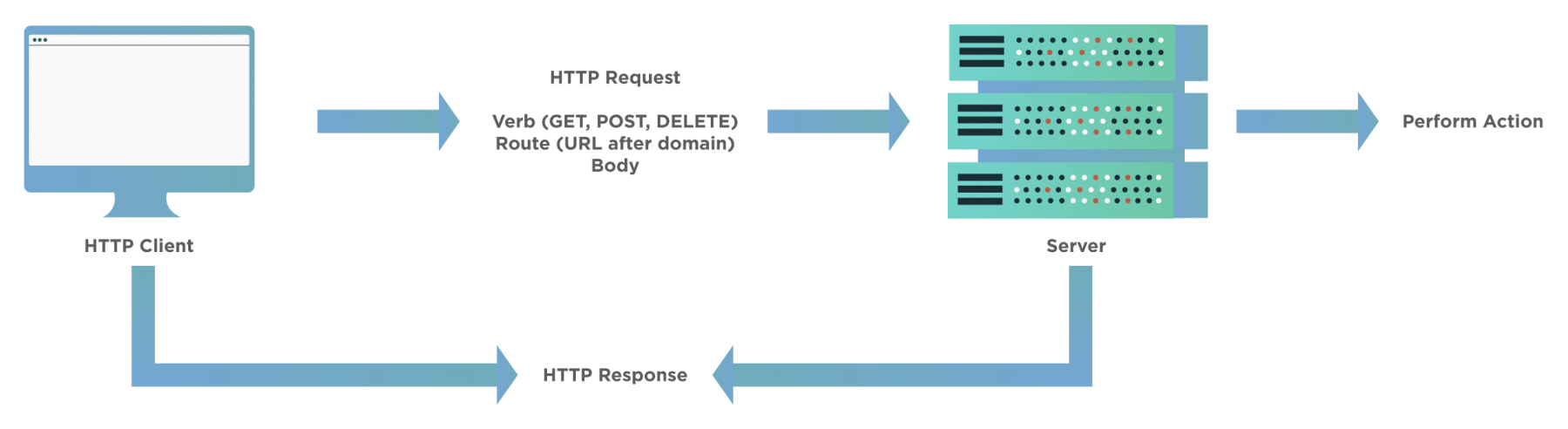


This image tells us about the ***core selenium webdriver architecture and the major selenium components*** which comprise WebDriver.

* ***Selenium WebDriver Client Libraries / Language Bindings*** – Software Testers want to select languages that they are comfortable with. Since WebDriver Architecture supports different languages, so there are bindings available for a range of languages like [***Java,***](https://www.toolsqa.com/java/java-tutorial/) C#, [***Python,***](https://www.toolsqa.com/python-tutorial/) Ruby, PHP, etc. Anyone who has a basic knowledge of working with any programming language can get specific language bindings and can start off. This is how Selenium Achitecture provides flexibility to testers to do automation in their comfort zone.
* ***JSON WIRE PROTOCOL*** - As per the Selenium Architecture above, the JSON Wire Protocol facilitates all the communication that is happening in Selenium between the browser and the code. This is the heart of Selenium. JSON Wire Protocol provides a medium for data transfer using a RESTful (Representational State Transfer)  API which provides a transport mechanism and defines a RESTful web service using  JSON over HTTP.
* ***Browser Drivers*** – Since there are various browsers that are supported by Selenium, each browser has its own implementation of the W3C standard that Selenium provides. As such browser-specific binaries are available that are specific to the browser and hides the implementation logic from the end-user. JSONWire protocol establishes a connection between the browser binaries and the client libraries.
* ***Browsers*** – Selenium will be only able to run tests on the browsers if they are locally installed, either on the local machine or on the server machines. So browser installation is necessary.

### *****How Selenium WebDriver works?*****

In the section above, we saw the architecture of Selenium. Now let's see how behind the scenes all the communication happens?  Take a look at the image below – this shows a view of how the actual workflow looks like.



When a user writes a ***WebDriver code in Selenium*** and executes it, the following actions happen in the background –

* An HTTP request generates, and it goes to the respective browser driver (Chrome, IE, Firefox). There is an individual request for each Selenium command.
* The browser driver receives the request through an HTTP server.
* The HTTP server decides which actions/instructions need to execute on the browser.
* The browser executes the instructions/steps as decided above.
* The HTTP server then receives the execution status and then sends back the status to an automation script, which then shows the result ( as passed or an exception or error).

## How to use Selenium WebDriver for Web Automation?

Selenium WebDriver provides a very seamless, user-friendly, and code friendly approach to automation using various browsers. Since it supports most of the major browser vendors, it's just a matter of using the respective browser driver and browser and setting up Selenium to use the same.

For any Selenium test script, there are generally the following 7 steps, which apply to all the test cases and all the applications under test (AUT):

1. ***Create an instance of WebDriver specific to the Browser:***

* Eg: To create an instance of the Firefox driver, we can use the following commands:

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

WebDriver driver = new FirefoxDriver();

1. ***Navigate to the desired Web page which needs to be automated:***

* Eg: To navigate to the [***"https://demoqa.com/text-box",***](https://demoqa.com/text-box) we can use the following command:

driver.get("https://demoqa.com/text-box")

1. ***Locate an HTML element on the Web page:***

* In order to interact with a web page, we need to locate the HTML elements on the web page. We can use any of the element locator strategies mentioned at [***"Selenium Locators"***](https://www.toolsqa.com/selenium-webdriver/selenium-locators/). Eg: if we want to get the "***Full Name***" text box, we can use the following commands:

import org.openqa.selenium.By;

import org.openqa.selenium.WebElement;

WebElement usernameElement = driver.findElement(By.id("userName"));

1. ***Perform an action on an HTML element:***

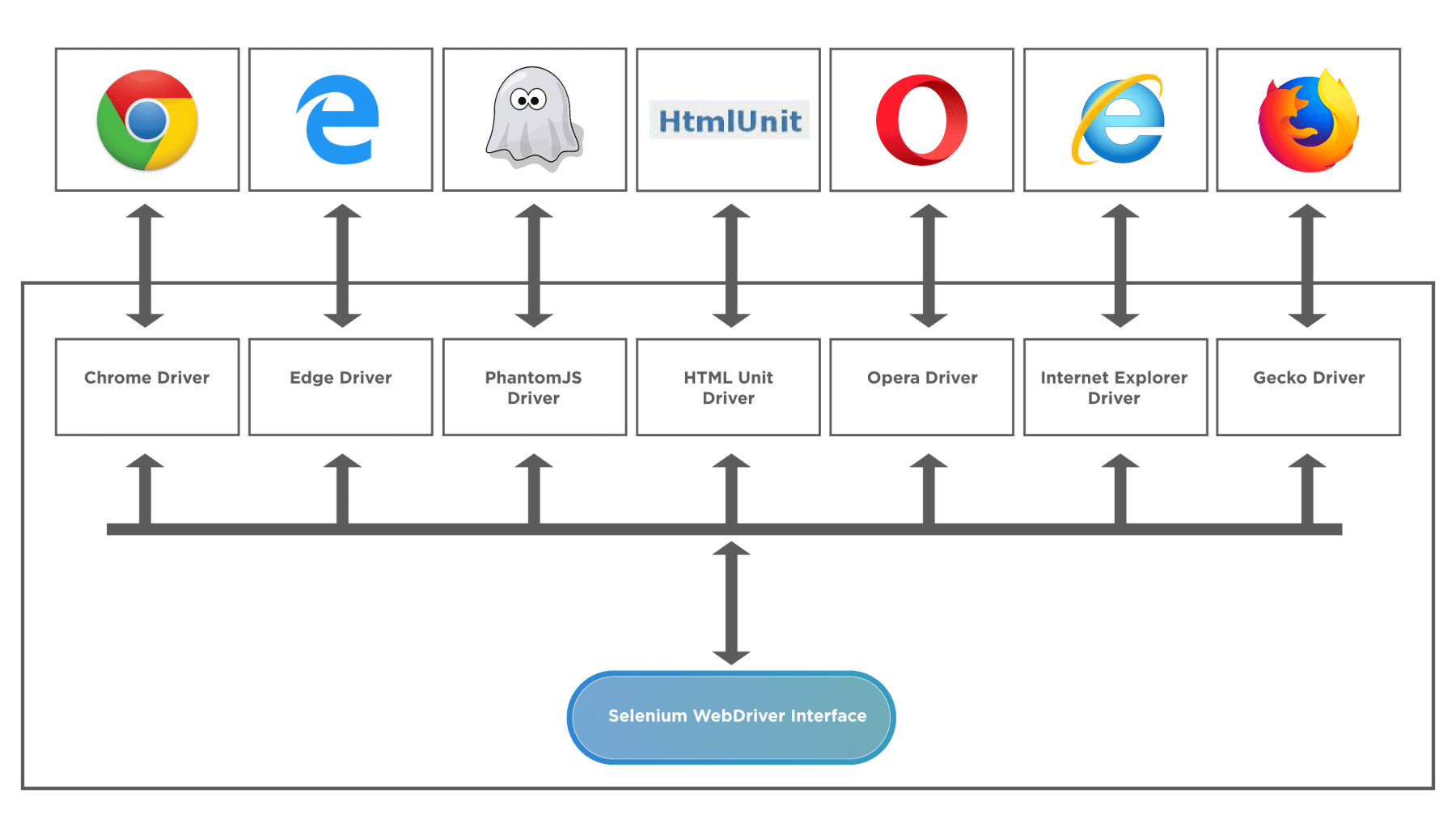
We can perform certain actions on the HTML elements, such as type something using the ***SendKeys*** method, click on the element if it is a button. Eg: if we want to type the name in the identified text box, we can use the following commands:

usernameElement.sendKeys("Ravinder Singh");

1. ***Run tests and record test results using a test framework.***

And, we are done with using the WebDriver to identify and perform the needed actions on the Web Application. Depending on the browser, on which we need to test our application, we can use the corresponding WebDriver.

Here is a list of various browsers and their respective browser drivers:



Recently Microsoft moved their Edge browser on the same platform as Chromium (which is the parent for Chrome), and due to this ChromeDriver can now also support Microsoft Edge Chromium.

## Common Questions

***What is the difference between Selenium and Selenium WebDriver?***

Although ***Selenium*** and ***Selenium WebDriver*** are part of a single ecosystem, they have certain differences. Let's take a look -

* Selenium is not a single tool, it is a framework that consists of various independent tools, like Selenium IDE, Selenium RC, WebDriver, and Selenium Grid. WebDriver, on the other hand, is a single tool, which is a component of Selenium.
* Selenium supports and provides structure and tools for QA's who can write code based automation as well as code-less automation. WebDriver only provides a code-based mechanism of automation.

***Is Selenium WebDriver a framework?***

A general definition of the framework is that it provides a collection of different plugins, libraries, compilers, software programs, API's, etc. that help build a whole software within one platform. Going by this definition, it is a huge component that handles multiple components of the software development lifecycle within itself.

Selenium WebDriver is essentially a set of API's that help a tester or QA achieve simulated user actions on the browser. It does this with finesse but lacks most of the attributes defined above. As such WebDriver isn’t a framework. However, if we talk about Selenium and all its components, then we will be called them a Framework.

***Is Selenium WebDriver a tool?***

There is not a single correct definition of a tool. Most of the time, in software development, people mix tool with the framework. A framework supports a range of functionalities, provides separate options, and is broader in nature. Whereas, a tool has a well-defined scope of functionality, meant to perform specialized tasks.

Talking about WebDriver, it has a well-defined scope of the automation of user actions. For example, clicking, double-clicking, selecting menu items, drop-downs, etc. It can combine with a test runner like TestNG, or pyTest to increase the capabilities.

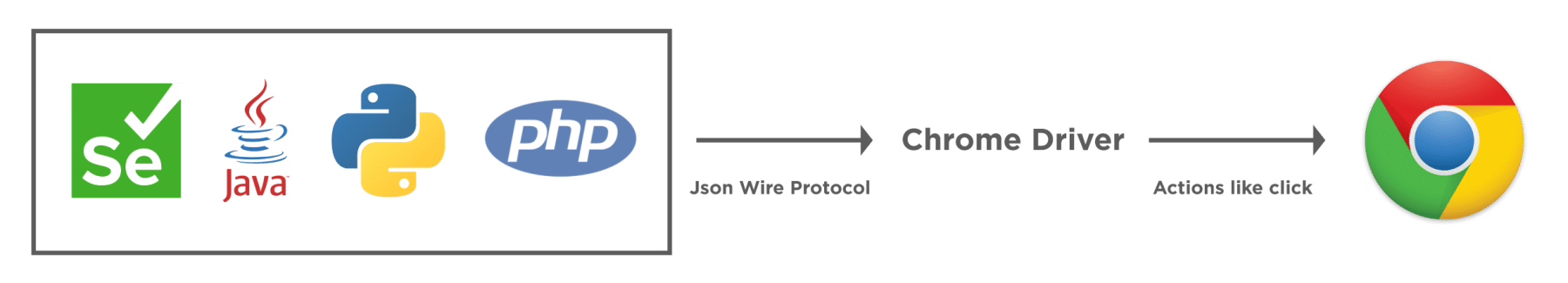
So Selenium WebDriver would roughly classify as a tool, although it is not a standalone tool, rather than a framework. It can combine with various other tools, components, libraries, and test runners to create a scalable test- framework.

## Key Takeaways

* Selenium WebDriver is a component of the Selenium family. It’s the introduction in the system lead to overcome a few of the shortcomings of Selenium RC.
* Selenium WebDriver is a set of APIs, which makes the interaction and actions on the browser very easy and quick.
* Selenium WebDriver provides quite a few unique features, such as it can automate dynamic web pages. Additionally, it can automate all the web applications, no matter in which programming language we use to develop them.
* Selenium WebDriver interacts with the browsers with the help of various drivers provided by corresponding Vendors.

## What is Selenium ChromeDriver?

***ChromeDriver*** is the communication medium that allows users to run their Selenium tests on Chrome browser. It is a standalone server that implements the open-source ***Selenium WebDriver Chromium Protocol***. The Selenium tests interact with the ***ChromeDriver*** using the ***JsonWireProtocol***, which translates the Selenium commands into corresponding actions on the Chrome browser.



The sole purpose of the ChromeDriver is to launch and interact with ***Google Chrome***. Without using ChromeDriver, it's not possible to run [***Selenium***](https://www.toolsqa.com/selenium-webdriver/selenium-testing/) tests on the chrome browser. It is why it is one of the most vital pre-requisite of the test execution on Chrome. A ChromeDriver can be used easily by instantiating the object of the ChromeDriver, assigning it to a WebDriver object, and using that object for browser-based actions.

### *****What are the pre-requisites for Selenium ChromeDriver?*****

Before we can start writing Selenium tests or setup ChromeDriver, there are few pre-requisites that we should have on our system:

1. ***Java JDK***: We require [***JDK***](https://www.infoworld.com/article/3296360/what-is-the-jdk-introduction-to-the-java-development-kit.html) or Java Development Kit for writing java programs. It contains [***JRE***](https://www.infoworld.com/article/3304858/what-is-the-jre-introduction-to-the-java-runtime-environment.html) and other development tools, including compiler and debugger. As we will be writing our selenium tests in java, having JDK is a must. You can get more information about JDK and read its installation guide from here: [***How to install Java?***](https://www.toolsqa.com/selenium-webdriver/install-java/)
2. ***Java IDE***: IDE or Integrated Development Environment helps in writing Java programs. It offers many different features to users to ease their programming requirements. For this tutorial, we will be using Eclipse IDE, although any other Java IDE will be perfectly fine. To more or know how to install Eclipse, visit here: [***Install Eclipse***](https://www.toolsqa.com/selenium-webdriver/download-and-start-eclipse/).
3. ***Selenium WebDriver***: To develop Selenium tests, we need Selenium WebDriver. You can download Selenium WebDriver from the [***official Selenium site***](https://www.selenium.dev/), and you can learn how to configure Selenium in the tutorial; [***Configure Selenium WebDriver***](https://www.toolsqa.com/selenium-webdriver/configure-selenium-webdriver-with-eclipse/). For this tutorial, we will be using ***Selenium 4***.

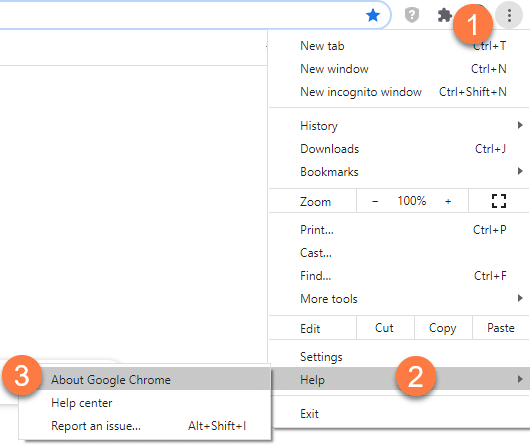
## How to install ChromeDriver on Windows?

Now, as we have learned what ChromeDriver is and why do, we need it for executing Selenium tests on the chrome browser. Let's move a little further and learn how to setup ChromeDriver with your Selenium Java project and execute your tests on Chrome. The first part will be to download the ChromeDriver. Let's see how we can do the same on the Windows platform?

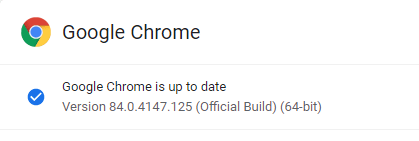
### *****How to download ChromeDriver on Windows?*****

Before we can download the ChromeDriver, we need to check the version of the Chrome browser on your system. ChromeDriver has a direct compatibility dependency on the Chrome browser version, so you need to download the compatible version of ChromeDriver. Follow the steps, as mentioned below, to download a. ChromeDriver  which is compatible with the Chrome browser on your system:

1. Firstly, to check the Chrome browser version on your machine, click on the ***three dots*** on the right top corner of the browser
2. Secondly, click on ***Help*** in the menu.
3. Thirdly, click on ***About Google Chrome*** in the sub-menu.

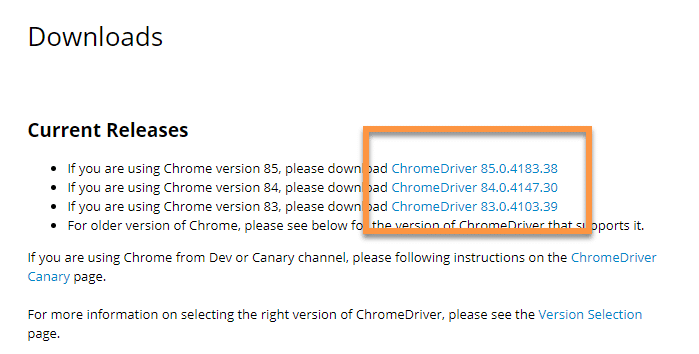


1. After clicking the "***About Google Chrome***" option, the following page will open. Consequently, you will get the Chrome version details as in the image below:

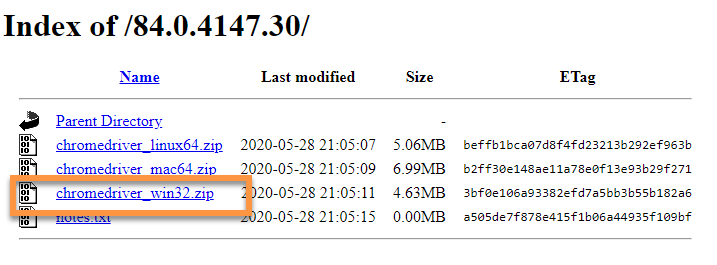


Now, as we have got the Chrome browser version so, we can download the compatible ChromeDriver. Additionally, to download ChromeDriver, navigate to the link of the official ***[ChromeDriver website](https://chromedriver.chromium.org/downloads)***. Follow the steps as mentioned below to download the ChromeDriver executable file:

1. On the ***[ChromeDriver download page](https://chromedriver.chromium.org/downloads)***, there will be links for different ChromeDriver version. Based on your Chrome browser version, download the corresponding ChromeDriver, as marked in the below image. Subsequently, click on the ChromeDriver version that you need to download. As we had the Chrome browser version as "***84*** ", so we will download the corresponding ChromeDriver.



1. Secondly, clicking on the "***ChromeDriver 84.0.4147.30*** " link will take you to the ChromeDriver index page. Here, you will get different options of ChromeDriver based on your operating system. Additionally, for the Windows operating system, you can choose the Win32 version as marked in the below image. Yes, even if you have a 64-bit Windows installed on your system, the Win32 version will work fine.



1. Thirdly, once the download is complete, extract the zip file and place the "***chromedriver.exe***"  at any preferred location on your system.

Now that we have downloaded the ChromeDriver, we will open Eclipse and create a new java project. Moreover, we will add all the selenium dependencies to the project. Additionally, to know more about setting up Selenium with Eclipse, you can visit our previous tutorial on the same at [***Configure Selenium WebDriver***](https://www.toolsqa.com/selenium-webdriver/configure-selenium-webdriver-with-eclipse/).

As a next step, we need to make the downloaded ChromeDriver executable available to the Selenium tests. Subsequently, let's see how we can setup ChromeDriver, so as we can use the same in the Selenium test cases:

### *****How to setup ChromeDriver on Windows?*****

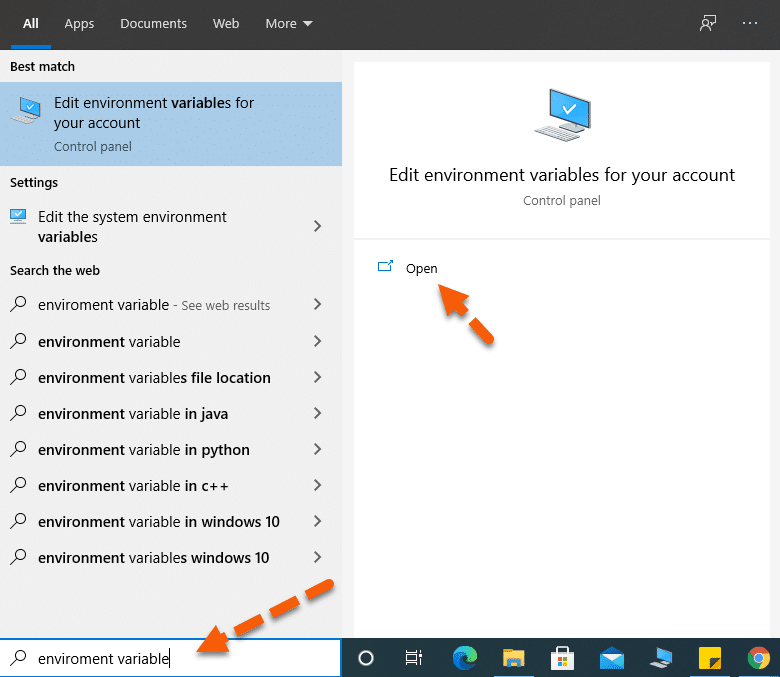
To set up and configure the ChromeDriver with the Selenium, the ChromeDriver executable file should be accessible in the test script. Additionally, Selenium tests can access the ChromeDriver  if any of the following ways sets it up:

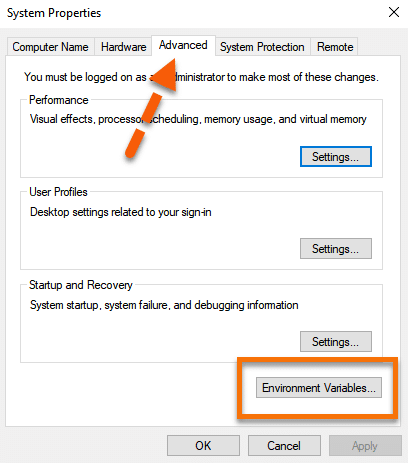
1. Setup ChromeDriver using System Properties in Environment Variables.
2. Setup ChromeDriver using System Properties in the test script.

Let’s understand all of these and try running our test code with ***Selenium 3*** or ***Selenium 4***.

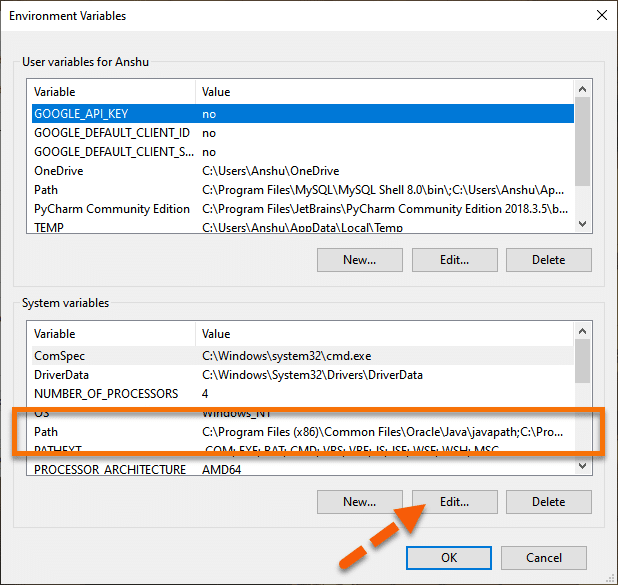
#### **How to setup ChromeDriver using System Properties in Environment Variables?**

On the Windows operating system, one of the ways to declare system-level variables is by using ***Environment Variables***. Users can define either user-level environment variables or system variables. Moreover, the variable defined here is accessible to all the programs running on the system. We can use the environment variables to set the path of the ChromeDriver. So, whenever we create an instance of the WebDriver, it will automatically detect the path of the ChromeDriver from the system variables and can use the same. Subsequently, let's have a look at steps through which we can do that.

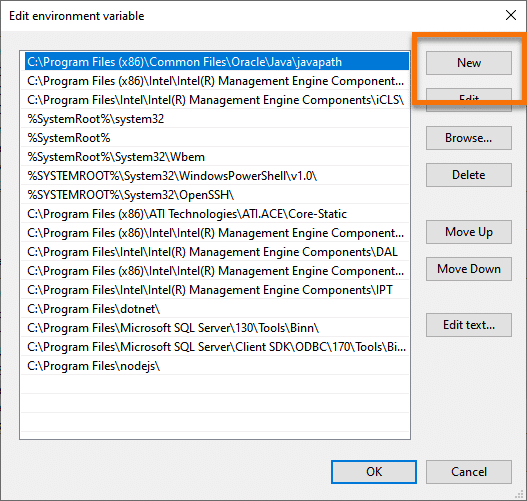
1. First, we need to open the Environment Variable pop-up. To do that, click on the search bar and search for "***Environment Variables***". It will search and display "***Edit environment variables for your account***", as shown in the image below. After that, click on the "***Open***" to open the System Properties pop-up.
2. Secondly, the "***System Properties*** " pop-up will open. In the pop-up, select the "***Advanced*** " tab as marked by the arrow. After that, in the Advanced tab, click on the "***Environment Variables*** " button.



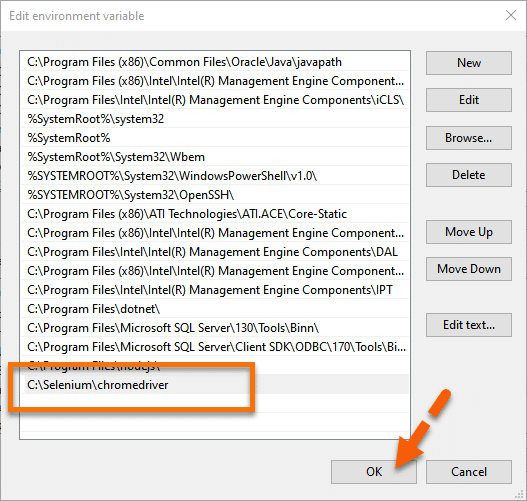
1. Thirdly, this will open the "***Environment Variables*** " pop-up. In the pop-up System variables section, look for the "***path*** " variable marked in the below image. After that, click on the path variable to select it. Once selected, click on the "***Edit*** " button as marked by the arrow.



1. Fourthly, once the "***Edit environment variable*** " pops-up, click on the "***New*** " button.



1. Fifthly, add the ChromeDriver's folder location to the path. We have placed our driver at the following location "***C:\Selenium\chromedriver***", so we have added the same as the path variable. Once done, click on the "***OK*** " button as denoted by the arrow.



