package automationFramework;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class WebDriverCommands {

public static void main(String[] args) {

// Create a new instance of the Chrome 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();

}

}

***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();

}

}

## 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.

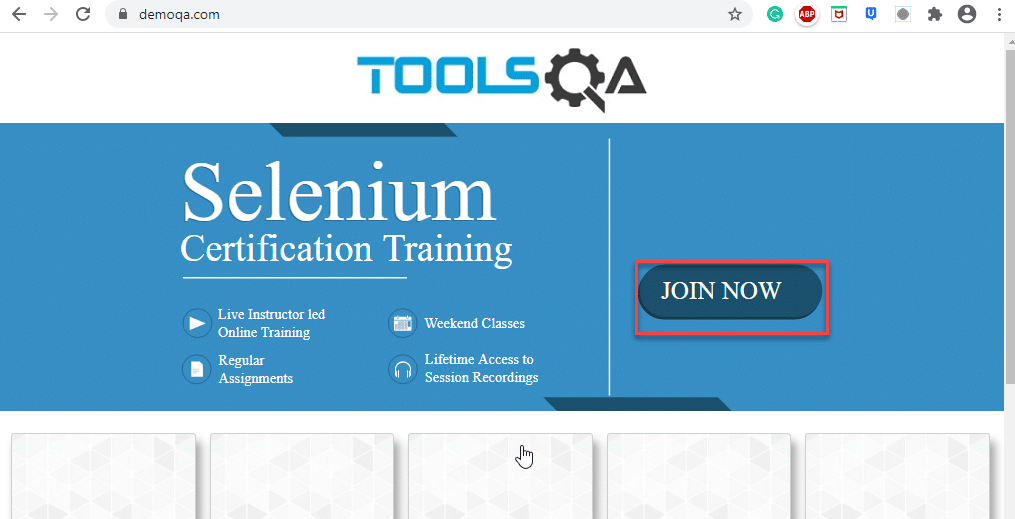
| ***Locator*** | ***Description*** |
| --- | --- |
| ***id*** | finds elements by ID attribute. The search value given should match the ID attribute. |
| ***name*** | Finds or Locates elements based on the NAME attribute. The name attribute is used to match the search value. |
| ***class name*** | Finds 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.

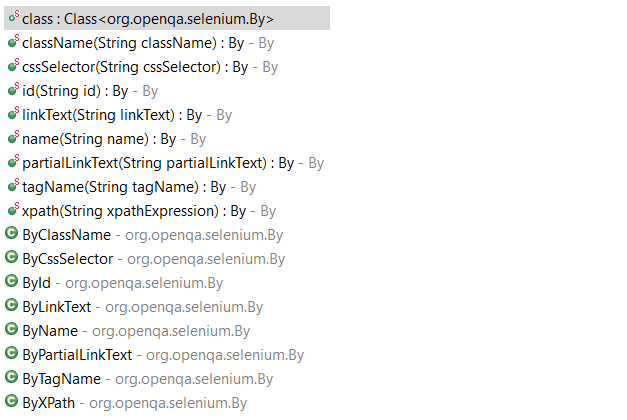
It’s syntax looks like below:

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**

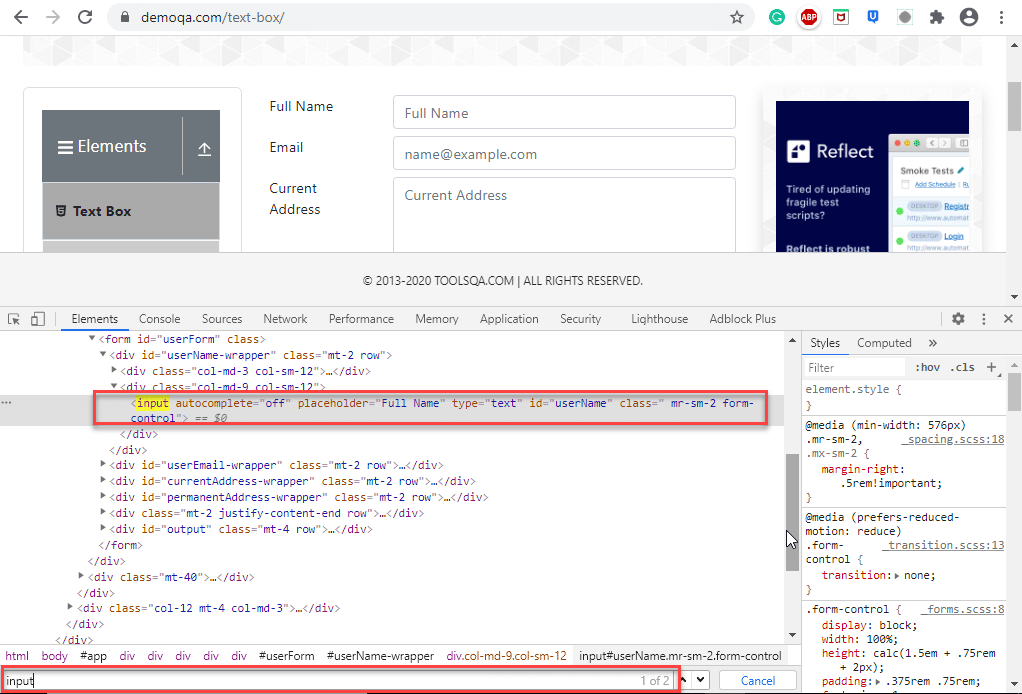
The command findElements() returns a list of web elements that match the specified criteria, unlike findElement() which returned a unique element. ***If there are no matching elements then an empty list returns.***

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"));

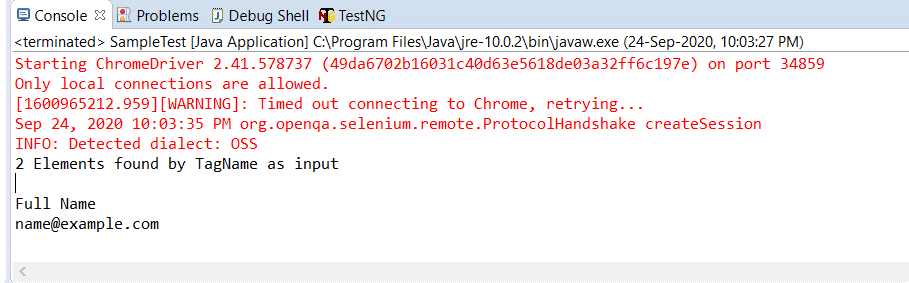
}

}

}

}

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?**

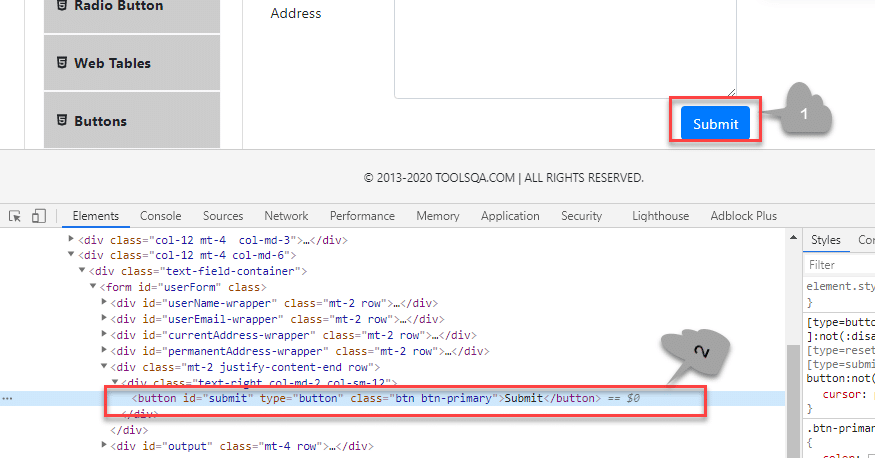
Using ***"id "*** to find an element is by far the most common strategy used to find an element. Suppose if the webpage uses dynamically generated ids, then this strategy returns the first web element that matches the id.

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) {

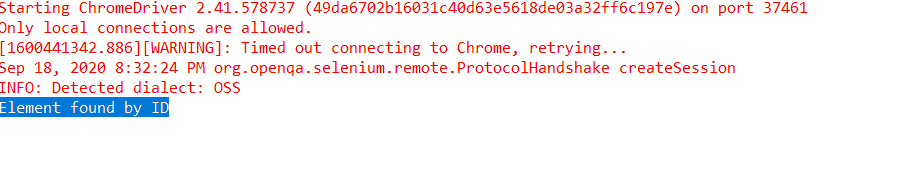
System.out.println("Element found by ID");

}

}

}

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.

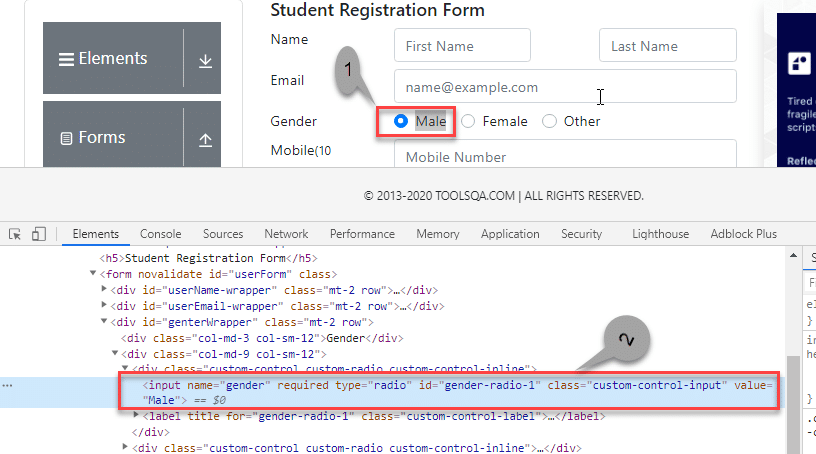
#### **How to find an element using the attribute "name" in Selenium?**

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"));

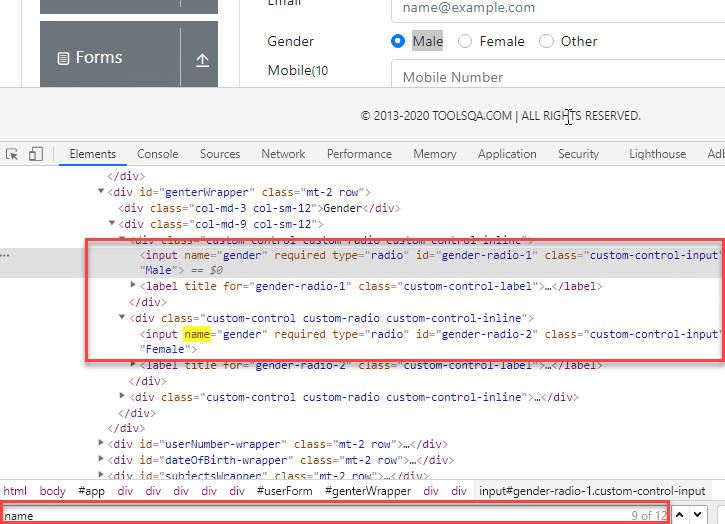
// Action can be performed on Input Text element

