**Day - 1**

**Introduction on Selenium**

Selenium is an open-source framework for automated browser testing and web application automation, offering a suite of tools to interact with web browsers programmatically.

**Different Selenium components**

Selenium comprises four main components:

1. Selenium IDE
2. Selenium RC
3. Selenium WebDriver
4. Selenium Grid

**Selenium WebDriver:** Allows programmatically interacting with web browsers to

automate testing

**Selenium IDE (Integrated Development Environment):** A browser

extension for recording and playback of browser interactions

**Selenium RC** (Remote Control) was the predecessor to WebDriver, providing a server-based architecture for automated web testing but is now deprecated in favor of WebDriver.

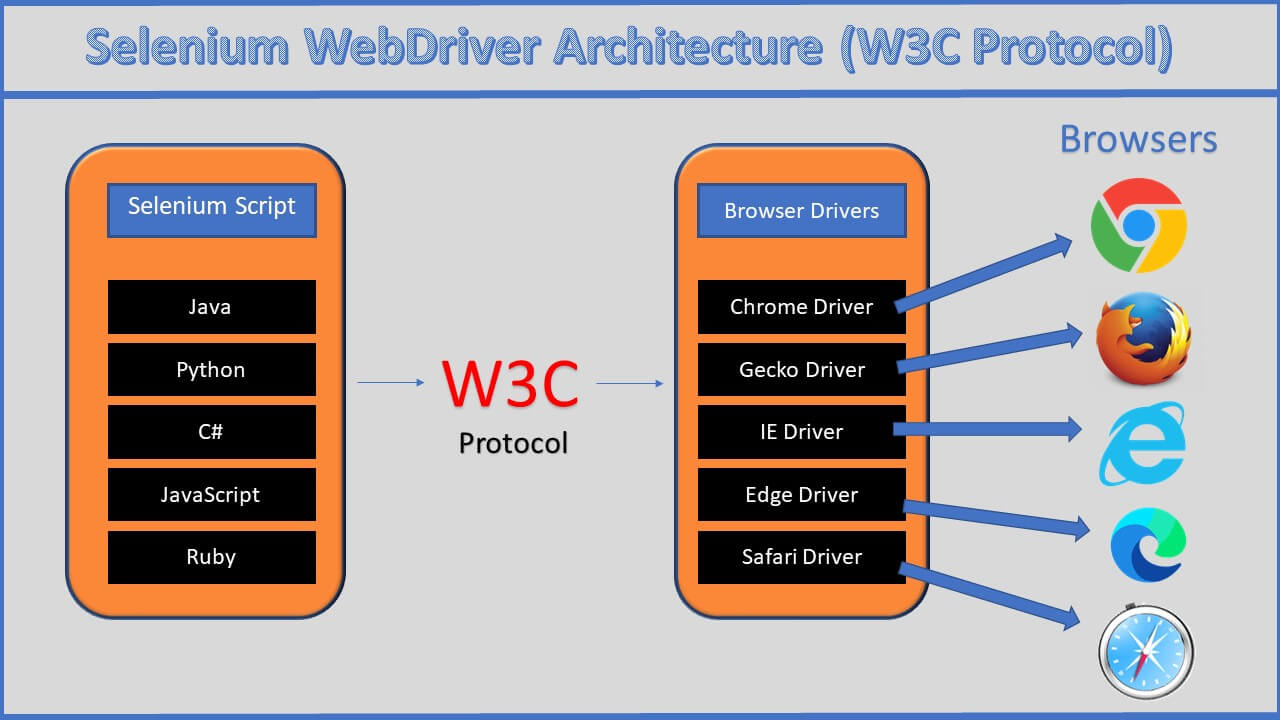
**Selenium Grid:** Enables parallel test execution across multiple machines and

browsers.

**Selenium Architecture**

Selenium WebDriver API provides communication facility between languages and browsers.

The following image shows the architectural representation of Selenium WebDriver.