#### **How to run Selenium tests on Chrome Browser using ChromeDriver?**

Conclusively, we can now directly initialize the WebDriver  instance using the ChromeDriver, as shown below:

package demoPackage;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class ChromeDriverDemo {

public static void main(String[] args) throws InterruptedException{

System.out.println("Execution after setting ChromeDriver path in System Variables");

WebDriver driver=new ChromeDriver();

driver.get("https://demoqa.com");

Thread.sleep(3000);

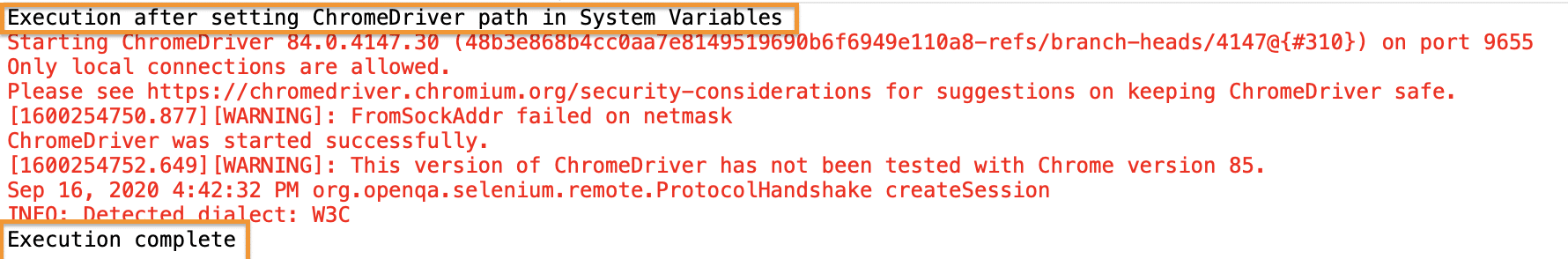
driver.quit();

System.out.println("Execution complete");

}

}

On executing the above code, you will see the results below.



Evidently from the console results, there is no WebDriver error, which implies that the WebDriver set up is correct. Moreover, you can see the print statements as the entry and exit points of our execution. Correspondingly you will be able to view the execution in your system.

#### **How to initialize ChromeDriver using System Properties in the Selenium test script?**

Instead of using the global instance of ChromeDriver, if we want to use a specific version of ChromeDriver, we can do the same by explicitly specifying the path of the ChromeDriver in the test script itself. In other words, we need to add a single line of code to set up the system properties for the ChromeDriver, as shown below:

System.setProperty("webdriver.chrome.driver", "<Path of the ChromeDriver Executable>");

Conclusively, let us modify the code we used above and see that we can launch the Chrome browser successfully. The modified code would look like this:

package demoPackage;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class ChromeDriverDemo {

public static void main(String[] args) throws InterruptedException{

System.out.println("Execution after setting ChromeDriver path in System setProperty method");

System.setProperty("webdriver.chrome.driver", "E:\\drivers\\ChromeDrivers\\85\\chromedriver.exe");

WebDriver driver=new ChromeDriver();

driver.get("https://demoqa.com");

Thread.sleep(3000);

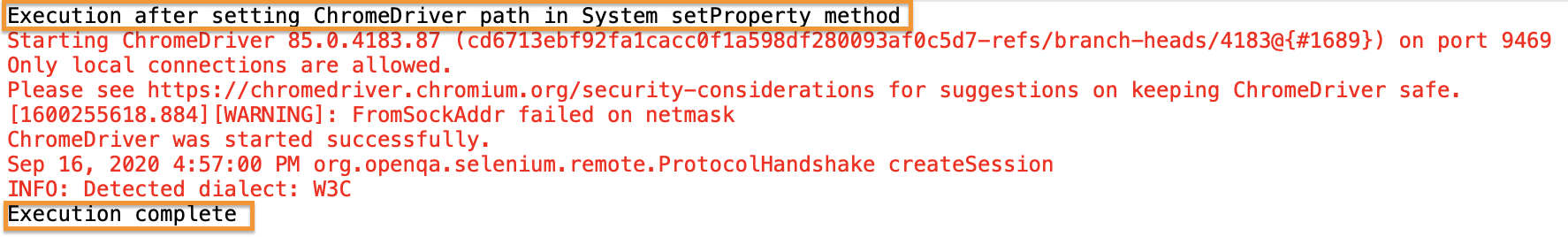
driver.quit();

System.out.println("Execution complete");

}

}

You will see that [***demoqa.com***](https://demoqa.com/)  opens in the Chrome browser without any error and exception.



If you noticed, here we used ChromeDriver version 85, instead of the default global ChromerDriver version of 84. The execution logs indicate that our WebDriver session started with the print statement displayed right at the beginning. The lines highlighted in red are some browser logs corresponding to the browser session. Moreover, you can see the browser opening up in your system, and after the website opens, the browser session is closed.

## How to install ChromeDriver on macOS?

The installation and setup of ChromeDriver on macOS is almost the same as that of the Windows platform. The only difference being the executable for macOS will be different, and the way we can include the ChromeDriver executable in the System’s PATH  variable is a bit different. Let’s see how we can install and setup the ChromeDriver on macOS :

### *****How To Download ChromeDriver on macOS?*****

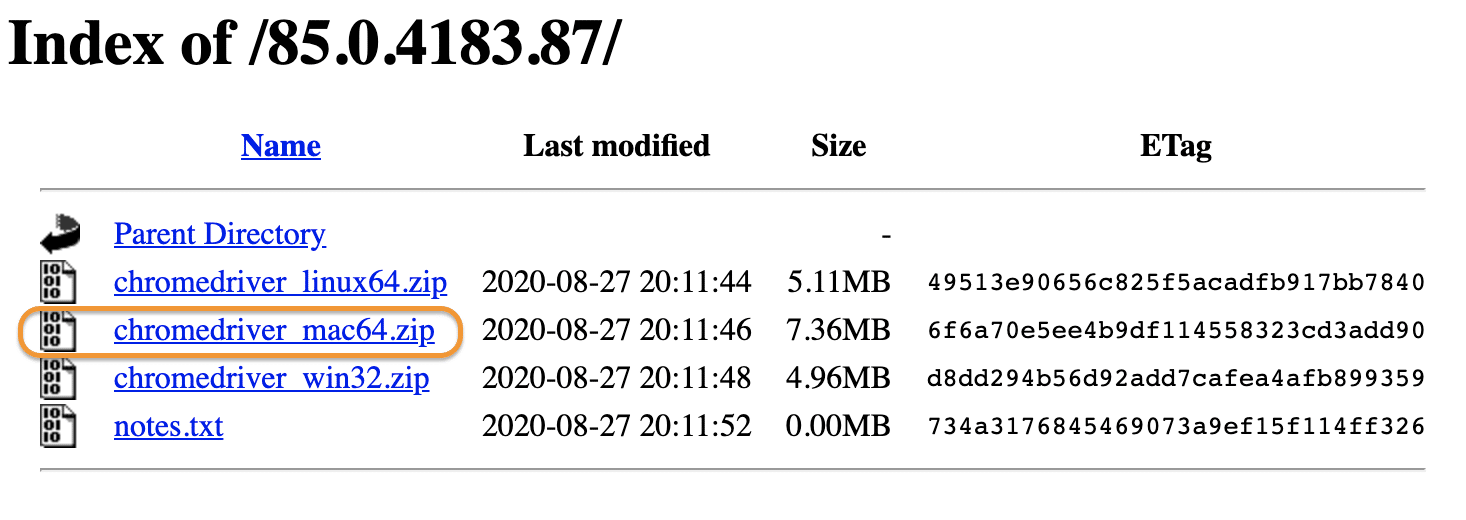
On macOS, we can download ChromeDriver using any of the following ways:

* Download executable file from the [***Chromium website***](https://chromedriver.chromium.org/)
* Download using any package manager, such as [***Homebrew***](https://brew.sh/).

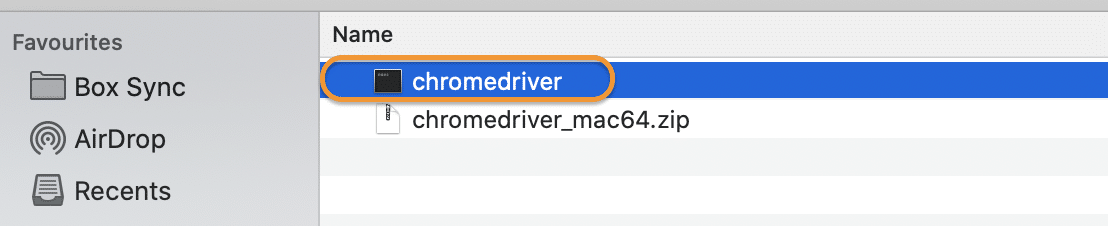
Let's understand the details of both of these ways of downloading and setting up ChromeDriver  on macOS :

#### **How to download ChromeDriver for macOS from the Chromium website?**

You can download the ChromerDrive for macOS, same as we did for the Windows platform, except for the difference that, now select the binary for the macOS platform as shown below:



It will download a zip file, which you can extract in any of the folders of your choice. After extracting, it will show the executable file of ChromeDriver, as shown below:



So, now you have the ChromeDriver executable file available on your machine, which we can use in our test scripts. Subsequently, let's see how to setup ChromeDriver  on macOS and use in the Selenium test scripts:

### *****How To Set Up ChromeDriver on macOS?*****

Now that you have downloaded the ChromeDriver, the next step is to set it up to use it in your test scripts. On macOS  also, we can follow the same ways, as on Windows, to set up the ChromeDriver:

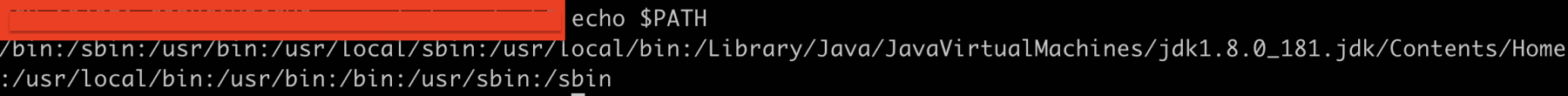
1. Setup ChromeDriver using the System’s PATH variable.
2. Setup ChromeDriver using System Properties in the test script.

The 2nd point is the same setup as the Windows platform, as we are using JAVA for test development, and JAVA being platform-independent, will have the same behavior across platforms. So, let's see how we can set up the ChromerDriver  using the System's PATH variable:

#### **How to Setup ChromeDriver using the System’s PATH variable?**

As we mentioned above, one of the easiest ways to make the executable available globally on the macOS is to copy the executable under any the folders which are already in the PATH variable. Let’s follow the steps mentioned below to achieve the same:

First, identify the folders included in the PATH variable using the command ‘echo $PATH ‘ on the terminal. It will give a sample output, as shown below:



1. Secondly, as we can see, multiple directories are already part of the PATH variable. Suppose we choose “***/usr/local/bin***” as a placeholder directory to hold the ChromeDriver executable.
2. Thirdly, copy the ChromeDriver  executable file from the downloaded directory to the “***/usr/local/bin***” directory using the mv command as shown below:

mv chromedriver /usr/local/bin/

Now your ChromeDriver is ready to be used in your Selenium test scripts. Consequently, now we will write a simple program and execute the same in the macOS platform.

package demoPackage;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class ChromeDriverDemo {

public static void main(String[] args) throws InterruptedException{

System.out.println("ChromeDriver execution on mac!!");

WebDriver driver=new ChromeDriver();

driver.get("https://demoqa.com");

Thread.sleep(3000);

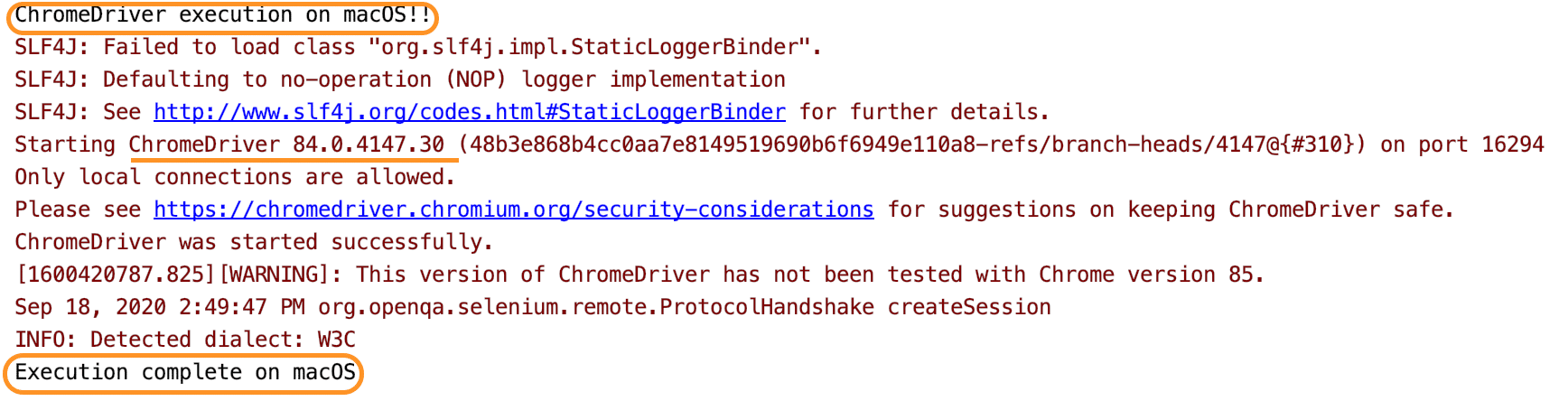
driver.quit();

System.out.println("Execution complete on macOS");

}

}

On executing the same, you can find the results in your console window:



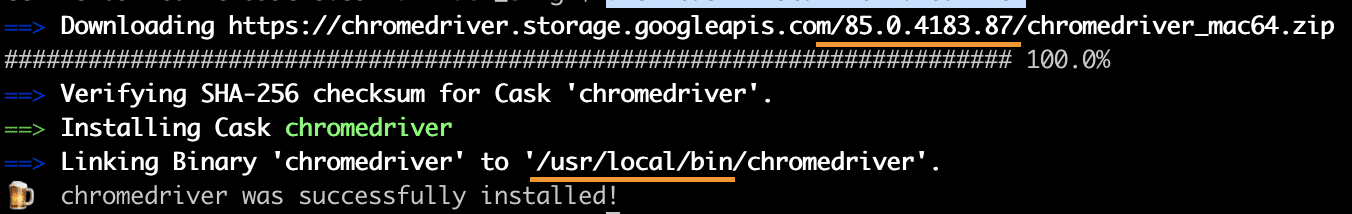
You can see the execution happening successfully without any error. Both the print statements are getting displayed, which indicates that our execution did not face any error. So did you see how easy it was to run\* ChromeDriver\* tests in macOS? Unlike the Windows system, where you have to remember the path of your driver executable, just placing the driver at a location in macOS makes our lives so easy!

### *****How to install ChromeDriver using Homebrew?*****

***Homebrew*** is one of the package managers available on macOS, which can download any binaries which register as a package with Homebrew. Luckily ChromeDriver is available as a Homebrew package, and we can download and set up the same, with a straightforward command as below:

brew cask install chromedriver

When we run the above command, it will download and install the ChromeDriver in the "***/usr/local/bin*** " directory, as can be validated from the following output of the above command:



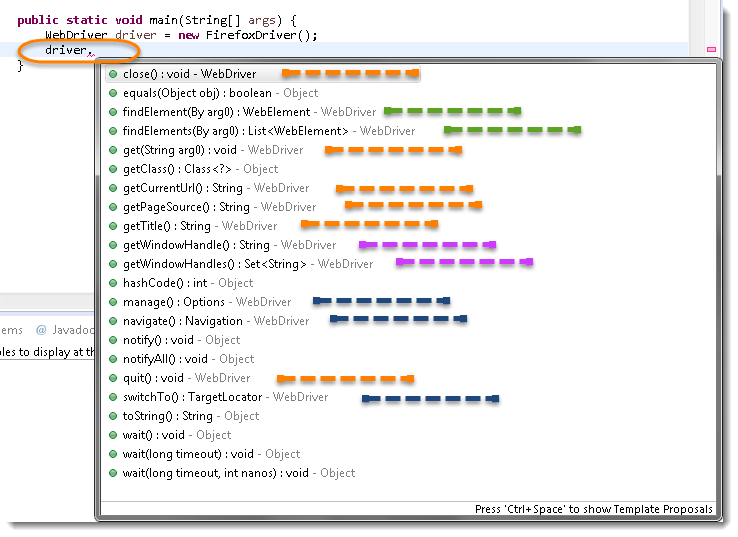
As we can see that the latest stable version of ChromeDriver was downloaded and installed in the "***/usr/local/bin***" directory, which makes it part of the PATH variable and accessible to all the applications on the system.

So, this way, ChromeDriver installed and setup on macOS with a single command. Other package managers, such as NPM, also provides the capabilities to install ChromeDriver, which can be explored based on which package manager you are using on your machine.

The very first question which comes to my mind and has been asked in many interviews is ***What is Selenium WebDriver?*** Is it an Automation Tool? Is it a Class? Is it an Interface or what actually it is? To answer this question we need to understand the ***Advance Java OOPs concepts*** first and then we would be able to visualize the ***WebDriver Implementation***. For the sake of simplicity, we will avoid this WebDriver Implementation topic for now and will cover this in later chapters. As of now, we start with all the methods we get from WebDriver.

## Selenium Webdriver Browser Commands

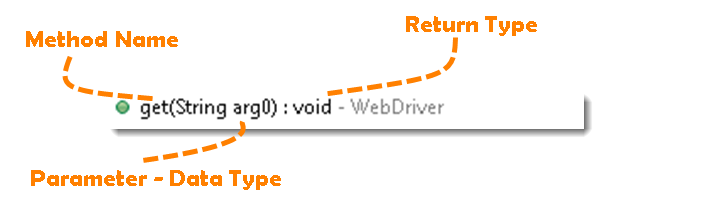
Now the next question is, How to access the methods of WebDriver? To check what all we have in WebDriver, create a driver object from WebDriver and press ***dot key.*** This will list down all the methods of WebDriver.



***Note:*** Methods followed by ***Object*** keyword are the generic methods gets from Object Class in Java. You will find these methods for every object of java language.

* The suggestions marked in ***Orange Color*** are the methods directly under WebDriver and will be covering these in this chapter only.
* The suggestions marked in ***Blue Color*** are Nested Classes under WebDriver and will be covered in detail separately in the following chapters.
* The suggestions marked in ***Green Color*** are also Interfaces like WebDriver and will be covered in detail separately in the following chapters.
* The suggestions marked in ***Violet Color*** are similar methods like ***Orange*** but will be covered in detail separately in the following chapters.

Let's just start discussing the ***Orange colored*** methods of ***Selenium WebDriver*** but before that try to understand the syntax of the displayed suggestion by Eclipse for WebDriver.



**Method**: A Java method is a collection of statements that are grouped together to perform an operation.

* ***Method Name:*** To access any method of any class, we need to create an object of a class and then all the public methods will appear for the object.
* ***Parameter***: It is an argument that is passed to a method as a parameter to perform some operation. Every argument must be passed with the same data type. For e.g. ***get(String arg0)*** : void. This is asking for a ***String type*** argument.
* ***Return Type***: Method can return a value or returning nothing (***void***). If the void is mentioned after the method, it means the method is returning no value. And if it is returning any value, then it must display the type of the value for e.g. getTitle() : String.

Now it would be very easy to understand the WebDriver commands in the below chapter. The very first thing you like to do with Selenium is to ***Opening*** a new browser, ***Perform*** a few tasks and ***Closing*** the browser. Below are the numbers of commands you can apply on the Selenium opened browser.

### *****Get-Command - How to Open a WebPage in Selenium?*****

***get(String arg0)***: ***void*** - This method ***Load*** a new web page in the current browser window. Accepts String as a parameter and returns nothing.

***Command - driver.get(appUrl);***

Where ***appUrl*** is the website address to load. It is best to use a fully qualified URL.

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

//Or can be written as

String URL = "https://www.DemoQA.com";

driver.get(URL);

### *****Get Title Command - How to get the Title of the Webpage in Selenium?*****

***getTitle(): String*** - This method fetches the ***Title*** of the current page. Accepts nothing as a parameter and returns a String value.

***Command - driver.getTitle();***

As the return type is a String value, the output must be stored in String object/variable.

driver.getTitle();

//Or can be used as

String Title = driver.getTitle();

### *****Get Current URL Command - How to read the URL of the Webpage in Selenium?*****

***getCurrentUrl(): String*** - This method fetches the string representing the ***Current URL*** which is opened in the browser. Accepts nothing as a parameter and returns a String value.

***Command - driver.getCurrentUrl();***

As the return type is String value, the output must be stored in String object/variable.

driver.getCurrentUrl();

//Or can be written as

String CurrentUrl = driver.getCurrentUrl();

### *****Get Page Source Command - How to read the page source of the WebPage in Selenium?*****

***getPageSource(): String*** - This method returns the ***Source Code*** of the page. Accepts nothing as a parameter and returns a String value.

***Command - driver.getPageSource();***

As the return type is String value, the output must be stored in String object/variable.

driver.getPageSource();

//Or can be written as

String PageSource = driver.getPageSource();

### *****Close Command - How to close the Browser in Selenium?*****

***close(): void*** - This method Close only the current window the WebDriver is currently controlling. Accepts nothing as a parameter and returns nothing.

***Command - driver.close();***

Quit the browser if it's the last window currently open.

driver.close();

### *****Quit Command - How to close all the Browser's Window in Selenium?*****

***quit(): void*** - This method ***Closes*** all windows opened by the WebDriver. Accepts nothing as a parameter and returns nothing.

***Command - driver.quit();***

Close every associated window.

driver.quit();

***Note***: Important to note that this command will only close the browser's window opened by the selenium in the same session. If any browser is opened manually, this will have no impact on the same. Also, there is no impact on the browsers opened in another run or session even by the Selenium. The same is with close() method as well.

## Selenium WebDriver Browser Command - Practice Exercises

### *****Practice Exercise - 1*****

1. Launch a new Chrome browser.
2. Open Shop.DemoQA.com
3. Get Page Title name and Title length
4. Print Page Title and Title length on the Eclipse Console.
5. Get Page URL and verify if it is a correct page opened
6. Get Page Source (HTML Source code) and Page Source length
7. Print Page Length on Eclipse Console.
8. Close the Browser.

#### **Solution**

package automationFramework;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class WebDriverCommands {

public static void main(String[] args) {

String driverExecutablePath = "D:\\Drivers\\chromedriver.exe";

System.setProperty("webdriver.chrome.driver", driverExecutablePath);

// Create a new instance of the FireFox driver

WebDriver driver = new ChromeDriver();

// Storing the Application Url in the String variable

String url = "https://www.shop.demoqa.com";

//Launch the ToolsQA WebSite

driver.get(url);

// Storing Title name in the String variable

String title = driver.getTitle();

// Storing Title length in the Int variable

int titleLength = driver.getTitle().length();

// Printing Title & Title length in the Console window

System.out.println("Title of the page is : " + title);

System.out.println("Length of the title is : "+ titleLength);

// Storing URL in String variable

String actualUrl = driver.getCurrentUrl();

if (actualUrl.equals(url)){

System.out.println("Verification Successful - The correct Url is opened.");

}

else {

System.out.println("Verification Failed - An incorrect Url is opened.");

//In case of Fail, you like to print the actual and expected URL for the record purpose

System.out.println("Actual URL is : " + actualUrl);

System.out.println("Expected URL is : " + url);

}

// Storing Page Source in String variable

String pageSource = driver.getPageSource();

// Storing Page Source length in Int variable

int pageSourceLength = pageSource.length();

// Printing length of the Page Source on console

System.out.println("Total length of the Pgae Source is : " + pageSourceLength);

//Closing browser

driver.close();

}

}

***Output***

Title of the page is : ToolsQA Demo Site – ToolsQA – Demo E-Commerce Site Length of the title is : 50 Verification Failed - An incorrect Url is opened. Actual URL is : http://shop.demoqa.com/ Expected URL is : http://www.shop.demoqa.com Total length of the Pgae Source is : 88952

### *****Practice Exercise - 2*****

1. Launch a new Chrome browser.
2. Open ToolsQA Practice Automation Page for Switch Windows: [***https://demoqa.com/browser-windows/***](https://demoqa.com/browser-windows/)
3. Use this statement to click on a New Browser Window button "***driver.findElement(By.id("New Browser Window")).click();***"
4. Close the browser using close() command

You will notice that only one window will close. Next time use quit() command instead of close(). At that time selenium will close both the windows.

package automationFramework;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class WebDriverCommands\_2 {

public static void main(String[] args) {

String driverExecutablePath = "D:\\Drivers\\chromedriver.exe";

System.setProperty("webdriver.chrome.driver", driverExecutablePath);

// Create a new instance of the FireFox driver

WebDriver driver = new ChromeDriver();

// Storing the Application Url in the String variable

String url = "https://demoqa.com/browser-windows/";

//Launch the ToolsQA WebSite

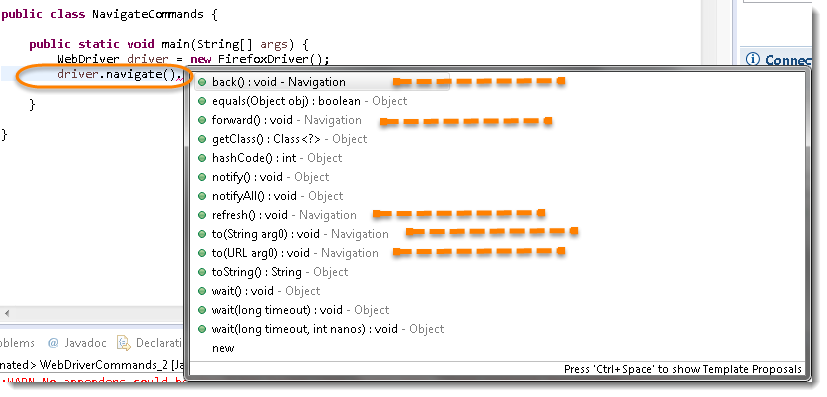
driver.get(url);

}

After successfully running our [***first test case on Firefox Browser***](https://www.toolsqa.com/selenium-webdriver/run-selenium-test/) now we are stepping towards grasping the essential ***Selenium Navigation Commands.*** Thus we are going to discuss about various Selenium Navigation Commands that we would be using in our day to day automation testing. The navigate interface exposes the ability to move backwards and forwards in the browser’s history.

## Selenium Webdriver - Browser Navigation Command

To access the navigation's method, just type ***driver.navigate().*** The IntelliSense feature of the eclipse will automatically display all the public methods of ***Navigate Interface*** shown in the below image.



***Note:*** Only methods that are followed by ***Navigation*** keyword are belongs to navigate. Rest followed by ***Object*** keyword are the generic methods gets from Object Class in Java. You will find these methods for every object of java language.

### *****Navigate To Command - How to Navigate to URL or How to open a webpage in Selenium Browser?*****

***to(String arg0) : void*** - This method ***Loads*** a new web page in the current browser window. It accepts a String parameter and returns nothing.

***Command - driver.navigate().to(appUrl);***

It does exactly the same thing as the ***driver.get(appUrl)*** method. Where ***appUrl*** is the website address to load. It is best to use a fully qualified URL.

driver.navigate().to("https://www.DemoQA.com");

### *****Forward Command - How to browser Forward in Selenium Browser?*****

***forward() : void*** - This method does the same operation as clicking on the ***Forward Button*** of any browser. It neither accepts nor returns anything.

***Command - driver.navigate().forward();***

Takes you forward by one page on the browser's history.

driver.navigate().forward();

### *****Back Command - How to browse backward in Selenium Browser?*****

***back() : void*** - This method does the same operation as clicking on the ***Back Button*** of any browser. It neither accepts nor returns anything.

***Command - driver.navigate().back();***

Takes youback by one page on the browser's history.

driver.navigate().back();

### *****Refresh Command - How to Refresh Selenium Browser?*****

***refresh() : void*** - This method ***Refresh*** the current page. It neither accepts nor returns anything.

***Command - driver.navigate().refresh();***

Perform the same function as pressing F5 in the browser.

driver.navigate().refresh();

## Practice Exercises for Selenium Navigation Commands

### *****Practice Exercise*****

1. Launch new Browser
2. Open DemoQA.com website
3. Click on Registration link using "driver.findElement(By.xpath(".//[@id='menu-item-374']/a")).click();"\*
4. Come back to Home page (Use 'Back' command)
5. Again go back to Registration page (This time use 'Forward' command)
6. Again come back to Home page (This time use 'To' command)
7. Refresh the Browser (Use 'Refresh' command)
8. Close the Browser

### *****Solution*****

package automationFramework;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

public class NavigateCommands {

public static void main(String[] args) {

// Create a new instance of the FireFox driver

WebDriver driver = new FirefoxDriver();

// Open ToolsQA web site

String appUrl = "https://www.DemoQA.com";

driver.get(appUrl);

// Click on Registration link

driver.findElement(By.xpath(".//\*[@id='menu-item-374']/a")).click();

// Go back to Home Page

driver.navigate().back();

// Go forward to Registration page

driver.navigate().forward();

// Go back to Home page

driver.navigate().to(appUrl);

// Refresh browser

driver.navigate().refresh();

// Close browser

driver.close();

}

}

WebElement represents an ***HTML element***. HTML documents are made up by HTML elements. HTML elements are written with a ***start*** tag, with an ***end*** tag, with the ***content*** in between: ***<tagname> content </tagname>***

The HTML element is everything from the start tag to the end tag: <p> ***My first HTML paragraph***. </p>

HTML elements can be nested (elements can contain elements). All HTML documents consist of nested HTML elements.

