Siddhi Institute

**SELENIUM**

*by*

Srikanth Pittala Sr. Automation Engineer

**Selenium Introduction:**

**What is selenium?**

* Selenium is an open-source tool/framework/set of API’S/project for a range of libraries used to automate web application on different browser & platform.
* Selenium provides remote control of browser instances which emulate user’s interaction with the browser.
* Selenium is a well-known antidote for mercury poison.

Limitation of Selenium :

* It supports only web-application automation.
* It doesn’t support windows-based application directly. However, third party tool **(eg: AutoIt)** can be integrated with selenium to automate windows-based applications.

Note :

1. Selenium community developed specific tool called **WINIUM** to automate windows-based applications.
2. Selenium community also developed tools to test mobile applications,
   * **Selendroid** - it supports only Android platform
   * **Appium** - it supports Android platform, MAC, Windows etc.

Note :

1. All the selenium related resources and documents can be found on the below website.

<http://www.seleniumhq.org>

Here, hq stands for head quarter

**Why Selenium is so popular and demanding ?**

Selenium is popular and demanding due to the following features.

* 1. it is an open-source tool freely available on internet
  2. No project cost involved
  3. No licence required
  4. It supports many IDE (Integrated Development Environment) like eclipse, Visual studio, PyCharm etc. to write automation scripts using different languages.
  5. Can be easily customized to integrate with other Test Management tools like ALM, Bugzilla etc.
  6. It supports many languages to write automation script, almost 13 diﬀerent software languages
     + Java
     + C#
     + Ruby
     + Python
     + Perl
     + Php
     + Javascript
     + Javascript (Node JS)
     + Haskell
     + R
     + Dart
     + TCL
     + Objective - C
  7. It supports almost all the browsers.(Firefox, Chrome, Internet Explorer etc) and hence, cross browser testing/compatibility testing can be performed using selenium.
  8. It supports almost all the Operating System (MAC, Windows, LINUX etc) and hence, cross platform testing can also be performed.

**What are the different tools available in Selenium suite?**

It contains following tool’s:

1. Selenium IDE (Integrated Development Environment)
2. Selenium RC (Remote Control - Version is 1.x)
3. Selenium WD (WebDriver - Version is 2.x , 3.x & 4.x)
4. Selenium Grid

Diagram

Description automatically generated

Selenium IDE:

* It is a Firefox extension that can automate web applications by Record & Playback feature.
* In 2006, Selenium IDE is merged with selenium project.

|  |  |
| --- | --- |
| Advantages | Dis-Advantages |
| Very easy to install | Available only for Firefox |
| No programming knowledge is required | No support for customized test operations |
| Built in test reports | Slow in execution compared with RC & WD |
| Open source | We can run the recorded scripts only in Firefox browser |

**Selenium RC :**

* It is a server written in java.
* It is the first automated web-testing tool that support all programming languages.
* Used for parallel execution of automation scripts on multiple remote systems
* Selenium RC merged with web driver in 2008 and formed selenium-2. After new updates & improvements introduced selenium-3.

|  |  |
| --- | --- |
| Advantages | Dis-Advantages |
| Faster than selenium IDE | Slower than selenium WD |
| Support for customized test operations | Need selenium RC server to execute test cases |
| Support parallel testing | Complicated in installation |
| Open source | Must have any programming skills |
| Support Data Driven framework | Internally RC used JavaScript to automate web-applications |
| Selenium RC has in-built HTML report generation features for test results | No support for Headless HTMLUnit browsers (Invisible browser) |

Diagram

Description automatically generated

**Selenium WD :**

* Founded by Simon Stewart in 2006.
* Selenium WebDriver was the first cross-platform testing framework that could control the browser from OS level.

Note :

* Selenium WebDriver version 3.x is no longer capable of running Selenium RC directly, rather it does through emulation and via an interface called **WebDriverBackedSelenium**.
* But **it does support Selenium Grid directly**.

|  |  |
| --- | --- |
| Advantages | Dis-Advantages |
| Faster than selenium RC & IDE | Installation is Complicated then selenium IDE |
| Support for customized test operations | Must have any programming skills |
| support parallel testing | It does not rely on JavaScript for automate web-applications |
| Open source | Does not provide test results directly without third party tools. |
| Interact directly with browser through Driver |  |
| Not required any server to execute testcases |  |
| Customized test results |  |
| Support for Headless HTMLUnit browsers (Invisible browser) |  |

**Selenium WebDriver Architecture:**

In order to understand Selenium WebDriver Architecture, we should first know what is a WebDriver API. Selenium Webdriver API helps in communication between languages and browsers. Each and every browser has different logic of performing actions on the browser. Below image discribe various components of Selenium WebDriver Architecture.

****

**It consists of four main components which are:**

1. Selenium Client Library
2. JSON WIRE PROTOCOL Over HTTP Client
3. Browser Drivers
4. Browsers

**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.

**JSON WIRE PROTOCOL Over HTTP Client**

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

Ref : <https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol>

**Browser Drivers**

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

**Browsers**

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

**Selenium Grid :**

* It is one of the components of selenium that is used to run automation scripts on multiple system simultaneously.
* It is used to carry out compatibility testing on multiple browsers and platforms.
* It has HUB and NODES features.
* Hub is a central source. it provides selenium commands to Nodes.
* Nodes perform actions that are initiated by Hub.

Diagram

Description automatically generated

**What are the key/Important topics of Selenium ?**

* Automation Framework - guidelines and rules to write selenium code
* GitHub - Central Repository to store code
* Maven - build dependency tool for auto update of selenium version
* Selenium Grid - to test on multiple OS and browsers
* Jenkins - Continuous Integration
* TestNG - framework for generation of Test Reports and running multiple test scripts in one go

**What are the Software’s required for Selenium ?**

1. Eclipse IDE
2. JDK 1.8
3. Seleniuim jar files
4. Selenium Server-Standalone jar

Download it from the given url : <https://www.selenium.dev/downloads/>

**Driver Executables**

* **For Firefox Browsere**
  + the name of the driver executable is : **geckodriver.exe**
  + Url to download : <https://github.com/mozilla/geckodriver/releases>
  + Version v**0.30** is recommended for firefox browser with version 94.0

**Note :** To stop auto update of firefox browser version, make sure to disconnect the internet connection and then install 54.0 version, now go to Setting/Option in firefox browser and check the below checkbox - Never check for updates

Graphical user interface, text, application, email

Description automatically generated

* **For Chrome Browser**
* The name of the driver executable is : **chromedriver.exe**
* Url to download : <https://chromedriver.chromium.org/downloads>
* If you are using Chrome version 97, please download ChromeDriver 97.0.4692.36
* If you are using Chrome version 96, please download ChromeDriver 96.0.4664.45
* If you are using Chrome version 95, please download ChromeDriver 95.0.4638.69
* **For Internet Explorer Browsere**
  + the name of the driver executable is : **IEDriverServer.exe**
  + Url to download : <https://www.selenium.dev/downloads/>
* **For Edge Browsere**
  + the name of the driver executable is : **msedgedriver.exe**
  + Url to download : [https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/](https://www.selenium.dev/downloads/)
* **For Opera Browsere**
  + the name of the driver executable is : **msedgedriver.exe**
  + Url to download : [https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/](https://www.selenium.dev/downloads/)

**Selenium Java Architecture - Detailed Level:**Diagram

Description automatically generated



1. SearchContext is the super most interface present in selenium webdriver.
2. An interface called WebDriver extends SearchContext interface.
3. A total of 13 interfaces are available in selenium, which is implemented by a super most class called RemoteWebDriver
4. RemoteWebDriver is again extended by few browsers’ specific child classes such as,
   1. FirefoxDriver class to automate on firefox browser.
   2. ChromeDriver class to automate on Chrome browser,
   3. InternetExplorerDriver class to automate on IE and so on…….

**Note:**

All the above-mentioned **interfaces and classes** are present in a package called **“org.openqa.selenium”.**

**WebDriver interface:**it is an interface provided by selenium. It consists constant and abstract methods. These abstract methods are implemented in RemoteWebDriver class and all respective browser driver class like, ChromeDriver , FirefoxDriver, InternetExplorerDriver, AndroidDriver etc. are extended this RemoteWebDriver class.

Example :

WebDriver driver = new ChromeDriver();

WebDriver driver = new FirefoxDriver();

WebDriver driver = new InternetExplorerDriver();

WebDriver driver = new OperaDriver();

**Launching Browsers :**

**Chrome Browser :**

**Note :** check the chrome browser version first and download chrome driver which is compatibility with that version.

**Syntax:**

System.setProperty("webdriver.chrome.driver","chromedriver path");

WebDriver chrome = new ChromeDriver();

* We need to specify driver and its path in set property method so that system can identify driver at the given path.
* Second line will launch the Chrome browser.

**Firefox Browser :**

**Note1** : check the firefox browser version first and download gecko driver which is compatibility with that version.

**Note2:** It's compatible with most browsers

**Syntax:**

System.setProperty("webdriver.gecko.driver","geckodriver path");

WebDriver firefox = new FirefoxDriver();

**IE Browser :**

**Note1 :**

* First, we must check the Protection Mode of our IE browser.
* **Launch Internet Explorer → move to Tools Menu → Internet Options → Security tab.**
* **Check** the option Select **Enable Protected Mode**. Also, we must choose the Internet as the zone. We can make the level as Medium-high. The other Protection zones like **Local Intranet** and **Trusted sites** (apart from **Restricted sites**) should also have the **same configuration**. Once all the configurations are done, we have to Apply and then click OK.
* Zoom level should be 100%.
* check the ie browser version first and download ie driver which is compatibility with that version.

**Note2:** Among all browsers, Internet Explorer is slower than the other drivers.

**Syntax:**

System.setProperty("webdriver.ie.driver","iedriver path");

WebDriver ie = new InternetExplorerDriver();

**Edge Browser :**

**Note1** : check the edge browser version first and download edge driver which is compatibility with that version.

**Syntax:**

System.setProperty("webdriver.edge.driver","edgedriver path");

WebDriver firefox = new EdgeDriver();

**Opera Browser :**

**Note1** : check the opera browser version first and download opera driver which is compatibility with that version.

**Syntax:**

System.setProperty("webdriver.opera.driver","operadriver path");

WebDriver firefox = new OperaDriver();

**Headless Browser (Non-UI mode) in selenium:**

**Headless browser:**

* A headless browser is same as a real browser except it do not have a GUI.
* All modern browsers provide the capabilities to run them in a headless mode
* when Selenium tests run in headless browsers, they execute in the background.
* Selenium supports headless browser testing using HtmlUnitDriver.
* HtmlUnitDriver is based on java framework HTMLUnit and is the one of the lightweight and fastest among all headless browsers.
* In older versions of Selenium, we used the HTMLUnitDriver mainly, a headless driver providing a non-GUI implementation of Selenium WebDriver.
* But with the latest versions of Selenium WebDriver 3 and Selenium WebDriver 4, Selenium also supports headless versions of real browsers like Chrome, Firefox, and Edge.

**Running Selenium test cases using HTMLUnitDriver :**

* Fastest execution of WebDriver compared to other browsers.
* HtmlUnitDriver is platform independent.
* HtmlUnitDriver supports JavaScript.
* Also, HtmlUnitDriver allows you to choose other browser versions to run your scripts. You can mention different browser versions of Chrome or Firefox in the HtmlUnitDriver itself.
* One of the significant benefits of using headless browsers is performance. Since headless browsers don't have a GUI, they are faster than real browsers.