****

### **Selenium Language Bindings / Selenium Client Libraries**

Selenium 4 supports various programming languages, and developers use Selenium client libraries to write automation scripts. Common languages include Java, Python, C#, Ruby, and JavaScript

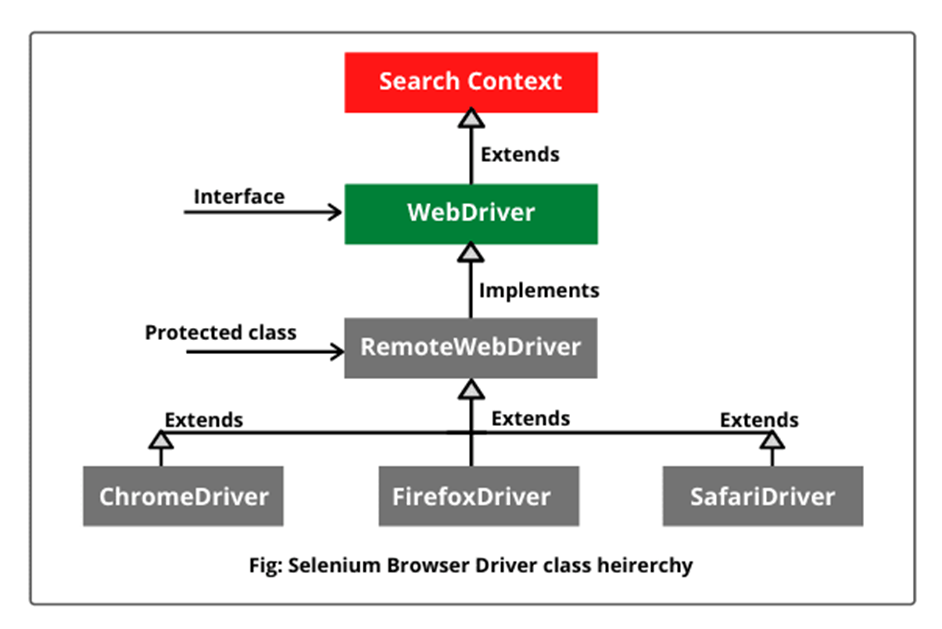
**W3C WebDriver Protocol**

The WebDriver protocol is the actual communication protocol used between the client (automation script or test code) and browser driver. It defines how commands are sent to the browser, how the browser processes those commands, and how the results are communicated back to the client

### **Browser Drivers**

Each browser (e.g., ChromeDriver, GeckoDriver, etc.) has its own driver that implements the WebDriver protocol. These drivers act as intermediaries between the Selenium scripts and the browsers, translating commands from the script into actions in the browser

**Hierarchy of Selenium Webdriver.**



**SearchContext -** is an Interface with 2 Abstract methods - FindElement() and FindElements().

**WebDriver -** is an interface which has 11+2 abstract methods.

**RemoteWebDriver -** is a protected class which is an implementation class for webdriver. It has 13 concrete methods.

Remote webdriver implements Webdriver.

**ChromeDriver -** A [WebDriver](https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/WebDriver.html) implementation that controls a Chrome browser running on the local machine.

It requires a chromedriver executable to be available in PATH**.**

**ChromeDriver extends RemoteWebDriver**

**Selenium Installation**

Installation of Selenium web driver, making sure all Environmental variables are set and installing selenium library in IDE.

**How to install Selenium -**

1) Downloaded Selenium Webdriver from Official Selenium Website.

2) Extracted Selenium files

3) In Eclipse, Properties of Project → Java Build path → Libraries —> Add External JARS→ Selected and opened all the Selenium files that we extracted.--> Apply and close.

4) Download Chromedriver (Make sure to download the latest version of chromedriver by checking against your chrome version / the chrome version used in your project)

5) We include the chromedriver file path in your script.

**First Selenium Script :**

public class FirstScript {

public static void main(String[] args) {

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

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

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

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

}

}

**System.setproperty** - It is used to enter the details of the driver(chromedriver,firefoxdriver,edgedriver) in a key value pair of Setproperty(key, value) key is the driver detail, value is the location of the driver.

**Web driver** is an **Interface**, driver is the reference variable, new is the keyword, ChromeDriver is the Constructor, together a new **ChromeDriver**() object is created and stored in the reference variable **driver**.

Here, All the driver methods are just declared in **Webdriver** Interface, and all the driver methods(get,close,quit,findelement etc) are implemented in Implementation class called **ChromeDriver**();

**Locators**

**I**n Automation , before performing any action such as click, clear, pass the data (send keys) we need to find the address of the element.

We do that by inspecting the element

To find the element, we use Locators

Locators are classified into 8 types.

1) ClassName.

2) CSS selector.

3) ID.

4) LinkText.

5) Name

6) PartialLinktext.

7) TagName

8) X path.

All of the Locators are static methods of By class.

By class is an abstract class.

Official info about Locators in **Selenium website** :

<https://www.selenium.dev/documentation/webdriver/elements/locators/>

### **What is a CSS Selector?**

CSS (Cascading Style Sheets) Selectors in Selenium are used to identify and locate web elements based on their id, class, name, attributes and other attributes.