<html>

<body>

<h1> My First Heading </h1>

<p> My first paragraph. </p>

</body>

</html>

## List of WebElement Commands/Actions

All interesting operations to do with interacting with a page will be performed through this ***WebElement Interface***.



***Note***: Methods followed by ***Object*** keyword are the generic methods gets from Object Class in Java. You will find these methods for every object of java language.

Before going through each and every action of WebElement, let's just understand that how we get a WebElement object/element. As in the previous chapters, we learned that every method of the ***WebDriver*** either returns something or return void(means return nothing). The same way ***findElement*** command of ***WebDriver*** returns ***WebElement***.



So, to get the WebElement object write the below statement:

***WebElement element = driver.findElement(By.id("UserName"));***

And now if you type ***element dot***, Eclipse's intellisence will populate the complete list of actions just like the above image.

One more thing to notice that WebElement can be of any type, like it can be a ***Text, Link, Radio Button, Drop Down, WebTable*** or any HTML element. But all the actions will always populate against any element irrespective of whether the action is valid on the WebElement or not. For e.g. ***clear() command,*** even if you have a link element still you get the option to choose clear() command on it, which if you choose may result in some error or may not does anything.

## Clear Command

***clear( ) : void*** - If this element is a text entry element, this will clear the value. This method accepts nothing as a parameter and returns nothing.

***Command - element.clear();***

This method has no effect on other elements. Text entry elements are ***INPUT*** and ***TEXTAREA*** elements.

WebElement element = driver.findElement(By.id("UserName"));

element.clear();

//Or can be written as

driver.findElement(By.id("UserName")).clear();

## SendKeys Command

***sendKeys(CharSequence... keysToSend ) : void*** - This simulates typing into an element, which may set its value. This method accepts CharSequence as a parameter and returns nothing.

***Command - element.sendKeys("text");***

This method works fine with text entry elements like ***INPUT*** and ***TEXTAREA*** elements.

WebElement element = driver.findElement(By.id("UserName"));

element.sendKeys("ToolsQA");

//Or can be written as

driver.findElement(By.id("UserName")).sendKeys("ToolsQA");

## Click Command

\*\*\*click( ) : void \*\*\*- This simulates the clicking of any element. Accepts nothing as a parameter and returns nothing.

***Command - element.click();***

Clicking is perhaps the most common way of interacting with web elements like text elements, links, radio boxes and many more.

WebElement element = driver.findElement(By.linkText("ToolsQA"));

element.click();

//Or can be written as

driver.findElement(By.linkText("ToolsQA")).click();

***Note***: Most of the time we click on the links and it causes a new page to load, this method will attempt to wait until the page has loaded properly before handing over the execution to next statement. But If click() causes a new page to be loaded via an event or is done by sending a native event for example through javascript, then the method will not wait for it to be loaded.

There are some preconditions for an element to be clicked. The element must be Visible and it must have a ***Height and Width*** greater than 0.

## IsDisplayed Command

***isDisplayed( ) : boolean*** - This method determines if an element is currently being displayed or not. This accepts nothing as a parameter but returns a boolean value(true/false).

***Command - element.isDisplayed();***

WebElement element = driver.findElement(By.id("UserName"));

boolean status = element.isDisplayed();

//Or can be written as

boolean staus = driver.findElement(By.id("UserName")).isDisplayed();

***Note***: Do not confuse this method with elements present on the page or not. This will return ***true*** if the element is present on the page and throw a ***NoSuchElementFound*** exception if the element is not present on the page. This refers to the property of the element, sometimes the element is present on the page but the property of the element is set to ***hidden***, in that case, this will return ***false***, as the element is present in the DOM but not visible to us.

## IsEnabled Command

***isEnabled( ) : boolean***- This determines if the element currently is ***Enabled or not?*** This accepts nothing as a parameter but returns boolean value(true/false).

***Command - element.isEnabled();***

This will generally return true for everything but I am sure you must have noticed many disabled input elements in the web pages.

WebElement element = driver.findElement(By.id("UserName"));

boolean status = element.isEnabled();

//Or can be written as

boolean staus = driver.findElement(By.id("UserName")).isEnabled();

//Or can be used as

WebElement element = driver.findElement(By.id("userName"));

boolean status = element.isEnabled();

// Check that if the Text field is enabled, if yes enter value

if(status){

element.sendKeys("ToolsQA");

}

## IsSelected Command

***isSelected( ) : boolean*** - Determine whether or not this element is selected or not. This accepts nothing as a parameter but returns boolean value(true/false).

***Command - element.isSelected();***

This operation only applies to input elements such as ***Checkboxes, Select Options***, and ***Radio Buttons***. This returns ***True*** if the element is currently selected or checked, ***false*** otherwise.

WebElement element = driver.findElement(By.id("Sex-Male"));

boolean status = element.isSelected();

//Or can be written as

boolean staus = driver.findElement(By.id("Sex-Male")).isSelected();

***Note***: In the later chapters of [***Check Box & Radio Buttons***](https://www.toolsqa.com/selenium-webdriver/selenium-checkbox/) and [***Drop Down & Multiple Selects***](https://www.toolsqa.com/selenium-webdriver/dropdown-in-selenium/), we have covered many examples around it.

## Submit Command

\*\*\*submit( ) : void- \*\*\*This method works well/better than the click() if the current element is a form, or an element within a form. This accepts nothing as a parameter and returns nothing.

***Command - element.submit();***

If this causes the current page to change, then this method will wait until the new page is loaded.

WebElement element = driver.findElement(By.id("SubmitButton"));

element.submit();

//Or can be written as

driver.findElement(By.id("SubmitButton")).submit();

## GetText Command

***getText( ) : String-*** This method will fetch the visible (i.e. not hidden by CSS) innerText of the element. This accepts nothing as a parameter but returns a String value.

***Command - element.getText();***

This returns an innerText of the element, including sub-elements, without any leading or trailing whitespace.

WebElement element = driver.findElement(By.xpath("anyLink"));

String linkText = element.getText();

## getTagName Command

***getTagName( ) : String***- This method gets the tag name of this element. This accepts nothing as a parameter and returns a String value.

***Command - element.getTagName();***

This does not return the value of the name attribute but return the tag for e.g. "***input***" for the element <input name="foo"/>.

WebElement element = driver.findElement(By.id("SubmitButton"));

String tagName = element.getTagName();

//Or can be written as

String tagName = driver.findElement(By.id("SubmitButton")).getTagName();

## getCssValue Command

***getCssvalue( ) : String***- This method Fetch CSS property value of the give element. This accepts nothing as a parameter and returns a String value.

***Command - element.getCssValue();***

Color values should be returned as rgba strings, so, for example if the "***background-color***" property is set as "***green***" in the HTML source, the returned value will be "***rgba(0, 255, 0, 1)***".

## getAttribute Command

***getAttribute***(String Name) : ***String***- This method gets the value of the given attribute of the element. This accepts the String as a parameter and returns a String value.

***Command - element.getAttribute();***

Attributes are Ids, Name, Class extra and using this method you can get the value of the attributes of any given element.

WebElement element = driver.findElement(By.id("SubmitButton"));

String attValue = element.getAttribute("id"); //This will return "SubmitButton"

## getSize Command

***getSize( ) : Dimension*** - This method fetch the width and height of the rendered element. This accepts nothing as a parameter but returns the Dimension object.

***Command - element.getSize();***

This returns the size of the element on the page.

WebElement element = driver.findElement(By.id("SubmitButton"));

Dimension dimensions = element.getSize();

System.out.println(“Height :” + dimensions.height + ”Width : "+ dimensions.width);

## getLocation Command

***getLocation( ) : Point*** - This method locate the location of the element on the page. This accepts nothing as a parameter but returns the Point object.

***Command - element.getLocation();***

This returns the ***Point object,*** from which we can get X and Y coordinates of specific element.

WebElement element = driver.findElement(By.id("SubmitButton"));

Point point = element.getLocation();

System.out.println("X cordinate : " + point.x + "Y cordinate: " + point.y);

We are aware that a webpage consists of numerous WebElements such as text boxes, buttons, lists, etc. We can perform a variety of actions on these WebElements using [***Selenium***](https://www.selenium.dev/documentation/en/) commands like search elements, associate events with web elements, etc. To perform these actions we first need to interact with a web page so that we can use WebElement Commands/actions. In this topic, we will discuss the different methods used to find an element on the webpage using Selenium so that we can perform actions on these elements. We will cover the following topics in this article.

* Find elements using Selenium WebDriver?
* Why do we need to find a web element in Selenium?
* How to find elements in Selenium?
* What is By class in Selenium?
* Difference between find Element and find Elements in Selenium.

## Find elements using Selenium WebDriver?

As mentioned above to interact with WebElements, we first have to find or locate these elements on the webpage. We can find elements on a web page by specifying the attributes such as ***Id*** of the element or ***class name*** of the element and such other parameters. These alternatives using which we can find elements on a webpage are called [***locator strategies.***](https://www.toolsqa.com/selenium-webdriver/selenium-locators/)

The following are the locator strategies we can use while locating the elements.

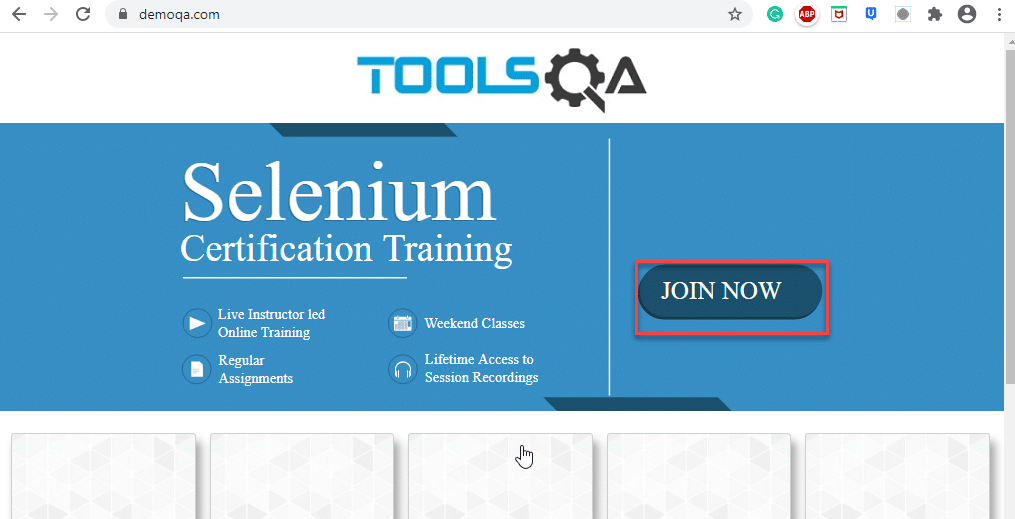
| ***Locator*** | ***Description*** |
| --- | --- |
| ***id*** | finds elements by ID attribute. The search value given should match the ID attribute. |
| ***name*** | Finds or Locates elements based on the NAME attribute. The name attribute is used to match the search value. |
| ***class name*** | Finds elements that match the class name specified. Note that compound classes are not allowed as strategy names. |
| ***tag name*** | Finds or Locates elements having tag names that match the search value. |
| ***CSS selector*** | Matches CSS selector to find the element. |
| ***XPath*** | Matches XPath expression to the search value and based on that the element is located. |
| ***link text*** | Here the visible text whose anchor elements are to be found is matched with the search value. |
| ***partial link text*** | Here also we match the visible text with the search value and find the anchor value. If we are matching multiple elements, only the first entry will be selected. |

Now before moving to how we can use these various types of locators to locate the elements, let's first understand why exactly there is a need to find the elements in Selenium?

### *****Why do we need to find an element in Selenium?*****

We know that we use Selenium mostly for UI testing of a web-based application. Since we need to perform automatic feature interaction with the web page, we need to locate web elements so that we can trigger some JavaScript events on web elements like click, select, enter, etc. or add/ update values in the text fields. To perform these activities it is important to first locate the element on the web page and then perform all these actions.

For example, suppose given a web page [***"demoqa.com"***](https://demoqa.com/) as shown below.



Now, let us say we need to perform some actions on the ***"JOIN NOW "*** button. So before implementing the code for the say click event on this button, we will have to first find this element on the web page. So, how we are going to find the element so that we can carry on with our actions?

We will use two methods 'findElement' and 'findElements' provided by Selenium WebDriver for this purpose. Now let us go ahead and understand the details of these methods.

### *****How to find elements in Selenium?*****

As discussed, Selenium WebDriver provides two methods using which we can find an element or list of elements on a web page. These are:

***findElement()***: This method uniquely finds a web element on the web page.

***findElements()***: This method finds a list of web elements on the web page.

Let's understand the usage and details of these methods in the following sections:

#### **findElement() in Selenium**

The findElement() method of the Selenium WebDriver finds a unique web element within the webpage.

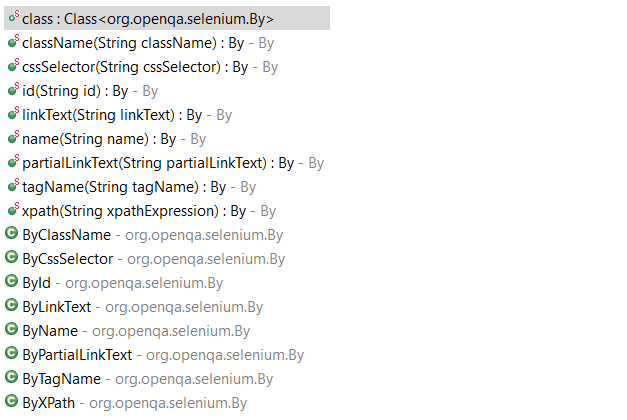
It’s syntax looks like below:

WebElement elementName = driver.findElement

(By.LocatorStrategy("LocatorValue"));

As shown in the above syntax, this command accepts the ***"By "*** object as the argument and returns a WebElement object.

The ***"By"*** is a locator or query object and accepts the locator specifier or strategies we discussed above. So if we write the line ***"driver.findElement( By.)"*** then the ***Eclipse IntelliSense*** will give the following locator strategies that we can associate with ***By object.***



The above screenshot shows all the options that we get when we write 'By'. We will explain each of these strategies in the later sections of this chapter.

***Note:*** In case there is no matching element found, the findElement command throws NoSuchElementException.

But what happens if there are multiple elements matching the criteria provided in the findElement() method? When such a case occurs, the ***findElement() method returns the first most element within the web page***.

#### **findElements() in Selenium**

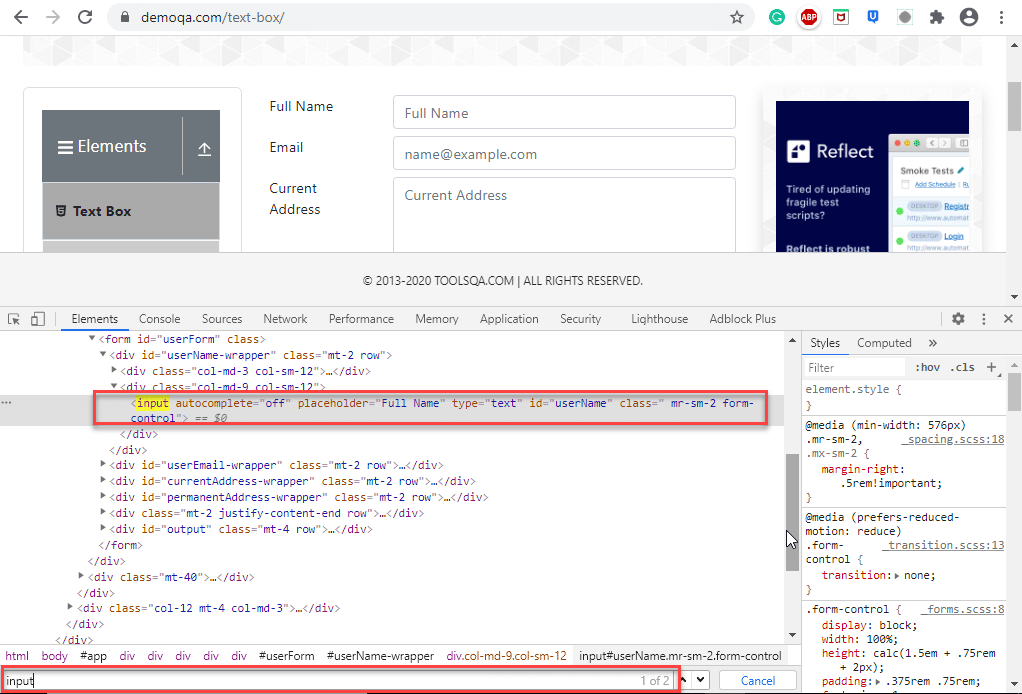
The command findElements() returns a list of web elements that match the specified criteria, unlike findElement() which returned a unique element. ***If there are no matching elements then an empty list returns.***

The general syntax of findElements() command in Selenium WebDriver is as below:

List<WebElement> elementName = driver.findElements(By.LocatorStrategy("LocatorValue"));

Like the findElement() command, this method also accepts the ***"By "*** object as the parameter and returns a ***WebElement***  list.

Let us consider an example wherein we need to find the number of elements having tag name as ***"input "*** on the ***[DemoQA text box page.](https://demoqa.com/text-box/)*** The inspect panel for this is as below.



In the above screenshot, when we search for the tag name 'input' two entries return (shown by the red rectangle around the search tool which says 1/2).

The following program shows the example of the ***findElements()*** method in which we provide the By object with tagName.

import java.util.List;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

public class FindElementByTagName {

public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver", "C:/testSelenium/chromedriver.exe");

WebDriver driver = new ChromeDriver();

driver.get("https://demoqa.com/text-box/");

// Find elements using tag name

List<WebElement> allInputElements = driver.findElements(By.tagName("input"));

if(allInputElements.size() != 0)

{

System.out.println(allInputElements.size() + " Elements found by TagName as input \n");

for(WebElement inputElement : allInputElements)

{

System.out.println(inputElement.getAttribute("placeholder"));

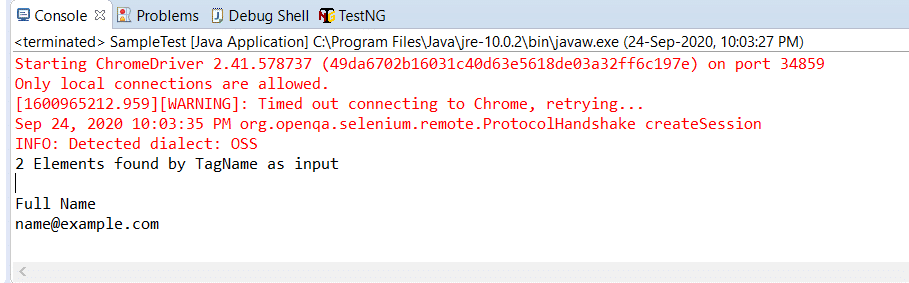
}

}

}

}

Following is the program output.



Next, let us understand how to use different locator strategies with findElement() and findElements() commands.

### *****What is By class in Selenium?*****

In this section, we will understand how to use Selenium WebDriver's findElement() and findElements() with different strategies using the ***By class.*** The 'By' class accepts various locator strategies explained above to find an element or elements on a web page. Let us discuss all the By class locator strategies.

#### **How to find an element using the attribute "id" in Selenium?**

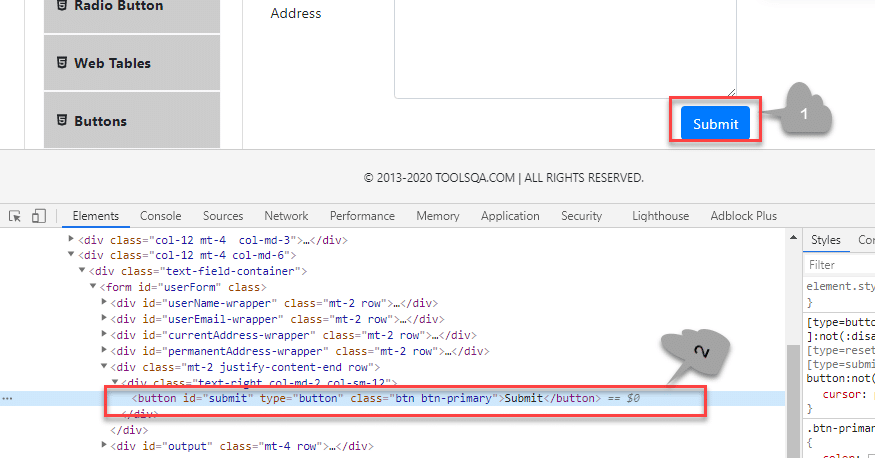
Using ***"id "*** to find an element is by far the most common strategy used to find an element. Suppose if the webpage uses dynamically generated ids, then this strategy returns the first web element that matches the id.

This strategy is preferred as most web pages are designed by associating ids with the elements. This is because using IDs is the easiest and quickest way to locate elements because of its simplicity while coding a web page. The ***value of the id attribute*** is a String type parameter.

The general syntax of findElement() command using ***By id*** strategy is :

WebElement elm = driver.findElement(By.id("Element\_Id"));

As an example consider the following element in the ***[DemoQA text box page:](https://demoqa.com/text-box/)***



Here we have selected the ***"submit "*** button (marked 1). The element code for this is marked 2 in the above screenshot.

The findElement() command corresponding to the above element:

WebElement element = driver.findElement(By.id("submit"));

// Action can be performed on Button element

element.submit();

***Note:*** If none of the web elements within the web page matches the id attribute then a ***"NoSuchElementException"*** is raised.

***Note:***  UI developers have to ensure that the ids on the page are unique. Auto-generated or dynamically generated ids are usually non-unique.

The complete program to find an element using the ***"By.id "*** object is as seen below:

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

public class FindElementById {

public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver", "C:/testSelenium/chromedriver.exe");

WebDriver driver = new ChromeDriver();

driver.get("https://demoqa.com/text-box/");

WebElement element = driver.findElement(By.id("submit"));

if(element != null) {

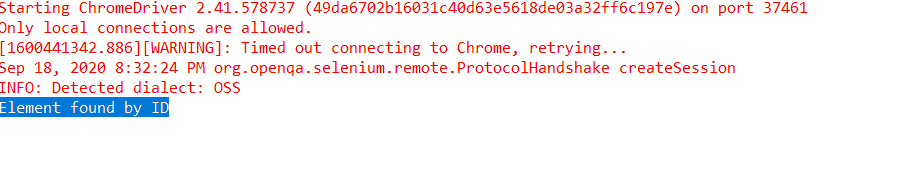
System.out.println("Element found by ID");

}

}

}

This program gives the following output.



The above program is a program to find an element using the id (By.Id) of that element. We provide an appropriate URL from which we need to search an element and then call ***"findElement() "*** with the argument By.id(***"elementID"***). This call returns the given element with the specified id.

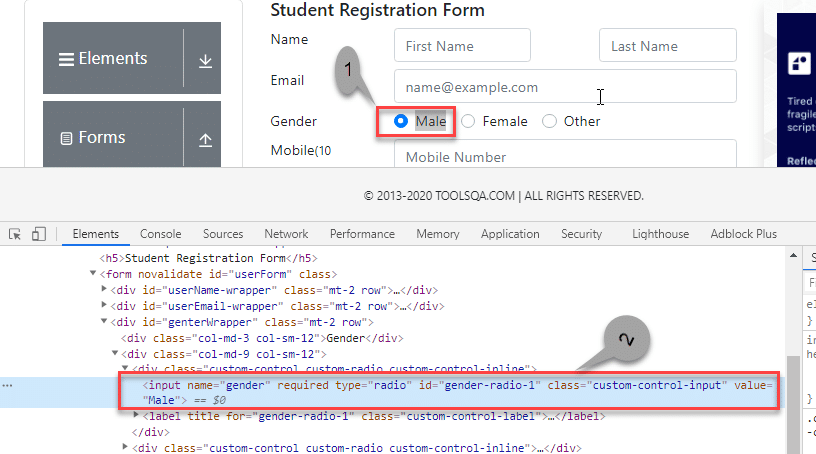
#### **How to find an element using the attribute "name" in Selenium?**

This strategy is the same as ***id*** except that the locator locates an element using the ***"name"*** instead of ***"id ".***

The ***value of the NAME*** attribute accepted is of type String. The general syntax of the findElement() method with By Name strategy is as below.

WebElement elm = driver.findElement(By.name("Element\_NAME"));

For example, consider the following element on the page ***[DemoQAAutomationPracticeForm :](https://demoqa.com/automation-practice-form)***



In the above screenshot, we select the first gender value (marked 1). Its corresponding element in the DOM  is highlighted (marked 2).

The corresponding findElement() method call for the above element is:

WebElement element = driver.findElement(By.name("gender"));

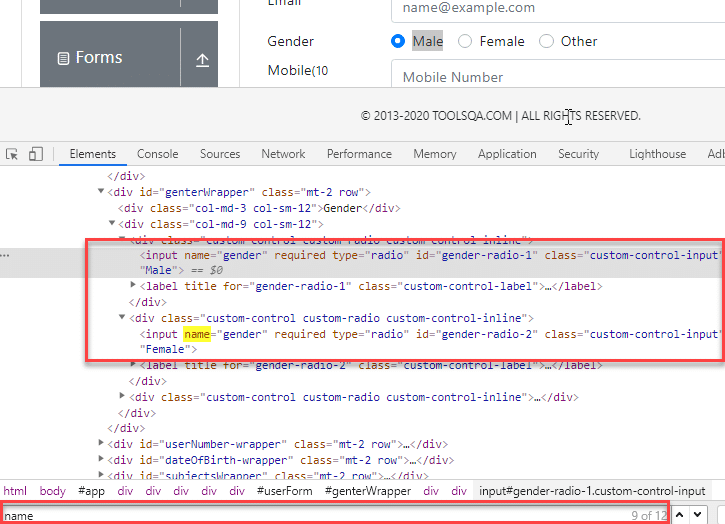
// Action can be performed on Input Text element

element.sendKeys("ToolsQA");

As a result of this method call, the first element matching the given name attribute value returns. If we can not find the match, ***NoSuchElementException*** raises.

Providing name as a strategy is also an efficient way to find an element but again if the names are not unique then the method suffers.

For example, consider the following element:



In the above screenshot, there are two elements with the same name =  gender. In this case, the findElement() method returns the first element.

Following code shows the program to find an element using Name (By.name):

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

public class FindElementByName {

public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver", "C:/testSelenium/chromedriver.exe");

WebDriver driver = new ChromeDriver();

driver.get("https://demoqa.com/automation-practice-form");

WebElement element = driver.findElement (By.name("gender"));

if(element != null) {

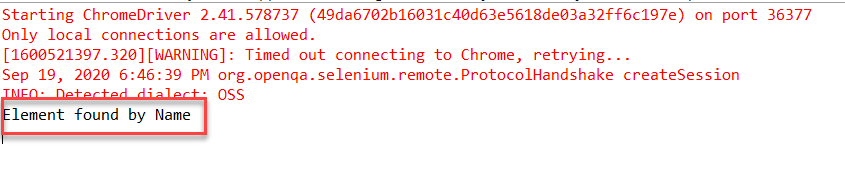
System.out.println("Element found by Name");

}

}

}

The program gives the following output.



The above program finds an element in Selenium using the name. We provide the name of the element that we can have to search as an argument to the By object in the 'findElement()' call.

#### **How to find an element using the attribute "class name" in Selenium?**

Here the value of the ***"class"*** attribute is passed as the locator. This strategy is mostly used to find multiple elements that use similar CSS classes.

The locator strategy 'By Class Name' finds the elements on the web page based on the ***CLASS attribute value.*** The strategy accepts a parameter of type String. The general syntax with the Class name strategy is given by:

List<WebElement> webList = driver.findElements(By.className(<Element\_CLASSNAME>)) ;

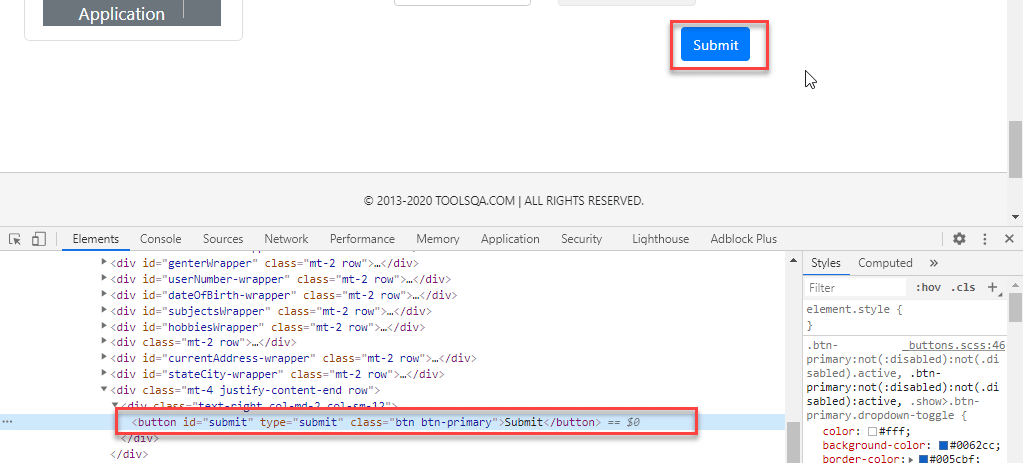
or

WebElement elm = driver.findElement(By.className(<Element\_CLASSNAME>)) ;

The first syntax is to obtain a list of matching elements based on Class Name while the second syntax is to get only one matching element.

