Locators

A locator is a way to identify elements on a page. It is the argument passed to the [Finding element](https://www.selenium.dev/documentation/webdriver/elements/finders/) methods.

Locating elements in Selenium WebDriver is performed with the help of findElement() and findElements() methods provided by WebDriver and WebElement class.

* findElement() returns a WebElement object based on a specified search criteria or ends up throwing an exception if it does not find any element matching the search criteria.
* findElements() returns a list of WebElements matching the search criteria. If no elements are found, it returns an empty list.

The following table lists all the Java syntax for locating elements in Selenium WebDriver.

|  |  |  |
| --- | --- | --- |
| **Method** | **Syntax** | **Description** |
| By ID | driver.findElement(By.id (<element ID>)) | Locates an element using the ID attribute |
| By name | driver.findElement(By.name (<element name>)) | Locates an element using the Name attribute |
| By class name | driver.findElement(By.className (<element class>)) | Locates an element using the Class attribute |
| By tag name | driver.findElement(By.tagName (<htmltagname>)) | Locates an element using the HTML tag |
| By link text | driver.findElement(By.linkText (<linktext>)) | Locates a link using link text |
| By partial link text | driver.findElement(By.partialLinkText (<linktext>)) | Locates a link using the link's partial text |
| By CSS | driver.findElement(By.cssSelector (<css selector>)) | Locates an element using the CSS selector |
| By XPath | driver.findElement(By.xpath (<xpath>)) | Locates an element using XPath query |

By XPath

XPath stands for XML path language. It is a query language for selecting nodes from an XML document. XPath is based on the tree representation of XML documents and provides the ability to navigate around the tree by selecting nodes using a variety of criteria.

Aaa

Xpath  
XPath = //tag\_name[@Attribute\_name = “Value of attribute”]

Xpath

Predicates:Predicates find a specific node/element by its index

xpath

// Pipe “|” - to locate both full name and Email label

List<WebElement> lst = driver.findElements(By.xpath(

"//label[@\*= 'userName-label']|//label[@\*= 'userEmail-label']"));

Types of xPath

Absolute :/ 🡪 starts the search from first or root node (/)

Relative :// 🡪 starts the search from the specified tag (//)

**Contains()** is a method used in XPath expression. It is used when the value of any attribute changes dynamically.

**XPath starts-with()** is a function used for finding the web element whose attribute value gets changed on refresh or by other dynamic operations on the webpage.

The **XPath text() function** is a built-in function of selenium webdriver which is used to locate elements based on text of a web element.

**What are XPath Functions in Selenium?**

XPath Contains() function

XPath Starts-with() function

XPath Text() function

//tag\_name[contains(@attribute,'value\_of\_attribute')]

//input[contains(@id, "userN")]

//tag\_name[starts-with(@attribute,'Part\_of\_Attribute\_value')]

//input[starts-with(@placeholder,"Fu")]

//tag\_name[text()='Text of the element']

//label[text()=”Email”]

**AND & OR operators**

//tag\_name[@name = 'Name value' and @id = ‘ID value’]

//tag\_name[@name = 'Name value' or @id = ‘ID value’]

**What are the XPath Axes in Selenium?**

Ancestor Axis

Parent Axis

Sibling

Child Axis

Descendant Axis

Following Axis

Following sibling Axis

Preceding Axis

### 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

### 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

### Child:

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

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

### Preceding:

Select all nodes that come before the current node

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

### 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

### Parent:

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

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

### Self:

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

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

### 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

**xPath axes**

//tag[@attribute ='Attribute\_Value']/ancestor::parent\_node

//label[text()="Full Name"]/ancestor::form

//tag[@attribute ='Attribute\_Value']//child::child\_node

//form[@id='userForm']/child::div[1]//label

//node[attribute='value of attribute']//descendant::attribute

//div[@class= 'custom-control custom-radio custom-control-inline']/descendant::input

//tag[@attribute ='Attribute\_Value']//ancestor::parent\_node

//input[@id="userName"]/following::textarea

//div[@class='col-md-3 col-sm-12']/following-sibling::div

//node[attribute='value of attribute']//preceding::attribute

//input[@id='userName']/preceding::label

Xpath preceding sibling

//td[text()=’Fransic’]//preceding-sibling::td//child::input

\*\*\*\*\*\*\*\*

<https://www.hyrtutorials.com/p/add-padding-to-containers.html>

//div[@class='container']//child::input[@type='text'][1]

The above path gets the div tag, with class =’container’. Then it is querying for child whose tag is input and attribute @type =’textbox’. Of all the listed input we are accessing the 1st input by providing the index [1]

//td[text()='Francisco Chang']//preceding-sibling::td//child::input

\*\*\*\*\*\*\*\*

**To access a parent of node use double dots (..)**

//span[@id=”username”]/.. 🡺 returns the parent of span

//input[@id=”username”]/../..

Nth sub element 🡺 use position() or specify the index

tr[position()=4]

By CSS

The CSS is used as a method to identify the web object, however NOT all browsers support CSS identification.

WebElement loginButton = driver.findElement(By.cssSelector("input.login"));

## **Relative Locators**

**Selenium 4** introduces Relative Locators (previously called as Friendly Locators). These locators are helpful when it is not easy to construct a locator for the desired element, but easy to describe spatially where the element is in relation to an element that does have an easily constructed locator.