CSS is a preferred locator strategy as it is simpler to write and faster as compared to XPath.

**By.cssSelector(String cssSelector)** method is used to locate the elements in Selenium WebDriver.

This method accepts a CSS Selector String as an argument which defines the selection method for the web elements.

**Types of CSS Selectors in Selenium (with Examples)**

There are five types of CSS Selectors in Selenium tests:

1. ID
2. Class
3. Attribute

#### **1. ID**

**In CSS, we can use “#” notation to select the “id” attribute of an element.**

**Syntax:**

<tagname>#<id\_value>

#<id\_value>

#### **2. Class**

In CSS, we can use “.” notation to select the “class” attribute of an element.

**Syntax:**

<tagname>.<class\_value>

.<class value>

#### **Attribute**

Apart from “**id**” and “**class**”, other attributes can also be used to locate web elements using CSS selector.

**Syntax:**

<tagname>[attribute\_name=’<attribute\_value>’]

**Xpath**

It is a path of the element in the HTML tree structure.

There are two kinds of Xpath -

1. **Absolute Xpath - only Forward slash** - Used to navigate from parent to immediate child tag.

Example – heroku Login Username xpath

/html/body/div[2]/div/div/form/div[1]/div/input

1. **Relative Xpath -**

**Double forward slash** - Used to travel directly to the specified tag.

We can also use combination of **Relative** and **Absolute** Xpaths.

Xpath Syntax - //Tagname[@key = ‘Value’];

**Xpath example** - **//input[@id='userName']**

**2.How to write Xpath with Multiple attributes.**

USing ‘**and’** operator

**Example** - //input[@class='nav-input nav-progressive-attribute' and @type="text"]

**3.How to use Text Function in Xpath -**

**Syntax :**

//tagname[text()=’Text Value’]

Example in Gmail SIgninPage :

//span[text()='Create account']

**4.How to write Xpath using Indexing -**

Syntax - **(//tag[@attribute=’Value’])[index]**

**Contains Function :**

**Syntax -**

//Tag[contains(text(),’textvalue’)]

Example -

(//div[@class="\_fluid-quad-image-label-v2\_style\_fluidQuadImageLabelBody\_\_3tld0"]//img)[3]

//a[contains(text(),'Grocery ')]

//\*[contains(text(),'Forgot')]

**5.Combination of Contains as well as Indexing in Xpath** (//\*[contains(text(),'akshay')])[2]

**How to use By.id in selenium script?**

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

**How to use By.name in selenium script?**

driver.findElement(By.name("checkBoxOption1"));

**Note :**

Whenever we give a incorrect Locator, we get **NoSuchElement Exception.**

**Day 2**

**13 Methods used from Webdriver Instance :**

1) close()

2) findElement()

3) findElements()

4) get()

5) getCurrentUrl()

6) getPageSource()

7) getTitle()

8) getWindowHandle()

9) getWindowHandles()

10) manage()

11) navigate()

12) quit()

13) switchTo()

**Web Element Commands: Edit Box, Button, Check box, Radio Button.**

**How to handle textbox :**

**Locate the Textbox/EditBox**

Use one of the available methods to locate the textbox element. Common methods include

**findElement(By.id())**,

**findElement(By.name())**,

**findElement(By.xpath())**

**Type into the Textbox using sendkeys() :**

Textbox.sendkeys(“Hello”)

**How to click on a button.**

First we have to Locate the Element , then we store it into a variable of WebElement

Then we click on it using click().

WebElement button = driver.findelement(By.id(“button\_id”));

Button.click();

**How to select a checkbox**

First we have to Locate the Element , then we store it into a variable of WebElement

Then we click on it using click().

WebElement checkbox = driver.findelement(By.id("checkbox\_id");

checkbox.click();

**How to select a RadioButton**

First we have to Locate the Element , then we store it into a variable of WebElement

Then we click on it using click().

WebElement radio = driver.findelement(By,id("radio\_id"));

radio.click();

### **Select in Selenium WebDriver**

How to Handle **Dropdowns**?

The 'Select' class in Selenium WebDriver is used for selecting and deselecting option in a dropdown.

The objects of Select type can be initialized by passing the dropdown webElement as parameter to its constructor.

WebElement DropDown = driver.findElement(By.id("testingDropdown"));

Select sel = **new** Select(DropDown);

WebDriver provides three ways to select an option from the drop-down menu.