**Note :** Required **Selenium standalone server 3.5.3** jar file.

**Syntax:**

HtmlUnitDriver driver = new HtmlUnitDriver();

**Running tests on different browser versions using HtmlUnitDriver :**

* **Chrome syntax:**

HtmlUnitDriver chrome = new HtmlUnitDriver(BrowserVersion.***CHROME***);

* **Firefox syntax:**

HtmlUnitDriver firefox = **new** HtmlUnitDriver(BrowserVersion.***FIREFOX***);

* **IE syntax:**

HtmlUnitDriver ie = **new** HtmlUnitDriver(BrowserVersion.***INTERNET\_EXPLORER***);

* **Edge syntax:**

HtmlUnitDriver edge = **new**  HtmlUnitDriver(BrowserVersion.***EDGE***);

**PhantomJS PhantomJS** :

PhantomJS PhantomJS is a headless browser with JavaScript API. It has integrated with GhostDriver and there is no need to separately install it.

**Note** : By PhantomJs provides log file

**Syntax:**

System.setProperty("phantomjs.binary.path"," phantomjs path");

WebDriver firefox = new FirefoxDriver();

**Headless Real Browsers in selenium:**

**Running Selenium headless browser tests using the headless Real Chrome browser.**

**NOTE** : Google Chrome is available with headless execution since version 59.

* Selenium WebDriver provides a class called "**ChromeOptions**"
* ChromeOptions can specify certain configurations to change the default behavior of Chrome.
* One of those configurations is the "**headless**" mode, which launches the Chrome in headless mode.

**Syntax:**

ChromeOptions options = **new** ChromeOptions();

options.addArgument("--headless"); **or** options.setHeadless(**true**);

WebDriver driver = **new** ChromeDriver(options);

**Running Selenium headless browser tests using the headless Real Firefox browser.**

* Selenium WebDriver provides a class called " **FirefoxOptions**"
* FirefoxOptions can specify certain configurations to change the default behavior of Firefox.

**Syntax**:

FirefoxOptions options = new FirefoxOptions();

options.addArgument("--headless"); **or** options.setHeadless(**true**);

WebDriver driver = **new** FirefoxDriver(options);

**Note :** IE does not have support for a headless mode

**Why we upcast the browser related child class to WebDriver, and not RemoteWebDriver class (RemoteWebDriver being the super most class in selenium) ?**

**Upcasting Example :**

WebDriver driver = new FirefoxDriver();

* Converting a child class object to super type is called Upcasting.
* In selenium, we use upcasting so that we can execute the same script on any browser.
* In selenium, we can upcast browser object to RemoteWebDriver, WebDriver, TakesScreenshot , JavascriptExecutor etc, but a standard practice is to upcast to WebDriver interface.
* This is as per the Selenium coding standard set by the Selenium community.
* WebDriver is the name of the key interface against which tests should be written in Java, the implementing classes one should use are listed as below:

ChromeDriver, EventFiring WebDriver, FirefoxDriver, HtmlUnitDriver, Internet ExplorerDriver, Phantom SDriver, RemoteWebDriver, SafariDriver

**Where did you use Upcasting in Selenium ?**

**WebDriver driver = new FirefoxDriver();**

**Explain the above statement**:

1. WebDriver is an interface in Selenium that extends the super most interface called SearchContext.
2. driver is the upcasted object or WebDriver interface reference variable.
3. “ = ” is an assignment operator.
4. new is a keyword using which object of the FirefoxDriver class is created.
5. FirefoxDriver() is the constructor of FirefoxDriver class which initializes the object, and it will also launch the firefox browser.

**WebDriverManager in selenium:**

WebDriverManager allows to automate the path declaration required in Selenium WebDriver.

Using WebDriverManager its not required to write system setproperty in script always.

**Note :** Required WebDriverManager 3.2.0 jar file.

**Syntax:**

* **Chrome syntax:**

WebDriverManager.chromedriver().setup();

WebDriver chrome = new ChromeDriver();

* **Firefox syntax:**

WebDriverManager.firefoxdriver().setup();

WebDriver ff = new FirefoxDriver();

* **IE syntax:**

WebDriverManager.iedriver().setup();

WebDriver ie = new InternetExplorerDriver();

* **Edge syntax:**

WebDriverManager.edgedriver().setup();

WebDriver edge = new EdgeDriver();

* **Opera syntax:**

WebDriverManager.operadriver().setup();

WebDriver edge = new OperaDriver();

**Selenium RemoteWebDriver Methods :**

1. **get**()

This method used to navigate the url passed as parameter. It waits until page load completely.

**Syntax:**

WebDriver driver = new ChromeDriver();

driver.get(“http://google.co.in”);

1. **getTitle**()

This method is used to get the title of the current tab or webpage.

**Syntax:**

WebDriver driver = new ChromeDriver();

String title = driver.getTitle();

System.out.println(title);

1. **getCurrentUrl**()

This method is used to get the URL of the current tab or webpage.

**Syntax:**

WebDriver driver = new ChromeDriver();

String url = driver.getCurrentUrl();

System.out.println(url);

**How do you ensure the required page is displayed or not** ?

We can use following checkpoints to validate the required page is displayed or not.

1. using title of the page
2. using URL of the page.
3. using any unique element on the page.
4. **getPageSource**()

This method is used to get the page source code of the current tab or webpage.

**Syntax:**

WebDriver driver = new ChromeDriver();

String pagesource = driver.getPageSource();

System.out.println(pagesource);

1. **close()**:

This method is used to close the current tab or current window.

**Syntax:**

WebDriver driver = new ChromeDriver();

driver.close();

1. **quit()**:

This method is used to close the current tab or current window.

**Syntax:**

WebDriver driver = new ChromeDriver();

driver.quit();

1. **getSessionID**():

This method used to get session-id of webpage.

Syntax:

WebDriver driver = new ChromeDriver();

String sessionid = ((RemoteWebDriver)driver).getSessionID();

System.out.println(sessionid);

1. **getCapabilities**():

The capabilities property is a dictionary containing information about the configuration**.**

1. **getBrowserName():**

This method used to get name of the browser.

Syntax:

ChromeDriver driver = new ChromeDriver();

String browsername = driver.getCapabilities().getBrowserName();

System.out.println(browsername);

**Or**

WebDriver driver = new ChromeDriver();

String browsername = ((RemoteWebDriver)driver).getCapabilities().getBrowserName();

System.out.println(browsername);

1. **getPlatform**()

This method used to get the name of the platform.

**Syntax:**

WebDriver driver = new ChromeDriver();

String platformname = ((RemoteWebDriver)driver).getCapabilities().getPlatform();

System.out.println(platformname);

1. **getVersion**()

This method used to get the version of the browser.

**Syntax:**

WebDriver driver = new ChromeDriver();

String version = ((RemoteWebDriver)driver).getCapabilities().getVersion);

System.out.println(version);

1. **getCapability**()

This method used to get the capability value based on capability name passed parameter.

**Syntax:**

WebDriver driver = new ChromeDriver();

String version = ((RemoteWebDriver)driver).getCapabilities().getCapability(“browserVersion”);

System.out.println(version);

1. **getCapabilityNames**()

This method used to get all capabilities names.

**Syntax:**

WebDriver driver = new ChromeDriver();

Set<String> caps = ((RemoteWebDriver)driver).getCapabilities().getCapabilityNames();

for(String cap: caps)

{

System.out.println(cap);

}

**Selenium RemoteNavigation class methods:**

1. **back()**

This method used to go back page from the current page.

**Syntax**:

WebDriver driver = new ChromeDriver();

driver.navigate().back();

1. **forward()**

This method used to go forward page from the current page.

**Syntax**:

WebDriver driver = new ChromeDriver();

driver.navigate().forward();

1. **refresh()**

This method used to refresh the current page.

**Syntax**:

WebDriver driver = new ChromeDriver();

Driver.navigate().refresh();

1. **to()**

This method used to navigate the url or string passed as parameter.

**Syntax**:

WebDriver driver = new ChromeDriver();

driver.navigate().to(“https://www.google.co.in/”);

OR

URL google = new URL(“https://www.google.co.in/”)

WebDriver driver = new ChromeDriver();

driver.navigate().to(google);

**Selenium RemoteWindow class methods:**

1. **maximize()**

This method used to maximize the browser window.

Syntax:

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

1. **fullScreen()**

This method used to set the screen as full screen.

**Syntax**:

driver.manage().window().fullScreen();

1. **newWindow(WindowType)**

This method used to open new tab or new window of the browser.

Note : Selenium-4 new Feature

For new tab:

Syntax:

driver.switchTo().newWindow(WindowType.TAB);

For new window:

Syntax:

driver.switchTo().newWindow(WindowType.WINDOW);

1. **getSize()**

This method used to get the size of the browser window.

Return type : Dimension

**Syntax**:

Dimension dimension = driver.manage().window().getSize();

System.out.println(dimension.getHeight());

System.out.println(dimension.getWidth());

1. **setSize()**

This method used to set the size of the browser window.

Parameter type : Dimension

**Syntax**:

Dimension dimension = new Dimension(500,600);

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

1. **getPosition()**

This method used to get the co-ordinates of browser window.

Return type : Point

**Syntax**:

Point point = driver.manage().window().getPosition();

System.out.println(point.getX());

System.out.println(point.getY());

1. **setPosition()**

This method used to set the co-ordinates of browser window.

Parameter type : Point

**Syntax**:

Point point = new Point (500,500);

driver.manage().window().setPosition(point);

**Selenium RemoteWebDriverOptions class methods:**

**Selenium cookie commands:**

**Cookie :** Cookie Is a temporary file which can store user information.

1. **getCookies()**

This method used to get all cookies present in a webpage.

Return type : Set <Cookie>

**Syntax**:

Set<Cookie> cookies = driver.manage().getCookies();

for(Cookie cookie: cookies)

{

System.out.println(cookie.getName());

System.out.println(cookie.getValue());

System.out.println(cookie.getPath());

}

1. **getCookieNamed()**

This method used to get a cookie by its name passed as a parameter.

Parameter Type : String

**Syntax**:

Cookie cookie = driver.manage().getCookieNamed(“cookiename”);

System.out.println(cookie);

1. **addCookie()**

This method used to add a cookie on webpage.

Parameter Type : Cookie

**Syntax**:

driver.manage().addCookie(“key”,”value”);

or

Cookie k = new Cookie(“key” ,”value”);

driver.manage().addCookie(k);

1. **deleteAllCookies()**

This method used to delete all cookies present on a webpage.

**Syntax**:

driver.manage().deleteAllCookies();

1. **deleteCookie()**

This method used to delete a cookie present on a webpage.

Parameter Type : Cookie

**Syntax**:

Cookie k = new Cookie(“key” ,”value”);

driver.manage().deleteCookie(k);

1. **deleteCookieNamed()**

This method used to delete a cookie present on a webpage.

Parameter Type : String

**Syntax**:

driver.manage().deleteCookieNamed(“cookieName”);

**HTML Web page components & tags.**

**Browser Inspectors.**

**Methods of SearchContext interface:**

1. **findElement():**

This method used to locate a WebElement on a webpage.