\* [***DOM***](https://www.w3.org/TR/WD-DOM/introduction.html) (Document Object Model),

1.  ***DOM*** can be accessed in Google Chrome either by pressing ***F12*** or by ***right click*** on the web page and then by selecting ***Inspect***

By.id("firstName");

By.name("gender");

By.className("practice-form-wrapper");

By.linkText("Home");

By.tagName("a");

By.xpath("//input[@id='userName']");

By.cssSelector("input[id= ‘userName’]");

The basic syntax of identifying a web element using ***CSS*** is as follows:

css=(HTML Page)[Attribute=Value]

CSS Selector

CSS Selector

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

tag id 🡺 tag#id

tag class 🡺 tag.class

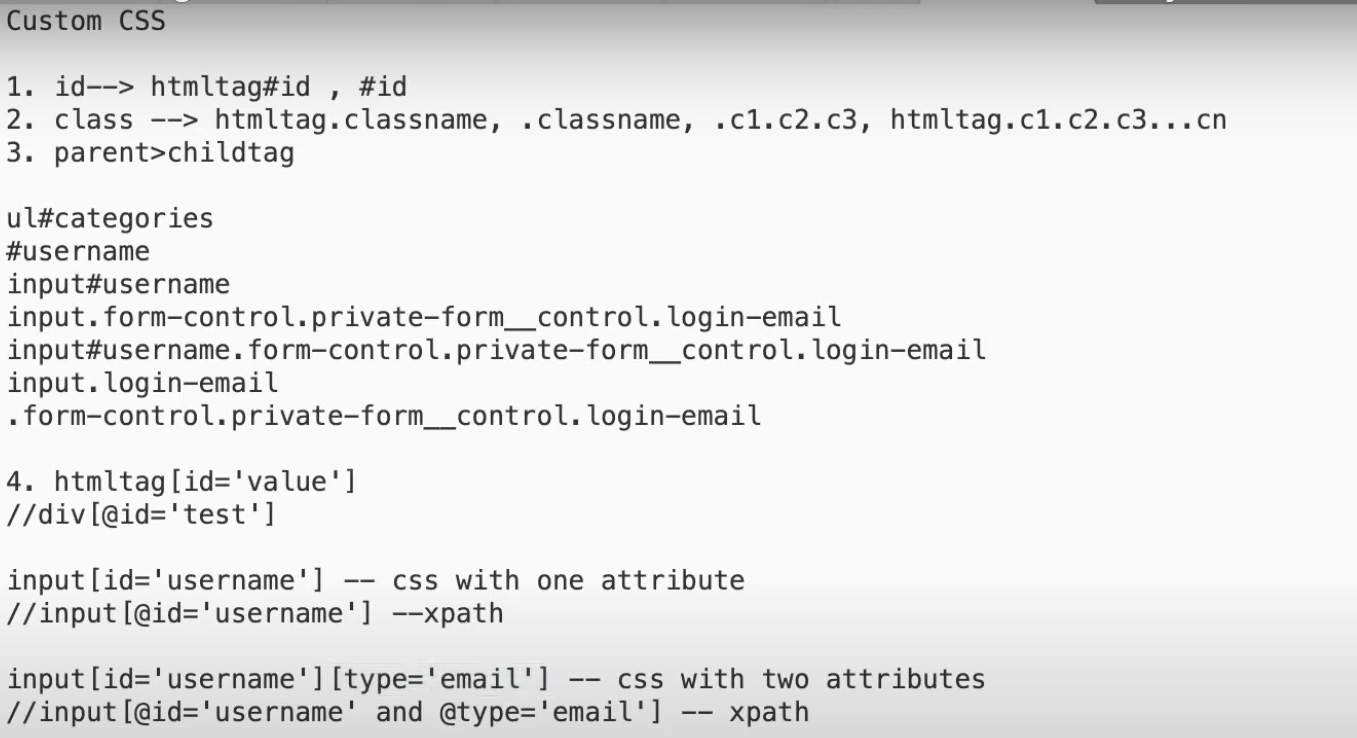
tag attribute 🡺 tag[attribute=value]

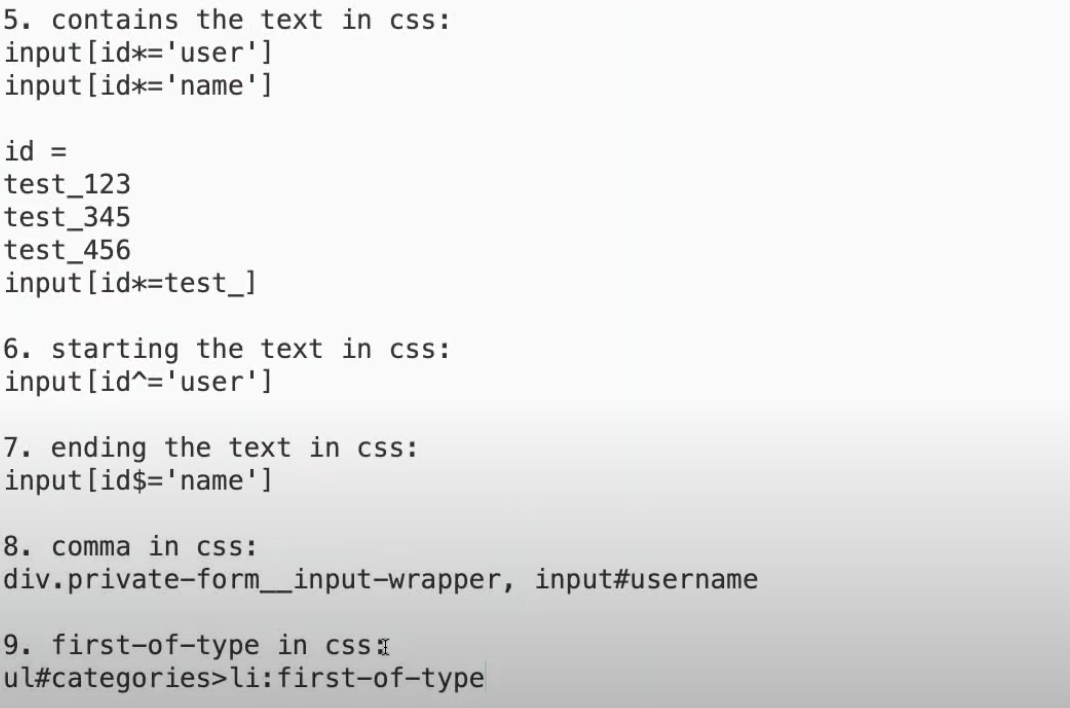
tag class attribute 🡺 tag.class[attribute=value]