In case the element has many classes, then this strategy will match each of the classes.

Consider the following element (submit button) on ***[DemoQAAutomationPracticeForm :](https://demoqa.com/automation-practice-form)***



The corresponding command for finding the element marked above is:

WebElement parentElement = driver.findElement(By.className("button"));

parentElement.submit();

***Note:*** Finding elements using class name strategy is helpful when we end up with non-unique IDs and names. That time we just go for the Class Name strategy and try to find the elements. When we use the Class Name strategy, once Selenium finds the particular class, it then looks for ID in the specified class.

The program to find an element using By.className is as below:

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

public class FindElementByClassName {

public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver", "C:/testSelenium/chromedriver.exe");

WebDriver driver = new ChromeDriver();

driver.get("https://demoqa.com/automation-practice-form");

WebElement parentElement = driver.findElement (By.className("button"));

if(parentElement != null) {

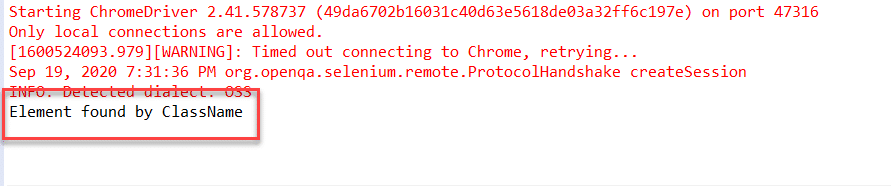
System.out.println("Element found by ClassName");

}

}

}

This program gives the following output.



In this program, we have provided a class name ***"button"*** as a By object argument in the 'findElement()' call. It scans the page and returns an element with className = ***"button"***.

#### **How to find an element using the attribute "HTML tag name" in Selenium?**

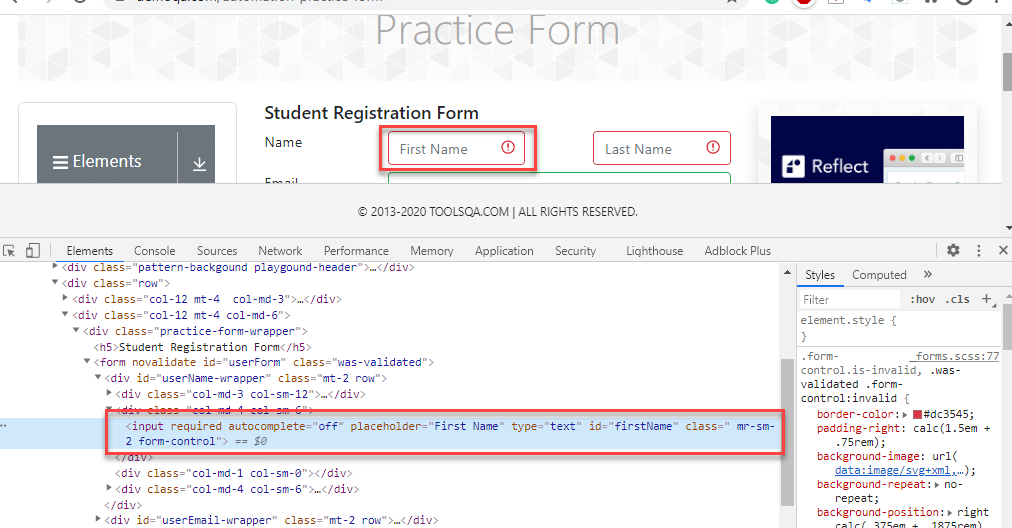
The ***Tag Name*** strategy uses an HTML tag name to locate the element. We use this approach rarely and we use it only when we are not able to find elements using any other strategies.

The ***value of the TAG attribute*** is a String type parameter. The syntax of the findElement() method using this strategy is as below.

WebElement elem =&nbsp; driver.findElement(By.tagName(“Element\_TAGNAME”));

As already mentioned, note that this strategy is not very popular and we use it only when there is no other alternative to locate the element.

As an example consider the following element on ***[DemoQAAutomationPracticeForm :](https://demoqa.com/automation-practice-form)***



The corresponding command for the above element (input tag) is as below:

WebElement element = driver.findElement(By.tagName("input"));

Following is the program to find elements using ***By.tagName*** object.

import java.util.List;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

public class FindElementByTagName {

public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver", "C:/testSelenium/chromedriver.exe");

WebDriver driver = new ChromeDriver();

driver.get("https://demoqa.com/automation-practice-form");

WebElement element = driver.findElement (By.tagName("input"));

if(element != null) {

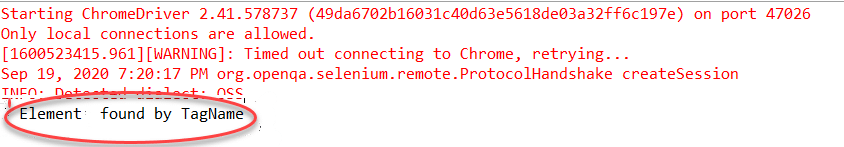
System.out.println("Element found by tagName");

}

}

}

The output of this program is as seen below.

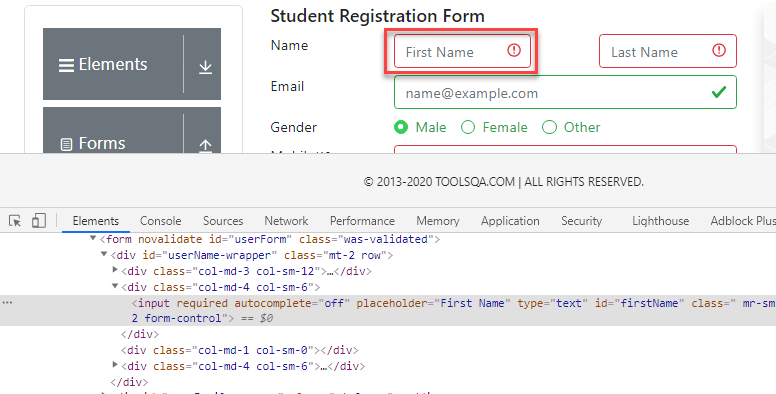


Above program uses ***By.tagName*** object in 'findElement()' call to find an element based on ***tagName = "input"***.

#### **How to find an element using the "CSS Selector" in Selenium?**

We can also use the [***CSS Selector strategy***](https://www.toolsqa.com/selenium-webdriver/css-selectors-in-selenium/) as an argument to By object when finding the element. Since CSS Selector has native browser support, sometimes the CSS Selector strategy is faster than the [***XPath***](https://www.toolsqa.com/selenium-webdriver/write-effective-xpaths/) strategy.

Again we will choose an element from the page ***[DemoQAAutomationPracticeForm :](https://demoqa.com/automation-practice-form)***



The CSS Selector for the above input field is ***#firstName.*** So the corresponding command to find element by CSS Selector is:

WebElement inputElem = driver.findElement(By.cssSelector("input[id = 'firstName']"));

inputElem.SendKeys("demoQA");

The following program shows how to find elements using the ***By.cssSelector*** construct.

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

public class FindElementByCssSelector {

public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver", "C:/testSelenium/chromedriver.exe");

WebDriver driver = new ChromeDriver();

driver.get("https://demoqa.com/automation-practice-form");

WebElement inputElem = driver.findElement (By.cssSelector("input[id = 'firstName']"));

if(inputElem != null) {

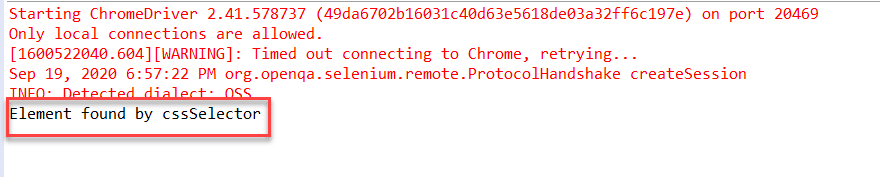
System.out.println("Element found by cssSelector");

}

}

}

The program gives the following output.



The above program finds an element using CSS Selector for the field 'firstNam'  by using the ***By.cssSelector*** locator strategy. The program returns an element having the specified CSS selector.

#### **How to find an element using the "XPath" in Selenium?**

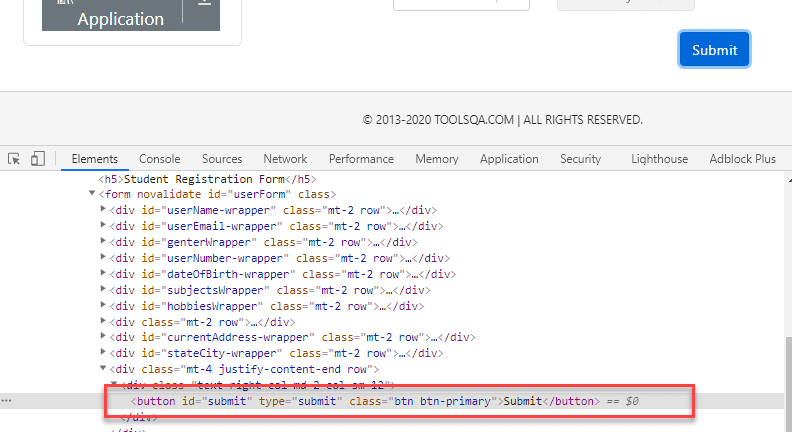
This strategy is the most popular one for finding elements. Using this strategy we navigate through the structure of the ***HTML*** or ***XML*** documents.

This strategy accepts a String type parameter, XPath Expression. The general syntax of using this strategy is as given below:

WebElement elem = driver.findElement(By.xpath(“ElementXPathExpression”));

Using [***XPath***](https://www.toolsqa.com/selenium-webdriver/write-effective-xpaths/) we can locate a single element using various ways. It provides many different and easy ways to locate elements.

As an example let us take the following element in the page ***[DemoQAAutomationPracticeForm :](https://demoqa.com/automation-practice-form)***



The XPath for the above button element is ***[@id="submit"].*** Please refer [***How To Inspect Elements***](https://www.toolsqa.com/selenium-webdriver/inspect-elements-using-browser-inspector/) using Web Inspector for more information. So we use it in the findElement() command as below:

WebElement buttonLogin = driver.findElement(By.xpath("//button[@id = 'submit']"));

A program to find elements using ***By.XPath*** is as follows:

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

public class FindElementByXpath {

public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver", "C:/testSelenium/chromedriver.exe");

WebDriver driver = new Chrome

Driver();

driver.get("https://demoqa.com/automation-practice-form");

WebElement buttonSubmit = driver.findElement( By.xpath("//button[@id = 'submit']"));

if(buttonSubmit != null) {

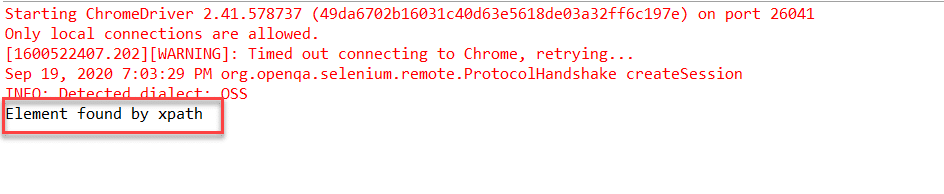
System.out.println("Element found by xpath");

}

}

}

This program displays the following output.



Here we have provided the ***XPath*** of the ***"submit "*** button as an argument to the ***By.xpath*** locator. The program returns the element that matches the specified XPath.

#### **How to find an element using the "Link Text/Partial Link Text" in Selenium?**

This strategy finds links within the webpage. It specially finds elements having links. This means we can use this strategy to find elements of ***"a "*** (links) tags that have matching link names or partial link names.

The strategy accepts the ***value of the LINKTEXT attribute*** as a String type parameter.

The syntax of findElement using this strategy is as seen below.

WebElement elem = driver.findElement(By.linkText(“Element LinkText”));

The above syntax is for finding elements using full link text. This is used when we know the link text that is used within the ***anchor*** (a) tag.

We can also use the partial link and find elements. Following is the syntax:

WebElement elem = driver.findElement(By.partialLinkText(“ElementLinkText”));

Here we can provide partial link names.

As an example, we can take the following element (***[DemoQAHomeLink](https://demoqa.com/links)*** ). We have highlighted the element as shown below:



We can use a link strategy if the targetted text is link text. So for the above link element, the findElement() command for the link and partial link strategy is as follows:

WebElement element = driver.findElement(By.linkText("Home"));

//Or can be identified as

WebElement element = driver.findElement(By.partialLinkText("HomehY");

In the first example, we use By.linkText strategy and provide the entire 'linkname'. This will look for a link with the ***"Home"*** word. In the second example, we use By.partialLinkText and only provide a part of 'linkname' ('HomehY'). Since this is a partial link, it will look for links starting with 'HomehY'. As shown above, there is a link 'HomehYtil' on the page. So ***By.partialLinkText*** will find this link.

Let us implement a code to find element using By.linkText/By.partialLinkText.

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

public class FindElementByLinkTextAndPartialLinkText {

public static void main(String[] args) {

System.setProperty("webdriver.chrome.driver", "C:/testSelenium/chromedriver.exe");

WebDriver driver = new ChromeDriver();

driver.get("https://demoqa.com/links");

WebElement element = driver.findElement (By.linkText("Home"));

if(element != null) {

System.out.println("Element found by LinkText");

}

element= driver.findElement (By.partialLinkText("HomehY");

if(element!= null) {

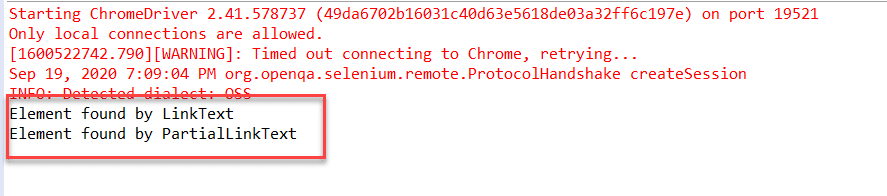
System.out.println("Element found by PartialLinkText");

}

}

}

The Output:



This program finds an element using By.linkText and By.partialLinkText locator. When By.linkText is specified, the 'findElement' matches the entire linkText specified. It matches the partial text when By.partialLinkText is specified.

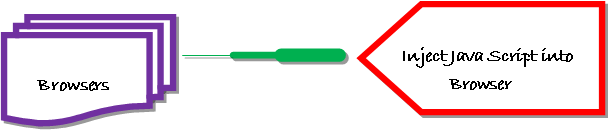
## Difference between find Element and find Elements in Selenium

Let us discuss some differences between findElement() and findElements() methods provided by Selenium WebDriver.

| ***FindElement()*** | ***FindElements()*** |
| --- | --- |
| Returns the first web element out of all the elements found by the same locator. | Finds and returns a list of web elements. |
| This method finds only one element. | This method returns the collection of elements matching the locator. |
| If no element matches the locator, an exception ***“NoSuchElementException”*** is thrown. | No exception is thrown if no matching elements are found. Simply returns an empty list. |
| NNo indexing required since only one element is returned. | Each web element is indexed starting from 0. |

**Java Script Executor in Selenium**

JavaScriptExecutor is an Interface that helps to execute JavaScript through Selenium Webdriver. JavaScriptExecutor provides two methods “executescript” & “executeAsyncScript” to run javascript on the selected window or current page.



## Why do we need JavaScriptExecutor?

In Selenium Webdriver, locators like XPath, CSS, etc. are used to identify and perform operations on a web page.

In case, these locators do not work you can use JavaScriptExecutor. You can use JavaScriptExecutor to perform an desired operation on a web element.

Selenium supports javaScriptExecutor. There is no need for an extra plugin or add-on. You just need to import (**org.openqa.selenium.JavascriptExecutor**) in the script as to use JavaScriptExecutor.

## JavaScriptExecutor Methods in Selenium

### executeScript

This method executes [JavaScript](https://www.guru99.com/interactive-javascript-tutorials.html)in the context of the currently selected frame or window in Selenium. The script used in this method runs in the body of an anonymous function (a function without a name). We can also pass complicated arguments to it.

**The script can return values. Data types returned are**

* Boolean
* Long
* String
* List
* WebElement.

**JavascriptExecutor syntax:**

JavascriptExecutor js = (JavascriptExecutor) driver;

js.executeScript(Script,Arguments);

* **Script** – This is the JavaScript that needs to execute.
* **Arguments** – It is the arguments to the script. It’s optional.

### executeAsyncScript

With Asynchronous script, your page renders more quickly. Instead of forcing users to wait for a script to download before the page renders. This function will execute an asynchronous piece of JavaScript in the context of the currently selected frame or window in Selenium. The JS so executed is single-threaded with a various callback function which runs synchronously.

## How to use JavaScriptExecutor in Selenium

Here is a step-by-step process on how to use JavaScriptExecutor in Selenium:

**Step 1)** Import the package.

import org.openqa.selenium.JavascriptExecutor;

**Step 2)** Create a Reference.

JavascriptExecutor js = (JavascriptExecutor) driver;

**Step 3)** Call the JavascriptExecutor method.

js.executeScript(script, args);

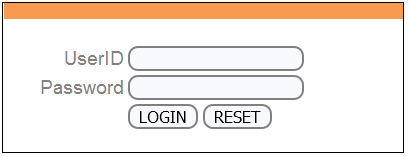
## Example of Click an Element using JavaScripExecutor in Selenium

For executeScript, we will see three different example one by one.

### 1) Example: Click a button to login and generate Alert window using JavaScriptExecutor.

In this scenario, we will use “Guru99” demo site to illustrate JavaScriptExecutor. In this example,

* Launch the web browser
* open the site <https://demo.guru99.com/V4/> and
* login with credentials



* Display alert window on successful login.

import org.openqa.selenium.By;

import org.openqa.selenium.JavascriptExecutor;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.firefox.FirefoxDriver;

import org.testng.annotations.Test;

public class JavaSE\_Test {

@Test

public void Login()

{

WebDriver driver= new FirefoxDriver();

//Creating the JavascriptExecutor interface object by Type casting

JavascriptExecutor js = (JavascriptExecutor)driver;

//Launching the Site.

driver.get("https://demo.guru99.com/V4/");

WebElement button =driver.findElement(By.name("btnLogin"));

//Login to Guru99

driver.findElement(By.name("uid")).sendKeys("mngr34926");

driver.findElement(By.name("password")).sendKeys("amUpenu");

//Perform Click on LOGIN button using JavascriptExecutor

js.executeScript("arguments[0].click();", button);

//To generate Alert window using JavascriptExecutor. Display the alert message

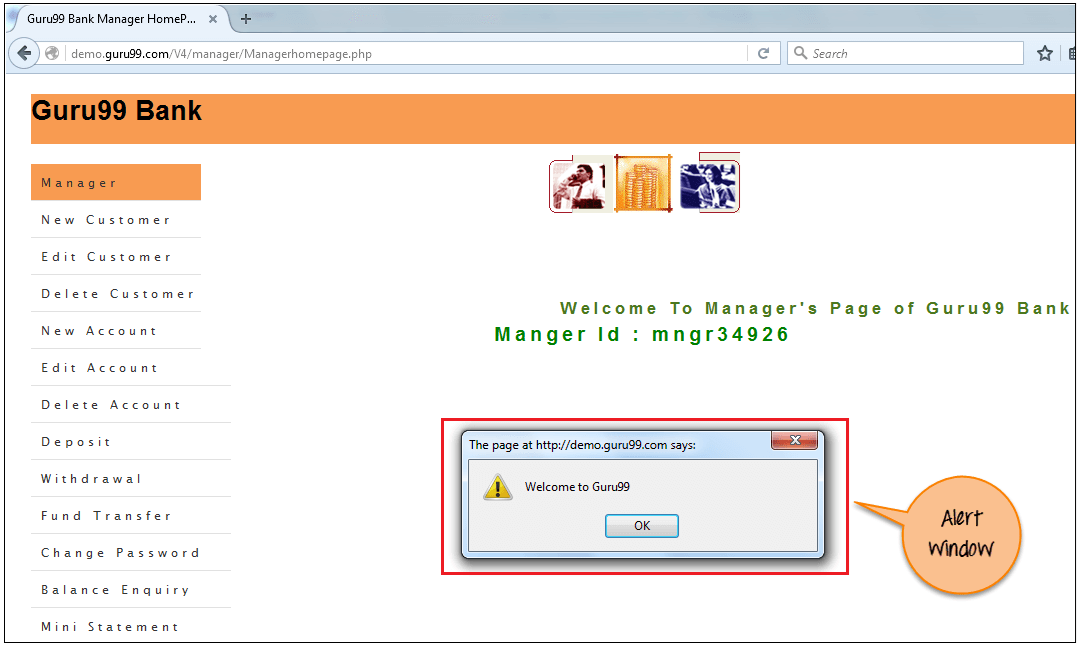
js.executeScript("alert('Welcome to Guru99');");

}

}

**Output:** When the code is executed successfully. You will observe

* Successful click on login button and the
* Alert window will be displayed (see image below).



### 2) Example: Capture Scrape Data and Navigate to different pages using JavaScriptExecutor.

Execute the below selenium script. In this example,

* Launch the site
* Fetch the details of the site like URL of the site, title name and domain name of the site.
* Then navigate to a different page.

import org.openqa.selenium.JavascriptExecutor;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

import org.testng.annotations.Test;

public class JavaSE\_Test {

@Test

public void Login()

{

WebDriver driver= new FirefoxDriver();

//Creating the JavascriptExecutor interface object by Type casting

JavascriptExecutor js = (JavascriptExecutor)driver;

//Launching the Site.

driver.get("https://demo.guru99.com/V4/");

//Fetching the Domain Name of the site. Tostring() change object to name.

String DomainName = js.executeScript("return document.domain;").toString();

System.out.println("Domain name of the site = "+DomainName);

//Fetching the URL of the site. Tostring() change object to name

String url = js.executeScript("return document.URL;").toString();

System.out.println("URL of the site = "+url);

//Method document.title fetch the Title name of the site. Tostring() change object to name

String TitleName = js.executeScript("return document.title;").toString();

System.out.println("Title of the page = "+TitleName);

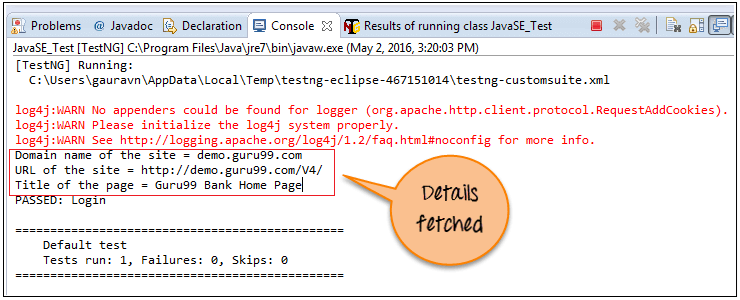
//Navigate to new Page i.e to generate access page. (launch new url)

js.executeScript("window.location = 'https://demo.guru99.com/'");

}

}

**Output:** When above code is executed successfully, it will fetch the details of the site and navigate to different page as shown below.



[TestNG] Running:

C:\Users\gauravn\AppData\Local\Temp\testng-eclipse-467151014\testng-customsuite.xml

log4j:WARN No appenders could be found for logger (org.apache.http.client.protocol.RequestAddCookies).

log4j:WARN Please initialize the log4j system properly.

log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.

Domain name of the site = demo.guru99.com

URL of the site = https://demo.guru99.com/V4/

Title of the page = Guru99 Bank Home Page

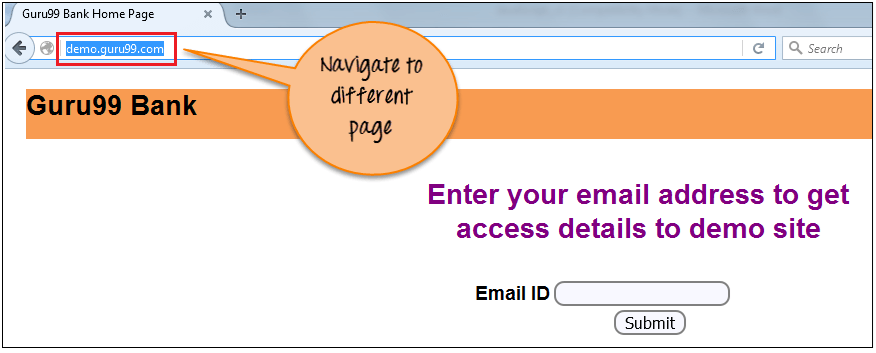
PASSED: Login

===============================================

Default test

Tests run: 1, Failures: 0, Skips: 0

===============================================



### 3) Example: Scroll Down using JavaScriptExecutor.

Execute the below selenium script. In this example,

* Launch the site
* Scroll down by 600 pixel

import org.openqa.selenium.JavascriptExecutor;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

import org.testng.annotations.Test;

public class JavaSE\_Test {

@Test

public void Login()

{

WebDriver driver= new FirefoxDriver();

//Creating the JavascriptExecutor interface object by Type casting

JavascriptExecutor js = (JavascriptExecutor)driver;

//Launching the Site.

driver.get("http://moneyboats.com/");

//Maximize window

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

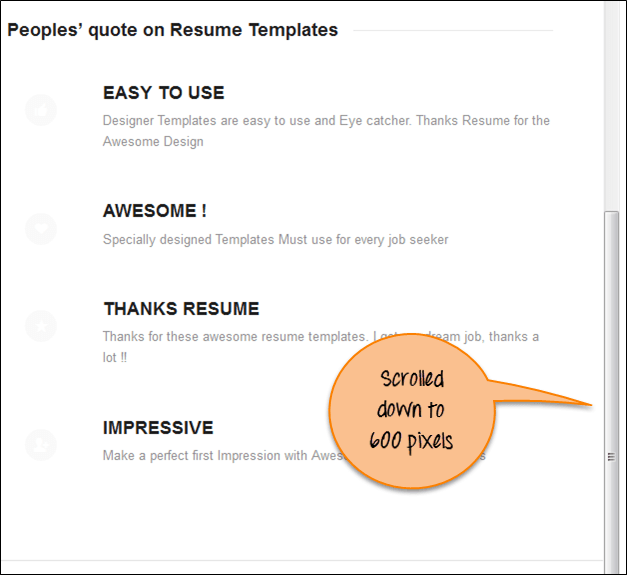
//Vertical scroll down by 600 pixels

js.executeScript("window.scrollBy(0,600)");

}

}

**Output**: When above code is executed, it will scroll down by 600 pixels (see image below).



## Example of executeAsyncScript in Selenium

Using the executeAsyncScript, helps to improve the performance of your test. It allows writing test more like a normal coding.

The execSync blocks further actions being performed by the Selenium browser but execAsync does not block action. It will send a callback to the server-side[Testing](https://www.guru99.com/software-testing.html)suite once the script is done. It means everything inside the script will be executed by the browser and not the server.

### Example 1: Performing a sleep in the browser under test.

In this scenario, we will use “Guru99” demo site to illustrate executeAsyncScript. In this example, you will

* Launch the browser.
* Open site <https://demo.guru99.com/V4/>.
* Application waits for 5 sec to perform a further action.

**Step 1)** Capture the start time before waiting for 5 seconds ( 5000 milliseconds) by using executeAsyncScript() method.

**Step 2)** Then, use executeAsyncScript() to wait 5 seconds.

**Step 3)** Then, get the current time.

**Step 4)**Subtract (current time – start time) = passed time.

**Step 5)** Verify the output it should display more than 5000 milliseconds

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.JavascriptExecutor;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

import org.testng.annotations.Test;

public class JavaSE\_Test {

@Test

public void Login()

{

WebDriver driver= new FirefoxDriver();

//Creating the JavascriptExecutor interface object by Type casting

JavascriptExecutor js = (JavascriptExecutor)driver;

//Launching the Site.

driver.get("https://demo.guru99.com/V4/");

//Maximize window

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

//Set the Script Timeout to 20 seconds

driver.manage().timeouts().setScriptTimeout(20, TimeUnit.SECONDS);

//Declare and set the start time

long start\_time = System.currentTimeMillis();

//Call executeAsyncScript() method to wait for 5 seconds

js.executeAsyncScript("window.setTimeout(arguments[arguments.length - 1], 5000);");

//Get the difference (currentTime - startTime) of times.

System.out.println("Passed time: " + (System.currentTimeMillis() - start\_time));

}

}

**Output:** Successfully displayed the passed time more than 5 seconds(5000 miliseconds) as shown below:

[TestNG] Running:

C:\Users\gauravn\AppData\Local\Temp\testng-eclipse-387352559\testng-customsuite.xml

log4j:WARN No appenders could be found for logger (org.apache.http.client.protocol.RequestAddCookies).

log4j:WARN Please initialize the log4j system properly.

log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.