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) {

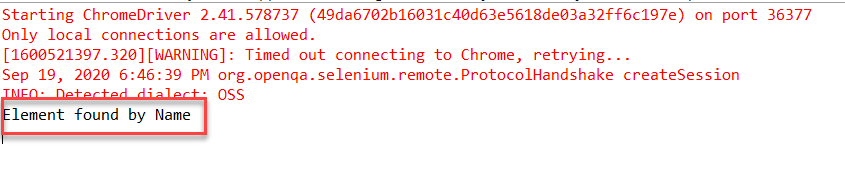
System.out.println("Element found by Name");

}

}

}

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>)) ;

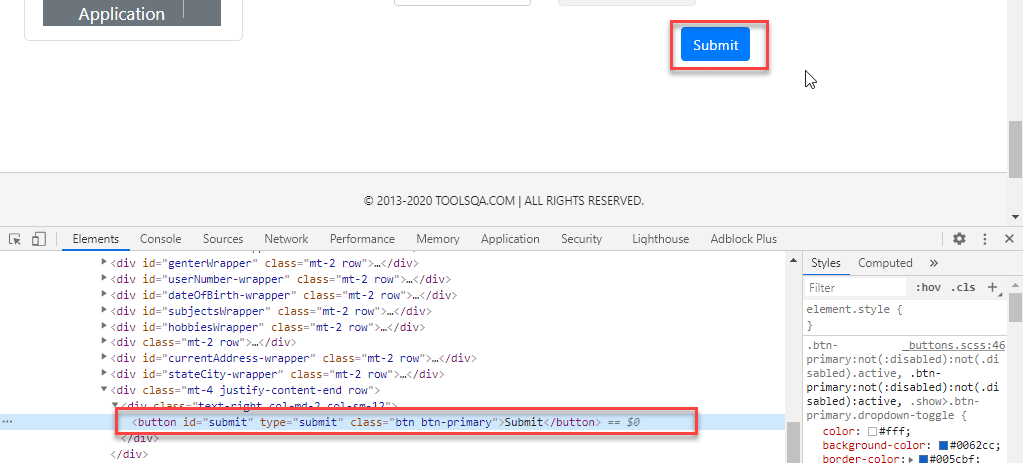
or

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) {

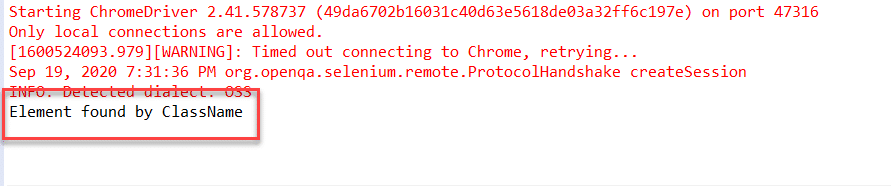
System.out.println("Element found by ClassName");

}

}

}

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?**

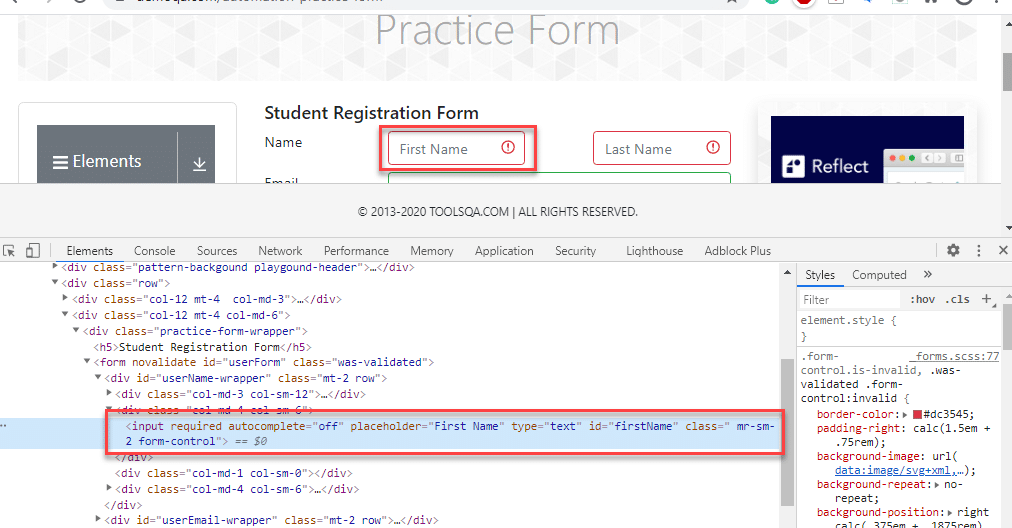
The ***Tag Name*** strategy uses an HTML tag name to locate the element. We use this approach rarely and we use it only when we are not able to find elements using any other strategies.

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) {

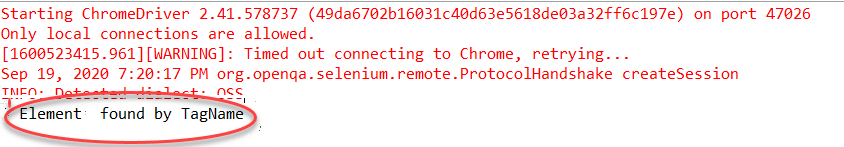
System.out.println("Element found by tagName");

}

}

}

The output of this program is as seen below.

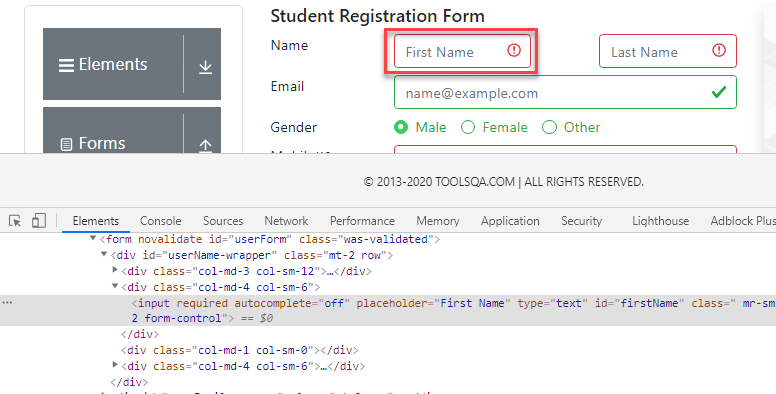


Above program uses ***By.tagName*** object in 'findElement()' call to find an element based on ***tagName = "input"***.

#### **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) {

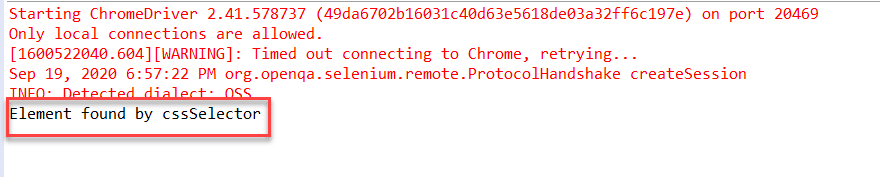
System.out.println("Element found by cssSelector");

}

}

}

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?**

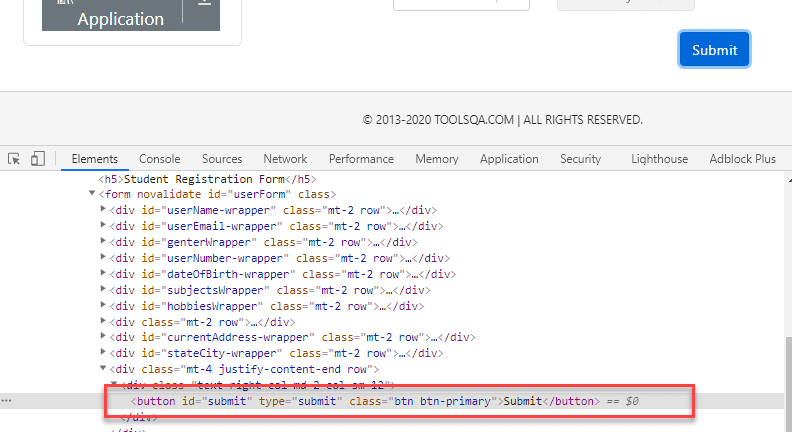
This strategy is the most popular one for finding elements. Using this strategy we navigate through the structure of the ***HTML*** or ***XML*** documents.

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) {

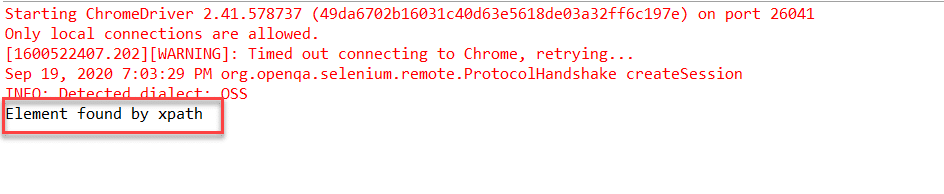
System.out.println("Element found by xpath");

}

}

}

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) {

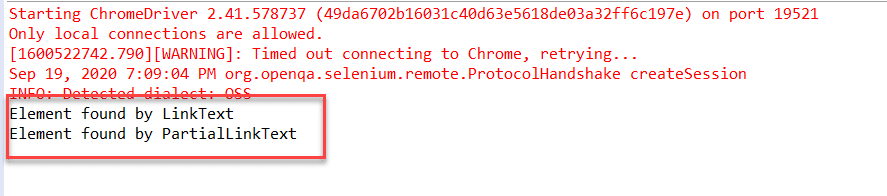
System.out.println("Element found by PartialLinkText");

}

}

}

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.