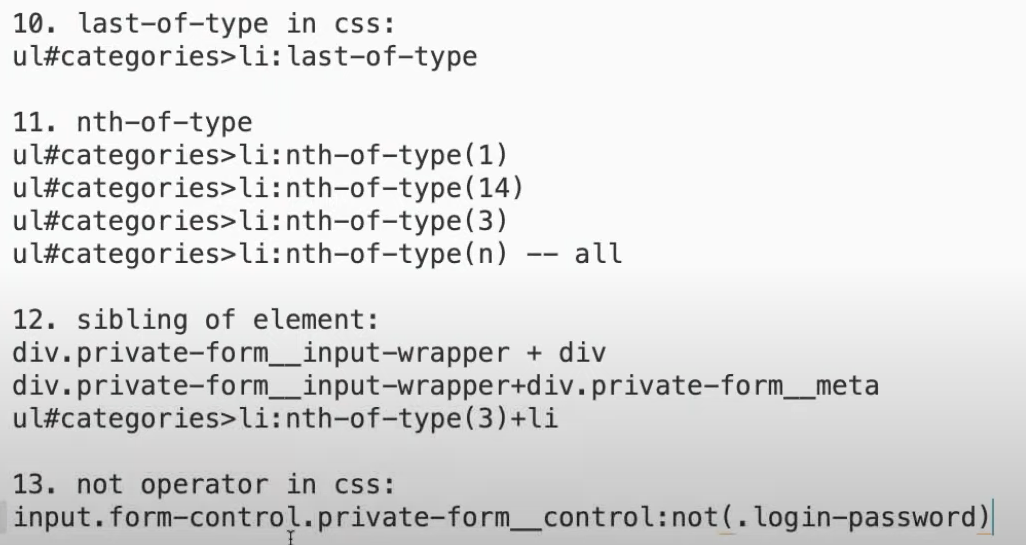
Universal selector 🡺 \* = driver.findElments(By.cssSelector(“\*”));

Parent> child

Ex: ul#categories>li[class='201429665']







here is the list of all CSS locators discussed in video:

**Custom CSS**

1. id--> htmltag#id , #id

2. class --> htmltag.classname, .classname, .c1.c2.c3, htmltag.c1.c2.c3...cn

3. parent>childtag

|  |
| --- |
|  |
| 1.ul#categories |
| #username |
| input#username |
| input.form-control.private-form\_\_control.login-email |
| 2input#username.form-control.private-form\_\_control.login-email |
| input.login-email |
| .form-control.private-form\_\_control.login-email |
|  |
| 4. htmltag[id='value'] |
| //div[@id='test'] |
|  |
| input[id='username'] -- css with one attribute |
| //input[@id='username'] –xpath |
|  |
| input[id='username'][type='email'] -- css with two attributes |
| //input[@id='username' and @type='email'] – xpath |
|  |
| 5. contains the text in css: |
| input[id\*='user'] |
| input[id\*='name'] |
|  |
| id = |
| test\_123 |
| test\_345 |
| test\_456 |
| input[id\*=test\_] |
|  |
| 6. starting the text in css: |
| input[id^='user'] |
|  |
| 7. ending the text in css: |
| input[id$='name'] |
|  |
| 8. comma in css: |
| div.private-form\_\_input-wrapper, input#username |
|  |
| 9. first-of-type in css: |
| ul#categories>li:first-of-type |
|  |
| 10. last-of-type in css: |
| ul#categories>li:last-of-type |
|  |
| 11. nth-of-type |
| ul#categories>li:nth-of-type(1) |
| ul#categories>li:nth-of-type(14) |
| ul#categories>li:nth-of-type(3) |
| ul#categories>li:nth-of-type(n) – all |
|  |
| 12. sibling of element: |
| div.private-form\_\_input-wrapper + div |
| div.private-form\_\_input-wrapper+div.private-form\_\_meta |
| ul#categories>li:nth-of-type(3)+li |
|  |
| 13. not operator in css: |
| input.form-control.private-form\_\_control:not(.login-password) |

<https://guide.blazemeter.com/hc/en-us>

|  |
| --- |
| ul#categories 🡪 by id  htmltag#ID |
| ul#categories>li>a>span.red\_trangle 🡺 15 items selected  htmlTag#id>htmltag>htmltag>htmltag.class |
| ul#categories>li:nth-of-type(2)>a.parent>span.red\_trangle:last-of-type  starts with htmltag  # 🡪 id  > 🡪 next htmltag  : 🡪 specifies the nth element  > 🡪 next htmltag  . 🡪 class  > 🡪 next htmltag  . 🡪 class  : 🡪 last element |
| ul#categories>li:first-of-type 🡪fist element  ul#categories>li:last-of-type 🡪last element  ul#categories>li:nth-of-type(5) 🡪 nth element  ul#categories>li:nth-of-type(n) 🡪 all elements |
| Sibling of element  div.private-form\_\_input-wrapper+div  htmltag.class+htmltag  next sibling of the specified element  ul#categories>li:nth-of-type(5)+li |
| <https://app.hubspot.com/login>  to capture username and password  class = "form-control private-form\_\_control login-email" ---> email element  class = "form-control private-form\_\_control login-password m-bottom-3" ==-> password element  Select the username  Class=”form-control private-form\_\_control login-email” 🡪 this element has 3 classes separated by space 🡪 while getting css replace space with “.”  input.form-control.private-form\_\_control.login-email  Not operator 🡪this select the password  input.form-control.private-form\_\_control:not(.login-email) |
|  |