Parameter Type : By

Return type : WebElement

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement loginBtn = driver.findElement(By.id(“loginbtn”));

1. **findElements():**

This method used to locate multiple WebElement’s on a webpage.

Parameter Type : By

Return type : List<WebElement>

**Syntax**:

WebDriver driver = new ChromeDriver();

List<WebElement> btns = driver.findElements(By.id(“loginbtn”));

**Types of locators in selenium or By class attributes.**

In selenium we have 8 types of locators.

**Note**: locator value should not have any space

1. Id

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement loginBtn = driver.findElement(By.id(“xxxxxx”));

1. name

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement loginBtn = driver.findElement(By.name(“xxxxxxx”));

1. className

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement loginBtn = driver.findElement(By.className(“xxxxxxx”));

1. tagname

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement loginBtn = driver.findElement(By.tagName(“xxxxx”));

1. linkText

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement loginBtn = driver.findElement(By.linkText(“xxxxxx”));

1. partialLinkText

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement loginBtn = driver.findElement(By.partialLinkText(“xxxxxx”));

1. xpath

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement loginBtn = driver.findElement(By.xpath(“xxxxx”));

**Important notes on LinkText and PartialLinkText locator:**

* Out of all the locators, linkText and PartialLinkText are used to identify only the links present on the webpage.(elements whose tagname is “a” -- a stands for anchor)
* LinkText locator should be used when the text of the link is constant
* PartialLinkText locator should be used when certain part of the link text is getting changed everytime the page is loaded. i.e for partially dynamically changing text, we use partialLinkText locator
* To handle those elements whose text changes completely, we can’t use partialLInkText locator. It should be handled by another locator called “**xpath**”
* If we use try to use these 2 locators on other type of elements (except Links), then we get “**NoSuchElementException**”

**Steps to write and evaluate Xpath/cssSelector expression in Chrome browser :**

1. In order to write xpath/cssSelector expression in chrome browser, we don’t need any add-ons as such.
2. Navigate to the web page -- > right click anywhere on the web page → Press F12 from keyboard or select inspect element, it will open the Developer tool section with Elements tab selected by default.
3. Press Ctrl+F and write the xpath/cssSelector expression, it will highlight the source code of the matching element.
4. Place the cursor on the highlighted source code, it will highlight the corresponding element present on the web page.

**Xpath in selenium:**

if we are unable to find any locator i.e., id , name, class, text etc. in any tag, then use xpath to locate that tag.

Xpath is categorized into two types

1. **Absolute xpath :**

* It starts from root tag/node of the html source code and ends at required tag.it is direct way of finding the WebElement.
* While writing the absolute xpath, we use single forward slash (/) to traverse through each immediate child element in the html tree.
* In the below sample html tree,

document

| html

|

---- body

|

------> a

* Absolute xpath can be written in the following ways.

html/body/a or ./html/body/a

(Note :- here, dot (.) refers to the current document or the current web page, using dot here is optional)

* The major dis-advantage of absolute xpath is, if there is any change in elements of the webpage then absolute xpath will not work.
* We can get absolute xpath directly by using any tool like Selenium IDE, firebug, chrome inspector etc.

**Relative xpath:**

it starts with double forward slash ( // ) and at any tag inside the html source.

Syntax :

//tagname[@attribute= ‘attribute value’]

Ex: //button[@id="hobbies"]

The major advantage of relative xpath is, if there is any change in WebElement of a webpage relative xpath will not affected.

**Different characters in xpath syntax:**

1. Single forward slash ( / ) :

it selects all child tags inside a parent tag.

**Syntax** : //tagname[@attribute=’attributeValue’]/tag

**Example** : //form[@id="userForm"]**/**div

1. Double forward slash( // ) :

it selects all selective tags inside a parent tag.

**Syntax** : //tagname[@attribute=’attributeValue’]//tag

**Example** **: //**form[@id="userForm"]//div

1. Address sigh ( @ ) :

it is used to select a tag based on attribute of a selective tag.

**Syntax** : //tagname[@attribute=’attributeValue’]/tag

**Example** : //form[**@**id="userForm"]

1. Dot ( . ) :

it selects current tag.

**Syntax** : //tagname[@attribute=’attributeValue’]/.

**Example** : //form[@id="userForm"]/**.**

1. Double dot (..) :

it selects parent tag of current tag.

**Syntax** : //tagname[@attribute=’attributeValue’]/..

**Example** : //form[@id="userForm"]/**..**

1. Asterisk ( \* ) :

it selects any tag of html source code.

**Syntax** : //\*[@attribute=’attributeValue’]

**Example** : //**\***[@id="userForm"]

1. Address and Asterisk ( @\* ) :

it selects any attribute of a given tag.

**Syntax** : //tagname[@\*]

**Example** : //form[**@\***]

1. or ( | ) :

it used to select any matched path. Its

**Syntax** : //tagname[@attribute=’attributeValue’] | //tagname[@attribute=’attributeValue’]

**Example** : //form[@id="userForm"] **|** //\*[@id="userNumber"]

**Note: xpath and CSS selector use 1-based index.**

**Predicates in xpath** :

Predicates are used to find specific element by its index.

1. index ( [ ] ) :

it selects WebElement based on index from grope of matched WebElement.

**Syntax** : //tagname[@attribute=’attributeValue’]/tag**[index]**

**Example** : //form[@id="userForm"]/div[2]

1. last() :

it selects last WebElement from grope of matched WebElement.

**Syntax** : //tagname[@attribute=’attributeValue’]/tag**[last()]**

**Example** : //form[@id="userForm"]/div[last()]

1. position() :

it selects specific positioned WebElement from grope of matched WebElement.

**Syntax** : //tagname[@attribute=’attributeValue’]/tag**[position()=x]**

**Example** : //form[@id="userForm"]/div[position()=1]

**Sample Html tree :**

****

**xpaths using Predicates to identify the elements in the above sample tree:**

**Table

Description automatically generated**

1. In Group index, we write xpath expression within the braces and then we write the index outside the braces.
2. Internally, it executes the xpath expression first and stores the result in an xpath array whose index starts with 1
3. last() is a function that is used to retrieve the last element present in the xpath array.

**Example2:**

**<table border="1">**

**<tr>**

**<td> CV</td>**

**<td> <input type="checkbox"/></td>**

**</tr>**

**<tr>**

**<td> cv</td>**

**<td> <input type="checkbox"/></td>**

**</tr>**

**<tr>**

**<td> cv</td>**

**<td><input type="checkbox"/></td>**

**</tr>**

**<tr>**

**<td> CV</td> <td> <input type="checkbox"/></td>**

**</tr>**

**<tr>**

**<td> cv</td>**

**<td> <input type="checkbox"/></td>**

**</tr>**

**</table>**

**Text

Description automatically generated**

**Functions in xpath** :

1. Contains():

It is used to select WebElement based on partial value of an attribute or text.

**Syntax** : //tagname[contains(@attribute,’partialAttributeValue’)]

( or )

**Syntax** : //tagname[contains(text() , ’partialtext’)]

**Example** : //input[contains(@id,'first')]

**Example** : //label[contains(text(),'Sport')]

1. Starts-with():

It is used to select WebElement based on starting value of an attribute or text.

**Syntax** : //tagname[starts-with(@attribute,’partialAttributeValue’)]

( or )

**Syntax** : //tagname[starts-with(text() , ’partialtext’)]

**Example** : //input[starts-with(@id,'first')]

**Example** : //label[starts-with(text(),'Spo')]

1. text():

It is used to select WebElement based on starting value of an attribute or text.

**Syntax** : //tagname[text() = ’text’]

**Or**

**Syntax** : //tagname[. = ’text’]

**Example** : //label[text()='Sports']

**Example** : //label[.='Sports']

**Operators in xpath** :

1. and :

It is used to combined two different conditions to select WebElement.

**Syntax** : //tagname[@attribute=’AttributeValue’ and @attribute=’AttributeValue’]

( or )

**Syntax** : //tagname[@attribute=’AttributeValue’ and text() =’text’]

**Example** : //input[@placeholder='First Name' and @id='firstName']

**Example** : //input[@placeholder='First Name' and text()='firstName']

1. or :

It is used any one condition to select WebElement.

**Syntax** : //tagname[@attribute=’AttributeValue’ or @attribute=’AttributeValue’]

( or )

**Syntax** : //tagname[@attribute=’AttributeValue’ or text() =’text’]

**Example** : //input[@placeholder='First Name' or @id='firstName']

**Example** : //input[@placeholder='First Name' or text()='firstName']

**Xpath Axes OR Special Attributes in xpath:**

* In xpath, navigating from one element to another element is called traversing.
* In order to traverse from one element to another, we use xpath axes.
* We have the following 8 xpath axes in selenium.
* child
* descendant
* parent
* ancestor
* following
* preceding
* following-sibling
* preceding-sibling

1. child:

It selects all child tags of the current node.

**Syntax** : //tagname[@attribute=’AttributeValue’]/child::tag

( or )

**Syntax** : //tagname[@attribute=’AttributeValue’]/tag

**Example** : //\*[@id='userEmail-wrapper']/child::input

**Example** : //\*[@id='userEmail-wrapper']/input

1. descendant:

it selects all descendant (children, grandchildren, etc.) of the current node.

**Syntax** : //tagname[@attribute=’AttributeValue’]/descendant::tag

( or )

**Syntax** : //tagname[@attribute=’AttributeValue’]//tag

**Example** : //\*[@id='userEmail-wrapper']/ descendant::input

**Example** : //\*[@id='userEmail-wrapper']//input

1. parent:

It selects parent tag or selective tag of the current tag.

**Syntax** : //tagname[@attribute=’AttributeValue’]/parent::tag

( or )

**Syntax** : //tagname[@attribute=’AttributeValue’]/..

**Example** : //\*[@id='userEmail-wrapper']/parent::input

**Example** : //\*[@id='userEmail-wrapper']/..

1. Ancestor:

It selects all ancestors (parent, grandparent, etc.) of the current node.

**Syntax** : //tagname[@attribute=’AttributeValue’]/ancestor::tag

**Example** : //\*[.='Label']/ancestor::table

1. following:

Selects every tag in the source after the current node.

**Syntax** : //tagname[@attribute=’AttributeValue’]/following::tag

**Example** : //\*[.='Label']/following::table

1. preceding:

it selects every tag in the source before the current node.

**Syntax** : //tagname[@attribute=’AttributeValue’]/preceding::tag

**Example** : //\*[.='Label']/preceding::table

1. following-sibling:

It selects all selective below sibling tags from the current tag.

**Syntax** : //tagname[@attribute=’AttributeValue’]/following-sibling::tag

**Example** : //\*[.='Label']/following-sibling::table

1. preceding-sibling:

It selects all selective above sibling tags from the current tag.

**Syntax** : //tagname[@attribute=’AttributeValue’]/preceding-sibling::tag

**Example** : //\*[.='Label']/preceding-sibling::table

1. CSS Selector

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement loginbtn = driver.findElement(By.cssSelector(“xxxxx”));

**CSS selector in selenium** : Cascading style sheets

CSS has better performance and speed than xpath because

* Xpath allows bidirectional flow which means the traversal can be both ways from parent to child and child to parent as well
* CSS allows only one directional flow which means the traversal is from parent to child only
* Xpath engines are different in each browser, hence make them inconsistent.
* IE does not have a native xpath engine, therefore selenium injects its own xpath engine for compatibility of its API.
* Every webpage has its CSS at the top in the <head> section and JavaScript at the bottom. This allows the browser to load your CSS first.