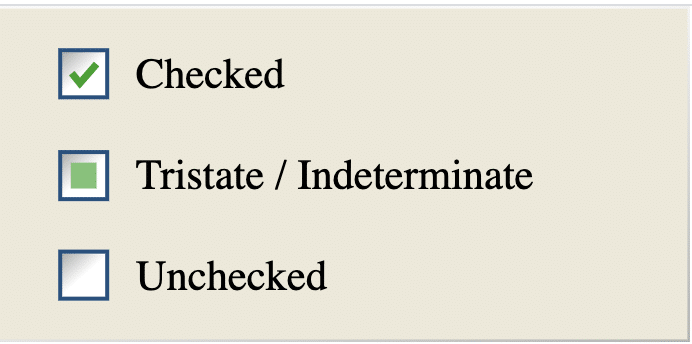
| ***FindElement()*** | ***FindElements()*** |
| --- | --- |
| Returns the first web element out of all the elements found by the same locator. | Finds and returns a list of web elements. |
| This method finds only one element. | This method returns the collection of elements matching the locator. |
| 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. |

## What is a CheckBox?

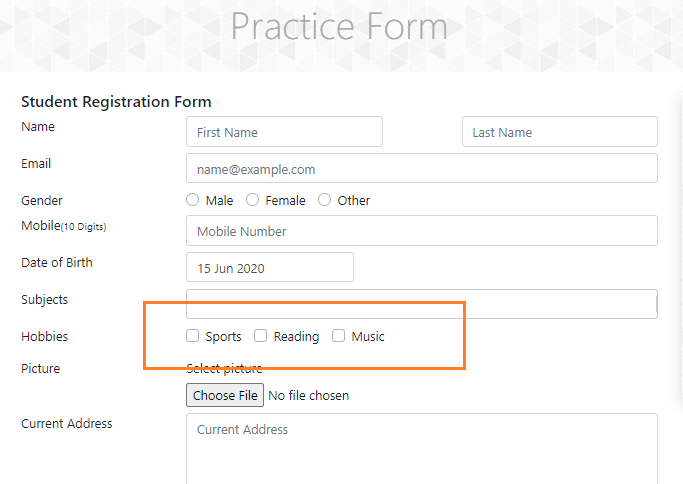
The checkbox is a GUI element that allows the user to make certain choices for the given options. Users may get a list of choices, and the checkbox records the choices made by the user. The checkbox allows users to select either single or multiple choices out of the given list.

We can define a checkbox in ***HTML*** using ***<input type="checkbox">*** ***tag.*** Any [***locator strategy***](https://www.toolsqa.com/selenium-webdriver/selenium-locators/)  that uses [***DOM***](https://en.wikipedia.org/wiki/Document_Object_Model) for locating web elements should use this tag and properties for recognizing the checkbox.

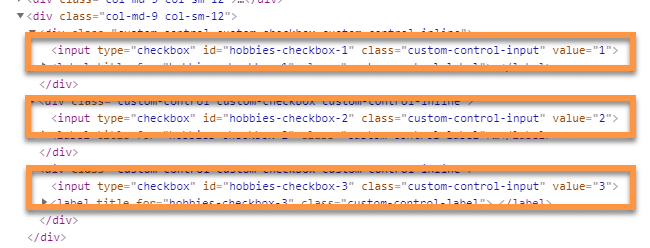
Apart from ***"Checked "*** and ***"UnChecked",*** sometimes applications provide a ***tri-state/intermediate,*** which we generally use when a neutral answer needs to provide for an option or which is an auto-selection for a parent when the child checkbox is selected. Based on these, checkboxes  will generally have the following states:



To understand more about CheckBoxes, let's consider the example of checkboxes (as highlighted below) given on the page [***"http://www.demoqa.com/automation-practice-form".***](https://www.demoqa.com/automation-practice-form)



If we look at the ***HTML*** structure of the above element, we will find that it starts with the ***<input type="checkbox">***  node. We have highlighted the ***HTML structure*** of the ***checkboxes***  in the image below:



As we can see, all the checkboxes are being created using the ***HTML tag <input>*** and have an attribute named ***“type”,*** which has a value ***“checkbox“,*** which signifies that the type of the input element is a checkbox.

Now, let’s see how we can locate and perform specific actions on the ***CheckBoxes*** using ***Selenium WebDriver?***

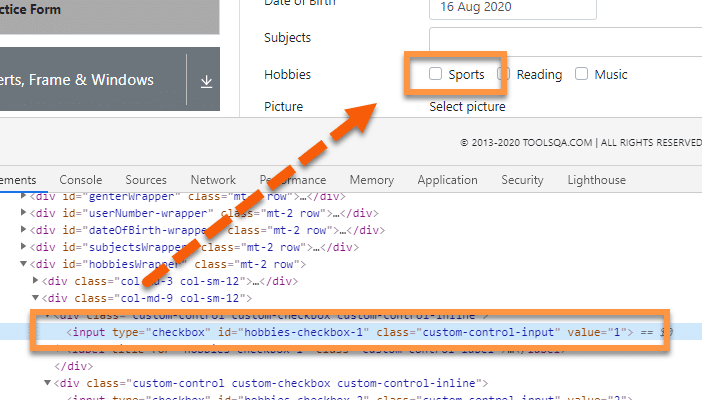
## How to handle a CheckBox in Selenium WebDriver?

As we know, Selenium provides various [***locator strategies,***](https://www.toolsqa.com/selenium-webdriver/selenium-locators/) and almost all of them can locate ***CheckBoxes*** using ***Selenium WebDriver.*** Let’s understand a few of them:

### *****How to locate and select a checkbox in Selenium using the ID locator?*****

If a checkbox has an ***id*** attribute that contains a unique value, then we can use the ***ID locator*** provided by the ***Selenium WebDriver*** for locating and selecting the element. To select a ***checkbox,*** the ***click operation*** needs to perform. So, once we locate the element, we need to perform a click to select it.

Let’s have a look at the below [***ToolsQA demo page***](https://demoqa.com/automation-practice-form) for understanding the details of the ***checkbox*** having ***id*** attribute:



In the above ***DOM,*** we can see that the ***input*** tag has an ***id*** attribute. Now, if we use the ***ID locator***  to recognize the element and perform the ***click***  operation, we will need to use the following Selenium code:

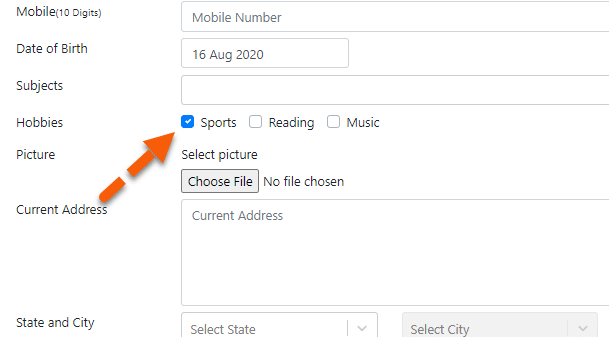
/\*\*

\* Locating and Clicking a CheckBox By using ID

\*/

driver.findElement(By.id("hobbies-checkbox-1")).click();

Using the above line of code, Selenium will locate the web element with ***"id "*** as ***"hobbies-checkbox-1 "*** and will perform the ***click*** operation on that. The execution of the above line of code will lead to the following state on the web page:



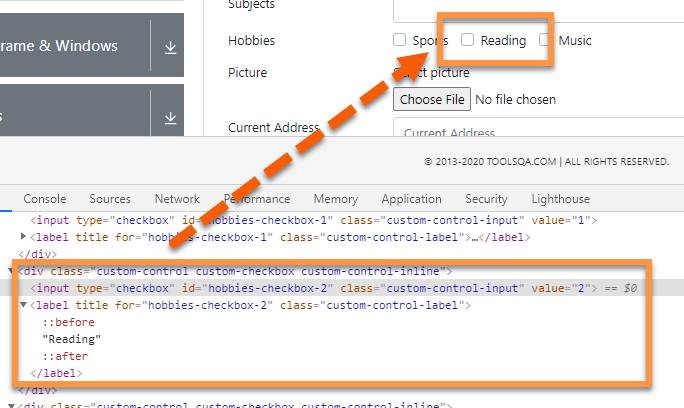
This way, we can select a ***CheckBox that*** has the unique ***"id "*** attribute and ***check*** the same by using the ***"click"*** operation.

### *****How to locate and select a checkbox in Selenium using the XPath locator?*****

***Selenium WebDriver*** can locate CheckBoxes using the XPath locator strategy, as it does the other web elements.

In the image given below, we can see the ***HTML structure*** of the highlighted checkbox in the ***DOM.*** We can use this to write the ***XPath*** and identify/locate the element.

Let's consider an example where we will try to select multiple checkboxes using XPath. We will choose both ***"Sports "*** and ***"Reading "*** checkbox to exhibit multiple selection scenarios.



Once we can uniquely identify the checkbox using ***XPath,*** we can use ***Selenium WebElement*** to perform a required operation such as ***"click "*** on the checkboxes. The code for locating the CheckBoxes using Xpath  and selecting the same will look as follows:

//Selecting the first checkbox using XPath

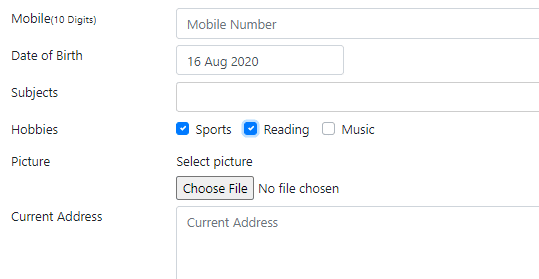
driver.findElement(By.xpath("//label[text()='Sports']")).click();

//Selecting the second checkbox using Xpath

driver.findElement(By.xpath("//label[text()='Reading']")).click();

***Note:*** In the above code, we have clicked on the label associated with the checkboxes. Generally, checkboxes can be checked/unchecked by clicking both the checkbox itself or the labels associated with the checkboxes.

Using the above line of code, Selenium will locate the web element with specified ***XPath*** and will perform the ***click*** operation on that. The execution of the above line of code will lead to the following state on the web page:

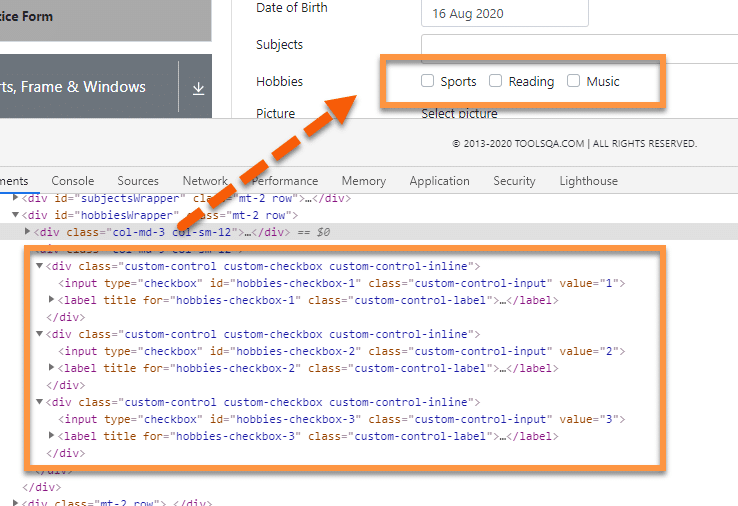


So, this way, we can select a ***CheckBox*** by using a unique ***XPath*** and ***select*** the same by using the ***"click "*** operation.

### *****How to locate and select a checkbox in Selenium using the CSS Selector locator?*****

As discussed in the previous section, sometimes it isn't easy to locate an element on the web page by using its identifier or even attribute value. The most efficient way is to use a ***CSS Selector*** to handle dynamic elements.

In the given image, we can see the element present in the ***DOM***  by highlighting it. We can use the given ***HTML*** structure to create a ***CSS expression*** that can recognize the element. We will try to select all three checkboxes using the ***CSS selectors.***



To locate the checkboxes, we will first need to write the ***CSS selector expression*** for each element. Then we can use these in ***Selenium WebElement*** and then perform select or click operation.