Interview quest : to fint the custom CSS path

**What Are The Main Traits Of A Software Test Automation Framework?**

* A test automation framework should have the following features.
* Flexible in the programming language selection
* Support of keywords and actions
* Provision of data sources for input
* Allow test case creation and modification
* Define test case priority
* Manual or automated execution
* Maintain test results history
* Generate test metrics such as test vs. code coverage
* Report creation
* CI tool integration such as Jenkins
* Cross-browser and cross-platform support

**What Type Of Test Framework Did You Create Using Selenium?**

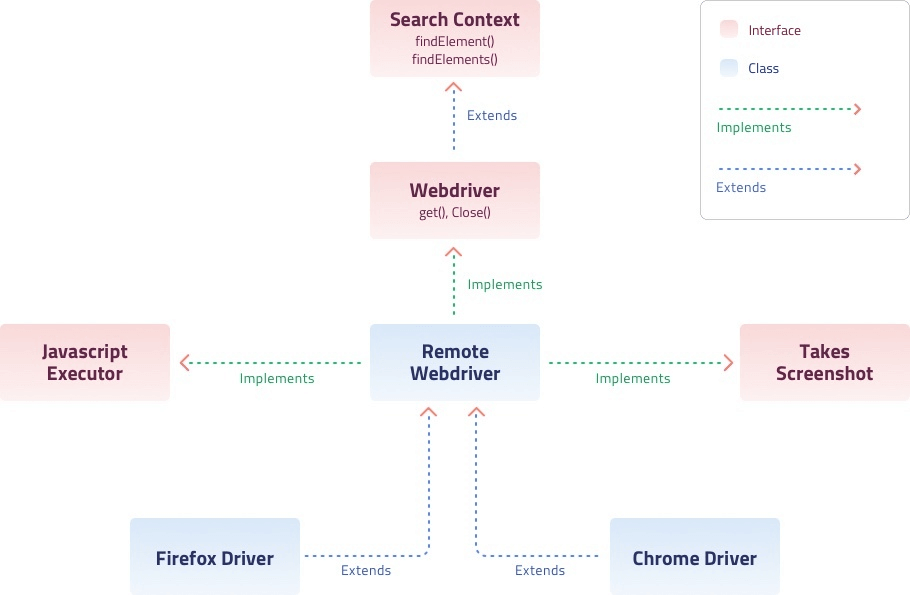
* While replying to such questions, stay focused, and keep your answer short and crisp. You can start by telling about the different components in your framework and then explain them one by one.
* Here is an illustration for your help.
* I worked on a framework built on top of the Page Factory framework.
* I’ve created a page class for every web page in my application. It keeps the objects and the handler functions.
* Every page class has a followup test class where I create tests for related use cases.
* I used separate packages to host the pages and their test classes. It’s a best practice to do that way.
* The framework also had a lib package for utility and some standard wrapper functions over Selenium APIs.
* Java is core programming language used for this project. It was primarily because the team had previous Java experience. Also, we could utilize the TestNG annotations and report features.
* Most test cases are data-driven. They require input from the external data source. So, I used Java property/POI class to read from the CSV/XLS files.
* We used the TestNG group feature for labeling test cases as P1, P2, and P3.
* The Log4J library provided the necessary support for tracing in our project.
* Instead of using the TestNG reporting, we preferred the Extent report. It has more graphical options and gives an in-depth analysis of the results.
* We built the framework with the help of Maven. Also, Jenkins provided support for automated build and execution.
* Bitbucket allowed us to manage our source code using git repositories.

**What Are Challenges Have You Faced With Selenium? And How Did You Overcome Them?**

Here are some of the problems that testers usually face while doing automation with Selenium.

* **Wrong implementation:** I used the [**page object model**](https://www.techbeamers.com/implement-page-object-model-pom-with-selenium-and-web-driver-2-0/) but had it implemented incorrectly. My classes were focusing on the web elements rather than they should have resembled the user actions.
* **Duplicate code:** The project had many category pages. Each category had a different search function instead of handling them at a central place.
* **Ineffective use of wait:** I used implicit wait with a fixed timeout. But some pages were timing out due to higher load time. I had to adopt the Fluent wait (with a variable timeout) to overcome this problem.
* **Improper error handling:** It was getting hard to debug the cause of a failed test. At some places, the {try-catch} blocks were missing, and hence, cases were skipping w/o giving a proper reason. Therefore, I had to refactor the code by adding asserts and exception handling.
* **Inconsistent XPath:** Most of the locators were using the XPath method. And the developers kept them changed while fixing new defects. I called up a discussion with them and agreed to have a fixed XPath or an ID for the web elements.
* **Performance & Localization:** We were using the flat files (CSV) initially to feed data to test cases. However, it had us failed in testing localization as well as beaten us on the performance. Ee migrated all of our test data to MySQL and fixed both issues.
* **Monolithic tests:** Earlier tests weren’t using the labeling. Honestly, there wasn’t a way to do it. Hence, we integrated our test suite with TestNG and got away with this limitation. Now, we have many test groups like features-based (F1, F2, F3…), priority-based (P1, P2, P3).