**1.** **selectByIndex** - It is used to select an option based on its index, beginning with 0.

sel.selectByIndex(5);

**2. selectByValue** - It is used to select an option based on its 'value' attribute

sel.selectByValue("Database");

**3. selectByVisibleText** - It is used to select an option based on the text over the option.

sel.selectByVisibleText("Database Testing");

**Find Elements**

In Selenium with Java, the **findElements** method is used to locate multiple elements on a

web page that match the specified locator strategy.

This method returns a list of **WebElement** objects, allowing you to interact with each matching element individually.

In list – we can store heterogeneous data.

In list we cannot store Duplicate values.

Here's a breakdown of how findElements works:

List<WebElement> elements = driver.findElements(By.Tagname("locatorValue"));

**Handling Multiple Checkboxes**

First we need to store the checkboxes using findelements, then iterating

Over a for loop we can click on it.

**Code :**

List<WebElement>checkboxes=driver.findElements(By.cssSelector("input[type='checkbox']"); // Check all checkboxes

for (WebElement checkbox : checkboxes) {

checkbox.click();

}

**Handling Multiple Radio buttons.**

First we need to store the radio buttons using findelements, then iterating

Over a for loop we can click on it.

**Code :**

List<WebElement>radio=driver.findElements(By.cssSelector("input[type='checkbox']"); // select all radio buttons.

for (WebElement checkbox : checkboxes) {

radio.click();

}

The Actions class in Selenium with Java provides a way to perform complex user interactions, such as mouse and keyboard actions, on a web page. It is part of the org.openqa.selenium.interactions package. The Actions class is often used for performing actions like drag-and-drop, mouse hovering, key press/release, etc

### **Create an instance of the Actions class**

Actions actions = new Actions(driver);

**For Mouse Hover :**

WebElement elementToHover = driver.findElement(By.id("elementId"));

actions.moveToElement(elementToHover).build().perform();

**For Click :**

WebElement elementToClick = driver.findElement(By.id("elementId"));

actions.click(elementToClick).perform();

**For Double Click**

WebElement elementToDoubleClick = driver.findElement(By.id("elementId"));

actions.doubleClick(elementToDoubleClick).perform();

**For Right Click :**

WebElement elementToRightClick = driver.findElement(By.id("elementId"));

actions.contextClick(elementToRightClick).perform();

**For Drag and Drop :**

WebElement sourceElement = driver.findElement(By.id("sourceElementId"));

WebElement targetElement = driver.findElement(By.id("targetElementId"));

actions.dragAndDrop(sourceElement, targetElement).perform();

**Day - 3**

**Handling Pop ups**

Handling Alerts/pop-ups in Selenium WebDriver using Java involves using the Alert interface. Here's a basic guide :

First we have to switch to the alert using switchto() :

// Switch to the alert

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

// Perform actions on the alert

alert.accept(); // To accept (click OK)

// or

alert.dismiss(); // To dismiss (click Cancel)

// Get text from the alert

String alertText = alert.getText();

**Handling Prompt Pop ups :**

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

// Send data to the prompt

alert.sendKeys("YourInput");

// Accept or dismiss

alert.accept(); // To accept (click OK)

// or

alert.dismiss(); // To dismiss (click Cancel)

// Get text from the alert

String alertText = alert.getText();

## **What are Frames in Selenium?**

The term "frames" in Selenium refers to segmenting an HTML document into several portions, each of which can contain a separate HTML document. These frames are also referred to as iframes (inline frames). Frames are frequently used when a web page has numerous sections or pages that must be displayed simultaneously or interacted with independently.

We must first identify the frame to handle frames in selenium using WebDriver commands. This can be done in three ways index, by name or id, and by web elements. Selenium WebDriver has a few simple steps to handle frames:

* Switch the driver's focus to the frame using the switchTo().frame() method.
* Using web driver commands, interact with the elements of the frame and perform the operations.
* Switch back to the web content by the switchTo().defaultContent() method

### **By Index**

Use the switchTo().frame() method in Selenium WebDriver and supply the frame's index as an argument to switch to a frame by index. The index of frames on a page starts from 0, so the first frame is at index 0 the next frame at index one, and so on

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

### **By Name or ID**

You can use the name or ID as a parameter to switch to the frame using the switchTo().frame() method in Selenium WebDriver. It's important to make sure the frame name or ID is unique to the frame because there may be other frames on a page with the same name or ID

// Switch to the frame by name

driver.switchTo().frame("frame-name");

// Switch to the frame by ID

driver.switchTo().frame("frame-id");

### **By Web Element**

We can also switch to the frame by web elements. Pass the web element as an argument in the switchTo().frame() method to switch to the frame using web elements. Use any locator method to locate the frame element before providing it to the switchTo command.