As discussed earlier, the checkboxes can be selected either by locating and clicking the input element or can also be selected by clicking the label associated with the checkboxes. So, if we use the labels associated with the checkboxes, the below code snippet will locate the elements using CSS Selectors and will click on them for selecting the same:

//Selecting the first checkbox

driver.findElement(By.cssSelector("label[for='hobbies-checkbox-1']")).click();

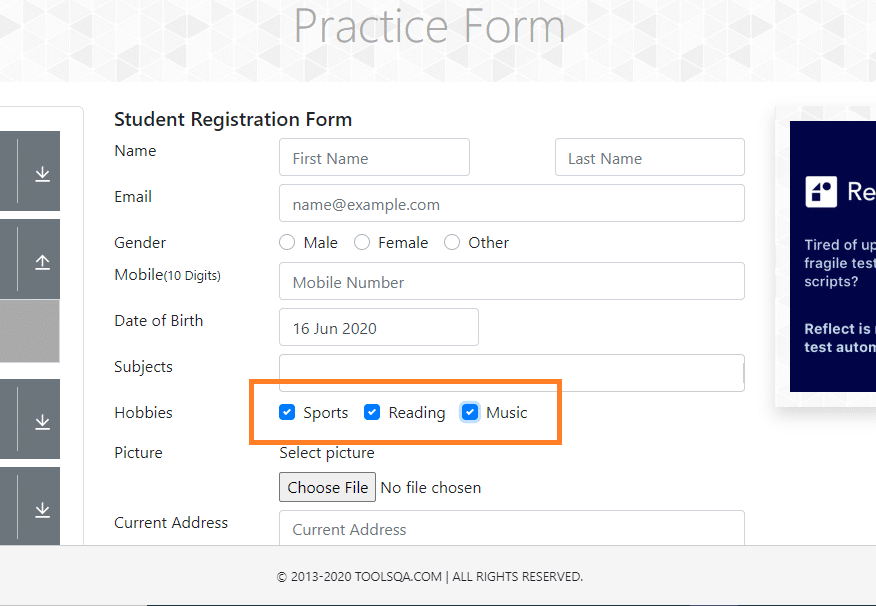
//Selecting the second checkbox

driver.findElement(By.cssSelector("label[for='hobbies-checkbox-2']")).click();

//Selecting the last check box

driver.findElement(By.cssSelector("label[for='hobbies-checkbox-3']")).click();

Using the above line of code, Selenium will locate the web element with specified ***CSS Locator*** and will perform the ***click*** operation on that. The execution of the above line of code will lead to the following state on the web page:



This way, we can select a ***Checkbox*** by using a unique ***CSS Locator*** and ***check*** the same by using the ***"click "*** operation.

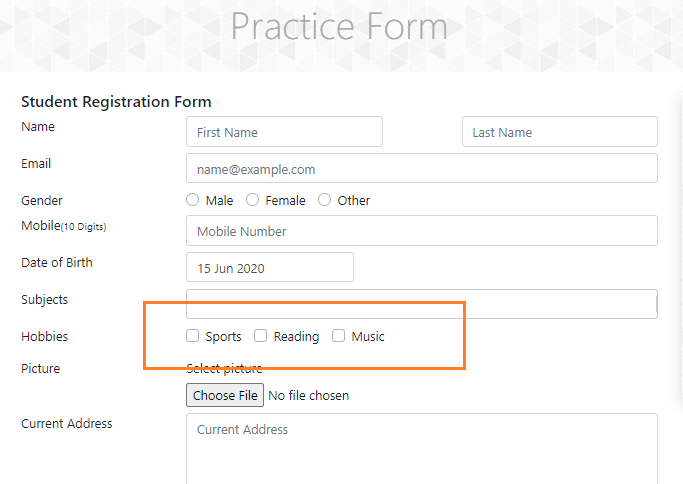
## How to perform validations on a CheckBox using Selenium WebDriver?

***Selenium WebDriver*** provides certain methods that we can use for a pre and post validation of the states of a ***CheckBox.*** Few of these methods are:

* isSelected(): Checks whether a checkbox is selected or not.
* isDisplayed(): Checks whether a checkbox displays on the web page or not.
* isEnabled(): Checks whether a checkbox is enabled or not

We can use these methods to validate the current state of the check boxes. E.g., to validate that after clicking the checkbox, whether we have checked or not, we can use the ***"isSelected()"*** method. In other words, it helps to validate the current state of the checkbox. Similarly, before clicking a checkbox, we can validate that whether is checkbox is displayed on the page and is enabled, then only click on the checkbox. So, such pre validations can be done using the ***"isDisplayed()"*** and ***"isEnabled()"*** methods.

Let's have a look at an example to understand how we can use all these validations. We will take the following checkboxes  as an example:



### *****How to use the isSelected() method to validate if the CheckBox is selected?*****

We can use the ***isSelected()*** method to validate the current state of the checkbox, whether we selected it or not. We can use it both in pre and post validation. E.g., we can perform the click operation on the checkbox when it is not already selected; otherwise, we can skip the operation, as shown by the below code snippet:

/\*\*

\* Validate Checkbox isSelected method and click

\*/

WebElement checkBoxElement = driver.findElement(By.cssSelector("label[for='hobbies-checkbox-1']"));

boolean isSelected = checkBoxElement.isSelected();

//performing click operation if element is not checked

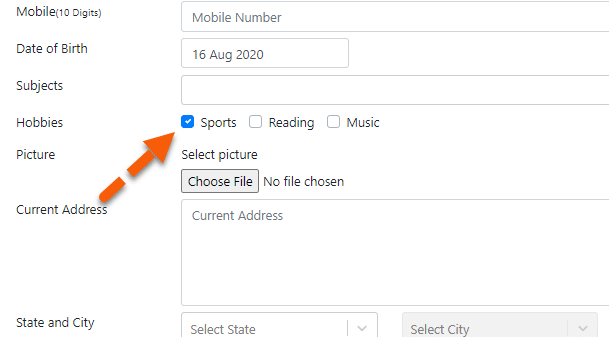
if(isSelected == false) {

checkBoxElement.click();

}

Once we ran this code, the code will first check whether the checkbox or not. Then an ***if condition*** will validate if the returned value is ***true*** or ***false.*** In case it's false, i.e., the checkbox will appear unchecked. The code inside the ***if condition*** will execute, and the checkbox will check.

The output of the above code will be:



This way, we can do a conditional select of the checkbox by first checking the state of the CheckBox.

### *****How to use isDisplayed() method to validate if the CheckBox is displayed?*****

Taking the same example as above, let's write a simple selenium code to validate if a given checkbox displays or not. If we can see it, then we will click and select it.

The code snippet for the scenario, as mentioned above, will be:

/\*\*

\* Validate Checkbox using isDisplayed() and click

\*/

WebElement checkBoxElement = driver.findElement(By.cssSelector("label[for='hobbies-checkbox-1']"));

boolean isDisplayed = checkBoxElement.isDisplayed();

// performing click operation if element is displayed

if (isDisplayed == true) {

checkBoxElement.click();

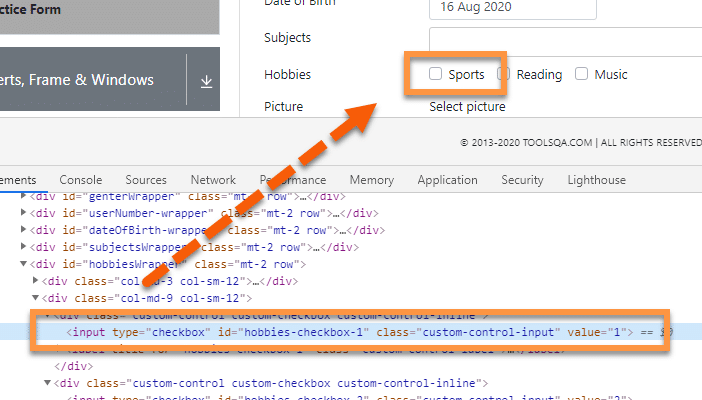
}

The above test will validate if the given checkbox displays on the page or not. If it is displayed, then it will make a selection. The output of the above code will be the same as was in the case of ***"isSelected()"*** as the specified checkbox is displayed, and it will check using the click option.

### *****How to use the isEnabled() method to validate if the CheckBox is enabled?*****

Assume we want to check a checkbox, but during the runtime, there may be a scenario where it is disabled. To handle such scenarios, Selenium offers a method called ***"isEnabled()".*** As the name suggests, this method validates if the given web element is enabled or not. This method will return the boolean value based on the status of the element. It will return 'true' if the element is in enabled status else 'false.'

For example, we want first to check if the ***"Sports "*** checkbox is in enabled status or not. Then we will proceed with other actions.



The below code snippet will click on the checkbox only when enabled:

/\*\*

\* Validate checkbox using isEnabled() and then click

\*/

WebElement checkBoxElement = driver.findElement(By.cssSelector("label[for='hobbies-checkbox-1']"));

boolean isEnabled = chckBxEnable.isEnabled();

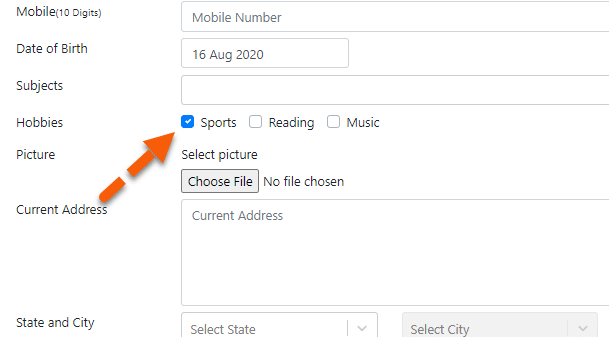
// performing click operation if element is enabled

if (isEnabled == true) {

checkBoxElement.click();

}

The above code will first check if the element is in an enabled state or not; if enabled, it will perform the click operation. If disabled, it will not perform any operation. As the checkbox enables, it will click the checkbox and will show the output as shown below:



The above code will first check if we have enabled the element; if it is, it will perform the click operation. If disabled, no operation will perform.