WebDriver



## **IDE**

[IDE](https://selenium.dev/selenium-ide) (Integrated Development Environment) is the tool you use to develop your Selenium test cases. It’s an easy-to-use Chrome and Firefox extension and is generally the most efficient way to develop test cases. It records the users’ actions in the browser for you, using existing Selenium commands, with parameters defined by the context of that element. This is not only a time-saver but also an excellent way of learning Selenium script syntax.

## **Grid**

Selenium Grid allows you to run test cases in different machines across different platforms. The control of triggering the test cases is on the local end, and when the test cases are triggered, they are automatically executed by the remote end.

**Alerts**

driver.findElement(By.*cssSelector*("input[value=\"Go!\"]")).click();

Thread.*sleep*(5000);

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

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

## Waits

In automation testing, wait commands **direct test execution to pause for a certain length of time before moving onto the next step**. This enables WebDriver to check if one or more web elements are present/visible/enriched/clickable, etc.

There are two kinds of waits.

1. Implicit wait – used to set the default waiting time throughout the program
2. Explicit wait – used to set the waiting time for a particular instance only

## Implicit Wait

* It is simpler to code than Explicit Waits.
* It is usually declared in the instantiation part of the code.
* You will only need one additional package to import.

To start using an implicit wait, you would have to import this package into your code.

First Selenium Webdriver Script: JAVA Code Example

Then on the instantiation part of your code, add this.

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

## Explicit Wait

**Explicit waits are done using the WebDriverWait and ExpectedCondition classes**. For the following Selenium WebDriver example, we shall wait up to 10 seconds for an element whose id is “username” to become visible before proceeding to the next command. Here are the steps.

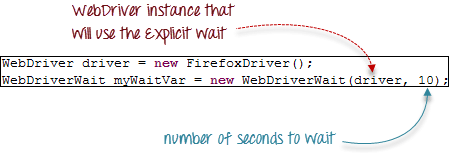
**Step 1**

Import these two packages:

**First Selenium Webdriver Script: JAVA Code Example**

**Step 2**

Declare a WebDriverWait variable. In this example, we will use “myWaitVar” as the name of the variable.



**Step 3**

Use myWaitVar with ExpectedConditions on portions where you need the explicit wait to occur. In this case, we will use explicit wait on the “username” (Mercury Tours HomePage) input before we type the text “tutorial” onto it.

First Selenium Webdriver Script: JAVA Code Example

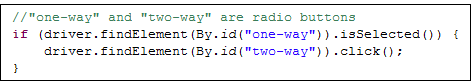
## Conditions

Following  methods are used  in conditional and looping operations —

* **isEnabled()** is used when you want to verify whether a certain element is enabled or not before executing a command.
* **isDisplayed()** is used when you want to verify whether a certain element is displayed or not before executing a command.

First Selenium Webdriver Script: JAVA Code Example

* **isSelected()** is used when you want to verify whether a certain **check box, radio button, or option in a drop-down box** is selected. It does not work on other elements.



## Using ExpectedConditions

The ExpectedConditions class offers a wider set of conditions that you can use in conjunction with WebDriverWait’s until() method.

Below are some of the most common ExpectedConditions methods.

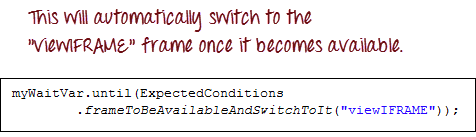
* **alertIsPresent()**– waits until an alert box is displayed.

First Selenium Webdriver Script: JAVA Code Example

* **elementToBeClickable()** – Waits until an element is visible and, at the same time, enabled. The sample Selenium Code below will wait until the element with id=”username” to become visible and enabled first before assigning that element as a WebElement variable named “txtUserName”.

First Selenium Webdriver Script: JAVA Code Example

* **frameToBeAvailableAndSwitchToIt()**– Waits until the given frame is already available, and then automatically switches to it.



* WebDriver provides these useful **get commands**:
* get()
* getTitle()
* getPageSource()
* getCurrentUrl()
* getText()
* WebDriver provides these useful **navigation commands**
* navigate().forward()
* navigate().back()
* navigate().to()
* navigate().refresh()
* The close() and quit() methods are used to close browser windows. **Close()** is used to close a single window; while **quit()** is used to close all windows associated to the parent window that the WebDriver object was controlling.
* The **switchTo().frame()** and **switchTo().alert()** methods are used to direct WebDriver’s focus onto a frame or alert, respectively.
* **Implicit waits** are used to set the waiting time throughout the program, while **explicit waits** are used only on specific portions.
* You can use the **isEnabled(), isDisplayed(),isSelected(),** and a combination of **WebDriverWait** and **ExpectedConditions** methods when verifying the state of an element. However, they do not verify if the element does not exists.
* When isEnabled(), isDisplayed(),or isSelected() was called while the element was not existing, WebDriver will throw a **NoSuchElementException**.
* When WebDriverWait and ExpectedConditions methods were called while the element was not existing, WebDriver would throw a **TimeoutException**.

**Note:**

driver.get() : It’s used to go to the particular website , But it doesn’t maintain the browser History and cookies so , we can’t use forward and backward button , if we click on that , page will not get schedule

driver.navigate() : it’s used to go to the particular website , but it maintains the browser history and cookies, so we can use forward and backward button to navigate between the pages during the coding of Testcase

## Locating by DOM (Document Object Model)

The [Document Object Model (DOM)](https://www.guru99.com/understanding-dom-fool-guide.html), in simple terms, is the way by which HTML elements are structured.