// Find the frame element

WebElement frameElement = driver.findElement(By.id("frame-id"));

// Switch to the frame using the web element

driver.switchTo().frame(frameElement);

// Switch to the parent frame

driver.switchTo().frame("parent-frame");

## **Handling Nested Frames in Selenium WebDriver**

In Selenium WebDriver, nested frames mean a situation when a web page has numerous frames nested inside a frame. It means a frame also contains frames as its element. Moving the driver's focus to the appropriate frame is important to interact with the items inside a nested frame when using Selenium.

// Switch to the child frame

driver.switchTo().frame("child-frame");

// Interact with elements inside the child frame

WebElement childElement = driver.findElement(By.id("child-element"));

childElement.click();

// Switch back to the parent frame

driver.switchTo().parentFrame();

// Interact with elements inside the parent frame

WebElement parentElement = driver.findElement(By.id("parent-element"));

parentElement.click();

// Switch back to the default content

driver.switchTo().defaultContent();

# **Working with windows and tabs**

### **Get window handle**

WebDriver does not make the distinction between windows and tabs. If your site opens a new tab or window, Selenium will let you work with it using a window handle. Each window has a unique identifier which remains persistent in a single session. You can get the window handle of the current window by using:

driver.getWindowHandle();

How to get the window handle if Child tab?

We use a method called driver.getWindowHandles();

This will give us the window handles of all the windows currently present.

### Get the handle of the parent window using the command:

### String parentWindowHandle = driver.getWindowHandle();

### Print the window handle of the parent window.

### Find the element on the web page using an ID which is an element locator.

### Open multiple child windows.

### Iterate through child windows.

### Get the handles of all the windows that are currently open using the command:

### Set<String> allWindowHandles = driver.getWindowHandles();

### which returns the set of handles.

### Use the SwitchTo command to switch to the desired window and also pass the URL of the web page

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

System.setProperty("webdriver.chrome.driver","Path to the driver");

WebDriver driver = new ChromeDriver();

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

// Load the website

driver.get("http://www.naukri.com/");

// It will return the parent window name as a String

String parent=driver.getWindowHandle();

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

// Now iterate using Iterator

Iterator<String> I1= s.iterator();

while(I1.hasNext())

{

String child\_window=I1.next();

if(!parent.equals(child\_window))

{

driver.switchTo().window(child\_window);

System.out.println(driver.switchTo().window(child\_window).getTitle());

driver.close();

}

}

//switch to the parent window

driver.switchTo().window(parent);

}

**Day - 4**

**Wait commands**

**Why do we need waits?**

**Because of Synchronization issue.**

**Synchronization issue – When the Script execution speed doesn’t match with the browser speed.**

### **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.(Synchronization issues)

While running Selenium tests, it is common for testers to get the message “*Element Not Visible Exception*“, ”NoSuchElementException” ,“NoAlertPresentException”.

This appears when a particular web element with which WebDriver has to interact, is **delayed in its loading**. To prevent this Exception, Selenium Wait Commands must be used.

In automation testing, Selenium Webdriver wait commands direct test execution to **pause for a certain length of time** before moving onto the next step. This enables WebDriver to check if one or more web elements are present/visible/enriched/clickable, etc when identifying certain elements. If an element is not located, then the “ElementNotVisibleException” appears. Selenium Wait commands help resolve this issue.

Important point – When the element is actually there,enabled but there is a browser delay issue.

### **Implicit Wait in Selenium**

Implicit Wait directs the Selenium WebDriver to wait for a certain measure of time before throwing an exception. Once this time is set, WebDriver will wait for the element before the exception occurs.

It sets a global wait for a certain amount of time for the entire script. Selenium will wait for a specified amount of time before throwing a **NoSuchElementException** if an element is not present

Once the command is run, Implicit Wait remains for the entire duration for which the browser is open. It’s default setting is 0, and the specific wait time needs to be set by the following protocol.

**Implicit Wait Syntax**

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

### 

### **Explicit Wait in Selenium**

By using the Explicit Wait command, the WebDriver is directed to wait until a **certain condition occurs** before proceeding with executing the code.

This is a conditional Wait.

Setting Explicit Wait is important in cases where there are certain elements that naturally take more time to load. If one sets an implicit wait command, then the browser will wait for the same time frame before loading **every web element**. This causes an unnecessary delay in executing the test script.

Explicit wait is more intelligent, but can only be applied for specified elements. However, it is an improvement on implicit wait since it allows the program to pause for dynamically loaded Ajax elements.