Passed time: 5022

PASSED: Login

===============================================

Default test

Tests run: 1, Failures: 0, Skips: 0

===============================================

#### Summary:

JavaScriptExecutor is used when Selenium Webdriver fails to click on any element due to some issue.

* JavaScriptExecutor provides two methods “executescript” & “executeAsyncScript” to handle.
* Executed the JavaScript using Selenium Webdriver.
* Illustrated how to click on an element through JavaScriptExecutor, if selenium fails to click on element due to some issue.
* Generated the ‘Alert’ window using JavaScriptExecutor.
* Navigated to the different page using JavaScriptExecutor.
* Scrolled down the window using JavaScriptExecutor.
* Fetched URL, title, and domain name using JavaScriptExecutor.

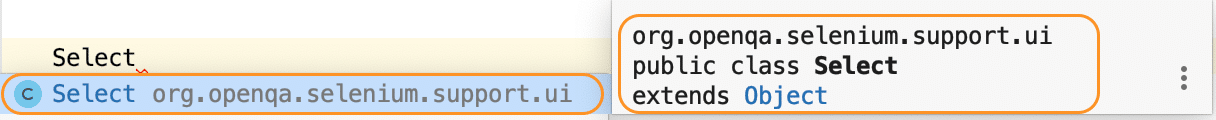
**Select Class in Selenium**

A web page consists of various web elements ranging from text fields, buttons, checkboxes, drop-downs, etc. With the wide use of forms on the websites these days, we come across ***DropDowns*** now and then. There are various types of dropdowns available these days, majorly of which are categorized as a ***single-select*** (which allows selecting only one value) or ***multi-select*** (which allows selecting multiple values). [***Selenium WebDriver***](https://www.toolsqa.com/selenium-webdriver/selenium-testing/) provides a class named ***"Select"***, which provides various methods to handle the dropdowns, be it single-select or multi-select dropdowns. In this article, we will understand the intricacies of the ***"Select "*** class of Selenium WebDriver and will understand how we can handle dropdown in Selenium by covering the details under the following topics:

* What is Select class in Selenium?
  + How to select a value from a dropdown in Selenium?
  + How to select multiple values from a dropdown in Selenium?
  + Also, how to get options from a dropdown in Selenium?
* How to deselect a value from a dropdown Selenium?
* Examples illustrating Select class usage in Selenium.
  + Example 1 - Handling dropdown using Selenium WebDriver.
  + Example 2 - Handling multi-select using Selenium WebDriver.

## What is Select Class in Selenium?

In HTML, the dropdowns are generally implemented either using the **<select>** tag or the **<input>** tag. To perform certain operations on the dropdowns, which are declared using the **<select> HTML tag,** Selenium WebDrivers provides a class called ***"Select "*** class. As soon as you start typing the ***"Select "*** in your [***IDE,***](https://www.toolsqa.com/selenium-webdriver/download-and-start-eclipse/) it will show the details as shown below:



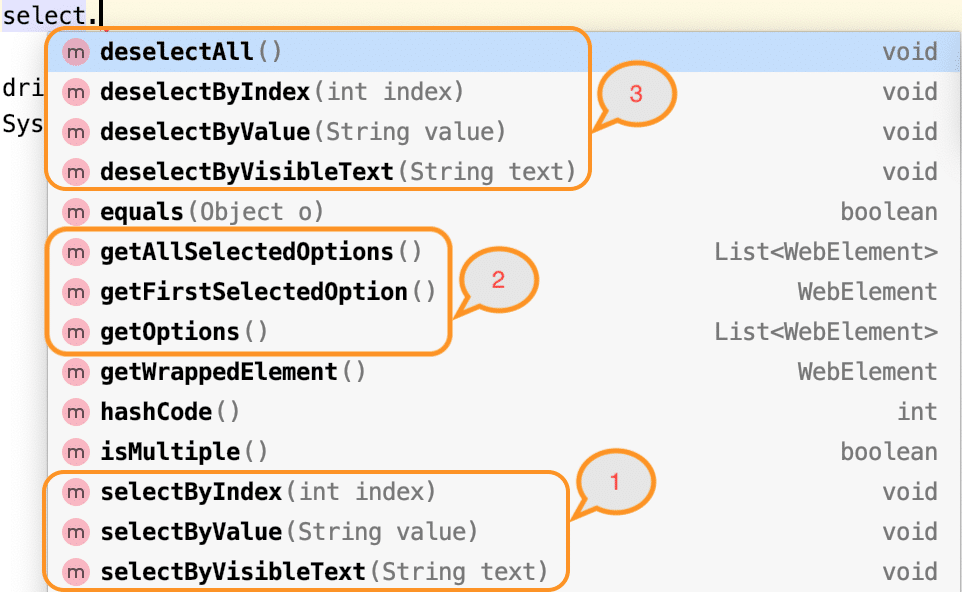
As we can see from the above screenshot, the ***"Select "*** class is provided by the ***"org.openqa.selenium.support.ui "*** package of Selenium WebDriver. You can create an object of the Select class, by-passing the object of the [***"WebElement"***](https://www.toolsqa.com/selenium-webdriver/webelement-commands/) class, which shows the object returned by the corresponding [***locator***](https://www.toolsqa.com/selenium-webdriver/selenium-locators/) of the WebElement.

So, you can create the object of the ***Select***  class by using the following syntax:

Select select = new Select(WebElement webelement);

The Select class constructor accepts one parameter: the WebElement object returned by the locators of the specified element.

The ***"Select "*** class provides various methods for handling the dropdown operations. The following figures list a few of them:



Let's now understand the syntax and usage of various select and deselect methods provided by the ***"Select "*** class in Selenium WebDriver.

### *****How to select a value from a dropdown in Selenium?*****

As highlighted in the above figure, the Select class of Selenium WebDriver  provides the following methods to select an option/value from a drop-down (as highlighted by marker 1 in the above image):

* selectByIndex
* selectByValue
* selectByVisibleText

Let's understand the syntax and usage of all these methods:

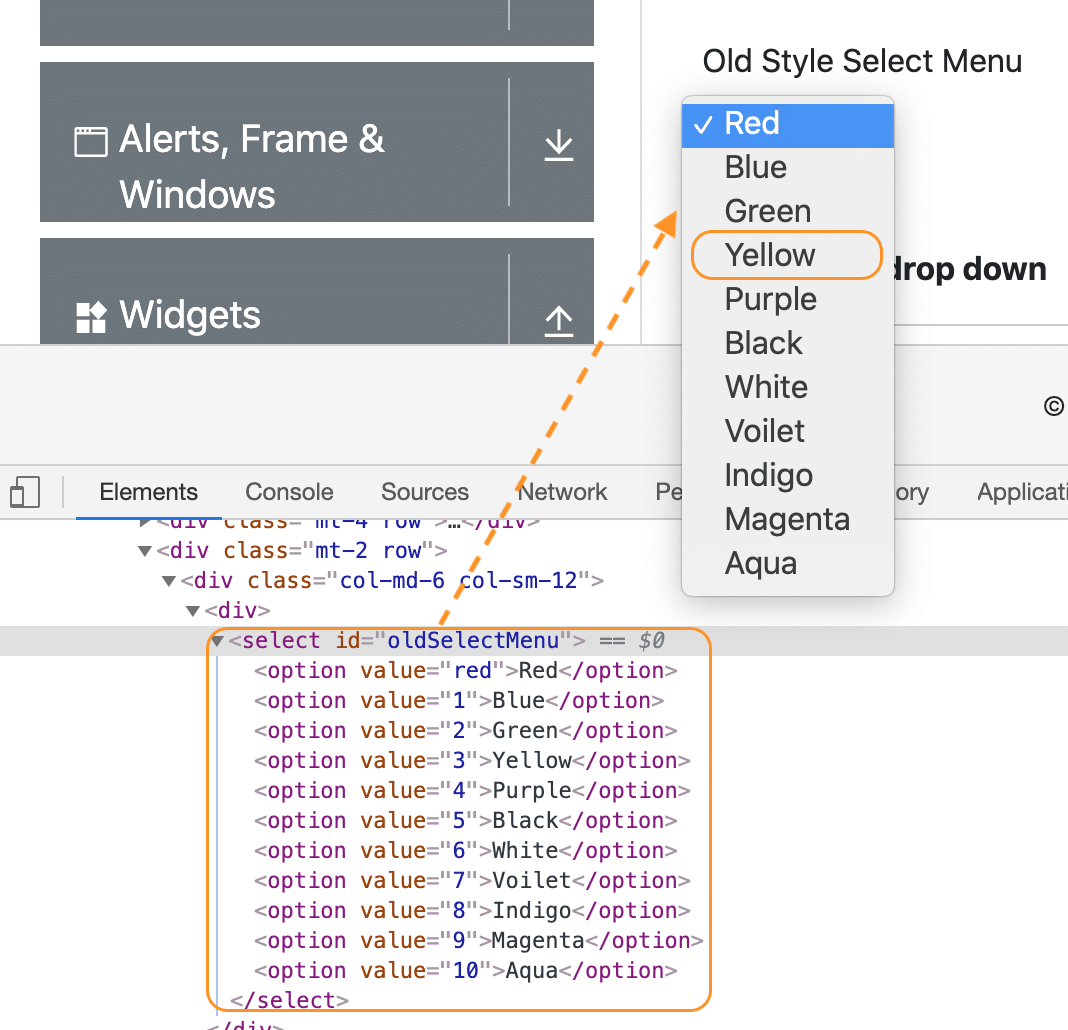
#### **selectByIndex:**

This method selects the dropdown option by its index number. We provide an integer value as the index number as an argument. It possesses the following syntax:

selectByIndex(int arg0) : void

i.e., it accepts the index of the dropdown value, which needs to be selected. The index starts at 0.

Suppose on the web page [***"https://demoqa.com/select-menu"***](https://demoqa.com/select-menu) we have the select the 4th value of the dropdown as highlighted below:



As we can see, the above dropdown is being implemented using the *<select>* HTML tag, so we can use the ***"Select "*** class of Selenium WebDriver to select the option ***"Yellow "*** using the index as shown below:

// Create object of the Select class

Select se = new Select(driver.findElement(By.xpath("//\*[@id='oldSelectMenu']")));

// Select the option by index

se.selectByIndex(3);

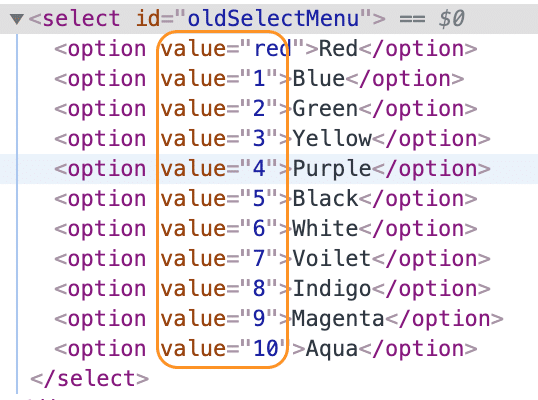
As we mentioned, the indices of dropdown start at 3, so the value ***"Yellow "*** can be selected using index 3.

#### **selectByValue**

This method selects the dropdown option by its value. We provide a string value as the value as an argument. It possesses the following syntax:

selectByValue(String arg0) : void

If we consider the same dropdown on page [***"https://demoqa.com/select-menu"***](https://demoqa.com/select-menu), as in the previous section, we can see that each of the options of the dropdown has an assigned value as shown below:



Now, if we have to select the option ***"White"***, we can use its value ***"6 ",*** as shown in the following code snippet:

// Create object of the Select class

Select se = new Select(driver.findElement(By.xpath("//\*[@id='oldSelectMenu']")));

// Select the option with value "6"

se.selectByValue("6");

As the value ***"6 "*** corresponds to the option ***"White,"*** so it will select the value **"White"** from the dropdown.

#### **selectByVisibleText**

This method enables one to select one option from the dropdown or multi-select dropdown based on the dropdown text. You need to pass the String value of the *<select>* element as an argument. It possesses the following syntax:

selectByVisibleText(String arg0): void

If we consider the same dropdown on page [***"https://demoqa.com/select-menu"***](https://demoqa.com/select-menu), as in the previous section, we can see that each of the options of the dropdown will have a text value, which is displayed on the web page also, so we can use that text to select the corresponding option, as shown below:

// Create the object of the Select class

Select se = new Select(driver.findElement(By.xpath("//\*[@id='oldSelectMenu']")));

// Select the option using the visible text

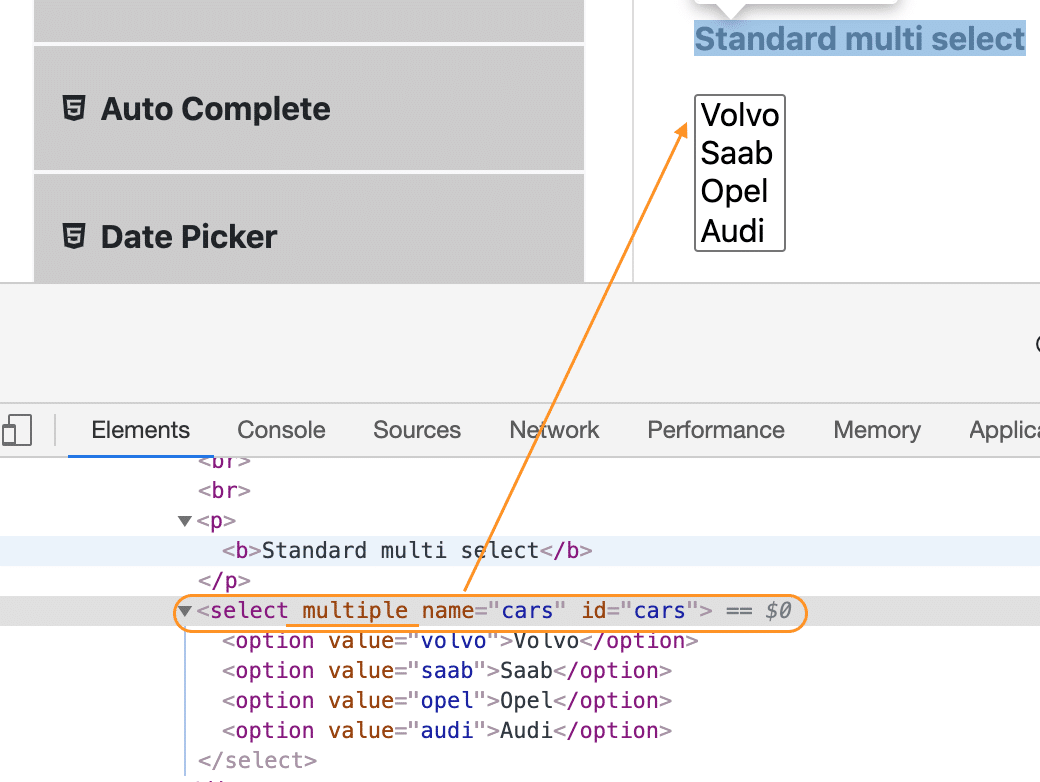
se.selectByVisibleText("White");

As the dropdown has one of the options having the text as ***"White"***, the same will be selected using the above code.

Apart from the dropdown types briefed above, the HTML *<select>* tag also provides ways to define dropdowns, which allows selecting multiple values. Let's see how a Multi-Select dropdown will be declared and how we can select multiple options in a dropdown using the ***"Select "*** class of Selenium WebDriver.

### *****How to select multiple values from a dropdown in Selenium?*****

If the *<select >* tag contains multiple attributes, it means that the dropdown allows selecting multiple values. As we can see in the following screenshot from the web page [***"https://demoqa.com/select-menu":***](https://demoqa.com/select-menu)



We can use any of the methods we used to select one value from the dropdown to select multiple values by invoking the methods multiple times for different values. The ***"Select "*** class provides a method, isMultiple(), using which we can first validate whether the dropdown allows us to select multiple values. Let's see how to use the isMultiple() method:

#### **How to check whether dropdown is Multi-Select?**

As we discussed, the Select class provides the ***"isMultiple() "*** method, which determines whether the web element in say supports multiple selections. It returns a boolean value, i.e., True/False, without taking any argument. It checks the attribute 'multiple' in the HTML code for the web element. Consequently, it possesses the following syntax:

isMultiple(): boolean

Once you determine whether the web element is multi-select or not, you can use the Select class's various select methods on the multiple values you intend to select. The below example code shows the same-

Select oSel = new Select(driver.findElement(By.xpath(//\*[@id='cars']);

if(oSel.isMultiple()){

//Selecting multiple values by index

oSel.selectByIndex(1);

oSel.selectByIndex(2);

//Or selecting by values

oSel.selectByValue("volvo");

oSel.selectByValue("audi");

//Or selecting by visible text

oSel.selectByVisibleText("Volvo");

oSel.selectByVisibleText("Opel");

}

And that is how you can select multiple values from a multi-select dropdown.

Now that we have understood how we can select values from a dropdown, be it single-select or multi-select, we should have some way to check which values the dropdown contains and what all values are selected in the dropdown. The ***"Select "*** class provides methods to get options from the dropdown. Let's understand the details and usage of these methods:

### *****How to get options from a dropdown in Selenium?*****

As highlighted by marker 2, in the image under the ***"Select "*** class section above, the Select class provides the following methods to get the options of a dropdown:

* getOptions()
* getFirstSelectedOption()
* getSelectedOptions()

Let's understand the details of all these methods:

#### **getOptions**

There are times when you need to get all the options in a dropdown or multi-select box. This is where you can use the getOptions() method of the Select class. It possesses the following syntax:

getOptions(): List<WebElement>

As we can see, this method returns all the options of the dropdown as a list of ***WebElement.*** The following code snippet shows how we can get all the options of the dropdown on the page ***"https://demoqa.com/select-menu":***

Select select = new Select(driver.findElement(By.id("oldSelectMenu")));

// Get all the options of the dropdown

List<WebElement> options = select.getOptions();

Using this method, we can retrieve all the options of a dropdown (be it single-select or multi-select ).

#### **getFirstSelectedOption()**

This method returns the first selected option of the dropdown. If it is a single-select dropdown, this method will return the selected value of the dropdown, and if it is a multi-select dropdown, this method will return the first selected value of the dropdown. It possesses the following syntax:

getFirstSelectedOption(): WebElement

As we can see, this method returns a ***WebElement.*** The following code snippet shows how we can get the first selected option of the dropdown on the page ***"https://demoqa.com/select-menu":***

Select select = new Select(driver.findElement(By.id("oldSelectMenu")));

// Get the first selected option of the dropdown

WebElement firstSelectedOption = select.getFirstSelectedOption();

Using this method, we can retrieve the first selected option of a dropdown (be it single-select or multi-select ).

#### **getAllSelectedOptions()**

This method returns all the selected options of the dropdown. If it is a single-select dropdown, this method will return the only selected value of the dropdown, and if it is a multi-select dropdown, this method will return all the selected values of the dropdown. It possesses the following syntax:

getAllSelectedOptions():List<WebElement>

As we can see, this method returns a list of ***WebElements.*** The following code snippet shows how we can get all the selected options of the dropdown on the page ***"https://demoqa.com/select-menu":***

Select select = new Select(driver.findElement(By.id("oldSelectMenu")));

// Get all the selected option of the dropdown

List<WebElement> selectedOptions = select.getAllSelectedOptions();

Using this method, we can retrieve all the selected options of a dropdown (be it single-select or multi-select ).

### *****How to deselect a value from a dropdown in Selenium?*****

Just like we select values in a DropDown & Multi-Select, we can deselect the values too. But the deselect method works only for Multi-Select. You can deselect pre-selected options from a Multi-select element using the different deselect methods discussed here.  As we would have observed in the screenshot showing methods of the ***"Select "*** class (shown by marker 3), the Select class provides the following methods to deselect values of a dropdown:

* deselectAll()
* deselectByIndex()
* deselectByValue()
* deselectByVisibleText()

Let's understand the details and usage of all these methods:

#### **deselectAll**

This method will clear all the selected entries of the dropdown. It possesses the following syntax:

deselectAll(): void

If there are few options already selected in a dropdown, you can deselect all the options using the method deselectAll(). The following code snippet shows a sample example, how we deselect all the values from the dropdown:

Select select = new Select(driver.findElement(By.id("oldSelectMenu")));

//Deselect all the options

select.deselectAll();

It will deselect all the options from the dropdown.

#### **deselectByIndex**

Similar to the selectByIndex() method, the Select class also provides the method to deselect an option from the dropdown using the deselectByIndex() method. You can use the option's index number to deselect it. It possesses the following syntax:

deselectByIndex(int arg0): void

So, if there are few options already selected in a dropdown, you can deselect one of the options using the method deselectByIndex(). The following code snippet shows a sample example, how we deselect one of the values from the dropdown by specifying its index:

Select select = new Select(driver.findElement(By.id("oldSelectMenu")));

//Deselect first value by index

select.deselectByIndex(1);

It will deselect the option at index 1 in the dropdown.

#### **deselectByValue**

Similar to the selectByValue() method, the Select class also provides the method to deselect an option from the dropdown using the ***deselectByValue()*** method. You can use the option's value to deselect it. It possesses the following syntax:

deselectByValue(String arg0): void

So, if there are few options already selected in a dropdown, you can deselect one of the options using deselectByValue(). The following code snippet shows a sample example, how we deselect one of the values from the dropdown by specifying its value:

Select select = new Select(driver.findElement(By.id("oldSelectMenu")));

//Deselect option with value "6"

select.deselectByValue("6");

It will deselect the option with value in the dropdown.

#### **deselectByVisibleText**

Similar to the selectByVisibleText() method, the Select class also provides the method to deselect an option from the dropdown using the ***deselectByVisibleText()*** method. You can use the option's text to deselect it. It possesses the following syntax:

deselectByVisibleText(String arg0): void

So, if there are few options already selected in a dropdown, you can deselect one of the options using the method deselectByVisibleText(). The following code snippet shows a sample example, how we deselect one of the values from the dropdown by specifying its text:

Select select = new Select(driver.findElement(By.id("oldSelectMenu")));

//Deselect option with text "White"

select.deselectByVisibleText("White");

It will deselect the option with the text ***"White "*** in the dropdown.

## Examples illustrating Select class usage in Selenium

We will cover two examples, one with a simple dropdown and another with a multi-select dropdown. We will be using the ***[ToolsQA demo](https://demoqa.com/select-menu)*** website to automate these scenarios. Let us now quickly start \*dropdown/\*multi-select automation.

### *****Example 1- Handling dropdown using Selenium WebDriver*****

Our use-case would follow the steps below-

* Launch the browser.
* Open ***"https://demoqa.com/select-menu"***.
* Select the Old Style Select Menu using the element id.
* Print all the options of the dropdown.
* Select 'Purple' using the index.
* After that, select 'Magenta' using visible text.
* Select an option using value.
* Close the browser

Using the methods of Select class as discussed above, the code would look like below-

import java.util.List;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

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

public class DropDown {

public static void main(String[] args) throws InterruptedException {

//Creating instance of Chrome driver

WebDriver driver = new ChromeDriver();

//Step#2- Launching URL

driver.get("https://demoqa.com/select-menu");

//Maximizing window

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

//Step#3- Selecting the dropdown element by locating its id

Select select = new Select(driver.findElement(By.id("oldSelectMenu")));

//Step#4- Printing the options of the dropdown

//Get list of web elements

List<WebElement> lst = select.getOptions();

//Looping through the options and printing dropdown options

System.out.println("The dropdown options are:");

for(WebElement options: lst)

System.out.println(options.getText());

//Step#5- Selecting the option as 'Purple'-- selectByIndex

System.out.println("Select the Option by Index 4");

select.selectByIndex(4);

System.out.println("Select value is: " + select.getFirstSelectedOption().getText());

//Step#6- Selecting the option as 'Magenta'-- selectByVisibleText

System.out.println("Select the Option by Text Magenta");

select.selectByVisibleText("Magenta");

System.out.println("Select value is: " + select.getFirstSelectedOption().getText());

//Step#7- Selecting an option by its value

System.out.println("Select the Option by value 6");

select.selectByValue("6");

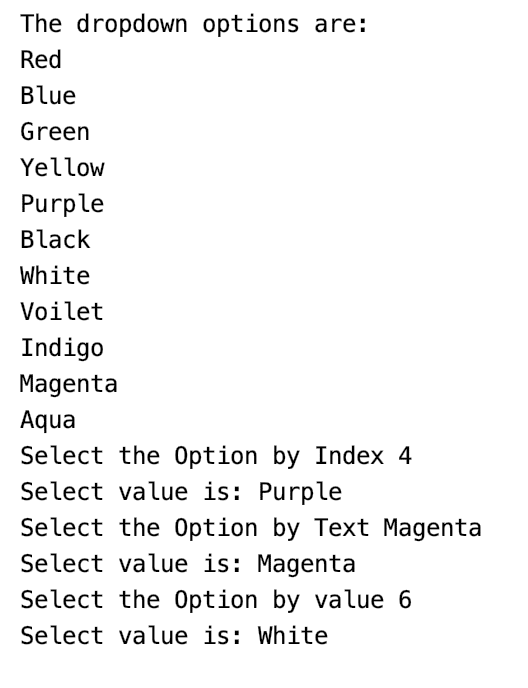
System.out.println("Select value is: " + select.getFirstSelectedOption().getText());

driver.quit();

}

}

On executing the code, you will notice that the dropdown selections are made as per the select method used, and the console window would print the options as shown below:



So this way, we can select and validate the values in a dropdown allowing single-select.

### *****Example 2- Handling multi-select using Selenium WebDriver*****

To automate multi-select using Selenium WebDriver's Select class, we will use the following use-case:

* Launch the browser.
* Open ***"https://demoqa.com/select-menu"***.
* Select the Standard Multi-Select using the element id.
* Verifying that the element is multi-select.
* Select 'Opel' using the index and deselect the same using index.
* Select 'Saab' using value and deselect the same using value.
* Deselect all the options.
* Close the browser.

We will use both the select and deselect methods of the Select class to automate the multi-select element. The code would look like below-

import java.util.List;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

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

public class MultiSelect {

public static void main(String[] args) throws InterruptedException {

//Creating instance of Chrome driver

WebDriver driver = new ChromeDriver();

// Navigate to the URL

driver.get("https://demoqa.com/select-menu");

//Maximizing window

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

//Selecting the multi-select element by locating its id

Select select = new Select(driver.findElement(By.id("cars")));

//Get the list of all the options

System.out.println("The dropdown options are -");

List<WebElement> options = select.getOptions();

for(WebElement option: options)

System.out.println(option.getText());

//Using isMultiple() method to verify if the element is multi-select, if yes go onto next steps else eit

if(select.isMultiple()){

//Selecting option as 'Opel'-- ByIndex

System.out.println("Select option Opel by Index");

select.selectByIndex(2);

//Selecting the option as 'Saab'-- ByValue

System.out.println("Select option saab by Value");

select.selectByValue("saab");

// Selecting the option by text

System.out.println("Select option Audi by Text");

select.selectByVisibleText("Audi");

//Get the list of selected options

System.out.println("The selected values in the dropdown options are -");

List<WebElement> selectedOptions = select.getAllSelectedOptions();

for(WebElement selectedOption: selectedOptions)

System.out.println(selectedOption.getText());

// Deselect the value "Audi" by Index

System.out.println("DeSelect option Audi by Index");

select.deselectByIndex(3);

//Deselect the value "Opel" by visible text

System.out.println("Select option Opel by Text");

select.deselectByVisibleText("Opel");

//Validate that both the values are deselected

System.out.println("The selected values after deselect in the dropdown options are -");

List<WebElement> selectedOptionsAfterDeselect = select.getAllSelectedOptions();

for(WebElement selectedOptionAfterDeselect: selectedOptionsAfterDeselect)

System.out.println(selectedOptionAfterDeselect.getText());

//Step#8- Deselect all values

select.deselectAll();

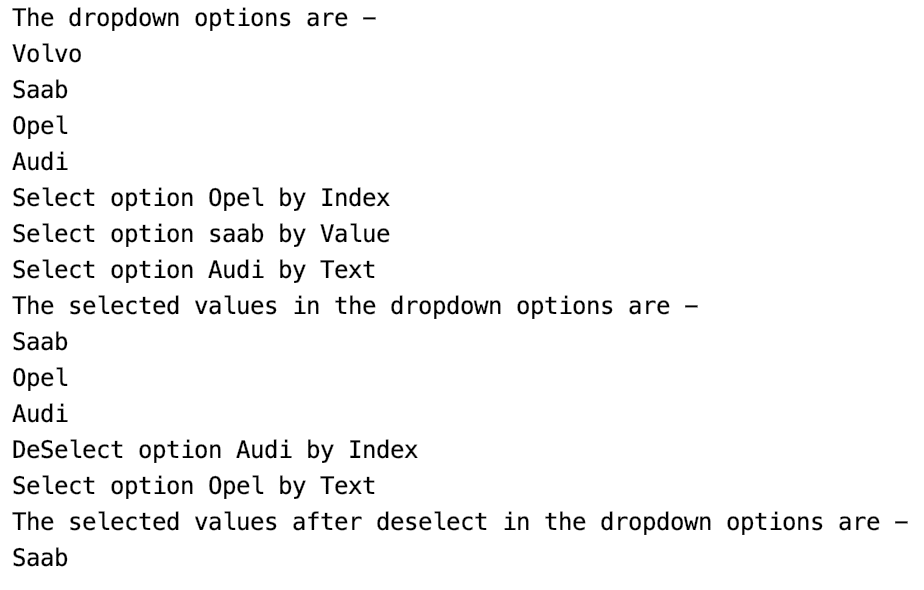
}

driver.quit();

}

}

The above code on execution would select and deselect multiple options from the multi-select and print the multi-select options, as shown below:



You can select multiple options, just like we selected all the options from the multi-select box as per your requirement. Now you can use the different Select class methods in Selenium automation and easily automate a dropdown or a multi-select box to ease your execution.