## Locating by DOM – getElementById

Let us focus on the first method – using the getElementById method of DOM in Selenium. The syntax would be:

**Syntax**

document.getElementById("id of the element")

* id of the element = this is the value of the ID attribute of the element to be accessed. This value should always be enclosed in a pair of parentheses (“”).

## Locating by DOM – getElementsByName

**Syntax**

document.getElementsByName(“name“)[index]

* name = name of the element as defined by its ‘name’ attribute
* index = an integer that indicates which element within getElementsByName’s array will be used.

**getElementsByName**

* It will get a collection of elements whose names are all the same.
* Each element is indexed with a number starting from 0 just like an array
* You specify which element you wish to access by putting its index number into the square brackets in getElementsByName’s syntax below.

## FindElement() and FindElements()

|  |  |
| --- | --- |
| **Find Element** | **Find Elements** |
| Returns the first most web element if there are multiple web elements found with the same locator | Returns a list of web elements |
| Throws exception NoSuchElementException if there are no elements matching the locator strategy | Returns an empty list if there are no web elements matching the locator strategy |
| Find element by XPath will only find one web element | It will find a collection of elements whose match the locator strategy. |
| Not Applicable | Each Web element is indexed with a number starting from 0 just like an array |

WebElement loginLink = driver.findElement(By.linkText("Login"));

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

## Troubleshooting NoSuchElementException()

If you encounter NoSuchElementException() while finding elements, it means that the element is not found in the page at the point the Web driver accessed the page.

1. Check your locator again using Firepath or Inspect Element in Chrome.
2. Check whether the value you used in the code is different from the one for the element in Firepath now.
3. Some properties are dynamic for few elements. In case, you find that the value is different and is changing dynamically, consider using By.xpath() or By.cssSelector() which are more reliable but complex ways.
4. Sometimes, it could be a wait issue too i.e., the Web driver executed your code even before the page loaded completely, etc.
5. Add a wait before findElement() using implicit or explicit waits.

## Accessing Image Links

Image links are the links in web pages represented by an image which when clicked navigates to a different window or page.

Since they are images, we cannot use the By.linkText() and By.partialLinkText() methods because image links basically have no link texts at all.

In this case, we should resort to using either By.cssSelector or By.xpath. The first method is more preferred because of its simplicity.

## What are Broken Links?

Broken links are links or URLs that are not reachable. They may be down or not functioning due to some server error

An URL will always have a status with 2xx which is valid. There are different HTTP status codes which are having different purposes. For an invalid request, HTTP status is 4xx and 5xx.

4xx class of status code is mainly for client side error, and 5xx class of status codes is mainly for the server response error.

|  |
| --- |
| int brokenLinkcount =0;  List <WebElement> links = driver.findElements(By.tagName("a"));  for(WebElement element : links) {  String str = element.getAttribute("href");  System.out.println("str = " + str);  if (str == null || str.isEmpty() ){  System.out.println("URL is empty");  continue;  }    URL link = new URL(str);  try {  HttpURLConnection httpconn = (HttpURLConnection) link.openConnection();  httpconn.connect();  if(httpconn.getResponseCode()>=400) {  System.out.println(httpconn.getResponseCode() + " " +str +" is broken link");  brokenLinkcount++;  }  else {  System.out.println(httpconn.getResponseCode() + " " +str +" is valid link");  }  } catch (Exception e) {  // TODO Auto-generated catch block  //e.printStackTrace();  System.out.println("Error occured " + e.getMessage());  }  }  System.out.println("Total broken links =" + brokenLinkcount); |

## What is Link Text in Selenium?

A **Link Text in Selenium** is used to identify the hyperlinks on a web page. It is determined with the help of an anchor tag.

## Select Class in Selenium for **Drop-Down Box**

The **Select Class in Selenium** provides implementation for HTML SELECT tag. The html select tag provides helper methods to select and deselect the elements.

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

fruits.selectByVisibleText("Banana");

fruits.selectByIndex(1);

|  |  |
| --- | --- |
| **Command** | **Description** |
| *selectByVisibleText()/*  *deselectByVisibleText()* | selects/deselects an option by its displayed text |
| *selectByValue()/*  *deselectByValue()* | selects/deselects an option by the value of its “value” attribute |
| *selectByIndex()/*  *deselectByIndex()* | selects/deselects an option by its index |
| *isMultiple()* | returns TRUE if the drop-down element allows multiple selection at a time; FALSE if otherwise |
| *deselectAll()* | deselects all previously selected options |

## Action Class

**Action Class** 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 |

|  |
| --- |
| Actions act = **new** Actions(driver);  WebElement sourceElement = driver.findElement(By.*id*("box3"));  WebElement targetElement = driver.findElement(By.*id*("box103"));  act.dragAndDrop(sourceElement, targetElement).build().perform();    sourceElement = driver.findElement(By.*id*("box5"));  targetElement = driver.findElement(By.*id*("box105"));  act.dragAndDrop(sourceElement, targetElement).build().perform();    System.***out***.println("Double click");  WebElement startElement = driver.findElement(By.*id*("box6"));  act.doubleClick(startElement).build().perform();  System.***out***.println("Done Double click");  act.release();      System.***out***.println("Performing clickAndHold");  startElement = driver.findElement(By.*id*("box7"));  act.doubleClick(startElement).build().perform();  WebElement endElement = driver.findElement(By.*id*("box107"));  act.moveToElement(endElement).build().perform();  System.***out***.println("Releasing");  act.release(endElement).build().perform();    System.***out***.println("clicking");        System.***out***.println("Performing dragAnd Drop by offset");  sourceElement=driver.findElement(By.*id*("countries"));  startElement = driver.findElement(By.*id*("box6"));  act.dragAndDropBy(sourceElement, 300, 100).build().perform();  System.***out***.println("Done Performing dragAnd Drop by offset"); |