Let's now try to write a single test, which will perform all the operations as mentioned above and validations on the radio buttons at the ***ToolsQA Demo site:***

package TestPackage;

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 CheckBoxOperations {

public static void main(String[] args) {

String exePath = "C:\\Selenium\\chromedriver\\chromedriver.exe";

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

WebDriver driver = new ChromeDriver();

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

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

/\*\*

\* Validate isSelected and click

\*/

WebElement checkBoxSelected = driver.findElement(By.cssSelector("label[for='hobbies-checkbox-1']"));

boolean isSelected = checkBoxSelected.isSelected();

// performing click operation if element is not selected

if(isSelected == false) {

checkBoxSelected.click();

}

/\*\*

\* Validate isDisplayed and click

\*/

WebElement checkBoxDisplayed = driver.findElement(By.cssSelector("label[for='hobbies-checkbox-1']"));

boolean isDisplayed = checkBoxDisplayed.isDisplayed();

// performing click operation if element is displayed

if (isDisplayed == true) {

checkBoxDisplayed.click();

}

/\*\*

\* Validate isEnabled and click

\*/

WebElement checkBoxEnabled = driver.findElement(By.cssSelector("label[for='hobbies-checkbox-1']"));

boolean isEnabled = checkBoxEnabled.isEnabled();

// performing click operation if element is enabled

if (isEnabled == true) {

checkBoxEnabled.click();

}

}

}

Radio Button

package TestPackage;

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 Selenium {

public static void main(String[] args) {

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

WebDriver driver = new ChromeDriver();

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

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

/\*\*

\* Find radio button using ID, Validate isSelected and then click to select

\*/

WebElement radioEle = driver.findElement(By.id("yesRadio"));

boolean select = radioEle.isSelected();

System.out.print(select);

// performing click operation if element is not already selected

if (select == false) {

radioEle.click();

}

/\*\*

\* Find radio button using Xpath, Validate isDisplayed and click to select

\*/

WebElement radioElem = driver.findElement(By.xpath("//div/input[@id='impressiveRadio']"))

boolean sel = radioEle.isDisplayed();

// performing click operation if element is displayed

if (sel == true) {

radioElem.click();

}

/\*\*

\* Find radio button using CSS Selector, Validate isEnabled and click to select

\*/

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

boolean selectNo = radioEle.isDisplayed();

// performing click operation if element is enabled

if (selectNo == true) {

radioNo.click();

}

}

}

***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");

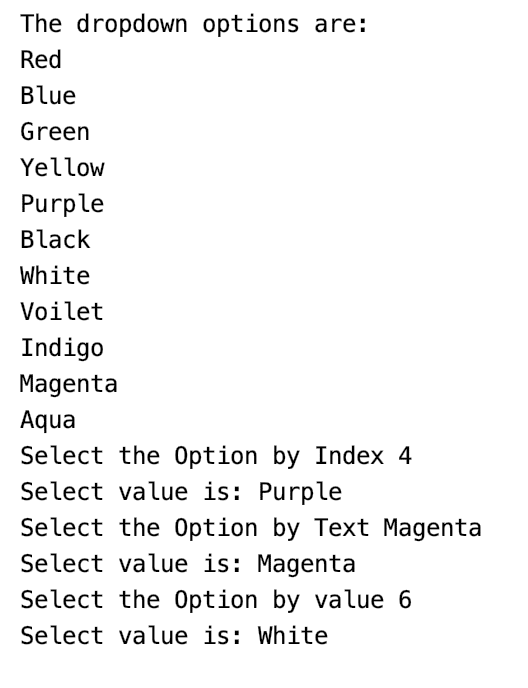
System.out.println("Select value is: " + select.getFirstSelectedOption().getText());

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();

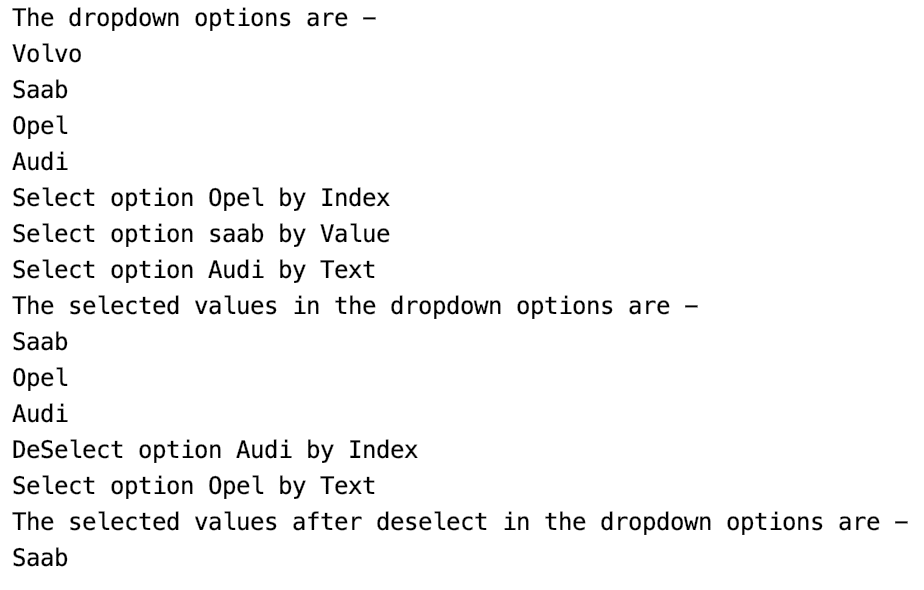
}

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.

***Practice Exercise 1***

1. *Launch new Browser*
2. *Open URL “****http://toolsqa.com/automation-practice-table/****”*
3. *Get the value from cell 'Dubai' and print it on the console*
4. *Click on the link 'Detail' of the first row and last column*

package automationFramework;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

public class PracticeTables {

public static void main(String[] args) {

WebDriver driver = new FirefoxDriver();

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

driver.get("https://toolsqa.com/automation-practice-table");

//Here we are storing the value from the cell in to the string variable

String sCellValue = driver.findElement(By.xpath(".//\*[@id='content']/table/tbody/tr[1]/td[2]")).getText();

System.out.println(sCellValue);

// Here we are clicking on the link of first row and the last column

driver.findElement(By.xpath(".//\*[@id='content']/table/tbody/tr[1]/td[6]/a")).click();

System.out.println("Link has been clicked otherwise an exception would have thrown");

driver.close();

}

}

***Practice Exercise 2***

1. *Launch new Browser*
2. *Open URL “****http://toolsqa.com/automation-practice-table/****”*
3. *Get the value from cell 'Dubai' with using dynamic xpath*
4. *Print all the column values of row 'Clock Tower Hotel'*

package automationFramework;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

public class PracticeTable\_2 {

public static void main(String[] args) {

WebDriver driver = new FirefoxDriver();

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

driver.get("https://toolsqa.com/automation-practice-table");

String sRow = "1";

String sCol = "2";

//Here we are locating the xpath by passing variables in the xpath

String sCellValue = driver.findElement(By.xpath(".//\*[@id='content']/table/tbody/tr[" + sRow + "]/td[" + sCol + "]")).getText();

System.out.println(sCellValue);

String sRowValue = "Clock Tower Hotel";

//First loop will find the 'ClOCK TWER HOTEL' in the first column

for (int i=1;i<=5;i++){

String sValue = null;

sValue = driver.findElement(By.xpath(".//\*[@id='content']/table/tbody/tr[" + i + "]/th")).getText();

if(sValue.equalsIgnoreCase(sRowValue)){

// If the sValue match with the description, it will initiate one more inner loop for all the columns of 'i' row

for (int j=1;j<=5;j++){

String sColumnValue= driver.findElement(By.xpath(".//\*[@id='content']/table/tbody/tr[" + i + "]/td["+ j +"]")).getText();

System.out.println(sColumnValue);

}

break;

}

}

driver.close();

}

}

Mouse action:-

Find below the steps of the scenario to be automated:

1. *Launch the web browser and launch our practice page https://demoqa.com/menu/*
2. *Find the required element i.e. ‘Music’ object in our sample*
3. *Now move the mouse on ‘Music’ option which will display the submenu list*
4. *Find the required element i.e. ‘Rock’ object in the sub menu list*
5. *Move mouse on ‘Rock’ option which will display one more submenu list*
6. *Find the required element i.e. ‘Alternative’ object in submenu list*
7. *Click on ‘Alternative’*
8. *Close the browser to end the program*

***Practice Exercise Code***

package com.toolsqa.tutorials.actions;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.interactions.Actions;

public class MouseHover1 {

public static WebDriver driver;

public static void main(String[] args) {

//Set system properties for geckodriver This is required since Selenium 3.0

System.setProperty("webdriver.gecko.driver", "C:\\Selenium\\Toolsqa\\lib\\geckodriver.exe");

// Launch the URL

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

System.out.println("demoqa webpage Displayed");

//Maximise browser window

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

//Adding wait

driver.manage().timeouts().implicitlyWait(10000, TimeUnit.MILLISECONDS);

//Instantiate Action Class

Actions actions = new Actions(driver);

//Retrieve WebElement 'Music' to perform mouse hover

WebElement menuOption = driver.findElement(By.xpath(".//div[contains(text(),'Music')]"));

//Mouse hover menuOption 'Music'

actions.moveToElement(menuOption).perform();

System.out.println("Done Mouse hover on 'Music' from Menu");

//Now Select 'Rock' from sub menu which has got displayed on mouse hover of 'Music'

WebElement subMenuOption = driver.findElement(By.xpath(".//div[contains(text(),'Rock')]"));

//Mouse hover menuOption 'Rock'

actions.moveToElement(subMenuOption).perform();

System.out.println("Done Mouse hover on 'Rock' from Menu");

//Now , finally, it displays the desired menu list from which required option needs to be selected

//Now Select 'Alternative' from sub menu which has got displayed on mouse hover of 'Rock'

WebElement selectMenuOption = driver.findElement(By.xpath(".//div[contains(text(),'Alternative')]"));

selectMenuOption.click();

System.out.println("Selected 'Alternative' from Menu");

// Close the main window

driver.close();

}

}

Find below the steps of the scenario to be automated:

1. *Launch the web browser and launch our practice page https://demoqa.com/slider/*
2. *Find the required element i.e. Slider object in our sample*
3. *Now specify x and y offset to move the mouse in the horizontal and vertical direction from top-left of the slider element*
4. *Move mouse on specified offset on the slider*
5. *Click on the slider*
6. *Close the browser to end the program*

***Practice Exercise Code***

package com.toolsqa.tutorials.actions;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.interactions.Actions;

public class MouseHover2 {

public static WebDriver driver;

public static void main(String[] args) {

//Set system properties for geckodriver This is required since Selenium 3.0

System.setProperty("webdriver.gecko.driver", "C:\\Selenium\\Toolsqa\\lib\\geckodriver.exe");

// Launch the URL

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

System.out.println("demoqa webpage Displayed");

//Maximise browser window

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

//Adding wait

driver.manage().timeouts().implicitlyWait(10000, TimeUnit.MILLISECONDS);

//Instantiate Action Class

Actions actions = new Actions(driver);

//Retrieve WebElemnt 'slider' to perform mouse hover

WebElement slider = driver.findElement(By.id("slider"));

//Move mouse to x offset 50 i.e. in horizontal direction

actions.moveToElement(slider,50,0).perform();

slider.click();

System.out.println("Moved slider in horizontal directions");

// Close the main window

driver.close();

}

}

## What are keyboard events in Selenium?

A Keyboard Event describes a user's interaction with the keyboard. When a user presses single or multiple keys, keyboard events generate. Selenium provides various ways to automate these Keyboard Events, a few of which are:

* [***Automate keyboard events using the sendKeys() method of WebElement class.***](https://www.toolsqa.com/selenium-webdriver/webelement-commands/)
* [***Automate keyboard events using Robot class.***](https://www.toolsqa.com/selenium-webdriver/robot-class-keyboard-events/)
* ***And Automate keyboard events using Actions class.***

We already discussed the first two ways of handling the keyboard events using the ***"sendKeys()"*** method of ***WebDriver's WebElement class*** and ***"Robot class"*** in the articles given by corresponding links. In this article, we will specifically cover the details of the ***"Actions "*** class in ***Selenium WebDriver.*** Before going deep to understand the concepts of the ***"Actions "*** class, let's first understand why specifically ***"Actions "*** class is needed to handle those Keyboard Events?