**Note1: xpath and CSS selector use 1-based index.**

**Note2: In CssSelector, we traverse through the element using this symbol “ > ”**

1. **Id**

**Syntax** : tag[id=’idvalue’]

**Example** : input[id=’firstname’] or input**#**firstname

**Note** : **#** symbol indicates ‘ID’ in CSS selector.

1. **Class**

**Syntax** : tag[class=’classvalue’]

**Example** : input[class=’firstname’] or input**.**firstname

**Note** : **.** symbol indicates ‘ID’ in CSS selector.

1. **Any attribute**

**Syntax** : tag[attribute=’attributevalue’]

**Example** : button[placeholder=’acceptcookies’]

**Multiple attribute:**

* **Id & attribute:**

**Syntax** : tag[id=’idvalue’][attribute=’attributevalue’] or tag#idvalue[attribute=’attributevalue’]

**Example** : input[id='firstName'][placeholder='First Name']

or

input#firstName[placeholder='First Name']

* **class & attribute:**

**Syntax** : tag[class=’classvalue’][attribute=’attributevalue’] or tag.classvalue[attribute=’attributevalue’]

**Example** : input[class=’firstname’][placeholder='First Name']

or

input.firstName+[placeholder='First Name']

**Using parent – child hierarchy**

**Syntax** : parenttag>childtag>childtag>…>…>…>

**Example** : form[id='userForm']>div>div>input[id='firstName']

or

form#userForm>div>div>input[id='firstName']

**Note** : **>** symbol indicates child tags in CSS selector.

**Index or nth-of-value()**

**Syntax** : parenttag>childtag : nth-of-type(index)

**Example** : form[id='userForm']>div:nth-of-type(1)>div:nth-of-type(2)>input[id='firstName']

or

form#userForm>div:nth-of-type(1)>div:nth-of-type(2)>input[id='firstName']

**Note** : nth-of-type(index) symbol indicate tags based on index in CSS selector.

**With text**

* Starting text

**Syntax** : tag[attribute^=’attributeStartingValue’]

**Example** : input[id^='firstNa']

**Note** : **^** symbol indicates Start-with text in CSS selector.

* Contains-text

**Syntax** : tag[attribute$=’attributeSomeTextValue’]

**Example** : input[id$='rstNa']

**Note** : **$** symbol indicates Contains-with text in CSS selector.

* Ending text

**Syntax** : tag[attribute\*=’attributeEndTextValue’]

**Example** : input[id\*='tName']

* **Note** : **\*** symbol indicates Ending text text in CSS selector.

**Difference between CssSelector and Xpath:**

|  |  |
| --- | --- |
| **CssSelector** | **Xpath** |
| It is faster | It is slower |
| text() function is not supported | text() function is supported |
| backward traversing is not supported | backward traversing is supported |
| groupIndex is not supported | groupIndex is supported |

**What is Page Factory in Selenium?**

Page Factory in Selenium is an inbuilt Page Object Model framework concept for Selenium WebDriver but it is very optimized. It is used for initialization of Page objects or to instantiate the Page object itself. It is also used to initialize Page class elements without using “FindElement/s.”

Page Factory is a class provided by Selenium WebDriver to support Page Object Design patterns. In Page Factory, testers use @FindBy annotation. The initElements method is used to initialize web elements.

**@FindBy**: An annotation used in Page Factory to locate and declare web elements using different locators. Below is an example of declaring an element using **@FindBy**

@FindBy(id=’elementId’) WebElement element;

Similarly, one can use @FindBy with different location strategies to find web elements and perform actions on them. Below are locators that can be used:

* ClassName
* CSS
* Name
* Xpath
* TagName
* LinkText
* PartialLinkText

**initElements()**: initElements is a static method in Page Factory class. Using the initElements method, one can initialize all the web elements located by @FindBy annotation.



**Example:**

public class BrowserPage

{

WebDriver driver;

@FindBy(xpath = "//input[@id=’idvalue’]")

WebElement Header;

@FindBy(xpath = "//\*[@id='signupButton']")

WebElement getStarted;

public BrowserPage(WebDriver driver)

{

this.driver = driver;

//This initElements method will create all WebElements

PageFactory.initElements(driver, this);

}

public void veryHeader()

{

String getheadertext = Header.getText();

assertEquals("App & Browser Testing Made Easy", getheadertext);

}

public void clickOnGetStarted()

{

getStarted.click();

}

}

**Selenium RemoteWebElement class methods:**

Diagram

Description automatically generated

1. **click();**

This method used to perform click on WebElement.

Syntax:

WebDriver driver = new ChromeDriver();

WebElement loginbutton = driver.findElement(By.id(“loginbtn”));

loginbutton.click();

or

WebDriver driver = new ChromeDriver();

driver.findElement(By.id(“loginbtn”)).click();

1. **submit();**

This method used to perform click or enter action on a WebElement.

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement button = driver.findElement(By.id(“btn”));

button.submit();

Or

WebDriver driver = new ChromeDriver();

driver.findElement(By.id(“btn”)).submit();

Note: We can use submit method only on WebElement’s which are having form tag or under form tag and its attribute **type** value must be **submit i.e.**

Text

Description automatically generated

1. **sendKeys();**

This method used to pass input keys to a WebElement.

Parameter Type : String

Syntax:

WebDriver driver = new ChromeDriver();

WebElement username = driver.findElement(By.id(“uname”));

username.sendKeys(“siddhi”);

or

WebDriver driver = new ChromeDriver();

driver.findElement(By.id(“uname”)).sendKeys(“siddhi”);

**Write a script to demonstrate different options to click on a button or on a link (Or any element)**

Using the below methods of WebElement interface.

* 1. click()
  2. sendkeys()
  3. submit()

**Write a script to demonstrate different options to click on a button or on a link (Or any element)**

Using the below methods of WebElement interface.

1. click()
2. sendkeys()
3. submit()

using **click()** method

driver.findElement(By.xpath(xp)).click();

using **sendKeys()** method

driver.findElement(By.xpath(xp)).sendKeys(Keys.ENTER);

using **submit()** method

This method will work only and only if the element has an attribute called type= 'submit'

driver.findElement(By.xpath(xp)).submit();

1. **clear();**

This method used to clear the text from the input field of a webpage.

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement username = driver.findElement(By.id(“uname”));

username.clear();

or

WebDriver driver = new ChromeDriver();

driver.findElement(By.id(“uname”)).clear();

**How do you remove value present in username text box of Actitime application without using clear() method ?**

Using sendKeys() method of WebElement interface.

Selenium code : driver.findElement(By.id("username")).sendKeys(**Keys.CONTROL + "a" + Keys.DELETE**);

1. **getAttribute();**

This method used to get value of an attribute of a WebElement.

Parameter Type : String

Return Type : String

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement username = driver.findElement(By.id(“uname”));

String attvalue = username.getAttribute(“type”);

or

WebDriver driver = new ChromeDriver();

String attvalue = driver.findElement(By.id(“uname”)).getAttribute();

1. **getCssValue();**

This method used to get cssvalue of a WebElement.

Parameter Type : String

Return Type : String

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement username = driver.findElement(By.id(“uname”));

String color = username.getCssValue(“color”);

or

WebDriver driver = new ChromeDriver();

String color = driver.findElement(By.id(“uname”)).getCssValue(“color”);

1. **getTagName();**

This method used to get tagname of a WebElement.

Return Type : String

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement username = driver.findElement(By.id(“uname”));

String tagname = username.getTagName();

or

WebDriver driver = new ChromeDriver();

String tagname = driver.findElement(By.id(“uname”)).getTagName();

1. **getText();**

This method used to get text of a WebElement.

Return Type : String

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement username = driver.findElement(By.id(“uname”));

String tagname = username.getText();

or

WebDriver driver = new ChromeDriver();

String tagname = driver.findElement(By.id(“uname”)).getText();

1. **isDisplayed();**

This method used to conform a WebElement is displayed or not on a webpage.

Return Type : Boolean

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement username = driver.findElement(By.id(“uname”));

Boolean user = username.isDisplayed();

or

WebDriver driver = new ChromeDriver();

Boolean user = driver.findElement(By.id(“uname”)).isDisplayed();

1. **isEnabled();**

This method used to conform a WebElement is enable or not on a webpage.

Return Type : Boolean

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement checkbox = driver.findElement(By.id(“checkbox”));

Boolean enable = checkbox.isEnabled();

Or

WebDriver driver = new ChromeDriver();

Boolean enable = driver.findElement(By.id(“checkbox”)).isEnabled();

1. **isSelected();**

This method used to conform a WebElement is selected or not on a webpage.

Return Type : Boolean

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement checkbox = driver.findElement(By.id(“checkbox”));

Boolean select = checkbox.isSelected();

Or

WebDriver driver = new ChromeDriver();

Boolean select = driver.findElement(By.id(“checkbox”)).isSelected();

1. **getLocation();**

This method used to get co-ordinates of a WebElement on a webpage.

Return Type : Point

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement button = driver.findElement(By.id(“btn”));

Point p = button.getLocation();

System.out.println(p.getX());

System.out.println(p.getY());

Or

WebDriver driver = new ChromeDriver();

System.out.println(driver.findElement(By.id(“btn”)).getLocation().getX());

System.out.println(driver.findElement(By.id(“btn”)).getLocation().getY());

1. **getSize()**

This method used to get the size of a WebElement.

Return type : Dimension

**Syntax**:

Dimension dimension = driver.findElement(By.id(“xx”)).getSize();

System.out.println(dimension.getHeight());

System.out.println(dimension.getWidth());

1. **getRect();**

This method used to get rectangle dimensions of a WebElement on a webpage.

Return Type : Rectangle

**Syntax**:

WebDriver driver = new ChromeDriver();

WebElement button = driver.findElement(By.id(“btn”));

Rectangle rec = button.getRect();

System.out.println(rec.getHeight());

System.out.println(rec.getWidth());

Or

WebDriver driver = new ChromeDriver();

System.out.println(driver.findElement(By.id(“btn”)).getRect().getHeight());

System.out.println(driver.findElement(By.id(“btn”)).getRect().getWidth());

1. **getScreenShotAs**()

This method used to get screen shot of a WebElement.

Parameter type : OutputType

**Syntax**:

File source = chrome.findElement(By.*name*("q")).getScreenshotAs(OutputType.***FILE***);

File dest = **new** File("D:\\xxxx\\test.png");

FileUtils.*copyFile*(source, dest);

**Secreenshot of a section in a page:**

**Example : table , grid or group of matched web elements**

File source = chrome.findElement(By.*name*("tablename")).getScreenshotAs(OutputType.*FILE*);

File dest = new File("D:\\xxxx\\test.png");

FileUtils.*copyFile*(source, dest);

**What are Wait commands in Selenium?**

* The wait commands are essential when it comes to executing Selenium tests. They help to observe and troubleshoot issues that may occur due to variation in time lag.
* In Selenium, “Waits” play an important role in executing tests.