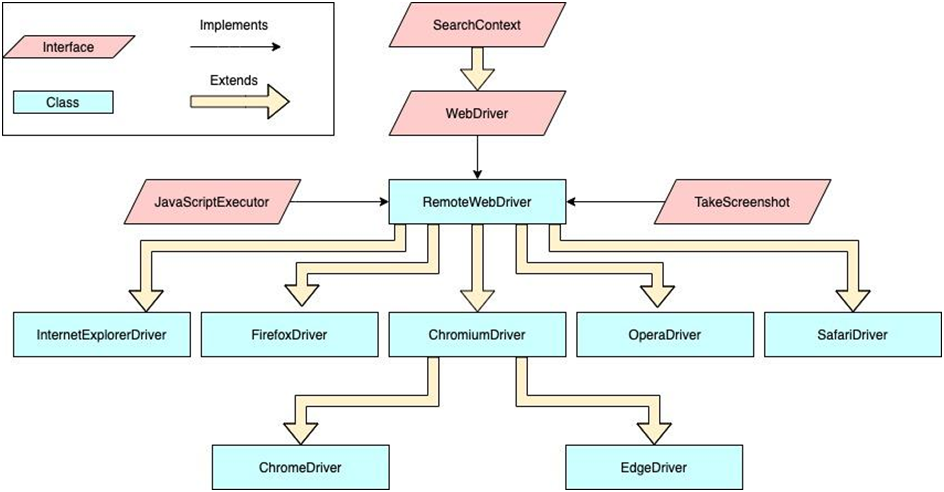
In order to declare explicit wait, one has to use ExpectedConditions. The following **Expected Conditions** can be used in Explicit Wait.

1. alertIsPresent()
2. elementSelectionStateToBe()
3. elementToBeClickable()
4. elementToBeSelected()
5. frameToBeAvailableAndSwitchToIt()
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()

**Explicit Wait Syntax**

WebDriverWait wait = new WebDriverWait(driver,30);

**JavaScriptExecutor**



In simple words, JavascriptExecutor is an interface that is used to execute JavaScript with Selenium.

To simplify the usage of JavascriptExecutor in Selenium, think of it as a medium that enables the WebDriver to interact with HTML elements within the browser.

JavaScript is a programming language that interacts with HTML in a browser, and to use this function in Selenium, JavascriptExecutor is required.

//importing the package

Import org.openqa.selenium.JavascriptExecutor;

//creating a reference

JavascriptExecutor js = (JavascriptExecutor) driver;

//calling the method

js.executeScript(script, args);

### **How JavascriptExecutor works in Selenium**

Let’s try to understand the working of JavascriptExecutor using a simple example and implementation of both the JavascriptExecutor methods.

**1.JavascriptExecutor in Selenium to click a button**

**js.executeScript(“document.getElementByID(‘element id ’).click();”);**

**2.JavascriptExecutor in Selenium to send text**

js.executeScript(“document.getElementByID(‘element id ’).value = ‘xyz’;”);

**3.JavascriptExecutor in Selenium to scroll down.**

js.executeScript("window.scrollBy(0,250)", "");

**JavascriptExecutor in Selenium to scroll down until the element is found into view**

// Scrolling down the page till the element is found

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

### **What are Broken Links?**

* To start with, a link is an HTML object that enables users to migrate from one web page to another when they click on it. It is a means to navigate between different web pages on the internet.
* A broken link, also often called a **dead** **link**, is one that does not work i.e. does not redirect to the webpage it is meant to.
* This usually occurs because the website or particular web page is **down or does not exist**. When someone clicks on a broken link, an error message is displayed.

### **Common Reasons for Broken Links**

* 404 Page Not Found – The destination web page has been removed by the owner
* 400 Bad Request – The server cannot process the HTTP request triggered by the link because the URL address requested is wrong
* Due to the user’s firewall settings, the browser cannot access the destination web page.
* The link is misspelt

### **How to identify broken links in Selenium WebDriver**

**Use the following steps to identify broken links in Selenium**

1. Use <a> tag to fetch all the links present on a web page
2. Send HTTP request for the link
3. Verify the HTTP response code for the link
4. Determine if the link is valid or it is broken based on the HTTP response code.
5. Repeat the process for all links captured with the first step

// Finding all the available links on webpage

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

// Iterating each link and checking the response status

for (WebElement link : links) {

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

verifyLink(url);

}

**public static void verifyLink(String url) {**

try {

URL link = new URL(url);

HttpURLConnection httpURLConnection = (HttpURLConnection) link.openConnection();

httpURLConnection.setConnectTimeout(3000); // Set connection timeout to 3 seconds

httpURLConnection.connect();

if (httpURLConnection.getResponseCode() == 200) {

System.out.println(url + " - " + httpURLConnection.getResponseMessage());

} else {

System.out.println(url + " - " + httpURLConnection.getResponseMessage() + " - " + "is a broken link");

}

} catch (Exception e) {

System.out.println(url + " - " + "is a broken link");

}

}

### 

### **How to take screenshot in Selenium WebDriver?**

To capture screenshots in Selenium, one has to utilise the Interface **TakesScreenshot** and use **getScreenshotAs**() This notifies WebDriver that it should take a screenshot in Selenium and store it.