### *****Why is Actions class needed to perform Keyboard actions using Selenium WebDriver?*****

When we interact with a web application, there will be various scenarios when the user performs the following actions:

* ***Type in capital/Camel case letters:*** Wherever user need to type a word or letter in caps, he/she will press the ***"SHIFT"*** key and will type the necessary characters, and whatever characters will type while pressing the ***"SHIFT"*** key, will type as a capital letter.
* ***Copy & Paste Text:*** When we need to copy some text from one text box to another, we select the text by pressing ***"CTRL+A"*** they copy the text using ***"CTRL+C"*** and paste the text in the new text box by simply clicking in the text box and pressing keys ***"CTRL+V"***.

These are very common user actions, which we perform on an almost daily basis. Now, as we discussed, ***Selenium WebDriver*** provides two ways to send any keyboard event to a web element:

* sendKeys() method of WebElement class.
* Actions class

Now let's try to understand in detail that if we want to automate the scenario as mentioned above of typing the letters in the capital (with SHIFT key pressed ), using the ***sendKeys()*** method of the ***WebElement*** class.

Consider the following scenario for quickly understanding the behavior:

* First, navigate to [***"https://demoqa.com/text-box."***](https://demoqa.com/text-box)
* Secondly, enter the Full name: "***Mr.Peter Haynes***".
* Thirdly, enter the Email: ***"PeterHaynes@toolsqa.com."***
* After that, Enter the Current Address: ***"43 School Lane London EC71 9GO"***.
* Fifthly, click on the Current Address text box and Copy the Current Address.
* After that, paste the copied address in the Permanent Address text box.
* Finally, validate that the text in the Current Address and Permanent Address is the same.

Let's try to automate the above scenario using the sendKeys() method of the WebElement class:

package automationFramework;

import static org.junit.Assert.assertEquals;

import org.openqa.selenium.By;

import org.openqa.selenium.Keys;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

public class KeyboardEvents {

public static void main(String[] args) {

// Initialize ChromeDriver

// Here we assume that the ChromeDriver path has been set in the System Global variables

WebDriver driver=new ChromeDriver();

//Navigate to the demo site

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

// Enter the Full Name

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

fullName.sendKeys("Mr.Peter Haynes");

//Enter the Email

WebElement email=driver.findElement(By.id("userEmail"));

email.sendKeys("PeterHaynes@toolsqa.com");

// Enter the Current Address

WebElement currentAddress=driver.findElement(By.id("currentAddress"));

currentAddress.sendKeys("43 School Lane London EC71 9GO");

// Copy the Current Address

currentAddress.sendKeys(Keys.CONTROL);

currentAddress.sendKeys("A");

currentAddress.sendKeys(Keys.CONTROL);

currentAddress.sendKeys("C");

//Press the TAB Key to Switch Focus to Permanent Address

currentAddress.sendKeys(Keys.TAB);

//Paste the Address in the Permanent Address field

WebElement permanentAddress=driver.findElement(By.id("permanentAddress"));

permanentAddress.sendKeys(Keys.CONTROL);

permanentAddress.sendKeys("V");

//Compare Text of current Address and Permanent Address

assertEquals(currentAddress.getAttribute("value"),permanentAddress.getAttribute("value"));

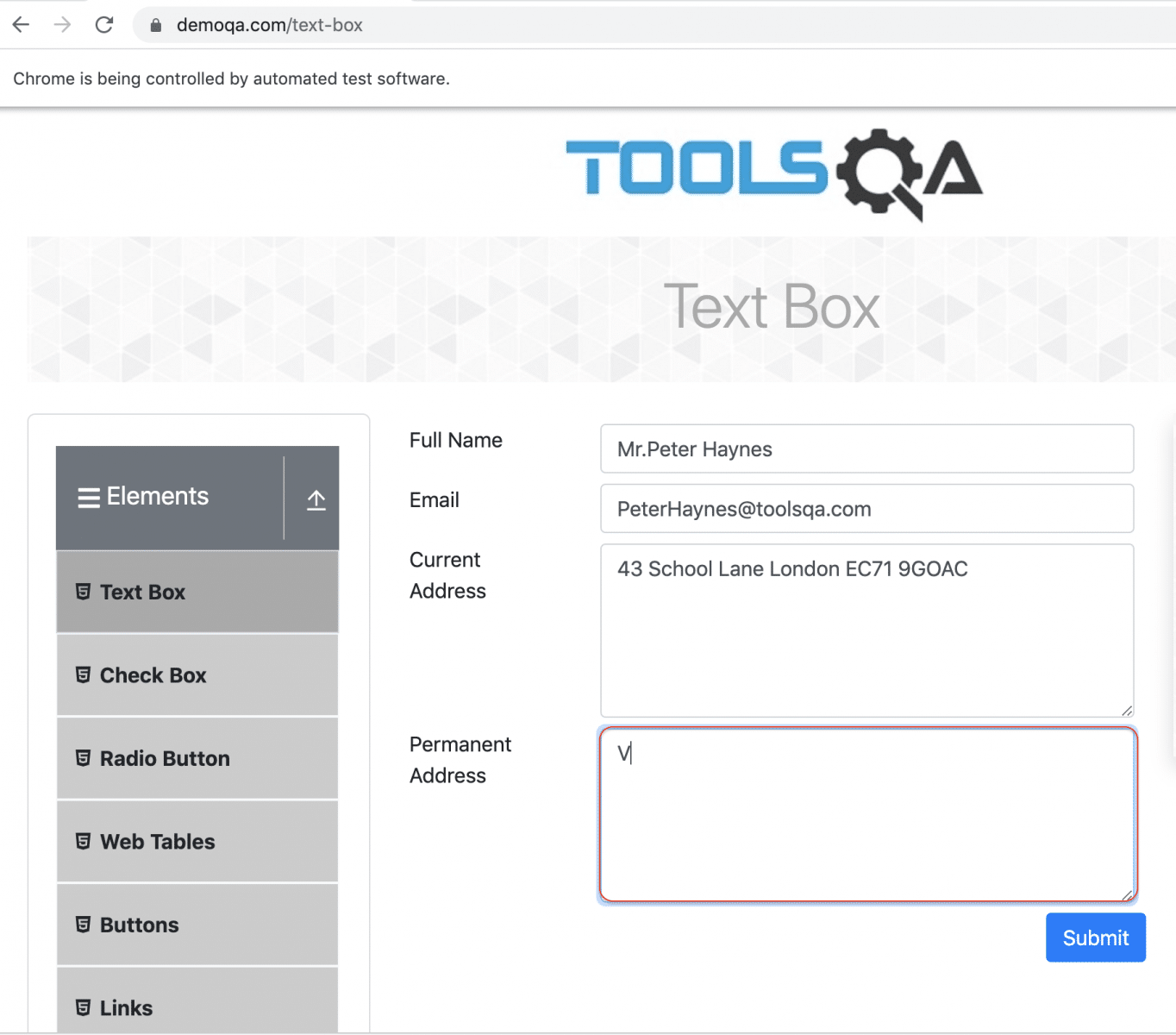
driver.close();

}

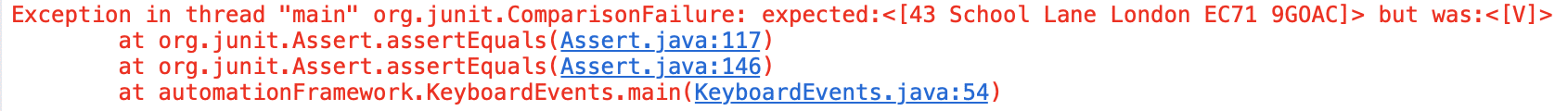
}

In the above code snippet, though the ***sendKeys()*** method of WebElementallows using the ***Control Key,*** it cannot do the copy and paste action, as it fails to combine the key sequences.

As we can see from the following screenshot, instead of pasting the content of Current Address in the Permanent Address text field, it just pasted the character ***"V".***



As is clear from the above screenshot, that the text of the Current Address text field did not copy to the Permanent Address text field, so it leads to failure of the assertion when the comparison of the text of these two fields happens. So, when we execute the above test, it fails with the following error message:

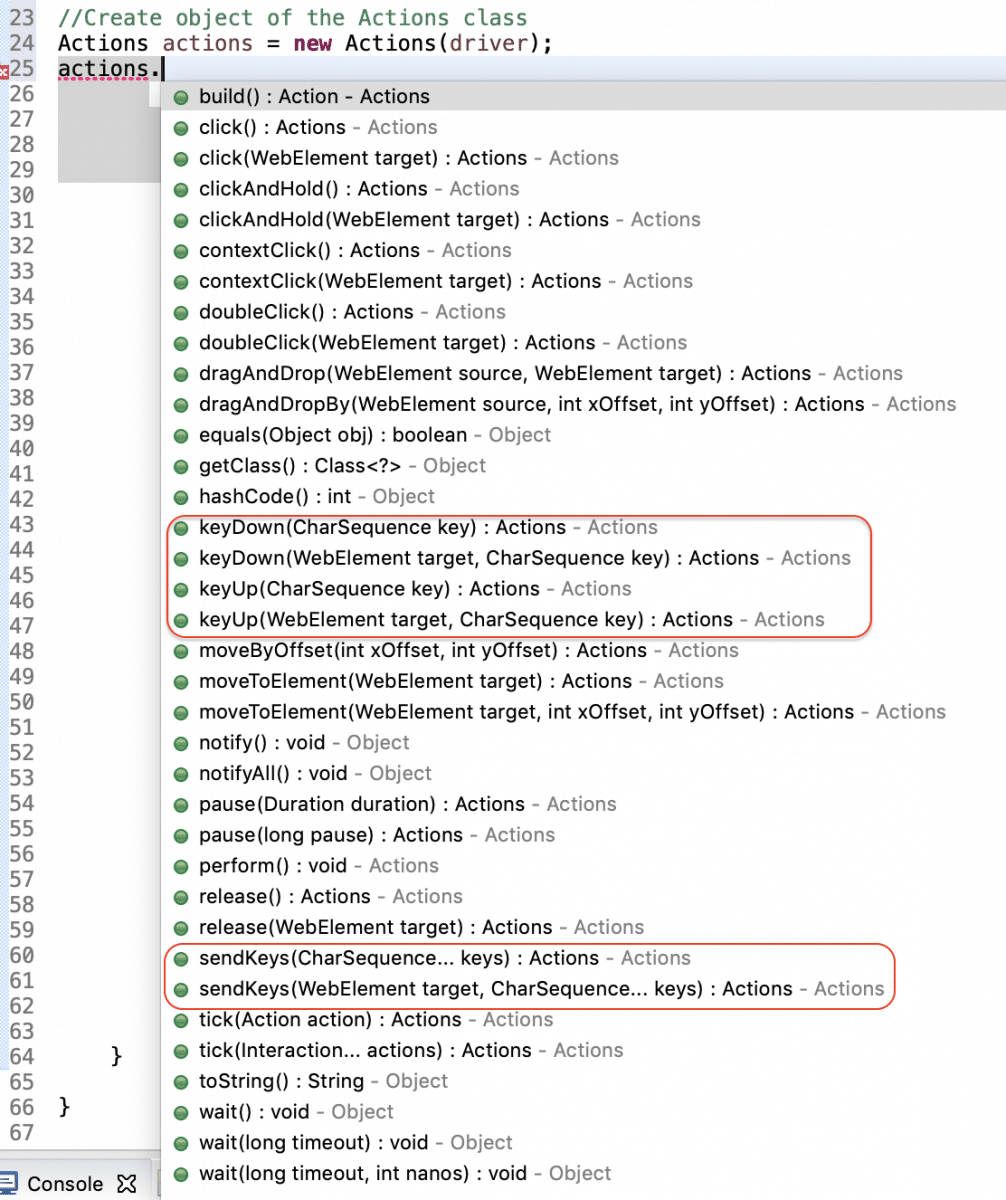


So, this is where the ***sendKeys() method of the WebElement*** class fails. In other words, it fails when we need to combine special keys such as ***"SHIFT"***, ***"CONTROL"***, etc. with the different key sequences, which we all know, is a prevalent scenario when we as a user are using any of the web applications.

Therefore, this is where the ***Actions class of Selenium WebDriver*** comes into the picture, which provides various methods to specifically handle operations of these meta keyboard keys, which need to press while performing operations on the other keyboard keys. Let's see how we can handle such keyboard actions, using the Actions class of Selenium WebDriver.

## What is Actions Class in Selenium WebDriver?

As we discussed above, ***Selenium WebDriver*** provides a class named ***"Actions"***, which provides various methods that can help in automating and simulating the ***Keyboard and Mouse actions.*** The below figure shows the exhaustive list of methods offered by Selenium Web Driver, and the highlighted ones are the most used methods for simulating the Keyboard actions:



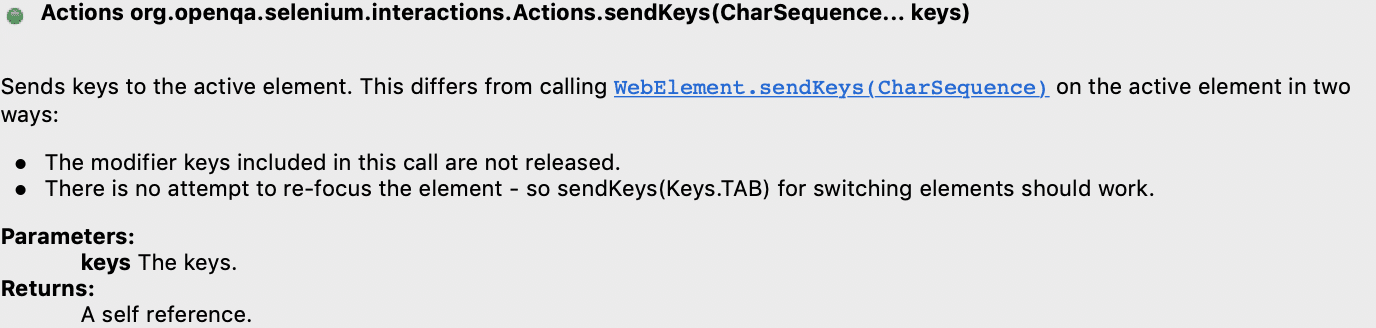
Let's understand the Keyboard specific methods provided by the ***Actions*** Class:

### *****What are the different methods provided by the Actions class for Keyboard Events?*****

As highlighted in the above screenshot, the Actions class majorly provide the following three methods for simulating the Keyboard events:

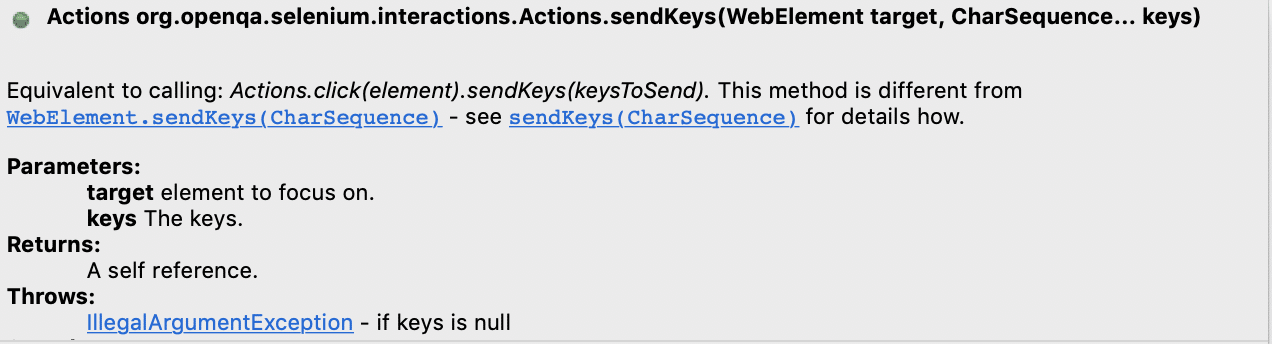
1. ***sendKeys():*** This method sends a series of keystrokes to a given web element. This method has two overloaded formats:

* ***sendKeys***(CharSequence... KeysToSend): The following screenshot shows the syntactical details of this method:



This method sends a sequence of keys to a currently focused web element, i.e., if we want to send specific characters to a web element, that element must be first focussed, then only the mentioned characters will go to that web element.

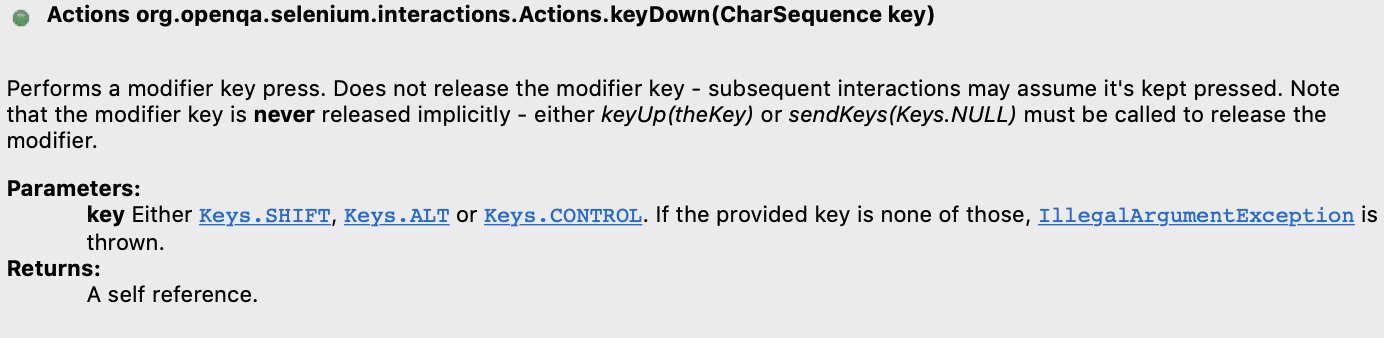
* ***sendKeys***(WebElement element, CharSequence... KeysToSend): The following screenshot shows the syntactical details of this method:



This implementation of the sendKeys() method sends a sequence of characters/keys to a specific web element, which passes as the first parameter to the method. This method first focuses on the target web element and then performs the same action as sendKeys(CharSequence keys).

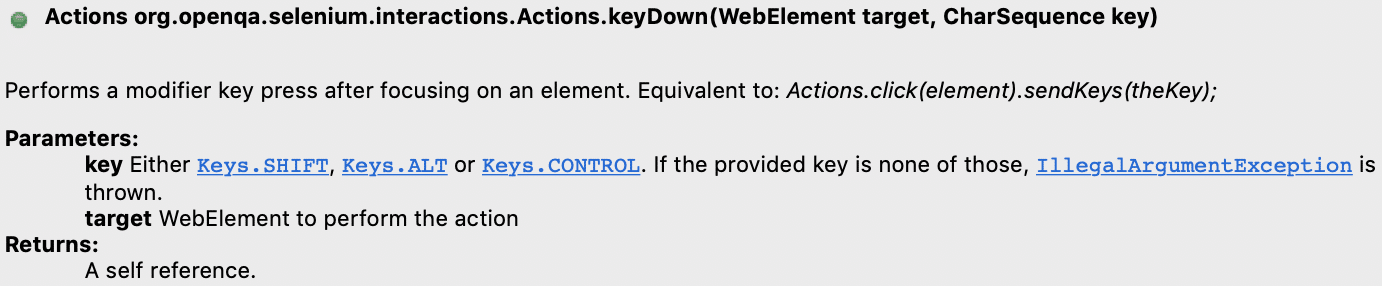
1. ***keyDown():*** This method simulates a keyboard action when a specific keyboard key needs to press. So, whenever you need to press a key and then perform specific other actions, we can use the keyDown() method to keep the key pressed. E.g., say a user has to type some characters in Capital. Then to simulate user behavior, where the user presses the ***SHIFT***  key and then presses the set of characters that need to type in Capital. This method is also available in the following two overloaded variants:

* ***keyDown***(CharSequence key): The following screenshot shows the syntactical details of this method:



This method presses the specified key on the currently focussed Web Element. This method generally presses the ***"Modifier keys"*** such as SHIFT, CTRL, etc. If you want to press the keyboard key on a specified web element, then that web element first needs to be focussed explicitly, and then this method needs to be invoked.

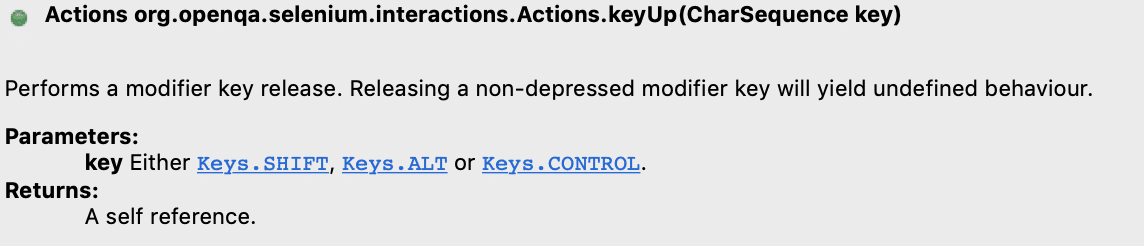
* ***keyDown***(WebElement element, CharSequence key): The following screenshot shows the syntactical details of this method:



This method first focusses on the web element, which has been passed as a parameter to the method and presses the mentioned key on that Web Element.