**Why Do We Need Waits In Selenium?**

* Most of the web applications are developed using Ajax and Javascript. When a page is loaded by the browser the elements which we want to interact with may load at different time intervals.
* Not only it makes this difficult to identify the element but also if the element is not located it will throw an “ElementNotVisibleException” exception. Using Selenium Waits, we can resolve this problem.
* Let’s consider a scenario where we have to use both implicit and explicit waits in our test. Assume that implicit wait time is set to 20 seconds and explicit wait time is set to 10 seconds.
* Suppose we are trying to find an element which has some “ExpectedConditions “(Explicit Wait), If the element is not located within the time frame defined by the Explicit wait(10 Seconds), It will use the time frame defined by implicit wait(20 seconds) before throwing an “ElementNotVisibleException“.

**Selenium Web Driver Waits:**

1. Implicit Wait
2. Explicit Wait

Diagram

Description automatically generated

**Implicit Wait in Selenium:**

* The Implicit Wait in Selenium is used to tell the web driver to wait for a certain amount of time before it throws a “No Such Element Exception”. The default setting is 0. Once we set the time, the web driver will wait for the element for that time before throwing an exception.
* Selenium Web Driver has borrowed the idea of implicit waits from Watir.
* We should note that implicit waits will be in place for the entire time the browser is open. This means that any search for elements on the page could take the time the implicit wait is set for.
* For example we have declared an implicit wait with the time frame of 10 seconds. It means that if the element is not located on the web page within that time frame, it will throw an exception.

**Implicit Wait syntax:**

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

Implicit wait will accept 2 parameters, the first parameter will accept the time as an integer value and the second parameter will accept the time measurement in terms of SECONDS, MINUTES, MILISECOND, MICROSECONDS, NANOSECONDS, DAYS, HOURS, etc.

**Explicit Wait in Selenium:**

* The Explicit Wait in Selenium is used to tell the Web Driver to wait for certain conditions (Expected Conditions) or maximum time exceeded before throwing “ElementNotVisibleException” exception. It is an intelligent kind of wait, but it can be applied only for specified elements. It gives better options than implicit wait as it waits for dynamically loaded Ajax elements.
* Once we declare explicit wait we have to use “ExpectedConditions” or we can configure how frequently we want to check the condition using Fluent Wait. These days while implementing we are using Thread.Sleep() generally it is not recommended to use

**Explicit Wait syntax:**

WebDriverWait wait = new WebDriverWait(WebDriverRefrence,TimeOut);

**Example:**

WebDriverWait wait = new WebDriverWait(driver,20);

WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(WebElement));

element.click();

**The following are the Expected Conditions that can be used in Selenium Explicit Wait:**

1. alertIsPresent()
2. elementSelectionStateToBe()
3. elementToBeClickable()
4. elementToBeSelected()
5. frameToBeAvaliableAndSwitchToIt()
6. invisibilityOfTheElementLocated()
7. invisibilityOfElementWithText()
8. presenceOfAllElementsLocatedBy()
9. presenceOfElementLocated()
10. textToBePresentInElement()
11. textToBePresentInElementLocated()
12. textToBePresentInElementValue()
13. titleIs()
14. titleContains()
15. visibilityOf()
16. visibilityOfAllElements()
17. visibilityOfAllElementsLocatedBy()
18. visibilityOfElementLocated()

**Fluent Wait in Selenium:**

* The Fluent Wait in Selenium is used to define maximum time for the web driver to wait for a condition, as well as the frequency with which we want to check the condition before throwing an “ElementNotVisibleException” exception. It checks for the web element at regular intervals until the object is found or timeout happens.
* Frequency: Setting up a repeat cycle with the time frame to verify/check the condition at the regular interval of time.
* Let’s consider a scenario where an element is loaded at different intervals of time. The element might load within 10 seconds, 20 seconds or even more then that if we declare an explicit wait of 20 seconds. It will wait till the specified time before throwing an exception. In such scenarios, the fluent wait is the ideal wait to use as this will try to find the element at different frequency until it finds it or the final timer runs out.

**Fluent Wait syntax:**

Wait wait = new FluentWait(WebDriver reference)

.withTimeout(Duration.ofSeconds(SECONDS))

.pollingEvery(Duration.ofSeconds(SECONDS))

.ignoring(Exception.class);

**Example:**

Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)

.withTimeout(30, TimeUnit.SECONDS)

.pollingEvery(5, TimeUnit.SECONDS)

.ignoring(NoSuchElementException.class);

**Window-handles in selenium:**

**Parent window** : The main window on which the user has currently landed or webpage on which the user is landed after hitting a link/URL. Such a webpage in Selenium is referred as the parent window also known as the main window.

**Child Window** : All the windows which will open inside your main window will be called as child windows.

**Window Handle**: A window handle stores the unique address of the browser windows. It is just like window navigator , whose return type is alphanumeric(**String**). The window handle in Selenium helps in handling multiple windows and child windows. Each browser/tab will have a unique window handle value with which we can uniquely identify it.

1. **getWindowHandle**() : This method will handle the **current window** that uniquely identifies it within this driver instance. Its return type is **String**.

**Syntax** :

String windowHandle = chrome.getWindowHandle();

System.out.println("Parent window Handle : "+windowHandle);

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

**Syntax**:

Set<String> windowsHandles = chrome.getWindowHandles();

for(String window:windowsHandles)

{

driver.switchTo().window(window).close();

}

**Selenium RemoteTargetLocator class methods:**

1. activeElement()

This method used to switch the web driver focus on active WebElement.

Return Type : WebElement

Syntax:

driver.switchTo().activeElement().click();

1. window()

This method used to switch the web driver focus on given windowHandle passed as parameter.

Return Type : WebDriver

Syntax:

driver.switchTo().window(“windowHandle”);

**Popups in selenium:**

**What is an alert?**

Alerts are small popup boxes/desktop-windows which display the messages/notifications and notify the user with some information seeking some permission on certain kinds of operations. Additionally, we can also use them for warning purposes. Sometimes, the user can enter a few details in the alert box as well.

In selenium, pop up are categorized into following types.

1. Javascript Popup
2. Hidden Division popup
3. File Upload popup (File upload window popup)
4. File download popup (File download window popup)
5. Child browser popup (Tabs)
6. Window popup (Any OS-based popup)

**Javascript Alert:**

This pop up is subdivided into below mentioned 3 pop ups.

**1.Simple alert**: These alerts are just informational alerts and have an OK button on them. Users can click on the OK button after reading the message displayed on the alert box. A simple alert box looks like below.

Graphical user interface, text, application, email

Description automatically generated

**2.Prompt Alert**: In Prompt alerts, some input requirement is there from the user in the form of text needs to enter in the alert box. A prompt alert box is displayed like below, where the user can enter his/her username and press the OK button or cancel the alert box without entering any details.

Graphical user interface, application, website, Teams

Description automatically generated

**3.Confirmation Alert**: These alerts get some confirmation from the user in the form of accepting or dismissing the message box. They are different from prompt alerts in a way that the user cannot enter anything as there is no text-box available. Users can only read the message and provide the inputs by pressing the OK/Cancel button.

Graphical user interface, application

Description automatically generated

1. **accept**() : This method is used to click on accept/ok/yes button on an alert.

* **dismiss**() : This method is used to click on cancel/no/reject button on an alert.
* **getText**() : This method is used to return the text present on an alert.
* **sendKeys**() : This method is used to send some commands to an alert.

**Syntax:**

WebDriver chrome = new ChromeDriver();

chrome.switchTo().alert().accept();

chrome.switchTo().alert().dismiss();

chrome.switchTo().alert().sendKeys("Hello Siddhi");

String alertText = chrome.switchTo().alert().getText();

System.out.println(alertText);

**Hidden Division Popup:**

**Characteristics:**

1. We cannot move the popup
2. We can Inspect the popup
3. When hidden division pop is opened, pop takes the focus from the application.
4. When pop up is closed, focus automatically goes to the application
5. Hidden division popup could be nested, i.e., a hidden division pop can have another hidden division pop up
6. Hidden division pop can hold other pop-ups/ alerts on it.
7. If the content is more then the pop-up size, pop shows scroll bar
8. It will be colorful

**Solution:** To handle Hidden Division Popup we use findElement() method.

**Ex: calendar popup, a type of hidden division popup**

**File download popup:**

**Characteristic features:**

* We can move this popup, but we can’t inspect it.
* This pop up will have 2 radio buttons : Open with and Save File

**How to handle File Download pop up:**

* In Google Chrome browser, when we click on Download link of Java language present on Selenium official website, it doesn’t show any file download pop up on the screen, instead, it automatically starts downloading the file in default location on the system. (i.e downloads folder)
* But, in firefox browser, on clicking on the same download link, we get a file download pop up on the screen. In order to handle this pop up, we use setPreference() method of FirefoxProfile class.
* setPreference() is used to change the settings of Firefox browser.
* setPreference() method is an overloaded method which takes 2 parameters (KEY, VALUE).

**“Key” will always be a String,**

**“Value” can be either String or int or boolean**

**Following example demonstrates how to use key and value with setPreference() method.**

FirefoxProfile profile = new FirefoxProfile();

// If the file type is .zip, then don’t display the popup, instead, download it directly.

String key = "browser.helperApps.neverAsk.saveToDisk";

String value = "application/zip";

profile.setPreference(key, value);

// 0 - save to desktop, 1 - save to downloads folder (default value),

// 2 - save the downloaded file to other folders in the system

profile.setPreference("browser.download.folderList", 2);

profile.setPreference("browser.download.dir", "D:\\");

In the above example, "application/zip" refers to MIME types. (Multi purpose Internet Mail Extension), which says what kind of file you want to download.

**Summary of the different popups in selenium and how to handle those is mentioned below.**Table

Description automatically generated

**IFrames in selenium:**

**What is iframe ?**

* + Iframe is webpage which is embedded in another web page

**(or)**

* + It is a HTML document embedded inside another HTML document
  + These frames are used in a webpage to add content from other source like advertisements in a webpage.
  + These iframe are defined with **<iframe>** tag.
  + If the specified frame is not present, we get an exception called “NoSuchFrameException”

**How to identity iframe on a web page?**

* We cannot detect the frames by just seeing the page or by inspecting on webpage.
* But when we try to inspect on web page, on context click options it will display

**View frame source**  or **Reload frame** in chrome browserand **This Frame** in Firefox bowser by these options we can confirm that webpage consists of frames.

Graphical user interface, text, application, email

Description automatically generated 

1. **frame(int )** : This method is used to switch the frame by its index.
2. **frame(String)** : This method is used to switch the frame by its name or id.
3. **frame(WebElement)** : This method is used to switch the frame by its webelement.
4. **parentFrame()** : This method is used to switch immediate parent frame from child frame.
5. **defaultContent()** : This method is used to switch main page from any frame.