## **XPath axes methods:**

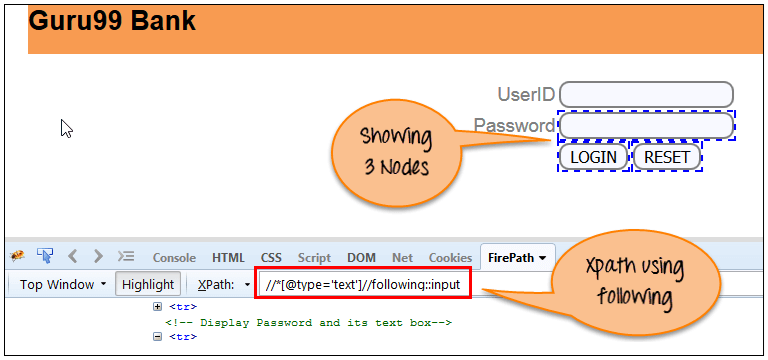
These XPath axes methods are used to find the complex or dynamic elements. Below we will see some of these methods.

For illustrating these XPath axes method, we will use the Guru99 bank demo site.

### 1) Following:

Selects all elements in the document of the current node( ) [ UserID input box is the current node] as shown in the below screen.

Xpath=//\*[@type='text']//following::input

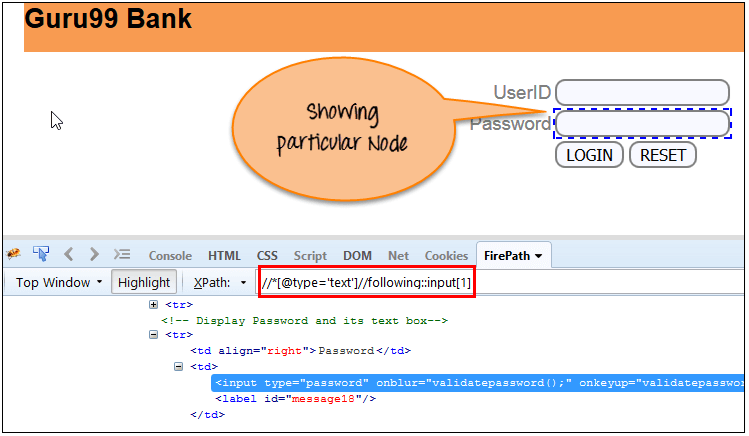


There are 3 “input” nodes matching by using “following” axis- password, login and reset button. If you want to focus on any particular element then you can use the below XPath method:

Xpath=//\*[@type='text']//following::input[1]

You can change the XPath according to the requirement by putting [1],[2]…………and so on.

With the input as ‘1’, the below screen shot finds the particular node that is ‘Password’ input box element.

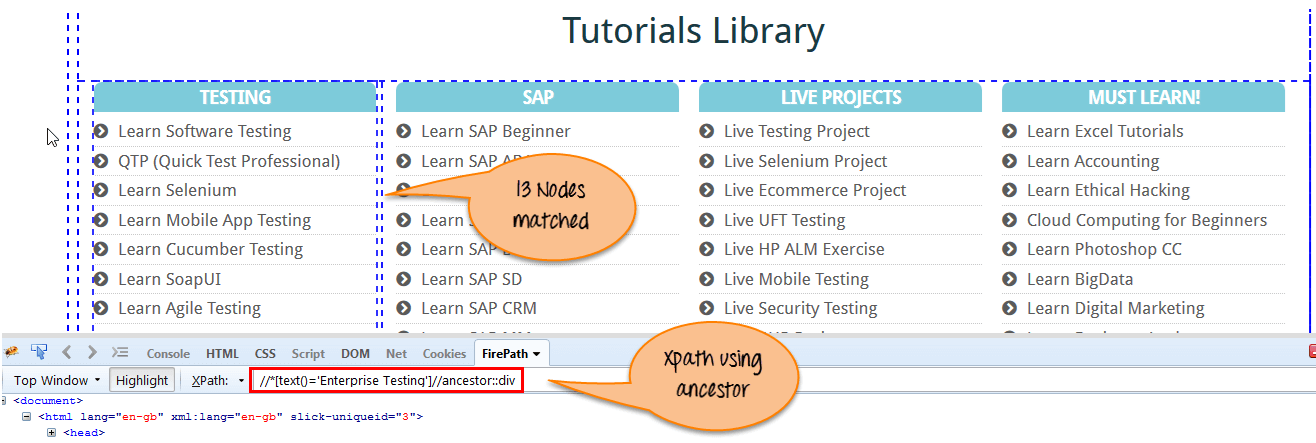


### 2) Ancestor:

The ancestor axis selects all ancestors element (grandparent, parent, etc.) of the current node as shown in the below screen.

In the below expression, we are finding ancestors element of the current node(“ENTERPRISE TESTING” node).

Xpath=//\*[text()='Enterprise Testing']//ancestor::div



There are 13 “div” nodes matching by using “ancestor” axis. If you want to focus on any particular element then you can use the below XPath, where you change the number 1, 2 as per your requirement:

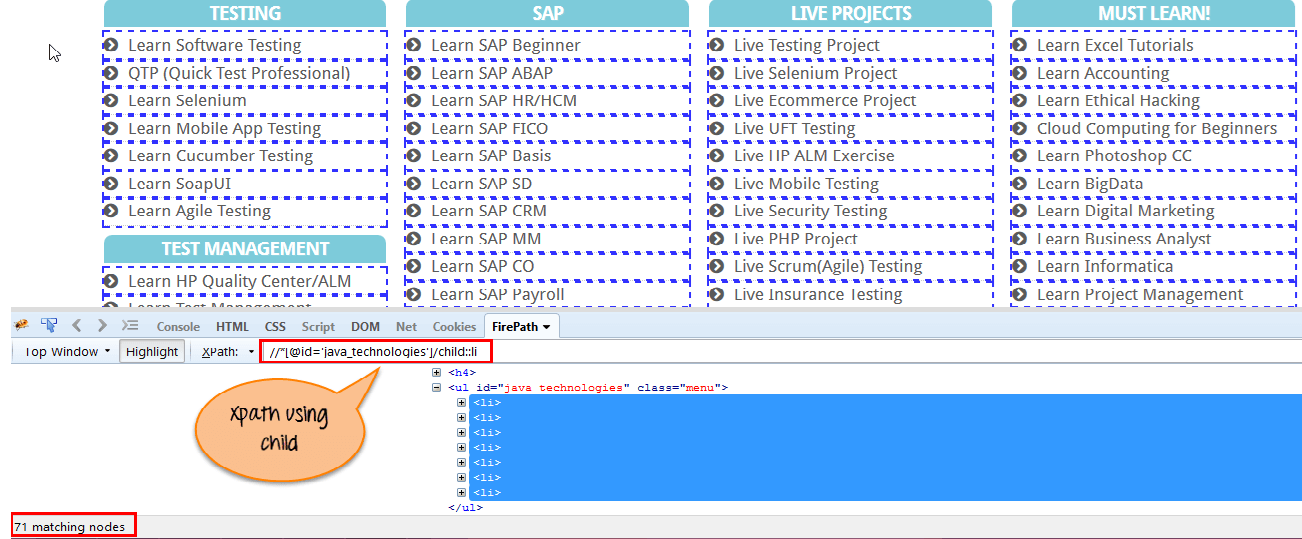
Xpath=//\*[text()='Enterprise Testing']//ancestor::div[1]

You can change the XPath according to the requirement by putting [1], [2]…………and so on.

### 3) Child:

Selects all children elements of the current node (Java) as shown in the below screen.

Xpath=//\*[@id='java\_technologies']//child::li



There are 71 “li” nodes matching by using “child” axis. If you want to focus on any particular element then you can use the below xpath:

Xpath=//\*[@id='java\_technologies']//child::li[1]

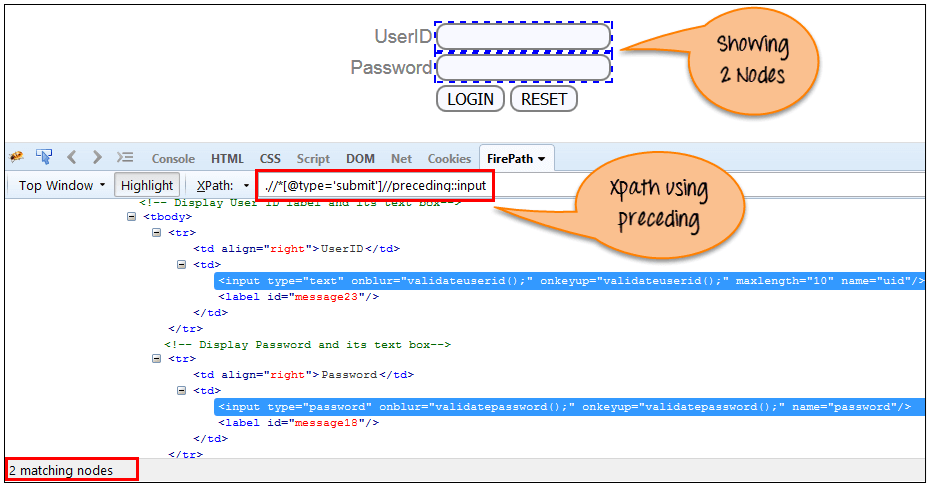
You can change the xpath according to the requirement by putting [1],[2]…………and so on.

### 4) Preceding:

Select all nodes that come before the current node as shown in the below screen.

In the below expression, it identifies all the input elements before “LOGIN” button that is **Userid** and **password** input element.

Xpath=//\*[@type='submit']//preceding::input



There are 2 “input” nodes matching by using “preceding” axis. If you want to focus on any particular element then you can use the below XPath:

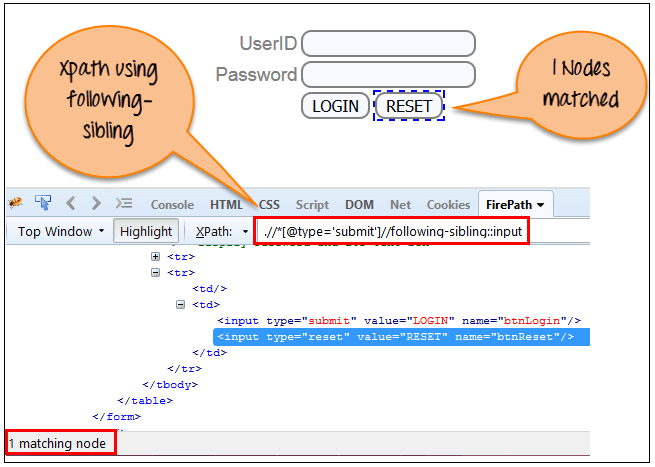
Xpath=//\*[@type='submit']//preceding::input[1]

You can change the xpath according to the requirement by putting [1],[2]…………and so on.

### 5) Following-sibling:

Select the following siblings of the context node. Siblings are at the same level of the current node as shown in the below screen. It will find the element after the current node.

xpath=//\*[@type='submit']//following-sibling::input

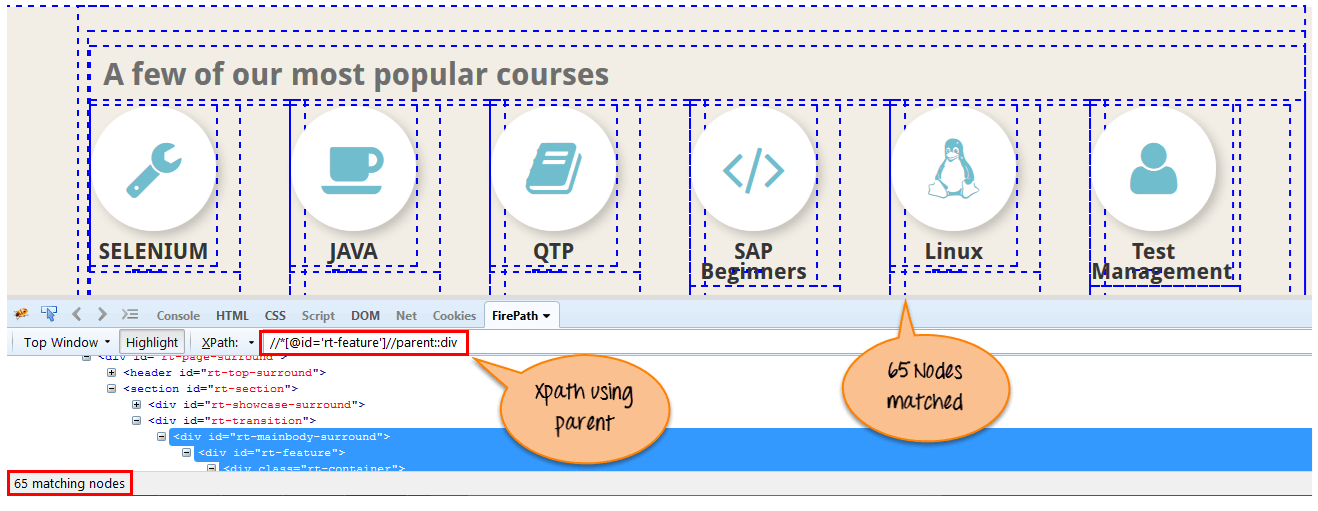


One input nodes matching by using “following-sibling” axis.

### 6) Parent:

Selects the parent of the current node as shown in the below screen.

Xpath=//\*[@id='rt-feature']//parent::div



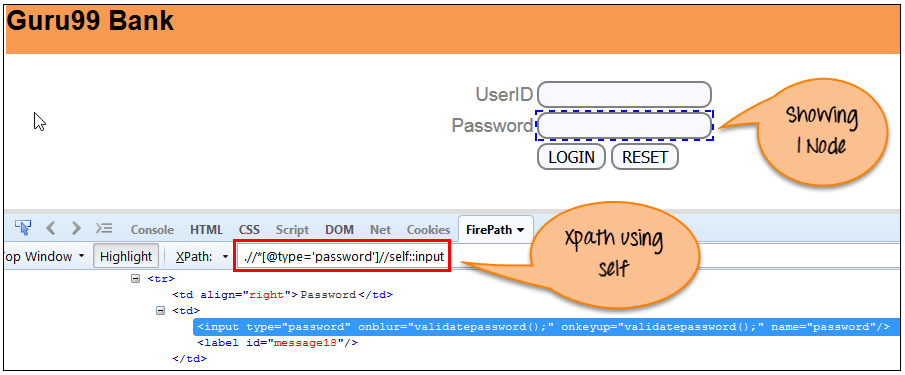
There are 65 “div” nodes matching by using “parent” axis. If you want to focus on any particular element then you can use the below XPath:

Xpath=//\*[@id='rt-feature']//parent::div[1]

You can change the XPath according to the requirement by putting [1],[2]…………and so on.

### 7) Self:

Selects the current node or ‘self’ means it indicates the node itself as shown in the below screen.



One node matching by using “self ” axis. It always finds only one node as it represents self-element.

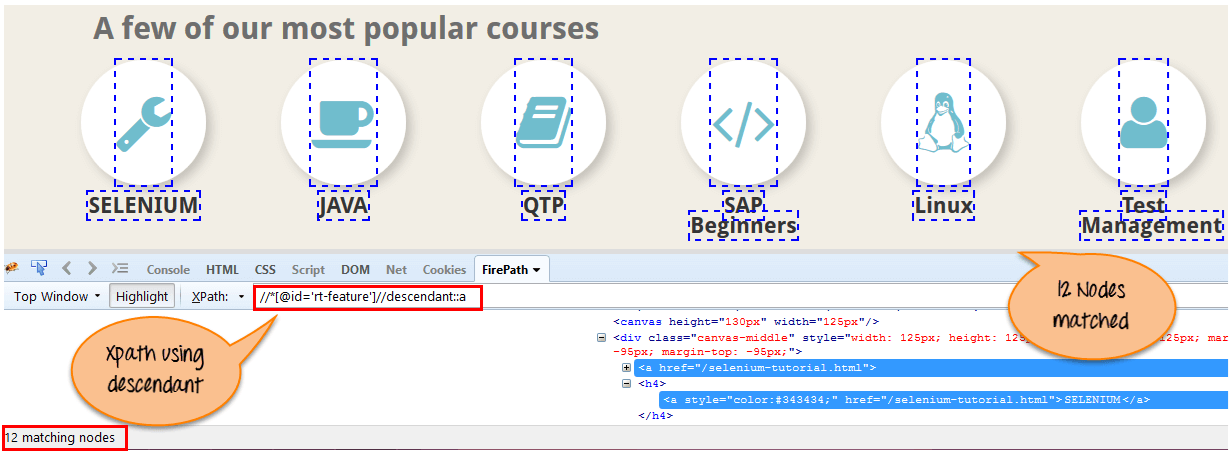
Xpath =//\*[@type='password']//self::input

### 8) Descendant:

Selects the descendants of the current node as shown in the below screen.

In the below expression, it identifies all the element descendants to current element ( ‘Main body surround’ frame element) which means down under the node (child node , grandchild node, etc.).

Xpath=//\*[@id='rt-feature']//descendant::a



There are 12 “link” nodes matching by using “descendant” axis. If you want to focus on any particular element then you can use the below XPath:

Xpath=//\*[@id='rt-feature']//descendant::a[1]

You can change the XPath according to the requirement by putting [1],[2]…………and so on.

#### Summary:

XPath is required to find an element on the web page as to do an operation on that particular element.

* There are two types of selenium XPath:
  + **Absolute XPath**
  + **Relative XPath**
* XPath Axes are the methods used to find dynamic elements, which otherwise not possible to find by normal XPath method
* XPath expression select nodes or list of nodes on the basis of attributes like ID , Name, Classname, etc. from the XML document .

**Alert and Pop Up in Selenium**

An **Alert in Selenium** is a small message box which appears on screen to give the user some information or notification. It notifies the user with some specific information or error, asks for permission to perform certain tasks and it also provides warning messages as well.

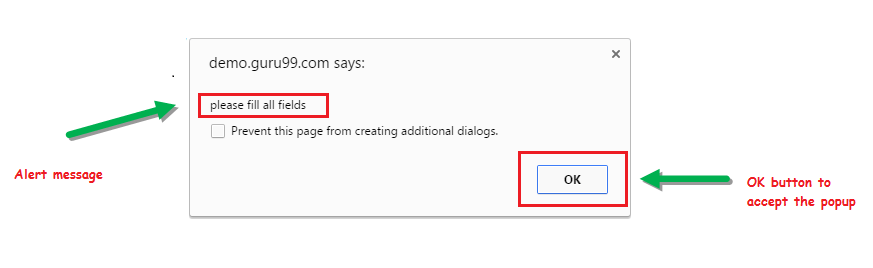
In this tutorial, we will learn how to handle popup in Selenium and different types of alerts found in web application [Testing](https://www.guru99.com/software-testing.html). We will also see how to handle Alert in Selenium WebDriver and learn how do we accept and reject the alert depending upon the alert types.

**Table of Content:**

## Types of Alerts in Selenium

### 1) Simple Alert

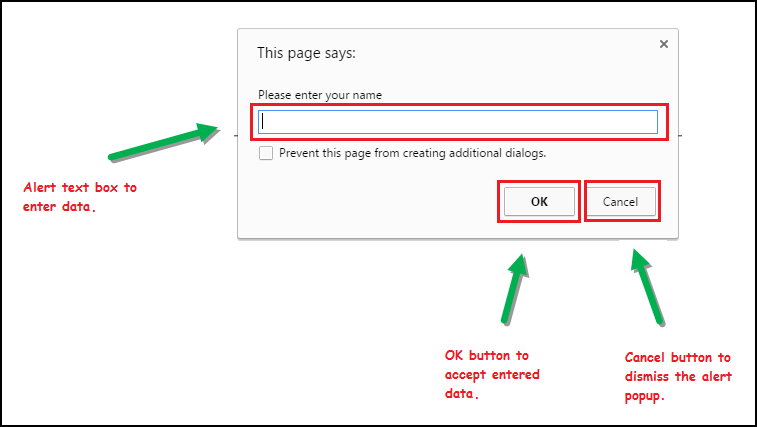
The simple alert class in Selenium displays some information or warning on the screen.



EXPLORE MORELearn Java Programming with Beginners Tutorial08:32Linux Tutorial for Beginners: Introduction to Linux Operating...01:35What is Integration Testing Software Testing Tutorial03:04What is JVM (Java Virtual Machine) with Architecture JAVA...02:24How to write a TEST CASE Software Testing Tutorial01:08Seven Testing Principles Software Testing05:01Linux File Permissions Commands with Examples13:29How to use Text tool in Photoshop CC Tutorial08:32What is NoSQL Database Tutorial02:00Important Linux Commands for Beginners Linux Tutorial15:03

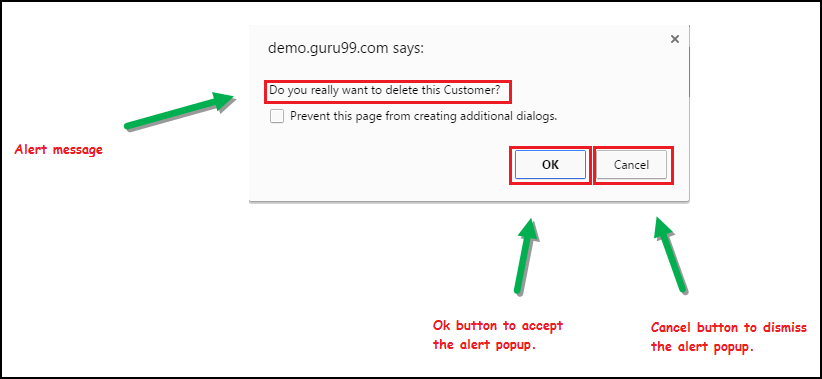
### 2) Prompt Alert

This Prompt Alert asks some input from the user and Selenium webdriver can enter the text using sendkeys(” input…. “).



### 3) Confirmation Alert

This confirmation alert asks permission to do some type of operation.



## How to handle Alert in Selenium WebDriver

Alert interface provides the below few methods which are widely used in [Selenium Webdriver](https://www.guru99.com/introduction-webdriver-comparison-selenium-rc.html).

1) **void dismiss()** // To click on the ‘Cancel’ button of the alert.

driver.switchTo().alert().dismiss();

2) **void accept()**// To click on the ‘OK’ button of the alert.

driver.switchTo().alert().accept();

3) **String getText()** // To capture the alert message.

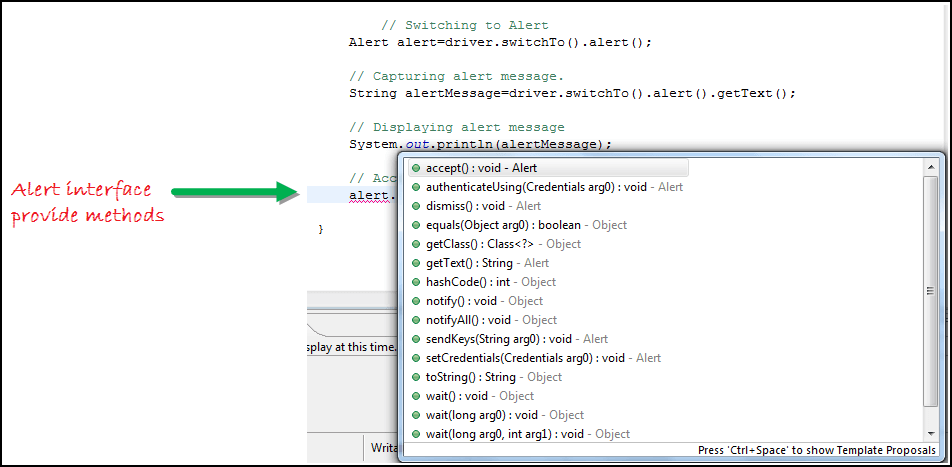
driver.switchTo().alert().getText();

4) **void sendKeys(String stringToSend)** // To send some data to alert box.

driver.switchTo().alert().sendKeys("Text");

You can see a number of Alert methods are displayed as shown in below screen suggested by Eclipse.

We can easily switch to alert from the main window by using Selenium’s **.switchTo()** method.

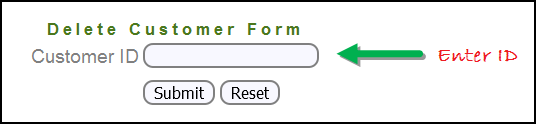


**Now we automate the given below scenario.**

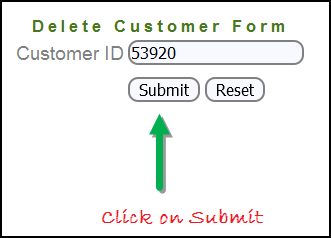
In this scenario, we will use Guru99 demo site to illustrate Selenium Alert handling.

**Step 1)** Launch the web browser and open the site <https://demo.guru99.com/test/delete_customer.php>

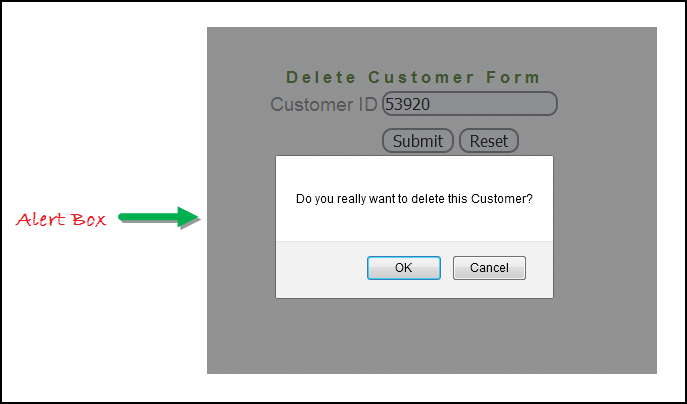
**Step 2)** Enter Any Customer id.



**Step 3)** After entering the customer ID, Click on the “Submit” button.



**Step 4)** Reject/accept the alert.



#### Handling Alert in Selenium Webdriver

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

import org.openqa.selenium.NoAlertPresentException;

import org.openqa.selenium.Alert;

public class AlertDemo {

public static void main(String[] args) throws NoAlertPresentException,InterruptedException {

System.setProperty("webdriver.chrome.driver","G:\\chromedriver.exe");

WebDriver driver = new ChromeDriver();

// Alert Message handling

driver.get("https://demo.guru99.com/test/delete\_customer.php");

driver.findElement(By.name("cusid")).sendKeys("53920");

driver.findElement(By.name("submit")).submit();

// Switching to Alert

Alert alert = driver.switchTo().alert();

// Capturing alert message.

String alertMessage= driver.switchTo().alert().getText();

// Displaying alert message

System.out.println(alertMessage);

Thread.sleep(5000);

// Accepting alert

alert.accept();

}

}

**Output:**

When you execute the above code, it launches the site. Try to delete Customer ID by handling confirmation alert that displays on the screen, and thereby deleting customer id from the application.

## How to handle Pop-up window using Selenium Webdriver

In automation, when we have multiple windows in any web application, the activity may need to switch control among several windows from one to other in order to complete the operation. After completion of the operation, it has to return to the main window i.e. parent window in Selenium. We will see this further in the article with an example.

In Selenium web driver there are methods through which we can handle multiple windows.

#### Driver.getWindowHandles();

To handle all opened windows by web driver, we can use “Driver.getWindowHandles()” and then we can switch window from one window to another in a web application. Its return type is Iterator<String>.

#### Driver.getWindowHandle();

When the site opens, we need to handle the main window by **driver.getWindowHandle()**. This will handle the current window that uniquely identifies it within this driver instance. Its return type is String.

For Window handling in Selenium, we will follow the below steps:

Now, we will automate the given below scenario to see how to handle multiple windows using Selenium Webdriver.

In this scenario, we will use “Guru99” demo site to illustrate window handling.