## What is Alert 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 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.

|  |
| --- |
| Set <String> sessionIDList =*driver*.getWindowHandles();  **for**(String sessionId : sessionIDList) {  System.***out***.println("SessionId = " + sessionId);  **if** (!mainWindow.equals(sessionId)) {  *driver*.switchTo().window(sessionId);    System.***out***.println("Entering emailID");  *driver*.findElement(By.*name*("emailid")).sendKeys("abc@gmail.com");  Thread.*sleep*(3000);    System.***out***.println("Cliclink login button");  *driver*.findElement(By.*name*("btnLogin")).click();  }  }  System.out.println("Switching to main window");  driver.switchTo().window(mainWindow);  System.out.println("Done"); |

## What is a Web Table in Selenium?

A **Web Table** in Selenium is a WebElement used for the tabular representation of data

## Tooltip in Selenium

A **Tooltip in Selenium** is a text that appears when a mouse hovers over an object on a web page. The object can be a link, an image, a button, a text area, etc. The tooltip text often gives more information about the object on which the user hovers over the mouse cursor.

* Tool Tips are implemented in different ways–
* The basic implementation is based on HTML’s “title” attribute. getAttribute(title) gets the value of the tooltip.
* Other tool tip implementation’s like JQuery, CSS tooltips require Interactions API to create mouse hover effect
* Advanced User Interactions API
* moveToElement(element) of Actions class is used to mouse hover an element.
* Build() method of Actions class builds the sequence of user actions into an Action object.
* Perform() of Action class executes all the sequence of user actions at once.
* In order to verify a tooltip, we have to first mouse-hover the element, then find the element that corresponds to the tool tip and get its text or other values to verify against the expected values.

|  |
| --- |
| String expectedTooltip="Github";  WebElement lnkGitHub = *driver*.findElement(By.*xpath*("//div[@class='soc-ico show-round']//a[position()=4]"));  String actualTooltip = lnkGitHub.getAttribute("title"); |

ToolTip for jquery plugins

|  |
| --- |
| WebElement lnkDownloadNow = *driver*.findElement(By.*xpath*("//a[@id='download\_now']"));  Actions builder = **new** Actions(*driver*);  builder.moveToElement(lnkDownloadNow).build().perform();  WebElement elementToolTip = *driver*.findElement(By.*xpath*("//div[@class='tooltip']"));  System.out.println("Tooltip = " + elementToolTip.getText()); |

## Desired Capabilities

**Desired Capabilities** is a class in Selenium used to set properties of browsers to perform cross browser testing of web applications. It stores the capabilities as key-value pairs and these capabilities are used to set browser properties like browser name, browser version, path of browser driver in the system, etc. to determine the behaviour of browser at run time.

* Desired capability can also be used to configure the driver instance of Selenium WebDriver.
* We can configure driver instance like FirefoxDriver, ChromeDriver, InternetExplorerDriver by using desired capabilities.

### How to scroll down a page using JavaScript?

|  |
| --- |
| JavascriptExecutor js = (JavascriptExecutor) driver;  js.executeScript("window.scrollBy(0,1000)"); |

### Is there a way to type in a textbox without using sendKeys()?

Yes! Text can be entered into a textbox using JavaScriptExecutor

JavascriptExecutor jse = (JavascriptExecutor) driver;

jse.executeScript("document.getElementById(‘email').value=“abc.efg@xyz.com”);

### What does the switchTo() command do?

switchTo() command is used to switch between windows, frames or pop-ups within the application. Every window instantiated by the WebDriver is given a unique alphanumeric value called “Window Handle”.

|  |
| --- |
| driver.switchTo().activeElement()  driver.switchTo().defaultContent()  driver.switchTo().alert()  driver.switchTo().window(sessionID) |

### How to upload a file in Selenium WebDriver?

You can achieve this by using sendkeys() or Robot class method. Locate the text box and set the file path using sendkeys() and click on submit button

Locate the browse button

   WebElement browse =driver.findElement(By.id("uploadfile"));

Pass the path of the file to be uploaded using sendKeys method

    browse.sendKeys("D:\\SeleniumInterview\\UploadFile.txt");

### How to set browser window size in Selenium?

The window size can be maximized, set or resized

To maximize the window

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

To set the window size

  Dimension d = new Dimension(400,600);

  driver.manage().window().setSize(d);

Alternatively,

The window size can be reset using JavaScriptExecutor

 ((JavascriptExecutor)driver).executeScript("window.resizeTo(1024, 768)");

### What is the difference between "/" and "//" in XPath?

The difference between "/" and "//" in XPath is that "/" is used to select an element based on its absolute location, while "//" is used to select an element based on its relative location.

For example, if you want to select the first <p> element on a page, you would use "/p". If you want to select all <p> elements on a page, regardless of their location, you would use "//p".

### How can we launch different browsers in Selenium WebDriver?

We can launch different browsers in Selenium WebDriver using several methods. For example, we can use the setWebDriver() method to specify the path to the browser's executable file. Alternatively, we can use the addCustomProfilePreference() method to add a custom profile preference for the browser. Finally, we can use the launchBrowser() method to launch the browser.

**Selenium WebDriver API:**

Selenium Webdriver API helps in communication between languages and browsers. Selenium supports many programming languages such as Java, C#, Python etc., and also it supports multiple browsers such as Google Chrome, Firefox, Internet Explorer etc.,

[***Real Time Selenium WebDriver Interview Questions***](https://www.softwaretestingmaterial.com/selenium-interview-questions/)

Every browser has different logic of performing actions like loading a page, closing the browser etc.