**Single Frame:**

Graphical user interface, application, Word

Description automatically generated

**Html code:**

<html>

<head> </head>

<body class=" darktheme">

<iframe frameborder="20" id="frame1" name="frame1" allowfullscreen="true">

<button type="submit1" id="loginbutton” form="nameform1" value="Submit">Submit1</button>

</iframe>

</body>

</html>

**Selenium code :**

**With index:**

chrome.switchTo().frame(0);

chrome.findElement(By.id("loginbutton")).click();

chrome.switchTo().defaultContent();

or

**With id or name:**

chrome.switchTo().frame("frame1");

chrome.findElement(By.id("loginbutton")).click();

chrome.switchTo().defaultContent();

or

**With web-element:**

chrome.switchTo().frame(chrome.findElement(By.id("iframe1")).);

chrome.findElement(By.id("loginbutton")).click();

chrome.switchTo().defaultContent();

**Multiple frame:**

Diagram

Description automatically generated

**Html code:**

<html>

<head> </head>

<body class=" darktheme">

<iframe frameborder="0" id="frame1" name="frame1" allowfullscreen="true">

<button type="submit" form="nameform" value="Submit">Submit1</button>

</iframe>

<iframe frameborder="0" id="frame2" name="frame2" allowfullscreen="true">

<button type="submit" form="nameform" value="Submit">Submit2</button>

</iframe>

<iframe frameborder="0" id="frame3" name="frame3" allowfullscreen="true">

<button type="submit" form="nameform" value="Submit">Submit3</button>

</iframe>

</body>

</html>

**Selenium code :**

chrome.switchTo().frame("iframe1");

chrome.findElement(By.id("loginbutton")).click();

chrome.switchTo().defaultContent();

chrome.switchTo().frame("iframe2");

chrome.findElement(By.id("loginbutton")).click();

chrome.switchTo().defaultContent();

chrome.switchTo().frame("iframe3");

chrome.findElement(By.id("loginbutton")).click();

chrome.switchTo().defaultContent();

**Nested Frames:**



<html>

<head> </head>

<body class=" darktheme">

<iframe frameborder="0" id="iframe2" name=" iframe2" allowfullscreen="true">

<button type="submit" form="nameform" id=" element2" value="Submit">element2</button>

<iframe frameborder="0" id="iframe1" name=" iframe1" allowfullscreen="true">

<button type="submit" form="nameform" " id=" element1" value="Submit">element1</button>

</iframe>

</iframe>

</body>

</html>

**Selenium syntax :**

chrome.switchTo().frame("iframe2");

chrome.findElement(By.id("element2")).click();

chrome.switchTo().frame("iframe1");

chrome.findElement(By.id("element1")).click();

chrome.switchTo().parentFrame();

chrome.switchTo().defaultContent();

**Select Class in selenium**:

Select is used to automate dropdown which are developed by using <select> HTML tag.

Graphical user interface, text, application

Description automatically generated

**Types of dropdowns**:

1.single selection dropdown

Example :



2. Multiple selection dropdown

Example :



**Select-Methods:**

**How to check whether dropdown is multi-Select?**

The Select class provides the "**isMultiple()** " method, which determines whether the web element is supports for multiple selections or not. It returns a boolean value, i.e., True/False, without taking any argument. It checks the attribute 'multiple' in the HTML code for the web element.

1. **isMultiple**

Syntax :

Select country = new Select(driver.findElement(By.name("xxxxxx")));

Boolean selection = country.isMultiple();

System.out.println(selection);

1. **selectByIndex()** :

This method selects the dropdown option by its index number. We provide an integer value as the index number as an argument.

The index starts at 0

**syntax**:

Select country = new Select(driver.findElement(By.name("xxxxxx")));

country.selectByIndex(3);

**2.** **selectByValue() :**

This method selects the dropdown option by its value. We provide a string value as the value as an argument.

**syntax**:

Select country = new Select(driver.findElement(By.name("xxxxxx")));

country.selectByValue("india");

3. **selectByVisibleText() :**

This method selects the dropdown option based on the dropdown text. We provide the dropdown text as a string as an argument.

**Syntax**:

Select country = new Select(driver.findElement(By.name("xxxxxx")));

country.selectByVisibleText("INDIA");

4. **getOptions()** :

This method is used to get all the options in a dropdown or multi-select box.

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

This method returns all the options of the dropdown as a list of WebElement (List<WebElement>).

**syntax**:

Select country = new Select(driver.findElement(By.name("xxxxxx")));

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

for(WebElement option:options)

{

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

}

5. **getFirstSelectedOption() :**

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

This method returns a WebElement.

**Syntax:**

Select country = new Select(driver.findElement(By.name("xxxxxx")));

System.out.println(country.getFirstSelectedOption());

6. **getAllSelectedOptions() :**

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

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

This method returns a WebElement’s.

**Syntax:**

Select country = new Select(driver.findElement(By.name("xxxxxx")));

List<WebElement>options = country. getAllSelectedOptions ();

for(WebElement option:options)

{

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

}

7. **deselectAll():**

It will deselect all the options from the dropdown.

**Syntax:**

Select country = new Select(driver.findElement(By.name("xxxxxx")));

country.deselectAll()

8. **deselectByIndex ():**

This method is used to deselect an option from the dropdown by using its index.

**Syntax:**

Select country = new Select(driver.findElement(By.name("xxxxxx")));

country. deselectByIndex(index);

9. **deselectByValue ():**

This method is used to deselect an option from the dropdown by using its value.

**Syntax:**

Select country = new Select(driver.findElement(By.name("xxxxxx")));

country.deselectByValue(“Value”);

10. **deselectByVisibleText():**

This method is used to deselect an option from the dropdown by using its Text.

**Syntax:**

Select country = new Select(driver.findElement(By.name("xxxxxx")));

country.deselectByVisibleText(“Text”);

**Write a script to print the UNIQUE content of the list box.**

Hint : Use **HashSet<>**

Selenium Code :

public class printUniqueElementinthelistbox extends BaseClass

{

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

{

Select s = new Select(listElement);

List<WebElement> allOptions = s.getOptions();

int count = allOptions.size();

System.out.println(count);

System.out.println("-----print the values in the list");

HashSet<String> allElements = new HashSet<String>();

for (WebElement option : allOptions)

{

String text = option.getText();

System.out.println(text);

allElements.add(text);

}

System.out.println(allElements);

}

}

**Write a script to print the UNIQUE content of the list box in SORTED order.**

Hint : Use TreeSet<>

Selenium Code :

public class printUniqueElement\_Sorted extends BaseClass

{

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

{

driver.get("url");

WebElement listElement = driver.findElement(By.id("mtr"));

Select s = new Select(listElement);

List<WebElement> allOptions = s.getOptions();

int count = allOptions.size();

System.out.println(count);

System.out.println("-----print the values in the list");

TreeSet<String> allElements = new TreeSet<String>();

for (WebElement option : allOptions)

{

String text = option.getText(); System.out.println(text);

allElements.add(text);

}

System.out.println(allElements);

}

}

**Write a script to print the content of the list box in sorted order**.

Selenium Code :

public class PrintListValues\_SortedOrder extends BaseClass

{

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

{

driver.get("url");

WebElement listElement = driver.findElement(By.id("mtr"));

Select s = new Select(listElement);

List<WebElement> allOptions = s.getOptions();

int count = allOptions.size();

System.out.println(count);

System.out.println("-----print the values in the list ");

ArrayList<String> list = new ArrayList<String>();

for (WebElement option : allOptions)

{

String text = option.getText(); System.out.println(text); list.add(text);

}

Collections.sort(list);

System.out.println("-----print the value in sorted order ");

for (String value : list)

{

System.out.println(value);

}

}

}

**Write a script to check whether listbox has duplicate or not ?**

Selenium Code :

public class checklisthasDUPLICATEvalues\_HashSet extends BaseClass

{

public static void main(String[] args)

{

driver.get("URL");

WebElement listbox = driver.findElement(By.id("mtr"));

Select s = new Select(listbox);

List<WebElement> allOptions = s.getOptions();

int count1 = allOptions.size();

System.out.println("Number of elements in the list is :" +count1);

HashSet<String> allElementText = new HashSet<String>();

for (int i = 0; i < count1; i++)

{

String text = allOptions.get(i).getText();

System.out.println(text);

allElementText.add(text);

}

int count2 = allElementText.size();

System.out.println("Number of elements in the hashset is :" +count2);

if (count1==count2)

{

System.out.println("list box has NO duplicate values");

}

else

{

System.out.println("list box has duplicate values");

}

System.out.println(allElementText); driver.close();

}

}

}

**Write a script to print the duplicate item in the list ?**

Selenium Code :

public class PrinttheDUPLICATEItem\_intheList\_HashSet extends BaseClass

{

public static void main(String[] args)

{

driver.get("url");

WebElement listbox = driver.findElement(By.id("mtr"));

Select s = new Select(listbox);

List<WebElement> allOptions = s.getOptions();

int count1 = allOptions.size();

System.out.println("Number of elements in the list is :" +count1);

HashSet<String> allElementText = new HashSet<String>();

for (int i = 0; i < count1; i++)

{

String text = allOptions.get(i).getText();

/\*allElementText.add(text) returns true if the element is not already added, and it returns false if the same element is trying to be added twice. \*/

if (!allElementText.add(text))

{

System.out.println(text +" is the duplicate item in the list box");

}

}

System.out.println(allElementText.size());

// it will print all the unique values in the HashSet object

System.out.println(allElementText);

driver.close();

}

}

**Print the number of occurrance of Poori in the list box.**

Selenium Code:

public class HashMapExample\_printtheOcuuranceOfPoori extends BaseClass

{

public static void main(String[] args)

{

driver.get("url");

WebElement list = driver.findElement(By.id("mtr"));

Select s = new Select(list);

List<WebElement> allElements = s.getOptions();

HashMap<String, Integer> hashMapObj = new HashMap<String, Integer>();

for (WebElement element : allElements)

{

String text = element.getText();

if (hashMapObj.containsKey(text))

{

Integer value = hashMapObj.get(text); value++;

hashMapObj.put(text, value);

}

else

{

hashMapObj.put(text, 1);

}

}

Set<String> allKeys = hashMapObj.keySet();

for (String key : allKeys)

{

Integer value = hashMapObj.get(key);

System.out.println(key +" -->"+ value);

if (value>1)

{

System.out.println("Occurance of " + key + " is :" + value);

}

}

}

}

**Actions Class in selenium:**

**Actions Class:**

* Actions class is provided by selenium to perform mouse events and handling keyboard operations.
* Actions classes are used to perform multiple actions at a time sequentially.
* **build()** method is always the final method, so that all the listed actions will be compiled into a single step
* **perform()** method is used to perform all compiled/stored actions.

**Different Methods in Actions class to perform Keyboard Events:**

1. **keyDown(Keys):**

This method used to perform a key press(Keyboard).

1. **keyDown(WebElement** , **Keys):**

This method performs, cursor focus on given webelement and key press(Keyboard).

1. **keyUp(Keys):**

This method used to perform a key release(Keyboard).

1. **keyUp(WebElement** , **Keys):**

This method performs, cursor focus on given webelement and key release(Keyboard).

1. **sendKeys(keys):**

This method used to perform Sendkeys to the active web element.

1. **sendKeys(WebElement** , **keys):**

This method performs, cursor focus on given webelement and Sendkeys.

Example for keyDown, KeyUp & sendKeys :

**Syntax:**

Actions action = new Actions(chrome);

WebElement firstname = chrome.findElement(By.name("firstName"));

action.click(firstname).keyDown(firstname,Keys.SHIFT)

.sendKeys(firstname, "siddhiInstitute")

.keyUp(firstname,Keys.SHIFT).build().perform();

**Different Methods in Actions class to perform Mouse Events:**

1. **moveToElement(WebElement):** This method used to move the mouse pointer to the middle of the given WebElement.