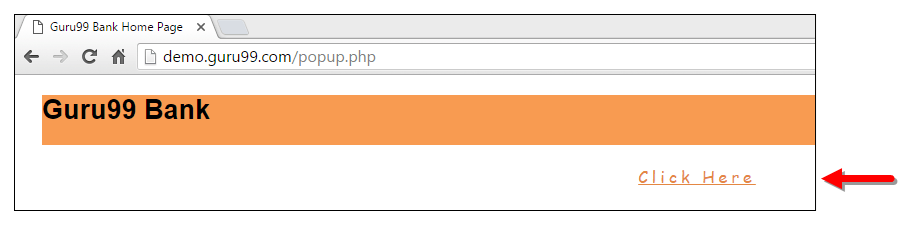
**Step 1)** Launch the site.

Launch the browser and open the site <https://demo.guru99.com/popup.php>



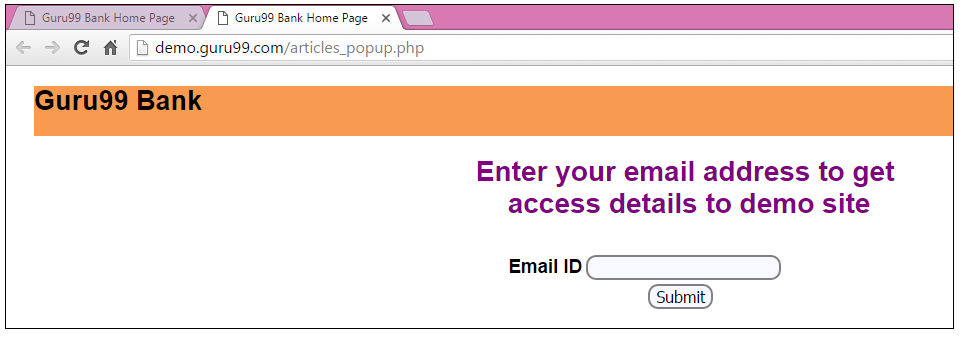
**Step 2)** Click on link “Click Here”.

When the user clicks on the ” Click Here ” link, new child window opens.

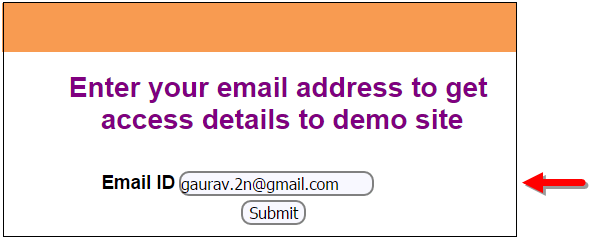


**Step 3)** New Child Window opens.

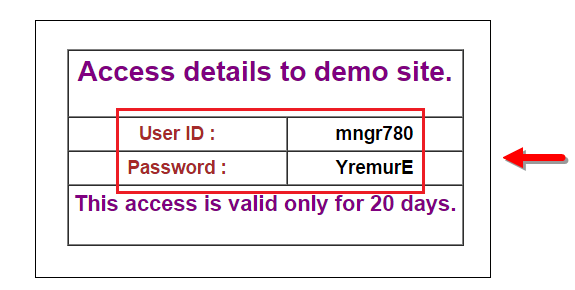
A new window opens, ask the user to enter email id and submit the page.



**Step 4)** Enter your email ID and submit.

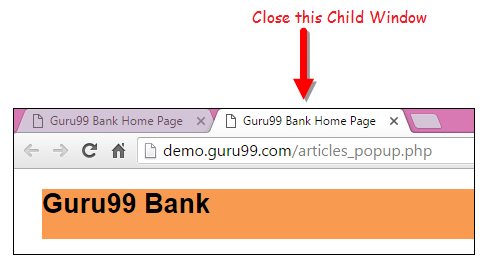


**Step 5)** Display the Access Credentials on submitting the page.

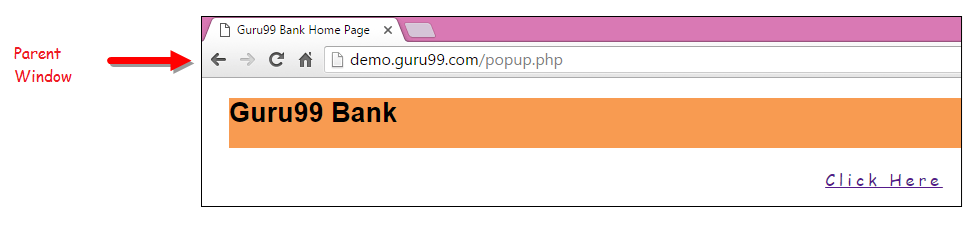


When you execute the code, you will see the child window is open in new tab.

1. Close the Child window on which credentials are displayed.



1. Switch to the parent window.



## How to Handle Multiple Windows in Selenium

import java.util.Iterator;

import java.util.Set;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

public class WindowHandle\_Demo {

public static void main(String[] args) throws InterruptedException {

WebDriver driver=new FirefoxDriver();

//Launching the site.

driver.get("https://demo.guru99.com/popup.php");

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

driver.findElement(By.xpath("//\*[contains(@href,'popup.php')]")).click();

String MainWindow=driver.getWindowHandle();

// To handle all new opened window.

Set<String> s1=driver.getWindowHandles();

Iterator<String> i1=s1.iterator();

while(i1.hasNext())

{

String ChildWindow=i1.next();

if(!MainWindow.equalsIgnoreCase(ChildWindow))

{

// Switching to Child window

driver.switchTo().window(ChildWindow);

driver.findElement(By.name("emailid"))

.sendKeys("gaurav.3n@gmail.com");

driver.findElement(By.name("btnLogin")).click();

// Closing the Child Window.

driver.close();

}

}

// Switching to Parent window i.e Main Window.

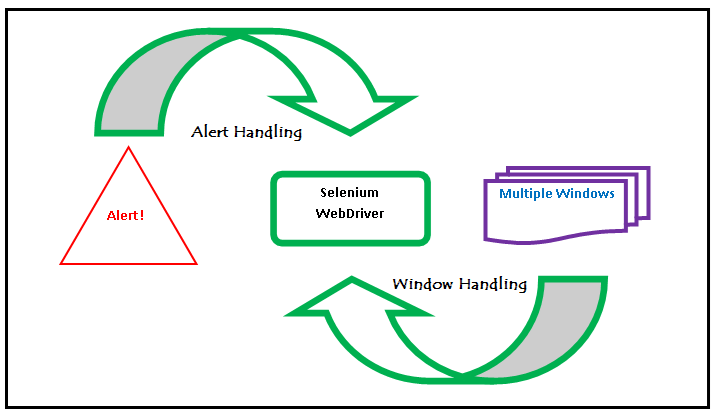
driver.switchTo().window(MainWindow);

}

}

**Output:**

When you execute the above code, it launches the site and on clicking the link “Click here,” it opens up a child window in a new tab. You can close the child window, and switch to the parent window once the operation is completely done. Hence handling more than one window in the application.



**Handling Frames in Selenium**

**iFrame in Selenium Webdriver** is a web page or an inline frame which is embedded in another web page or an HTML document embedded inside another HTML document. The iframe is often used to add content from other sources like an advertisement into a web page. The iframe is defined with the <**iframe**> tag.

## How to Handle Frames in Selenium using WebDriver Commands

Basically, we can switch over the elements and handle frames in Selenium using 3 ways.

* **By Index**
* **By Name or Id**
* **By Web Element**

### Method 1: Switch to the frame by index

Index is one of the attributes for frame handling in Selenium through which we can switch to it.

Index of the iframe starts with ‘0’.

Suppose if there are 100 frames in page, we can switch to frame in Selenium by using index.

* driver.switchTo().frame(0);
* driver.switchTo().frame(1);

### Method 2: Switch to the frame by Name or ID

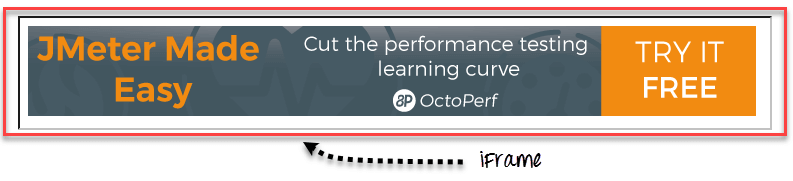
Name and ID are attributes for handling frames in Selenium through which we can switch to the iframe.

* driver.switchTo().frame("iframe1");
* driver.switchTo().frame("id of the element");

**Example of Switching to iframe through ID:**

to the frame by Web Element

We can access this iframe through this below URL: <https://demo.guru99.com/test/guru99home/>



It is impossible to click iframe directly through[XPath](https://www.guru99.com/xpath-selenium.html)since it is an iframe. First we have to switch to the frame and then we can click using xpath.

**Step 1)**

WebDriver driver = new FirefoxDriver();

driver.get("https://demo.guru99.com/test/guru99home/");

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

* We initialise the Firefox driver.
* Navigate to the “guru99” site which consist the iframe.
* Maximized the window.

**Step 2)**

driver.switchTo().frame("a077aa5e");

* In this step we need to find out the id of the iframe by inspecting through Firebug.
* Then switch to the iframe through ID.

**Step 3)**

driver.findElement(By.xpath("html/body/a/img")).click();

* Here we need to find out the xpath of the element to be clicked.
* Click the element using web driver command shown above.

**Here is the complete code:**

public class SwitchToFrame\_ID {

public static void main(String[] args) {

WebDriver driver = new FirefoxDriver(); //navigates to the Browser

driver.get("https://demo.guru99.com/test/guru99home/");

// navigates to the page consisting an iframe

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

driver.switchTo().frame("a077aa5e"); //switching the frame by ID

System.out.println("\*\*\*\*\*\*\*\*We are switch to the iframe\*\*\*\*\*\*\*");

driver.findElement(By.xpath("html/body/a/img")).click();

//Clicks the iframe

System.out.println("\*\*\*\*\*\*\*\*\*We are done\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

**Output:**

Browser navigates to the page consisting the above iframe and clicks on the iframe.

### Method 3: Switch to the frame by Web Element

We can even switch to the iframe using web element .

* driver.switchTo().frame(WebElement);

**How to switch back to the Main Frame**

We have to come out of the iframe.

To move back to the parent frame, you can either use switchTo().parentFrame() or if you want to get back to the main (or most parent) frame, you can use switchTo().defaultContent();

driver.switchTo().parentFrame();

driver.switchTo().defaultContent();

**How to switch over the frame, if we CANNOT switch using ID or Web Element:**

Suppose if there are 100 frames in the page, and there is no ID available, in this case, we just don’t know from which iframe required element is being loaded (It is the case when we do not know the index of the frame also).

The solution for the above concern is, we must find the index of the iframe through which the element is being loaded and then we need to switch to the iframe through the index.

Below are the steps for finding the index of the Frame by which the element is being loaded by using below snippet

**Step 1)**

WebDriver driver = new FirefoxDriver();

driver.get("https://demo.guru99.com/test/guru99home/");

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

* Initialise the Firefox driver.
* Navigate to the “guru99” site which consisting the iframe.
* Maximized the window.

**Step 2)**

int size = driver.findElements(By.tagName("iframe")).size();

* The above code finds the total number of iframes present inside the page using the tagname ‘iframe’.

**Step 3)**

**Objective for**this step would be finding out the index of iframe.

for(int i=0; i<=size; i++){

driver.switchTo().frame(i);

int total=driver.findElements(By.xpath("html/body/a/img")).size();

System.out.println(total);

driver.switchTo().defaultContent();}

Above “forloop” iterates all the iframes in the page and it prints ‘1’ if our required iframe was found else returns ‘0’.

**Here is the complete code till step 3:**

public class IndexOfIframe {

public static void main(String[] args) {

WebDriver driver = new FirefoxDriver();

driver.get("https://demo.guru99.com/test/guru99home/");

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

//driver.manage().timeouts().implicitlyWait(100, TimeUnit.SECONDS);

int size = driver.findElements(By.tagName("iframe")).size();

for(int i=0; i<=size; i++){

driver.switchTo().frame(i);

int total=driver.findElements(By.xpath("html/body/a/img")).size();

System.out.println(total);

driver.switchTo().defaultContent();}}}

**Execute this program and output would be like below:**

**Output:**

1

0

0

0

0

0

Verify the output, you can find the series of 0’s and 1’s.

* Wherever you find the ‘1’ in output that is the index of Frame by which the element is being loaded.
* Since the index of the iframe starts with ‘0’ if you find the 1 in the 1stplace, then the index is 0.
* If you find 1 in 3rd place, the index is 2.

We can comment out the for loop, once we found the index.

**Step 4)**

driver.switchTo().frame(0);

* Once you find the index of the element, you can switch over the frame using above command.
* driver.switchTo().frame(index found from the Step 3);

**Step 5)**

driver.findElement(By.xpath("html/body/a/img")).click();

* The above code will clicks the iframe or element in the iframe.

**So the complete code would be like below:**

public class SwitchToframe {

public static void main(String[] args) throws NoSuchElementException{

WebDriver driver = new FirefoxDriver();

driver.get("https://demo.guru99.com/test/guru99home/");

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

//int size = driver.findElements(By.tagName("iframe")).size();

/\*for(int i=0; i<=size; i++){

driver.switchTo().frame(i);

int total=driver.findElements(By.xpath("html/body/a/img")).size();

System.out.println(total);

driver.switchTo().defaultContent(); //switching back from the iframe

}\*/

//Commented the code for finding the index of the element

driver.switchTo().frame(0); //Switching to the frame

System.out.println("\*\*\*\*\*\*\*\*We are switched to the iframe\*\*\*\*\*\*\*");

driver.findElement(By.xpath("html/body/a/img")).click();

//Clicking the element in line with Advertisement

System.out.println("\*\*\*\*\*\*\*\*\*We are done\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

**Output:**

Browser navigates to the page consisting the above iframe and clicks on the iframe.

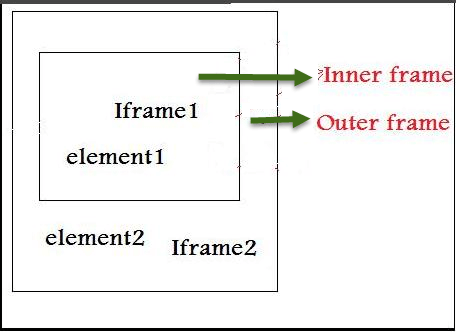
## **Concept of Nested Frames in Selenium:**

Let’s assume that there are two frames one inside other like shown in below image and our requirement is printing the text in the outer frame and inner frame.

**In the case of nested frames,**

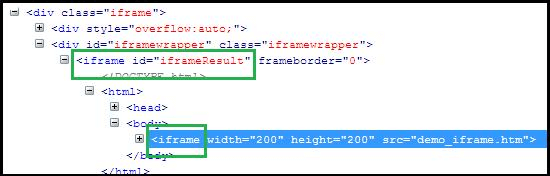
* At first we must switch to the outer frame by either Index or ID of the iframe
* Once we switch to the outer frame we can find the total number of iframes inside the outer frame, and
* We can switch to the inner frame by any of the known methods.

While exiting out of the frame, we must exit out in the same order as we entered into it from the inner frame first and then outer frame.



Nested iFrames in Selenium WebDriver

**The Html code for the above nested frame is as shown below.**



The above HTML code clearly explains the iframe tag (highlighted in green) within another iframe tag, indicating presence of nested iframes.

Below are the steps for switching to outer frame and printing the text on outer frames:

**Step 1)**

WebDriver driver=new FirefoxDriver();

driver.get("Url");

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

driver.manage().timeouts().implicitlyWait(2, TimeUnit.SECONDS);

int size = driver.findElements(By.tagName("iframe")).size();

System.out.println("Total Frames --" + size);

// prints the total number of frames

driver.switchTo().frame(0); // Switching the Outer Frame

System.out.println (driver.findElement(By.xpath("xpath of the outer element ")).getText());

* Switch to the outer Frame.
* Prints the text on outer frame.

Once we switch to the outer frame, we should know whether any inner frame present inside the outer frame

**Step 2)**

size = driver.findElements(By.tagName("iframe")).size();

// prints the total number of frames inside outer frame

System.out.println("Total Frames --" + size);

* Finds the total number of iframes inside outer frame.
* If size was found ‘0’ then there is no inner frame inside the frame.

**Step 3)**

driver.switchTo().frame(0); // Switching to innerframe

System.out.println(driver.findElement(By.xpath("xpath of the inner element ")).getText());

* Switch to the inner frame
* Prints the text on the inner frame.

**Here is the complete code:**

public class FramesInsideFrames {

public static void main(String[] args) {

WebDriver driver=new FirefoxDriver();

driver.get("Url");

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

driver.manage().timeouts().implicitlyWait(2, TimeUnit.SECONDS);

int size = driver.findElements(By.tagName("iframe")).size();

System.out.println("Total Frames --" + size);

// prints the total number of frames

driver.switchTo().frame(0); // Switching the Outer Frame

System.out.println (driver.findElement(By.xpath("xpath of the outer element ")).getText());

//Printing the text in outer frame

size = driver.findElements(By.tagName("iframe")).size();

// prints the total number of frames inside outer frame

System.out.println("Total Frames --" + size);

driver.switchTo().frame(0); // Switching to innerframe

System.out.println(driver.findElement(By.xpath("xpath of the inner element ")).getText());

//Printing the text in inner frame

driver.switchTo().defaultContent();

}

}

**Action Class in Selenium**

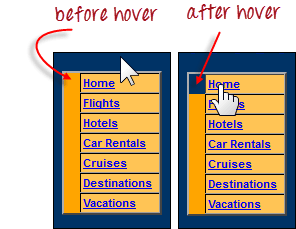
**Action Class in Selenium** is a built-in feature provided by the selenium for handling keyboard and mouse events. It includes various operations such as multiple events clicking by control key, drag and drop events and many more. These operations from the action class are performed using the advanced user interaction API in Selenium Webdriver.

## Handling Keyboard & Mouse Events

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

|  |  |
| --- | --- |
| **Method** | **Description** |
| **clickAndHold()** | Clicks (without releasing) at the current mouse location. |
| **contextClick()** | Performs a context-click at the current mouse location. (Right Click Mouse Action) |
| **doubleClick()** | Performs a double-click at the current mouse location. |
| **dragAndDrop(source, target)** | Performs click-and-hold at the location of the source element, moves to the location of the target element, then releases the mouse.  **Parameters:**  source- element to emulate button down at.  target- element to move to and release the mouse at. |
| **dragAndDropBy(source, x-offset, y-offset)** | Performs click-and-hold at the location of the source element, moves by a given offset, then releases the mouse.  **Parameters**:  source- element to emulate button down at.  xOffset- horizontal move offset.  yOffset- vertical move offset. |
| **keyDown(modifier\_key)** | Performs a modifier key press. Does not release the modifier key – subsequent interactions may assume it’s kept pressed.  **Parameters**:  modifier\_key – any of the modifier keys (Keys.ALT, Keys.SHIFT, or Keys.CONTROL) |
| **keyUp(modifier \_key)** | Performs a key release.  **Parameters**:  modifier\_key – any of the modifier keys (Keys.ALT, Keys.SHIFT, or Keys.CONTROL) |
| **moveByOffset(x-offset, y-offset)** | Moves the mouse from its current position (or 0,0) by the given offset.  **Parameters**:  x-offset- horizontal offset. A negative value means moving the mouse left.  y-offset- vertical offset. A negative value means moving the mouse down. |
| **moveToElement(toElement)** | Moves the mouse to the middle of the element.  **Parameters**:  toElement- element to move to. |
| **release()** | Releases the depressed left mouse button at the current mouse location |
| **sendKeys(onElement, charsequence)** | Sends a series of keystrokes onto the element.  **Parameters**:  onElement – element that will receive the keystrokes, usually a text field  charsequence – any string value representing the sequence of keystrokes to be sent |

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



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

**Step 1:**Import the **Actions** and **Action** classes.

Keyboard & Mouse Event using Action Class in Selenium Webdriver

**Step 2:**Instantiate a new Actions object.

Keyboard & Mouse Event using Action Class in Selenium Webdriver

**Step 3:**Instantiate an Action using the Actions object in step 2.

Keyboard & Mouse Event using Action Class in Selenium Webdriver

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

**Step 4:**Use the perform() method when executing the Action object we designed in Step 3.

Keyboard & Mouse Event using Action Class in Selenium Webdriver

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

package newproject;

import org.openqa.selenium.\*;

import org.openqa.selenium.firefox.FirefoxDriver;

import org.openqa.selenium.interactions.Action;

import org.openqa.selenium.interactions.Actions;

public class PG7 {

public static void main(String[] args) {

String baseUrl = "http://demo.guru99.com/test/newtours/";

System.setProperty("webdriver.gecko.driver","C:\\geckodriver.exe");

WebDriver driver = new FirefoxDriver();

driver.get(baseUrl);

WebElement link\_Home = driver.findElement(By.linkText("Home"));

WebElement td\_Home = driver

.findElement(By

.xpath("//html/body/div"

+ "/table/tbody/tr/td"

+ "/table/tbody/tr/td"

+ "/table/tbody/tr/td"

+ "/table/tbody/tr"));

Actions builder = new Actions(driver);

Action mouseOverHome = builder

.moveToElement(link\_Home)

.build();

String bgColor = td\_Home.getCssValue("background-color");

System.out.println("Before hover: " + bgColor);

mouseOverHome.perform();

bgColor = td\_Home.getCssValue("background-color");

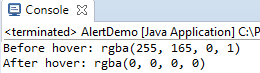
System.out.println("After hover: " + bgColor);

driver.close();

}

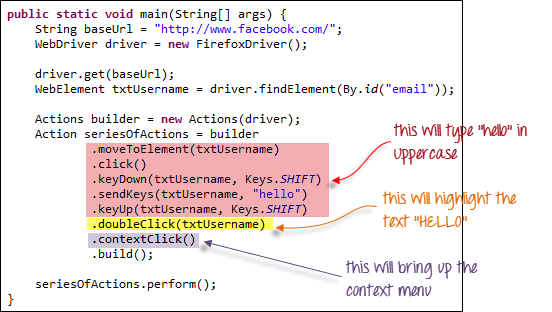
}

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



## Building a Series of Multiple Actions

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



public static void main(String[] args) {

String baseUrl = "http://www.facebook.com/";

WebDriver driver = new FirefoxDriver();

driver.get(baseUrl);

WebElement txtUsername = driver.findElement(By.id("email"));

Actions builder = new Actions(driver);

Action seriesOfActions = builder

.moveToElement(txtUsername)

.click()

.keyDown(txtUsername, Keys.SHIFT)

.sendKeys(txtUsername, "hello")

.keyUp(txtUsername, Keys.SHIFT)

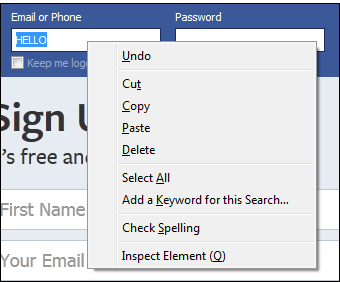
.doubleClick(txtUsername)

.contextClick()

.build();

seriesOfActions.perform() ;

}



### Summary

* Handling special keyboard and mouse events are done using the AdvancedUserInteractions API.
* Frequently used Keyword and Mouse Events are doubleClick(), keyUp, dragAndDropBy, contextClick & sendKeys.

**Dynamic WebTable in Selenium**

* A table with rows and columns on a web page is called a web table. In HTML, a table is created using  tag.
* Table contains  headers represented as, rows asand Coloums as tag
* There are two types of web Tables

1. Static web table**:** Number of rows and columns will be static means constant that will not be changed. Here we just need to find the WebElement of the particular row or column and simply can perform operations on the cell.
2. **Dynamic table:** Number of rows and columns will be dynamic. It will be kept on changing based on data and this the area where we will be focussing on.

**3. Ways to Handle Dynamic WebTable:**

1. The standard way where we will iterate over the rows and columns and reach to a particular cell and get the cell value.
2. By using custom XPath

Before proceeding further, let’s see how dynamic table looks like :

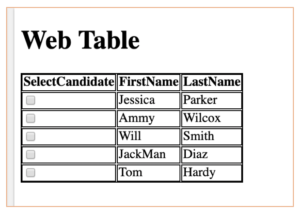


Table with checkbox , rows and coloumns

And below is the HTML representation of the above web table:



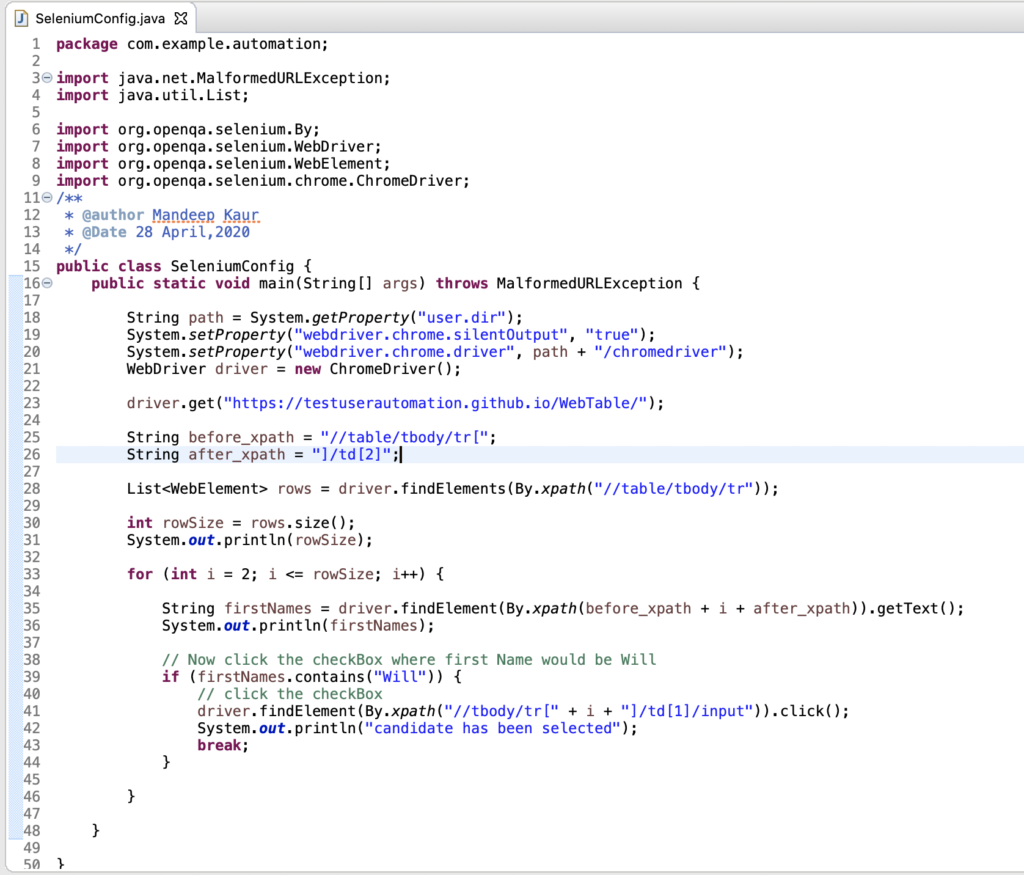
So, based on the data input we can increase or decrease our number of rows or columns.

Since we have seen what a web table is and how it is created, now let’s take the simple scenario :

1)click on checkbox where firstName is Will and remember it’s a dynamic table, hence “WILL” now on 3rd row can be shifted to other row based on the insertion or deletion of data.

* **Let’s implement it by First Method:**

Let’s accomplish this task first by looking at the code snippet below



**We’ll break this code into steps to get a deeper understanding of it:**

Step1 : Capture xpath of the FirstName Row and observe the pattern

* First Row xpath: //table/tbody/tr[2]/td[2] –>jessica
* second Row xpath: //table/tbody/tr[3]/td[2] –>Ammy
* Third Row xpath: //table/tbody/tr[4]/td[2] –>Will and so on
* Here can we say that our rowis getting changed and our columnis constant.

Step2: Catch before and after XPath from the point our row started changing as done in the code.

Step 3: Now, we have captured the before and after XPath.Find out the total row size by size() function of the java collections.