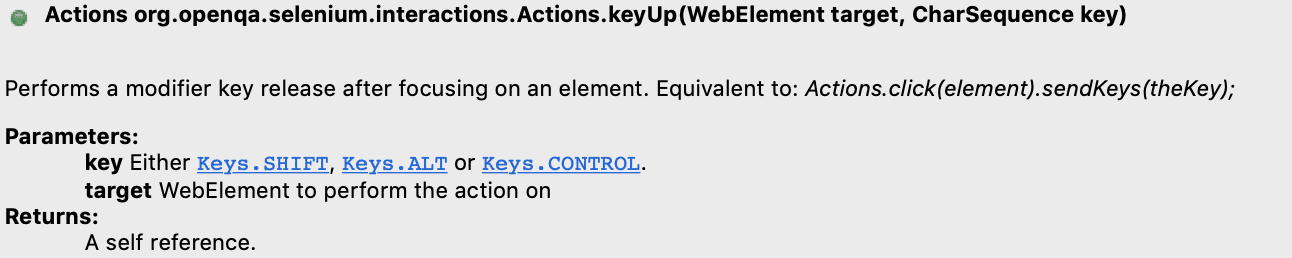
1. ***keyUp():*** We use this method majorly in collaboration with the ***keyDown()*** method. The keyboard key which presses using the ***keyDown()*** method, doesn't get released automatically, so the same need to be explicitly released using the ***keyUp()*** method. So, similar to the ***keyDown()*** method, this method has two overloaded variants:

* ***keyUp***(CharSequence key): The following screenshot shows the syntactical details of this method:



This method releases the specified key on the currently focussed Web Element. If you want to release the keyboard key on a specified web element, then that web element first needs to be focussed explicitly, and then this method needs to be invoked.

* ***keyUp***(WebElement element, CharSequence key): The following screenshot shows the syntactical details of this method:



This method first focusses on the web element, which gets passed as a parameter to the method. Then, it releases the mentioned key on that Web Element.

Conclusively, we are clear about all the keyboard specific methods provided by the Actions class. Subsequently, let's see how we can automate the user as mentioned above scenario using the methods provided by the Actions class of Selenium WebDriver.

Let's modify the above-written code-snippet to use the methods of Actions class, instead of using the methods of WebElement class:

package automationFramework;

import static org.junit.Assert.assertEquals;

import org.openqa.selenium.By;

import org.openqa.selenium.Keys;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

import org.openqa.selenium.interactions.Actions;

public class KeyboardEventsUsingActions {

public static void main(String[] args) {

// Initialize ChromeDriver

// Here we assume that the ChromeDriver path has been set in the System Global variables

WebDriver driver=new ChromeDriver();

//Navigate to the demo site

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

//Create object of the Actions class

Actions actions = new Actions(driver);

// Enter the Full Name

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

fullName.sendKeys("Mr.Peter Haynes");

//Enter the Email

WebElement email=driver.findElement(By.id("userEmail"));

email.sendKeys("PeterHaynes@toolsqa.com");

// Enter the Current Address

WebElement currentAddress=driver.findElement(By.id("currentAddress"));

currentAddress.sendKeys("43 School Lane London EC71 9GO");

// Select the Current Address using CTRL + A

actions.keyDown(Keys.CONTROL);

actions.sendKeys("a");

actions.keyUp(Keys.CONTROL);

actions.build().perform();

// Copy the Current Address using CTRL + C

actions.keyDown(Keys.CONTROL);

actions.sendKeys("c");

actions.keyUp(Keys.CONTROL);

actions.build().perform();

//Press the TAB Key to Switch Focus to Permanent Address

actions.sendKeys(Keys.TAB);

actions.build().perform();

//Paste the Address in the Permanent Address field using CTRL + V

actions.keyDown(Keys.CONTROL);

actions.sendKeys("v");

actions.keyUp(Keys.CONTROL);

actions.build().perform();

//Compare Text of current Address and Permanent Address

WebElement permanentAddress=driver.findElement(By.id("permanentAddress"));

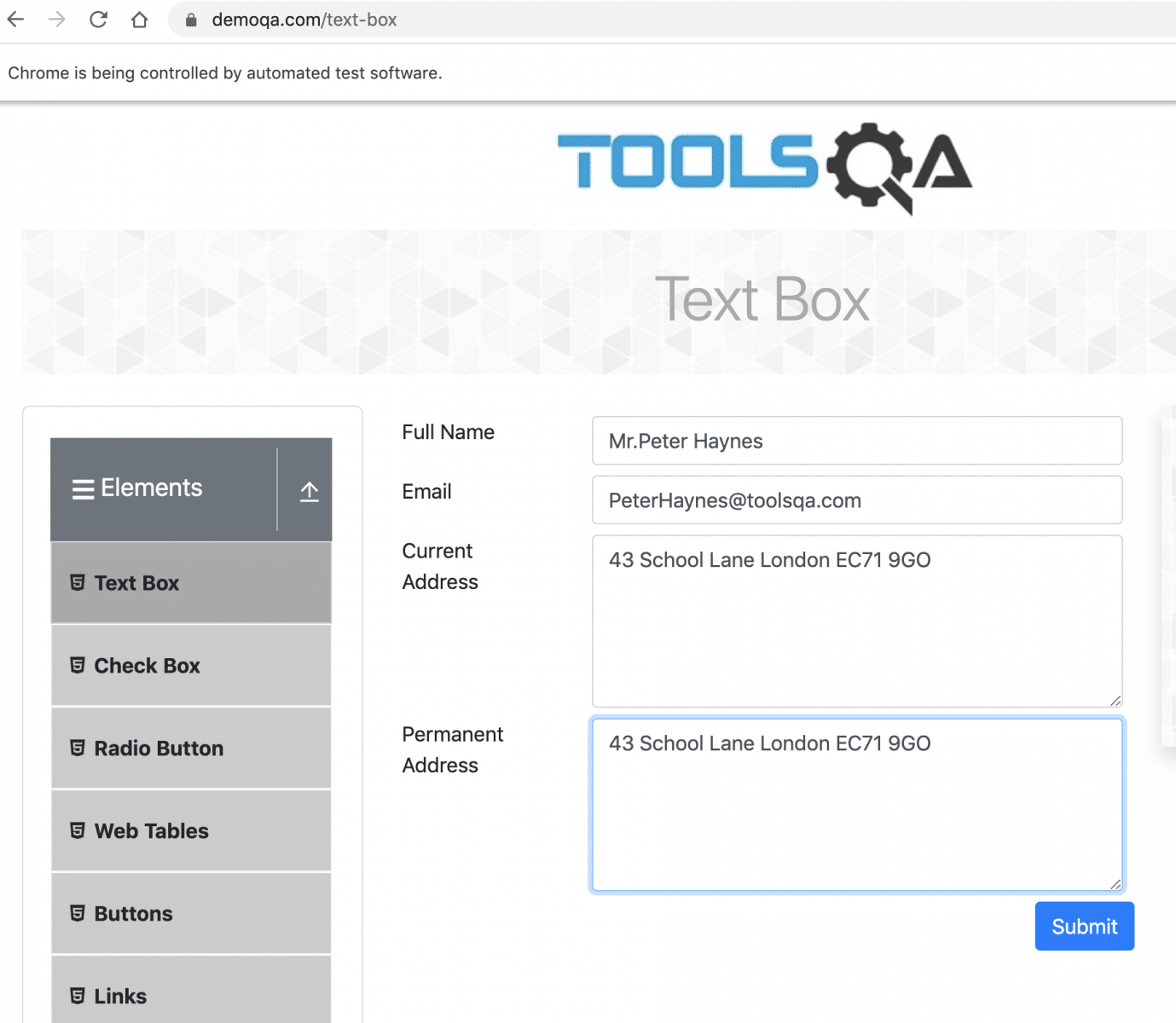
assertEquals(currentAddress.getAttribute("value"),permanentAddress.getAttribute("value"));

driver.close();

}

}

When we run the above code snippet, we get a sample output, as shown below:



As we can see in the above screenshot, copying of the address from the ***"Current Address "*** field to the ***"Permanent Address "*** field was successful. Few more point, which we should explicitly focus on the above code:

* Whichever ***META*** key (Eg ***CONTROL*** in the above use case) we press using the ***keyDown()*** method, it must be released using the ***keyUp()*** method. Otherwise, it will remain pressed and can cause side-effects on the next line of code.
* All the commands of the ***"Actions"*** class perform/ execute their operations when we invoke the ***"build()"*** and ***"perform()"*** methods. So, each of the expected actions/commands should follow by these methods.
* The above code will work on the Windows platform only, as the ***CTRL+C, etc.,*** is only the Windows-specific operation. We can update the platform-specific keys when we need to run the same program on other platforms.

## How to handle contiguous Keyboard Actions using Actions Class?

As noticed in the above sections, all the methods of the ***Actions*** class briefed above returns an ***object of the Actions class*** only. So, this gives us the flexibility of using the [***"Chaining of Methods"***](https://stackoverflow.com/questions/2872222/how-to-do-method-chaining-in-java-o-m1-m2-m3-m4), where we can club all the method invocations specific to one operation in one line of code only.

Let's modify the above-written code to trim it down further. We will use the Chaing of Methods  and will handle the various contiguous Keyboard operations in a single go:

package automationFramework;

import static org.junit.Assert.assertEquals;

import org.openqa.selenium.By;

import org.openqa.selenium.Keys;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

import org.openqa.selenium.interactions.Actions;

public class KeyboardEventsUsingActions {

public static void main(String[] args) {

// Initialize ChromeDriver

// Here we assume that the ChromeDriver path has been set in the System Global variables

WebDriver driver=new ChromeDriver();

//Navigate to the demo site

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

//Create object of the Actions class

Actions actions = new Actions(driver);

// Enter the Full Name

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

fullName.sendKeys("Mr.Peter Haynes");

//Enter the Email

WebElement email=driver.findElement(By.id("userEmail"));

email.sendKeys("PeterHaynes@toolsqa.com");

// Enter the Current Address

WebElement currentAddress=driver.findElement(By.id("currentAddress"));

currentAddress.sendKeys("43 School Lane London EC71 9GO");

// Select the Current Address

actions.keyDown(Keys.CONTROL).sendKeys("a").keyUp(Keys.CONTROL).build().perform();

// Copy the Current Address

actions.keyDown(Keys.CONTROL).sendKeys("c").keyUp(Keys.CONTROL).build().perform();

//Press the TAB Key to Switch Focus to Permanent Address

actions.sendKeys(Keys.TAB).build().perform();

//Paste the Address in the Permanent Address field

actions.keyDown(Keys.CONTROL).sendKeys("v").keyUp(Keys.CONTROL).build().perform();

//Compare Text of current Address and Permanent Address

WebElement permanentAddress=driver.findElement(By.id("permanentAddress"));

assertEquals(currentAddress.getAttribute("value"),permanentAddress.getAttribute("value"));

driver.close();

}

}