**Syntax:**

Actions action = new Actions(chrome);

WebElement nextbutton = chrome.findElement(By.name("xxxxxx"));

action.moveToElement(nextbutton).build().perform();

1. **click():** This method used to click at the **current** mouse pointer.

**Syntax:**

Actions action = new Actions(chrome);

action.click().build().perform();

1. **click(WebElement):** This method used to perform click on the WebElement, which is passed an argument.

**Syntax:**

Actions action = new Actions(chrome);

WebElement nextbutton = chrome.findElement(By.name("xxxxxx"));

action.click(nextbutton).build().perform();

1. **doubleClick():** This method used to double-click at the current mouse pointer.

**Syntax:**

Actions action = new Actions(chrome);

action.doubleClick().build().perform();

1. **doubleClick(WebElement):** This method used to perform double-click on the WebElement, which is passed an argument.

**Syntax:**

Actions action = new Actions(chrome);

WebElement nextbutton = chrome.findElement(By.name("xxxxxx"));

action.doubleClick(nextbutton).build().perform();

1. **contextClick()** : This method used to perform a context-click(mouse right click) at current mouse pointer.

**Syntax:**

Actions action = new Actions(chrome);

action.contextClick().build().perform();

1. **contextClick(WebElement)** : This method used to perform a context-click(mouse right click) at middle of the given WebElement.

**Syntax:**

Actions action = new Actions(chrome);

WebElement menu = chrome.findElement(By.name("xxxxxx"));

action.contextClick(menu).build().perform();

1. **clickAndHold():** This method used to perform mouse left click (without releasing) at current mouse pointer.

**Syntax:**

Actions action = new Actions(chrome);

action.clickAndHold().build().perform();

1. **clickAndHold(WebElement):** This method used to perform mouse left click (without releasing) at the middle of the given WebElement.

**Syntax:**

Actions action = new Actions(chrome);

WebElement imagetohold = chrome.findElement(By.name("xxxxxx"));

action.clickAndHold(imagetohold).build().perform();

1. **release():** This method used to release the depressed left mouse button at the current mouse location.

**Syntax:**

Actions action = new Actions(chrome);

action.release().build().perform();

1. **release(WebElement):** This method used to release the depressed left mouse button on the given webelement.

**Syntax:**

Actions action = new Actions(chrome);

WebElement imagetohold = chrome.findElement(By.name("xxxxxx"));

action.clickAndHold(imagetohold).build().perform();

action.release(imagetohold).build().perform();

1. **dragAndDrop(source we, target we):** This method used to perform click-and-hold at the location of the source webelement, moves and release at the location of the target webelement.

**Syntax**:

WebElement source = chrome.findElement(By.xpath(“xxx"));

WebElement target = chrome.findElement(By.xpath(“xxx"));

action.dragAndDrop(source, target).build().perform();

1. **dragAndDropBy(source, xOffset, yOffset):** This method used to perform click-and-hold the mouse at the location of the source webelement, moves and release at given offset.

**Syntax**:

WebElement source = chrome.findElement(By.xpath(“xxx"));

action.dragAndDrop(source, x, y).build().perform();

1. **moveByOffset(x-offset, y-offset):** This method used to move the mouse from its current position (or 0,0) to the given offset.

**Syntax**:

WebElement source = chrome.findElement(By.xpath(“xxx"));

action.moveByOffset(x, y).build().perform();

1. **pause(long):** This method is used to pause/delay the execution between actions for a specified duration in milli seconds.

**Syntax**:

Actions action = new Actions(chrome);

action.click().pause(2000).click().build().perform();

1. **pause(Duration ):** This method is used to pause/delay the execution between actions for a specified duration.

**Syntax**:

Actions action = new Actions(chrome);

action.click(firstname).pause(Duration.ofSeconds(2)).sendKeys("xx”).build().perform();

**JavaScriptExecuter in selenium:**

**What is JavaScriptExecuter ?**

* Javascriptexecutor is an interface which is used to execute JavaScript with selenium webdriver.
* We use JavascriptExecutor when we fail to perform some actions using selenium.
* Javascriptexecutor allows you to run pure JavaScript code irrespective of the Selenium language binding(Java, C#, Python etc.) you are using.

**Why JavaScriptExecuter?**

* JavaScript is extremely efficient and accurate when comparing with scripts written with Selenium-Java/c#/python etc.
* some features we can’t handle using Java-selenium so we need scripting language also which can control server side or client-side scripting so we will use JavaScript in our Selenium

It consists of two methods used to run JavaScript

1. executeScript : This method is used to execute JavaScript in sequential order.
2. executeAsyncScript :

* This method is used to execute asynchronous script in random order.(Time constants)
* we don’t use this for automation

**Note** : Both methods will block the WebDriver execution until they complete execution

**What is the usage of JavascriptExecutor ?**

1. To scroll on the webpage.
2. To handle the disabled elements
3. To use as an alternate solution when selenium inbuilt methods ( eg : clear(), click(), sendkeys() ) doesn’t work

**JavaScriptExecuter Actions:**

1. **Navigating URL :**

JavascriptExecutor js = (JavascriptExecutor) driver; js.executeScript("window.location=’url’");

1. **Title of a Webpage :**

JavascriptExecutor js = (JavascriptExecutor) driver;

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

System.***out***.println(title);

1. **Domain of a Webpage :**

JavascriptExecutor js = (JavascriptExecutor) driver;

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

System.***out***.println(domain);

1. **URL of a Webpage**

JavascriptExecutor js = (JavascriptExecutor) driver;

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

System.***out***.println(url);

1. **Inner Text of a WebElement :**

JavascriptExecutor js = (JavascriptExecutor) driver;

String text = js.executeScript("return arguments[0].innerText",WebElement).toString();

System.***out***.println(text);

(or)

String text1 = js.executeScript("return document.getElementById('xx').innerText").toString();

System.***out***.println(text1);

1. **Attribute of a WebElement:**

JavascriptExecutor js = (JavascriptExecutor) driver;

String arg = js.executeScript("return arguments[0].getAttribute('xxx')",WebElement).toString();

System.***out***.println(arg);

(or)

String arg1 = js.executeScript("return document.getElementById('xxx').getAttribute('xxx')").toString();

System.***out***.println(arg1);

1. **click on WebElement**

JavascriptExecutor js = (JavascriptExecutor) driver;

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

(or)

js.executeScript("document.getElementById('xxx').click();");

1. **sendkeys on WebElement**

JavascriptExecutor js = (JavascriptExecutor) driver;

js.executeScript("arguments[0].value='xxxx'", WebElement);

(or)

js.executeScript("document.getElementById('xx').value='xxx'");

1. **vertical scroll**

JavascriptExecutor js = (JavascriptExecutor) driver;

js.executeScript("window.scrollTo(0,300)");

1. **horizontal scroll**

JavascriptExecutor js = (JavascriptExecutor) driver;

js.executeScript("window.scrollTo(300,0)");

1. **scroll bottom of the webpage**

JavascriptExecutor js = (JavascriptExecutor) driver;

js.executeScript("window.scrollTo(0,document.body.scrollHeight)");

1. **scroll top of the webpage**

JavascriptExecutor js = (JavascriptExecutor) driver;

js.executeScript("window.scrollTo(document.body.scrollHeight,0)");

1. **Height of a webpage**

JavascriptExecutor js = (JavascriptExecutor) driver;

String height = js.executeScript("return window.innerHeight").toString();

System.***out***.println(height);

1. **Width of a webpage**

JavascriptExecutor js = (JavascriptExecutor) driver;

String width = js.executeScript("return window.innerWidth").toString();

System.***out***.println(width);

1. **changing the border color**

JavascriptExecutor js = (JavascriptExecutor) driver;

js.executeScript("arguments[0].style.borderColor='yellow'", WebElement);

1. **scroll Into View**

JavascriptExecutor js = (JavascriptExecutor) driver; js.executeScript("document.getElementById('xx').scrollIntoView(true)"); (or)

js.executeScript("arguments[0].scrollIntoView(true);", WebElement);

1. **multiple actions**

JavascriptExecutor js = (JavascriptExecutor) driver; js.executeScript("arguments[0].click();arguments[1].click();",

WebElement1, WebElement2);

1. **Example for asyncscript**

JavascriptExecutor js = (JavascriptExecutor) driver; js.executeAsyncScript("window.setTimeout(arguments[arguments.length-1],5000);alert('Hello');");

1. **alert creation**

JavascriptExecutor js = (JavascriptExecutor) driver;

js.executeScript("alert('Hello');");

1. **Refresh webpage**

JavascriptExecutor js = (JavascriptExecutor) driver;

js.executeScript("history.go(0)");

**Screenshot in Selenium Webdriver :**

**TakesScreenshot** is an interface which is used to take screenshot of a web page during the execution for debugging the failures.

Why capturing screenshot of the web pages is important in the project **?**

Answer :

* We capture screenshots in order to debug the failed test scripts.
* It actually helps the automation test engineer to find the exact root cause of the issue in the application at the earliest.

To take screen shot first we need to convert webdriver instance into TakesScreenshot.

Jar files required to add project : **Apache Commons IO**

**TakesScreenshot Methods :**

1. **getScreehshotAs(OutputType) :** This method used to take screen shot and convert into image file.

**Syntax :**

TakesScreenshot screenshot = (TakesScreenshot) chrome;

File source = screenshot.getScreenshot(OutputType.***FILE***);

File dest = **new** File("path where to store");

FileUtils.*copyFile*(source, dest);

**Why we need to typecast the WebDriver Instance?**

WebDriver driver = new ChromeDriver();

JavascriptExecutor jsExecutor = (JavascriptExecutor) driver;

TakesScreenshot takesScreenshot = (TakesScreenshot) driver;

**FileUpload in selenium:**

We can use webdriver sendKeys method to upload file in file upload option. But that file upload WebElement must have **<input> tag** and attribute **type=’file’**.

Example of file upload html :

Text

Description automatically generated

**Syntax :**

chrome.findElement(By.*id*("xxxx")).sendKeys("path");

**FileUpload with Robot Class:**

Robot class used to operate all keyboard event.

chrome.findElement(By.*id*("uploadbutton")).click();

StringSelection path = **new** StringSelection("C:\\xxxxx\\xxxx.png");

Toolkit.*getDefaultToolkit*().getSystemClipboard().setContents(path,**null**);

Robot robot = **new** Robot();

robot.keyPress(KeyEvent.***VK\_ENTER***);

robot.keyRelease(KeyEvent.***VK\_ENTER***);

robot.keyPress(KeyEvent.***VK\_CONTROL***);

robot.keyPress(KeyEvent.***VK\_V***);

robot.keyRelease(KeyEvent.***VK\_V***);

robot.keyRelease(KeyEvent.***VK\_CONTROL***);

robot.keyPress(KeyEvent.***VK\_ENTER***);

robot.keyRelease(KeyEvent.***VK\_ENTER***);

**FileUpload with Auto IT:**

**Software required to download :**

1. **Autoit :** To read window information.
2. **Script Editor :** To write autoit script.

**What is AutoIT?**

* + It is an open-source window-based automation tool.
  + It can be downloaded from below mentioned site :

<https://www.autoitscript.com/site/autoit/downloads>



* + Download the above Editor on your system.
  + Double click on the Setup file.
  + Follow the default instruction to install autoIT.

How Auto IT identifies objects on window popup ?