**Let’s explore this function with a three-step process –**

**Step 1 – Convert web driver object to TakeScreenshot instance**

TakesScreenshot scrShot =((TakesScreenshot)webdriver);

**Step 2 – Call getScreenshotAs method to create image file**

File SrcFile=scrShot.getScreenshotAs(OutputType.FILE);

**Step 3 – Copy file to Desired Location**

FileUtils.copyFile(SrcFile, DestFile);

## **How to Upload File in Selenium**

File upload is performed when there is a need of uploading any file or a document on a specific website such as forms, registration pages, document uploaders, etc.

Uploading a file process includes browsing a file from the desired location or from your computer and uploading it to the website

WebElement uploadElement=driver.findElement(By.id("uploadfile\_0"));

// enter the file path onto the file-selection input field

uploadElement.sendKeys("C:\\newhtml.html");

// click the "UploadFile" button

driver.findElement(By.name("send")).click()

**Remember following two things when uploading files in WebDriver**

1. There is no need to simulate the clicking of the “Choose File” button. WebDriver automatically enters the file path onto the file-selection text box of the <input type=”file”> element
2. When setting the file path in your Java IDE, use the proper escape character for the back-slash.

;

**Download file in Selenium**

**Locate the download link:**

Find the element representing the link to the file you want to download. You can use various locators, such as By.id, By.xpath, etc

WebElement downloadLink = driver.findElement(By.id("download-link"));

**Simulate a click on the link:**

Click the download link to initiate the file download:

downloadLink.click();

**Day – 5**

**Introduction to Data Driven Testing using Apache POI & implementation.**

Data-driven testing in Selenium using **Apache POI** involves reading test data from external Excel files, enhancing test robustness and flexibility by **separating** data from the test scripts.

Data-driven testing using Apache POI (Poor Obfuscation Implementation) in Java Selenium allows you to read test data from external sources like Excel files.

Add Apache POI dependencies:

Include the Apache POI dependencies in your project. You can add these dependencies to your Maven pom.xml

For Maven:

<dependency>

<groupId>org.apache.poi</groupId>

<artifactId>poi</artifactId>

<version>5.0.0</version>

</dependency>

<dependency>

<groupId>org.apache.poi</groupId>

<artifactId>poi-ooxml</artifactId>

<version>5.0.0</version>

</dependency>

**Components :**

1. External Excel File.
2. Apache POI jars.
3. Test Script – java Selenium script.

**Create an Excel file with test data:**

Create an Excel file with test data. For example, let's say you have a sheet named "TestData" with columns "Username" and "Password."

Read data from Excel using Apache POI:

**Create a method to read data from the Excel file. Here's a simple example:**

* import org.apache.poi.ss.usermodel.\*;
* public class ExcelUtils {
* public static Object[][] getTestData(String filePath, String sheetName) {
* Object[][] testData = null;
* try {
* Workbook workbook = WorkbookFactory.create(new FileInputStream(filePath));
* Sheet sheet = workbook.getSheet(sheetName);
* int rowCount = sheet.getPhysicalNumberOfRows();
* int colCount = sheet.getRow(0).getPhysicalNumberOfCells();
* testData = new Object[rowCount - 1][colCount]; // Exclude header row
* for (int i = 1; i < rowCount; i++) {
* Row row = sheet.getRow(i);
* for (int j = 0; j < colCount; j++) {
* Cell cell = row.getCell(j);
* testData[i - 1][j] = cell.toString();
* }
* }
* } catch (IOException e) {
* e.printStackTrace();
* }
* return testData;
* }
* }

**Use the test data in your Selenium tests:**

In your test script, use the data obtained from the Excel file:

import org.testng.annotations.DataProvider;

import org.testng.annotations.Test;

public class YourTestScript {

@Test(dataProvider = "getData")

public void yourTest(String username, String password) {

// Your test logic using username and password

System.out.println("Username: " + username);

System.out.println("Password: " + password);

}

@DataProvider

public Object[][] getData() {

String filePath = "path/to/your/testdata.xlsx";

String sheetName = "TestData";

return ExcelUtils.getTestData(filePath, sheetName);

}

}

Junit – Installation Steps –

1. Add junit dependency into your pom.xml.

**What is Junit ?**

Junit is a Unit Testing framework available in java used to write unit test scripts by developers.