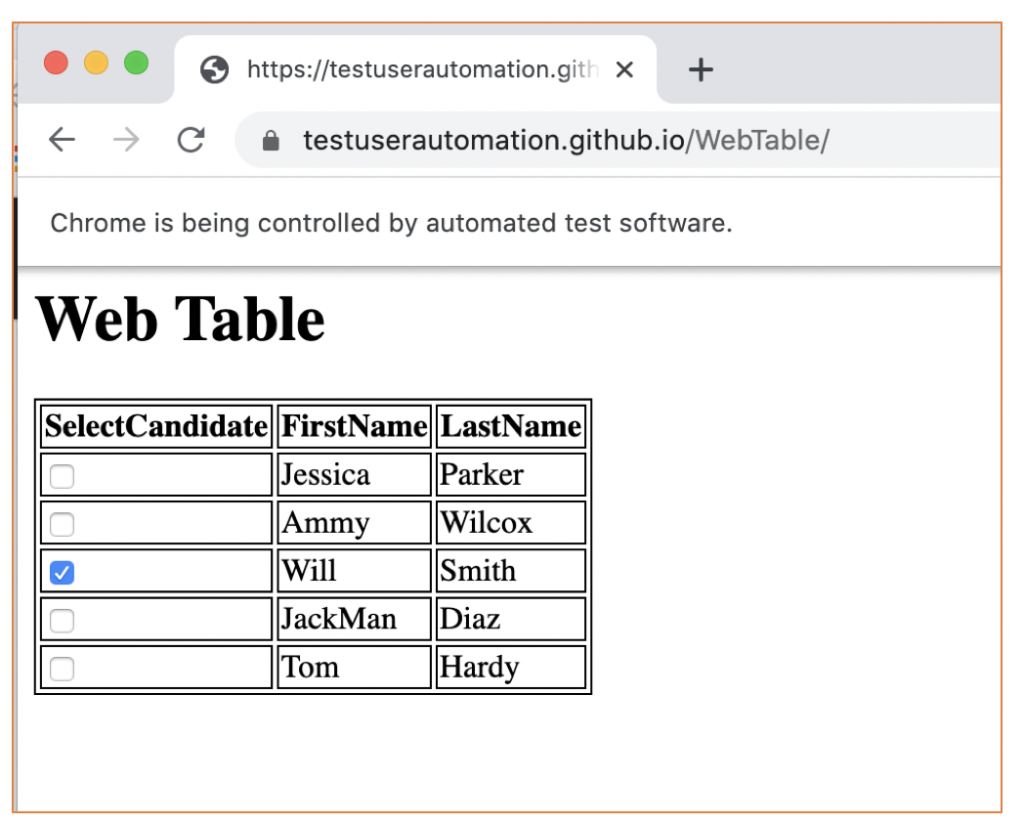
Step 4: Start iterating over the rows, start the row index with 2 as the first row would be reserved for headers.

Step 5:Get the text of all the rows and store it to the variable firstNames and print it to see if we are able to capture it

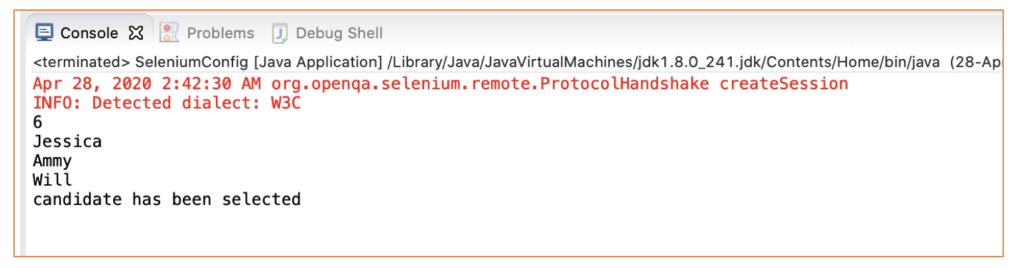
Step 6 : Final step of our task , that click the checkbox where FirstName is Will

For this step, we put one condition that if firstNames contains WILL then only checkbox has to be clicked. if it exists it will click the checkbox prior to WILL.

Since WILL is existing in the table , it will click the checkbox and we can capture it from the browser:



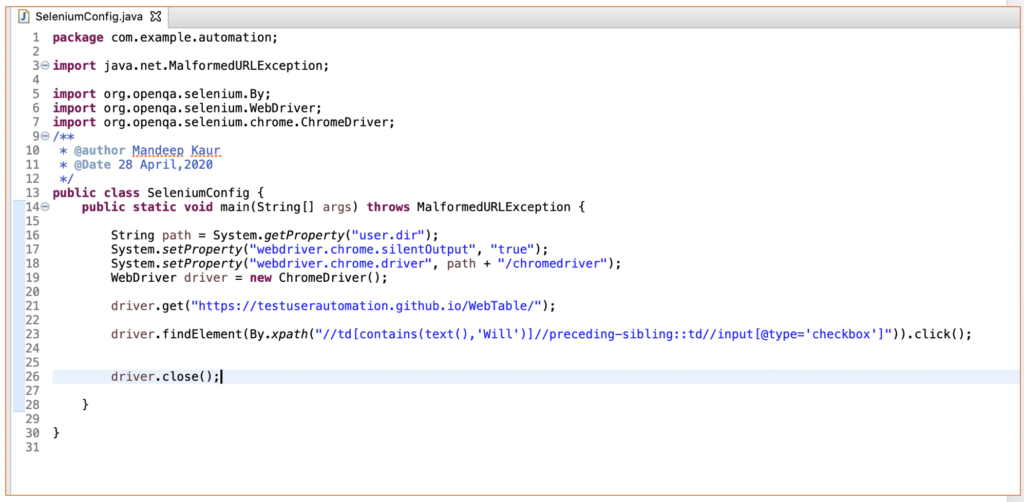
and our output on the console is similiar to following :

* 
* **By Second Method** : **with Custom XPath**

We can achieve our task with single statement :

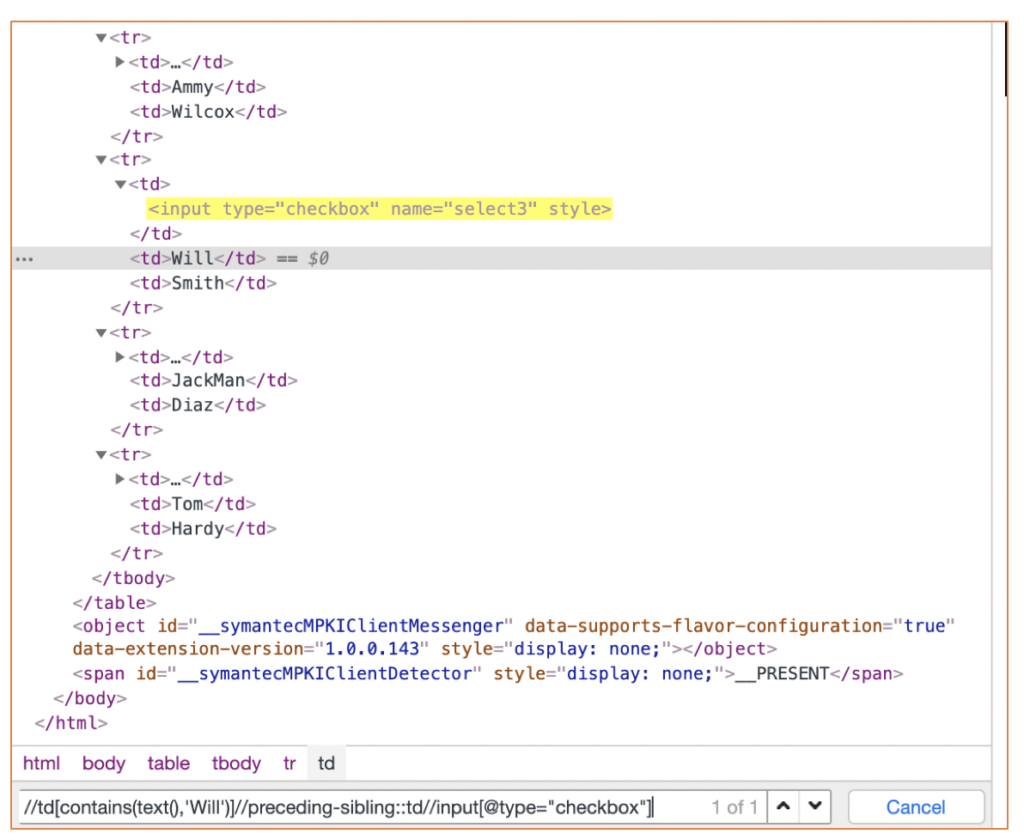
**driver.findElement(By.xpath(“//td[contains(text(),’Will’)]//preceding-sibling::td//input[@type=’checkbox’]”)).click();**

yes , that ‘it , below code for your reference

* 

Let’s try to understand this method :

firstly visit over [XPath\_Axes](https://www.w3schools.com/xml/xpath_axes.asp) , here you will get to know how to traverse from one node to another node.

* 

Next, in the above we have spied over FirstName with text Will and we found out Will Sibling is td when we proceed upwards and then we get our input as a checkbox and from here we can simply capture it.

**4. Which Method to choose**

As can be conveyed from the above methods, the custom method is the best way to handle web dynamic table :

Reasons :

1. In the First Method, it’s a lengthy method whereas the second method is just a single line statement
2. In the first Method, we had to use for loop with conditions and then selecting the value, on the other hand, we need not have to use for loop
3. As we are iterating through till all the rows it makes code slower in the first Method considering the other method can be seen fast, dynamic and efficient

**Code Snippet with both methods :**

package com.example.automation;  
import java.util.List;  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.chrome.ChromeDriver;  
/\*\*  
@author Mandeep Kaur  
@Date 28 April,2020  
\*/  
public class SeleniumConfig {  
public static void main(String[] args) {  
String path = System.getProperty("user.dir"); System.setProperty("webdriver.chrome.silentOutput", "true"); System.setProperty("webdriver.chrome.driver", path + "/chromedriver");  
WebDriver driver = new ChromeDriver(); driver.get("https://testuserautomation.github.io/WebTable/");  
// Method 1  
String before\_xpath = "//table/tbody/tr["; String after\_xpath = "]/td[2]";  
List rows = driver.findElements(By.xpath("//table/tbody/tr"));  
int rowSize = rows.size();  
System.out.println(rowSize);  
for (int i = 2; i <= rowSize; i++) {  
String firstNames = driver.findElement(By.xpath(before\_xpath + i + after\_xpath)).getText();  
System.out.println(firstNames);  
// Now click the checkBox where first Name would be Will  
if (firstNames.contains("Will")) {  
// click the checkBox  
driver.findElement(By.xpath("//tbody/tr[" + i + "]/td[1]/input")).click(); System.out.println("candidate has been selected");  
break;  
}  
}  
// Method 2 : By custom xpath:  
driver.findElement(By.xpath("//td[contains(text(),'Will')]//preceding-sibling::td//input[@type='checkbox']")) .click();  
driver.close();  
}  
}

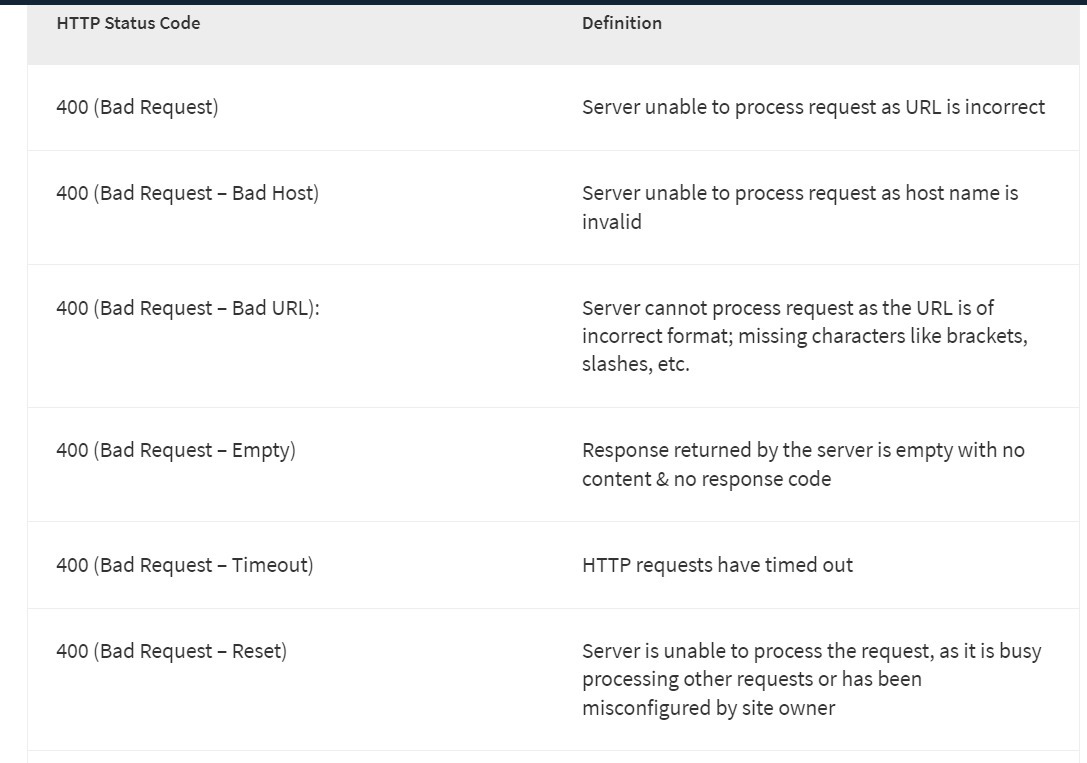
**Broken Links in Web Page**

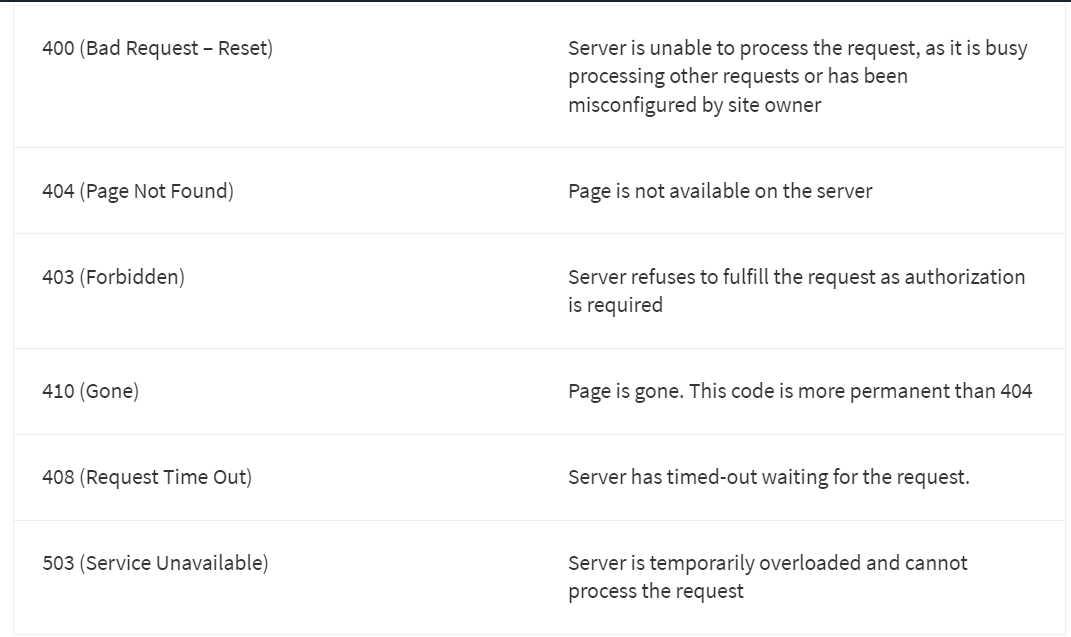
To start with, a link is an HTML object that enables users to migrate from one web page to another when they click on it. It is a means to navigate between different web pages on the internet.

A broken link, also often called a dead link, is one that does not work i.e. does not redirect to the webpage it is meant to. This usually occurs because the website or particular web page is down or does not exist. When someone clicks on a broken link, an error message is displayed.

Broken links may exist due to some kind of server error, which, in turn, causes the corresponding page to malfunction and not be displayed. A valid URL will have a 2xx HTTP status code.  Broken links, which are essentially invalid HTTP requests have 4xx and 5xx status codes.

The 4xx status code refers to a client-side error, while the 5xx status code usually points to a server response error.

****

****

**Why check for broken links**

If a user clicks on a broken link, they will be directed to an error page. This obviously contributes to sub-par user experience. Broken links defeat the purpose of having the website in the first place because users cannot find the information or service they are looking for.

Every link on a website must be tested to ensure that it is functioning as expected. However, given that most websites have hundreds (sometimes thousands) of links required to make them work, [manual testing](https://www.browserstack.com/guide/manual-testing-tutorial) of each link would require excessive amounts of time, effort, and resources. Moreover, with [automated Selenium testing](https://www.browserstack.com/automate) being an option, it would be completely unnecessary.

**Common Reasons for Broken Links**

* **404 Page Not Found** – The destination web page has been removed by the owner
* **400 Bad Request** – The server cannot process the HTTP request triggered by the link because the URL address requested is wrong
* Due to the user’s firewall settings, the browser cannot access the destination web page
* The link is misspelled

### ****How to identify broken links in Selenium WebDriver****

To check broken links in Selenium, the process is simple. On a web page, hyperlinks are implemented using the HTML Anchor (<a>) tag. All the script needs to do is to locate every anchor tag on a web page, get the corresponding URLs, and run through the links to check if any of them are broken.

Use the following steps to identify broken links in Selenium

1. Collect all the links present on a web page based on the <a> tag
2. Send HTTP request for each link
3. Verify the HTTP response code
4. Determine if the link is valid or broken based on the HTTP response code
5. Repeat the process for all links captured with the first step

If you’re wondering how to find broken images using Selenium WebDriver, use the same process.

**Read More:** [How to perform Double Click in Selenium](https://www.browserstack.com/guide/double-click-in-selenium)

### ****Finding Broken Links in Selenium: Example****

package automationPractice;

import java.io.IOException;

import java.net.HttpURLConnection;

import java.net.MalformedURLException;

import java.net.URL;

import java.util.Iterator;

import java.util.List;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

public class BrokenLinks {

private static WebDriver driver = null;

public static void main(String[] args) {

// TODO Auto-generated method stub

String homePage = "http://www.zlti.com";

String url = "";

HttpURLConnection huc = null;

int respCode = 200;

driver = new ChromeDriver();

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

driver.get(homePage);

List<WebElement> links = driver.findElements(By.tagName("a"));

Iterator<WebElement> it = links.iterator();

while(it.hasNext()){

url = it.next().getAttribute("href");

System.out.println(url);

if(url == null || url.isEmpty()){

System.out.println("URL is either not configured for anchor tag or it is empty");

continue;

}

if(!url.startsWith(homePage)){

System.out.println("URL belongs to another domain, skipping it.");

continue;

}

try {

huc = (HttpURLConnection)(new URL(url).openConnection());

huc.setRequestMethod("HEAD");

huc.connect();

respCode = huc.getResponseCode();

if(respCode >= 400){

System.out.println(url+" is a broken link");

}

else{

System.out.println(url+" is a valid link");

}

} catch (MalformedURLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

driver.quit();

}

}

[Run Selenium Tests on Real Devices for Free](https://www.browserstack.com/users/sign_up?ref=guide-find-broken-links-in-selenium-mid&product=automate)

Let’s go through the code for a closer understanding of its functionality.

#### **1. Import Packages**

Import the package below along with default packages:

import java.net.HttpURLConnection;

The methods in this package allow the tester to send HTTP requests and capture HTTP response codes when they are returned by the code.

#### **2. Collect all links on the web page**

Find all the links on the webpage and place them in a list:

List<WebElement> links = driver.findElements(By.tagName("a"));

Obtain Iterator to move through the list of links:

Iterator<WebElement> it = links.iterator();

#### **3: Identify and Validate URLs**

This step is about checking if a certain URL belongs to a third-party domain or if it is empty/null.

The code below will retrieve the href of the anchor tag and store it in the URL variable.

url = it.next().getAttribute("href");

If the URL is null or Empty, skip the steps after this.

if(url == null || url.isEmpty()){

System.out.println("URL is either not configured for anchor tag or it is empty");

continue;

}

If the URL belongs to the main domain, continue. If it belongs to a third party domain, skip the steps after this.

if(!url.startsWith(homePage)){

System.out.println("URL belongs to another domain, skipping it.");

continue;

}

#### **4. Send HTTP request**

Methods in the **HttpURLConnection** class will send HTTP requests and capture the HTTP response code. Therefore, the output of **openConnection()** method **(URLConnection)** is type casted to HttpURLConnection.

huc = (HttpURLConnection)(new URL(url).openConnection());

If testers set Request type as “HEAD” instead of “GET”, only headers will be returned, not the document body.

huc.setRequestMethod("HEAD");

When the tester invokes the connect() method, the actual connection to the URL is established and the HTTP request is sent.

huc.connect();

#### **5. Validate Links**

Use the getResponseCode() method to get the HTTP response code for the previously sent HTTP request.

respCode = huc.getResponseCode();

Check link status (broken or not) based on the response code

if(respCode >= 400){

System.out.println(url+" is a broken link");

}

else{

System.out.println(url+" is a valid link");

}

Finding broken links in Selenium is an integral part of website development and testing. By using the method described in this article, testers can identify malfunctioning links quickly and correctly. Allowing broken links to pass into the production stage would severely damage the user experience and must be prevented with extreme thoroughness. This is why knowing how to test broken links in [Selenium](https://www.browserstack.com/selenium) is an important part of a tester’s toolkit.

WebDriver can generally be said to have a blocking API. Because it is an out-of-process library that instructs the browser what to do, and because the web platform has an intrinsically asynchronous nature, WebDriver does not track the active, real-time state of the DOM. This comes with some challenges that we will discuss here.

From experience, most intermittent issues that arise from use of Selenium and WebDriver are connected to race conditions that occur between the browser and the user’s instructions. An example could be that the user instructs the browser to navigate to a page, then gets a **no such element** error when trying to find an element.

Consider the following document:

<!doctype html>

<meta charset=utf-8>

<title>Race Condition Example</title>

<script>

var initialised = false;

window.addEventListener("load", function() {

var newElement = document.createElement("p");

newElement.textContent = "Hello from JavaScript!";

document.body.appendChild(newElement);

initialised = true;

});

</script>

Copy

The WebDriver instructions might look innocent enough:

* Java
* Python
* CSharp
* Ruby
* JavaScript
* Kotlin

driver.get("file:///race\_condition.html");

WebElement element = driver.findElement(By.tagName("p"));

assertEquals(element.getText(), "Hello from JavaScript!");

Copy

The issue here is that the default [page load strategy](https://www.selenium.dev/documentation/webdriver/drivers/options/#pageloadstrategy) used in WebDriver listens for the document.readyState to change to "complete" before returning from the call to navigate. Because the p element is added after the document has completed loading, this WebDriver script might be intermittent. It “might” be intermittent because no guarantees can be made about elements or events that trigger asynchronously without explicitly waiting—or blocking—on those events.

Fortunately, the normal instruction set available on the [WebElement](https://www.selenium.dev/documentation/webdriver/elements/) interface—such as WebElement.click and WebElement.sendKeys—are guaranteed to be synchronous, in that the function calls will not return (or the callback will not trigger in callback-style languages) until the command has been completed in the browser. The advanced user interaction APIs, [Keyboard](https://www.selenium.dev/documentation/webdriver/actions_api/keyboard/) and [Mouse](https://www.selenium.dev/documentation/webdriver/actions_api/mouse/), are exceptions as they are explicitly intended as “do what I say” asynchronous commands.

Waiting is having the automated task execution elapse a certain amount of time before continuing with the next step.

To overcome the problem of race conditions between the browser and your WebDriver script, most Selenium clients ship with a wait package. When employing a wait, you are using what is commonly referred to as an [explicit wait](https://www.selenium.dev/documentation/webdriver/waits/#explicit-wait).

## **Explicit wait**

Explicit waits are available to Selenium clients for imperative, procedural languages. They allow your code to halt program execution, or freeze the thread, until the condition you pass it resolves. The condition is called with a certain frequency until the timeout of the wait is elapsed. This means that for as long as the condition returns a falsy value, it will keep trying and waiting.

Since explicit waits allow you to wait for a condition to occur, they make a good fit for synchronising the state between the browser and its DOM, and your WebDriver script.

To remedy our buggy instruction set from earlier, we could employ a wait to have the findElement call wait until the dynamically added element from the script has been added to the DOM:

* Java
* Python
* CSharp
* Ruby
* JavaScript
* Kotlin

WebDriver driver = new ChromeDriver();

driver.get("https://google.com/ncr");

driver.findElement(By.name("q")).sendKeys("cheese" + Keys.ENTER);

// Initialize and wait till element(link) became clickable - timeout in 10 seconds

WebElement firstResult = new WebDriverWait(driver, Duration.ofSeconds(10))

.until(ExpectedConditions.elementToBeClickable(By.xpath("//a/h3")));

// Print the first result

System.out.println(firstResult.getText());

Copy

We pass in the condition as a function reference that the wait will run repeatedly until its return value is truthy. A “truthful” return value is anything that evaluates to boolean true in the language at hand, such as a string, number, a boolean, an object (including a WebElement), or a populated (non-empty) sequence or list. That means an empty list evaluates to false. When the condition is truthful and the blocking wait is aborted, the return value from the condition becomes the return value of the wait.

With this knowledge, and because the wait utility ignores no such element errors by default, we can refactor our instructions to be more concise:

* Java
* Python
* CSharp
* Ruby
* JavaScript
* Kotlin

WebElement foo = new WebDriverWait(driver, Duration.ofSeconds(3))

.until(driver -> driver.findElement(By.name("q")));

assertEquals(foo.getText(), "Hello from JavaScript!");

Copy

In that example, we pass in an anonymous function (but we could also define it explicitly as we did earlier so it may be reused). The first and only argument that is passed to our condition is always a reference to our driver object, WebDriver. In a multi-threaded environment, you should be careful to operate on the driver reference passed in to the condition rather than the reference to the driver in the outer scope.

Because the wait will swallow no such element errors that are raised when the element is not found, the condition will retry until the element is found. Then it will take the return value, a WebElement, and pass it back through to our script.

If the condition fails, e.g. a truthful return value from the condition is never reached, the wait will throw/raise an error/exception called a timeout error.

### Options

The wait condition can be customised to match your needs. Sometimes it is unnecessary to wait the full extent of the default timeout, as the penalty for not hitting a successful condition can be expensive.

The wait lets you pass in an argument to override the timeout:

* Java
* Python
* CSharp
* Ruby
* JavaScript
* Kotlin

new WebDriverWait(driver, Duration.ofSeconds(3)).until(ExpectedConditions.elementToBeClickable(By.xpath("//a/h3")));

Copy

### Expected conditions

Because it is quite a common occurrence to have to synchronise the DOM and your instructions, most clients also come with a set of predefined expected conditions. As might be obvious by the name, they are conditions that are predefined for frequent wait operations.

The conditions available in the different language bindings vary, but this is a non-exhaustive list of a few:

* alert is present
* element exists
* element is visible
* title contains
* title is
* element staleness
* visible text

You can refer to the API documentation for each client binding to find an exhaustive list of expected conditions:

* Java’s [org.openqa.selenium.support.ui.ExpectedConditions](https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/support/ui/ExpectedConditions.html) class
* Python’s [selenium.webdriver.support.expected\_conditions](https://seleniumhq.github.io/selenium/docs/api/py/webdriver_support/selenium.webdriver.support.expected_conditions.html?highlight=expected) class
* JavaScript’s [selenium-webdriver/lib/until](https://seleniumhq.github.io/selenium/docs/api/javascript/module/selenium-webdriver/lib/until.html) module

## **Implicit wait**

There is a second type of wait that is distinct from [explicit wait](https://www.selenium.dev/documentation/webdriver/waits/#explicit-wait) called implicit wait. By implicitly waiting, WebDriver polls the DOM for a certain duration when trying to find any element. This can be useful when certain elements on the webpage are not available immediately and need some time to load.

Implicit waiting for elements to appear is disabled by default and will need to be manually enabled on a per-session basis. Mixing [explicit waits](https://www.selenium.dev/documentation/webdriver/waits/#explicit-wait) and implicit waits will cause unintended consequences, namely waits sleeping for the maximum time even if the element is available or condition is true.

Warning: Do not mix implicit and explicit waits. Doing so can cause unpredictable wait times. For example, setting an implicit wait of 10 seconds and an explicit wait of 15 seconds could cause a timeout to occur after 20 seconds.

An implicit wait is to tell WebDriver to poll the DOM for a certain amount of time when trying to find an element or elements if they are not immediately available. The default setting is 0, meaning disabled. Once set, the implicit wait is set for the life of the session.

* Java
* Python
* CSharp
* Ruby
* JavaScript
* Kotlin

WebDriver driver = new FirefoxDriver();

driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));

driver.get("http://somedomain/url\_that\_delays\_loading");

WebElement myDynamicElement = driver.findElement(By.id("myDynamicElement"));

Copy

## **FluentWait**

FluentWait instance defines the maximum amount of time to wait for a condition, as well as the frequency with which to check the condition.

Users may configure the wait to ignore specific types of exceptions whilst waiting, such as NoSuchElementException when searching for an element on the page.

* Java
* Python
* CSharp
* Ruby
* JavaScript
* Kotlin

// Waiting 30 seconds for an element to be present on the page, checking

// for its presence once every 5 seconds.

Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)

.withTimeout(Duration.ofSeconds(30))

.pollingEvery(Duration.ofSeconds(5))

.ignoring(NoSuchElementException.class);

WebElement foo = wait.until(driver -> {

return driver.findElement(By.id("foo"));

});