* Elements present on window pop up are known as CONTROLS.
* In order to inspect these controls, AutoIT uses AutoIT Window Info”.
* In order to open “AutoIT Window Info”, navigate to the below path.

Go to Start → All Programs → AutoIT V3 → Select AutoIT Window Info.

* As a result, the below window opens up.

Graphical user interface, application

Description automatically generated

* In the above image, drag the “Finder Tool” option and drop it on any element/control present on the window pop up for which you want to identify the properties.
* It will display the properties of the same controls such as Class, Name, ID and Text.
* These properties are also known as CONTROL ID, using which AutoIT locates elements/controls on window pop up.
* General syntax for using single Control ID is :

[Control ID : Value]

* We can use multiple Control IDs as well using semicolon as the delimiter to identify the controls using below syntax.

[ Control ID 1: Value1 ; Control ID 2 : Value2 ; Control ID 3 ; Value3]

**Steps to write and execute AutoIT script :**

* Navigate to the below path and open the Editor to write the autoIT script

Go to Start → All Programs → AutoIt → Select SciTE Script Editor

* Write the autoIT script and save the file with .au3 extension
* Go to Tool → Select Compile and compile script. As a result, it will generate an .exe file
* Navigate to the folder location where .exe file is located and double click on this .exe file to execute the autoIT script.
* We can also execute the script from eclipse by using RunTime class of Java **RunTime.getRunTime.exec(“path of the compiled au3.exe file”);**

**Syntax:**

chrome.findElement(By.*id*("uploadbutton")).click();

ControlFocus("Open","","Edit1")

ControlSetText("Open","","Edit1","path of file to upload.png")

ControlClick("Open","","Button1")

Runtime.getRuntime().exec("path of compiled autoit script.exe");

**Keys class in selenium:**

Keys class used to perform keyboard actions.

Syntax:

chrome.findElement(By.*xpath*("xx")).sendKeys(Keys.***CONTROL***);

**WebTable :**

Table present on the web page is called WebTable.

**Types of Web Tables** :

Depending on the data in the table, web tables can be classified as Static web tables and Dynamic web tables.

1. **Static Web Tables :**

These tables have fixed data that remains unchanged throughout. Due to the static nature of their content, they are called Static web tables.

1. **Dynamic Web Tables :**

These tables have data that changes over time, and hence the number of rows and columns might also change depending upon the data shifts. Due to the dynamic nature of their content, they are called Dynamic web tables.

**Create a webtable as shown below.**

<table border = "1">

<tr>

<th>SL No</th>

<th>Product</th>

<th>Price</th>

</tr>

<tr>

<td>1</td>

<td>CRM</td>

<td>98</td>

</tr>

<tr>

<td>2</td>

<td>ERP</td>

<td>99</td>

</tr>

<tr>

<td>3</td>

<td>PMP</td>

<td>97</td>

</tr>

</table>

The webpage looks like this as shown below.

**Table

Description automatically generated**

Program :

In the below webtable, find the following scenarios :

1. print the total number of ROWS present
2. print the total number of COLUMNS present
3. print the total number of CELLS present
4. print ONLY the NUMERIC values present
5. Count the TOTAL number of NUMERIC values present
6. print the SUM of all the numeric values in the table

**Selenium Exceptions:**

An exception is a runtime error that happens at the time of execution of a program.

The Exception Handling in selenium is used to handle the runtime errors so that the normal flow of the execution can be maintained.

**Exceptions in Selenium WebDriver :**

1. **AWTException [Abstract Window Toolkit]** : [java - checked] [While handling Robot object]
2. **ElementNotVisibleException** : In spite of the element being present in the DOM, it is not visible (can not be interactive). For example, elements defined in HTML with type =”hidden”
3. **ElementNotSelectableException**: An element is disabled (can not be clicked/selected) in spite of being present in the DOM.
4. **NoSuchElementException**: Webdriver is not able to determine the elements during runtime, i.e., the FindBy method cannot find a particular component.
5. **NoSuchFrameException**: Webdriver attempts to switch to an invalid frame, which is unavailable.
6. **NoAlertPresentException**: Webdriver is trying to switch to an invalid alert, which is unavailable.
7. **NoSuchWindowException**: Webdriver is trying to switch to an invalid window, which is unavailable.
8. **StaleElementReferenceException**: The referenced element is no longer present on the DOM page (a reference to a component is now Stale). For example, the item belongs to a different frame than the current one or the user has navigated away to another page.
9. **SessionNotFoundException**: Webdriver is acting immediately after ‘quitting’ the browser.
10. **TimeoutException**: The command did not complete in the specified time. For example, the element didn’t display at the specified time. This is especially encountered when working with waits.
11. **WebDriverException**: Webdriver is acting immediately after ‘closing’ the browser
12. **ConnectionClosedException** : This type of Exception takes place when there is a disconnection in the driver.
13. **ElementClickInterceptedException** : The command may not be completed as the element receiving the events is concealing the element which was requested clicked.
14. **ElementNotInteractableException** : This Selenium exception is thrown when any element is presented in the DOM. However, it is impossible to interact with such an element.
15. **ErrorHandler.UnknownServerException** : Exception is used as a placeholder in case if the server returns an error without a stack trace.
16. **IllegalStateException :**  [Java - Unchecked] (driver exe path not set)
17. **ImeActivationFailedException :** This expectation will occur when IME engine activation has failed.
18. **ImeNotAvailableException :** It takes place when IME support is unavailable.
19. **InterruptedException** : [Java - Checked ] (Thread.sleep)
20. **IOException** : [Java-Checked][File handling scenario]
21. **InsecureCertificateException :** Navigation made the user agent to hit a certificate warning. This can cause by an invalid or expired TLS certificate.
22. **InvalidArgumentException :** It occurs when an argument does not belong to the expected type.
23. **InvalidCookieDomainException :** This happens when you try to add a cookie under a different domain instead of current URL.
24. **InvalidCoordinatesException :** This type of Exception matches an interacting operation that is not valid.
25. **InvalidElementStateExceptio :** It occurs when command can’t be finished when the element is invalid.
26. **InvalidSessionIdException :** This Exception took place when the given session ID is not included in the list of active sessions. It means the session does not exist or is inactive either.
27. **InvalidSwitchToTargetException :** This occurs when the frame or window target to be switched does not exist.
28. **JavascriptException :** This issue occurs while executing JavaScript given by the user.
29. **JsonException :** It occurs when you afford to get the session when the session is not created.
30. **NoSuchAttributeException :** This kind of Exception occurs when the attribute of an element could not be found.
31. **MoveTargetOutOfBoundsException :** It takes place if the target provided to the ActionChains move() methodology is not valid. For Example, out of the document.
32. **NoSuchCookieException :** This Exception occurs when no cookie matching with the given pathname found for all the associated cookies of the currently browsing document.
33. **NotFoundException :** This Exception is a subclass of WebDriverException. This will occur when an element on the DOM does not exist.
34. **RemoteDriverServerException :** This Selenium exception is thrown when the server is not responding because of the problem that the capabilities described are not proper.
35. **ScreenshotException :** It is not possible to capture a screen.
36. **SessionNotCreatedException :** It happens when a new session could not be successfully created.
37. **UnableToSetCookieException :** This occurs if a driver is unable to set a cookie.
38. **UnexpectedTagNameException**  **:** Happens if a support class did not get a web element as expected.
39. **UnhandledAlertException :** This expectation occurs when there is an alert, but WebDriver is not able to perform Alert operation.
40. **UnexpectedAlertPresentException :** It occurs when there is the appearance of an unexpected alert.
41. **UnreachableBrowserException :** This Exception occurs only when the browser is not able to be opened or crashed because of some reason.

**ChromeOptions class in selenium:**

Ref : **https://peter.sh/experiments/chromium-command-line-switches/**

This class used for customizing the ChromeDriver session. It helps to perform various operations like opening Chrome in maximized mode, disable existing extensions, disable pop-ups, etc.

**List of arguments :**

1. disable-extensions : Disable extensions
2. headless : Run in headless mode, i.e., without a UI.
3. no-sandbox : Sandbox removes unnecessary privileges from the processes that don't need them in Chrome, for security purposes.
4. window-size=1200x600 : Sets the initial window size. Provided as string in the format "800,600".
5. verbose : WebView will log additional debugging information.
6. disable-web-security : Don't enforce the same-origin policy
7. ignore-certificate-errors : ignore all certification issue raised will browsing.
8. allow-running-insecure-content : allow all sights
9. incognito : open browser in incognito mode
10. start-maximized : Opens Chrome in maximize mode
11. disable-popup-blocking : Disables pop-up blocking.
12. disable-notifications : Disables the Web Notification and the Push APIs
13. make-default-browser : Makes Chrome default browser
14. disable-infobars : Prevents Chrome from displaying the notification ‘Chrome is being controlled by automated software
15. make-default-browser : Makes Chrome default browser
16. version : Prints chrome browser version

**Broken Links & Images on webpages:**

**What are Links?**

Hyperlinks, usually called as links and these are the HTML tags which enables users to redirection from one web page to another when they click on it.

**What are Broken Links?**

A broken link, also called as dead link, i.e it does not work, does not redirect to the webpage. This usually occurs because

* The destination web page is down, moved, or no longer exists.
* A web page moved without adding a redirect link.
* The user entered an improper/misspell URL.
* The web page link removed from the website.
* With activated firewall settings, also the browser cannot access the destination web page at times.

When someone clicks on a broken link, an error message is displayed.

Broken links may exist due to some kind of server error. A valid URL will have a 2xx HTTP status code.

Broken links, which are contain invalid HTTP requests have 4xx and 5xx status codes. The 4xx status code refers to a client-side error, while the 5xx status code usually points to a server response error.

**What are Broken images?**

A broken image on a web page is a link that is associated with the image, and the link is not working.

* Firstly, the image file is not located in the same path specified in your <img src " "> tag.
* Secondly, the image does not have the same path or filename.
* Thirdly, the image file at the location is either corrupt or undergoes damage, or maybe it's not compatible with a specific browser, and rendering fails in that browser only.

**Note:** Images can be broken on a web page, even if the link is valid on the page. In such a case, the issue is with either the image file itself or the browser's image rendering.

**What are HTTP Status Codes?**

A server generates HTTP Status codes in response to the request submitted by the client to the server.

There are different HTTP status codes, and a few of them are as below:

* 200 – Valid Link/success
* 301/302 - Page redirection temporary/permanent
* 404 – Page not found
* 400 – Bad request
* 401 – Unauthorized
* 500 – Internal Server Error

**Sytax:**

**public** **class** BrokenLinks {

**public** **static** **void** main(String[] args) **throws** InterruptedException, IOException

{

System.*setProperty*("webdriver.chrome.driver","driverpath");

WebDriver chrome = **new** ChromeDriver();

chrome.get("https://www.zensar.com/");

Thread.*sleep*(15000);

List<WebElement> links = chrome.findElements(By.*tagName*("a"));

**for**(WebElement link : links)

{

String linkurl = link.getAttribute("href");

URL url = **new** URL(linkurl);

HttpURLConnection huc = (HttpURLConnection) url.openConnection();

huc.setConnectTimeout(5000);

huc.connect();

**if**(huc.getResponseCode()>=400) {

System.***out***.println("link broken " + linkurl);

}**else** {

System.***out***.println("Valid link");

}

}

chrome.close();

chrome.quit();

}

}