Here is the Selenium WebDriver Framework Architecture Diagram

Selenium WebDriver Architecture

There are four components of Selenium Architecture:

1. Selenium Client Library
2. JSON Wire Protocol over HTTP
3. Browser Drivers
4. Browsers

***Must Read:***[***Top TestNG Interview Questions***](https://www.softwaretestingmaterial.com/testng-interview-questions/)

**Selenium Client Libraries/Language Bindings:**

Selenium supports multiple libraries such as Java, Ruby, Python, etc., Selenium Developers have developed language bindings to allow Selenium to support multiple languages. Check out Selenium Libraries in the [official site](http://www.seleniumhq.org/download/#client-drivers).

**JSON WIRE PROTOCOL Over HTTP Client:**

JSON stands for JavaScript Object Notation. It is used to transfer data between a server and a client on the web. JSON Wire Protocol is a REST API that transfers the information between HTTP server. Each BrowserDriver (such as FirefoxDriver, ChromeDriver etc.,)  has its own HTTP server.

**Browser Drivers:**

Each browser contains separate browser driver. Browser drivers communicate with respective browser without revealing the internal logic of browser’s functionality. When a browser driver is  received any command then that command will be executed on the respective browser and the response will go back in the form of HTTP response..

**Browsers:**

Selenium supports multipe browsers such as Firefox, Chrome, IE, Safari etc.,

**Selenium Grid**

Selenium grid allows to run the tests on multiple machine at the same time. It allows to run the test on different environment and different browser

Pre-requisite

* Download selenium-server-standalone.jar—hub and all your nodes.

Suggested : place this jar file at the same location (dir structure must be same on hub and all the nodes)

* Turnoff your firewalls

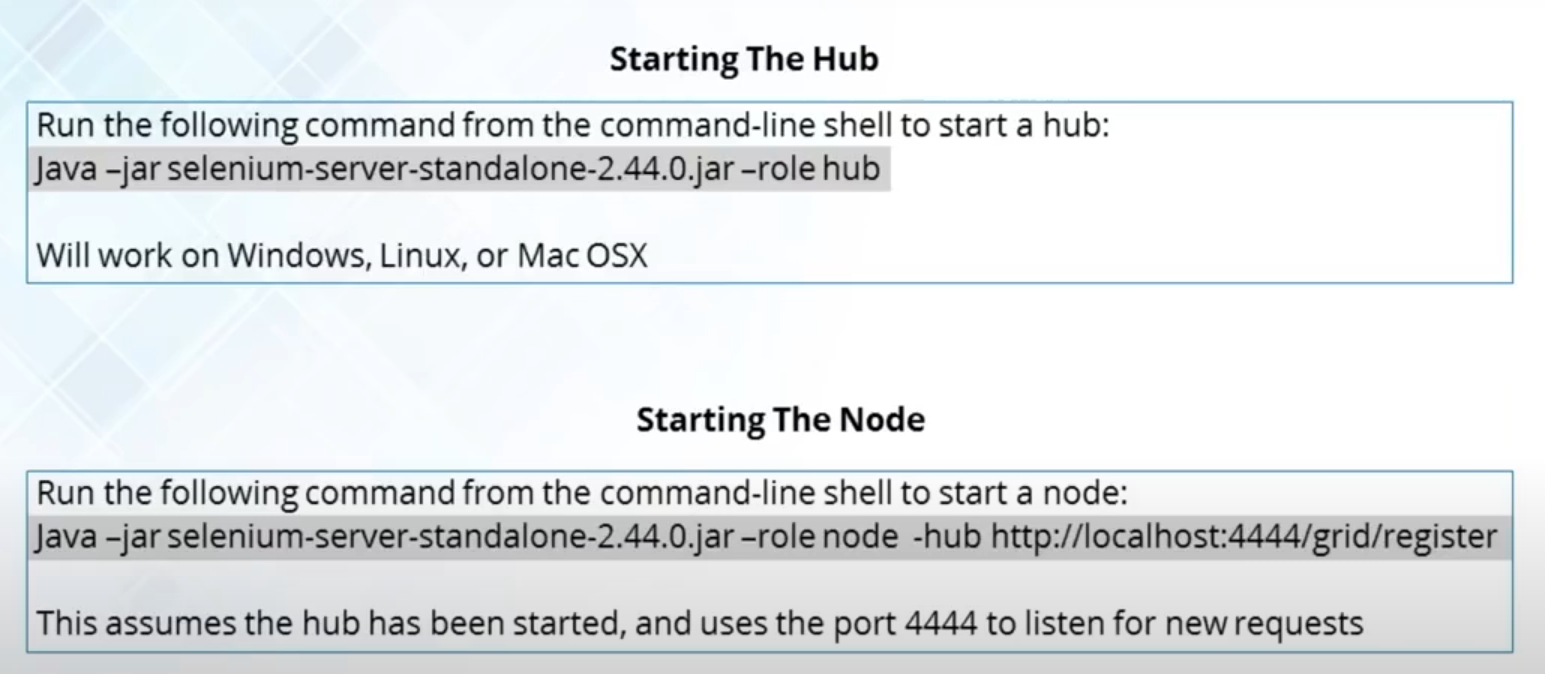
Starting an hub and node

Start Server

java -jar selenium-server-standalone-2.44 -role hub

Start Node

java -jar -d webdriver.chrome.driver= C:\Drivers\chromedriver\_win32\chromedriver.exe selenium-server-standalone-2.44 -role node -hub http://localhost:4444/grid/register



J