**Who uses Junit?**

Developers writing Unit Test cases will make use of Junit to write and Execute Unit Test cases.

Also Test engineers make use of Junit To write Test Scripts.

**Introduction to TestNG annotations**

TestNG (Test Next Generation) is a testing framework for Java that is widely used in Selenium automation testing. It provides powerful features for test configuration, parallel execution, grouping, and reporting.

**Installation steps of TestNG:**

Goto Eclipse window

Click on help option-> Eclipse Marketplace

1. Write TestNG in find edit box and click on go button.
2. Find TestNG for eclipse division and click on install button
3. Click on confirm button and I accept the terms and conditions and click on finish

To use TestNG with Selenium, you need to add the TestNG library to your project. If you're using Maven, add the following dependency to your pom.xml file:

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>7.4.0</version> <!-- Use the latest version available -->

<scope>test</scope>

</dependency>

**TestNG Annotations:**

TestNG uses annotations to define the behaviour of test methods. Some commonly used annotations in Selenium automation are:

@**Test**: Marks a method as a test method.

@**BeforeMethod**: Executes before each test method.

@**AfterMethod**: Executes after each test method.

@**BeforeClass**: Executes once before the test class.

@**AfterClass**: Executes once after the test class.

@**BeforeTest**: Executes before a test (a group of classes).

The @BeforeTest annotation is used to designate a method that should run before any test method belonging to a <test> tag in the XML suite file.

@**AfterTest**: Executes after a test (a group of classes).

@**dataprovider** – Helps us to run a single testcase with Multiple sets of data.

**Difference between Junit & TestNG.**

JUnit and TestNG are both popular testing frameworks for Java, and they share similarities in terms of functionality, but they also have some differences. Here's a comparison between JUnit and TestNG:

**1.Annotation Support:**

**JUnit**: JUnit 4 uses annotations such as @Test, @Before, @After, etc.

**TestNG**: TestNG provides a broader set of annotations, including @Test, @BeforeMethod, @AfterMethod, @BeforeClass, @AfterClass, @BeforeTest, @AfterTest, etc. TestNG's annotations offer more fine-grained control over test execution.

**2.Parameterization:**

**JUnit**: JUnit supports parameterized tests using the @Parameterized annotation.

**TestNG**: TestNG has built-in support for parameterization using the @DataProvider annotation, making it easier to run tests with different sets of data.

**3.Dependencies:**

**JUnit**: JUnit does not have built-in support for test dependencies. Developers often use @Before and @After annotations for setup and teardown.

**TestNG**: TestNG provides a dependsOnMethods attribute that allows you to specify test dependencies explicitly, ensuring the order of test execution.

**4.Groups and Prioritization:**

JUnit: JUnit doesn't have built-in support for test groups or prioritization. Developers often use naming conventions or custom runners for grouping.

**TestNG**: TestNG allows you to group tests using the **groups** attribute in annotations. It also supports test **prioritization** using the priority attribute.

**5.Parallel Execution:**

**JUnit**: JUnit does not provide native support for parallel execution. Developers often rely on external tools for parallel test execution.

Parallel Execution – Execution of Multiple test scripts at once wrt different browsers/ platform

**TestNG**: TestNG has built-in support for parallel test execution at various levels (methods, classes, suites), making it easy to parallelize test runs.

**6.Suite Configuration:**

**JUnit**: JUnit relies on test suite classes or test runners for suite-level configuration.

**TestNG**: TestNG allows you to define suite-level configurations in XML files, providing more flexibility in setting up test environments.

**7.Listeners:**

**JUnit**: JUnit has limited support for test listeners.

**TestNG**: TestNG provides a rich set of listeners that allow you to customise test execution and report generation.

Listeners – With the help of Listeners we can rerun some failed test scripts and also used for screenshot implementation.

**8.Reporting:**

**JUnit**: JUnit's reporting capabilities are basic, and developers often use additional tools for more detailed reports.

**TestNG**: TestNG generates detailed HTML reports by default, providing comprehensive information about test results.

In summary, while both JUnit and TestNG are powerful testing frameworks, TestNG often offers more features out of the box and provides additional capabilities for test configuration, parameterization, and parallel execution.