

# Selenium

## What is Selenium?

Selenium is a free (open-source) automated testing framework used to validate web applications across different browsers and platforms.

You can use multiple programming languages like Java, C#, Python etc to create Selenium Test Scripts.

## Component of selenium

- Selenium IDE
- Selenium RC (Remote Control)
- Selenium WebDriver
- Selenium Grid

## Selenium Grid

Selenium Grid is a tool **used together with Selenium RC to run parallel tests** across different machines and different browsers all at the same time. Parallel execution means running multiple tests at once.

### Features:

- Enables **simultaneous running of tests** in **multiple browsers and environments**.
- **Saves time** enormously.
- Utilizes the **hub-and-nodes** concept. The hub acts as a central source of Selenium commands to each node connected to it.

## Selenium RC

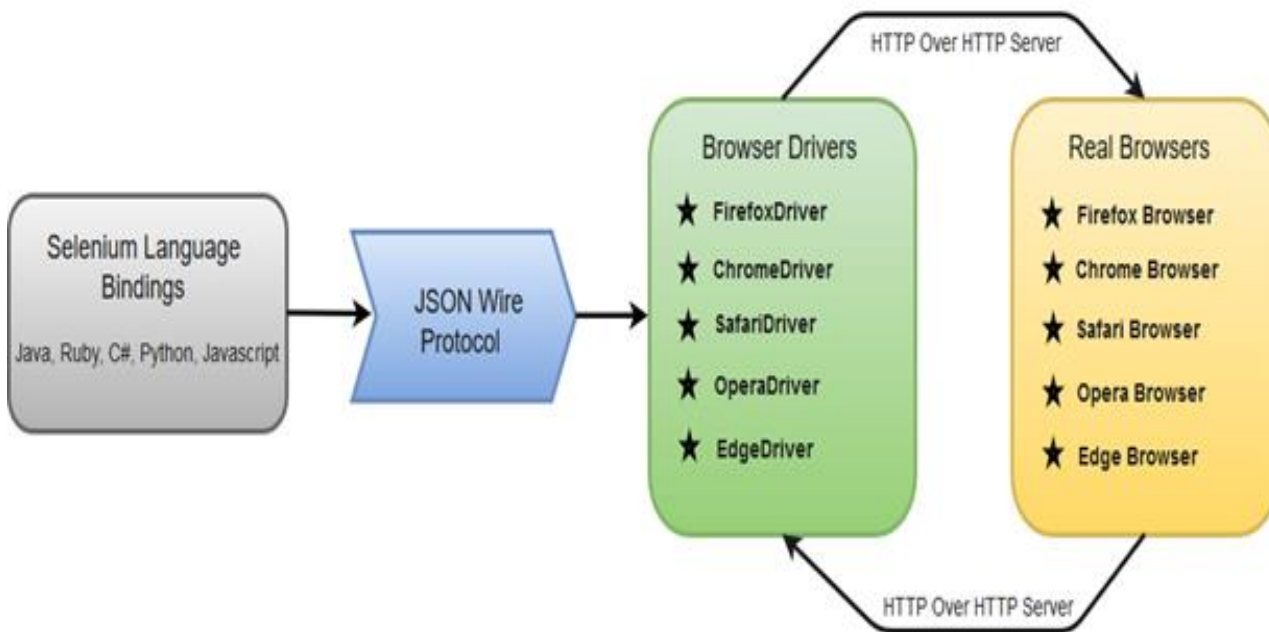
- To design a test using a more expressive language than Selenese
- To run your test against different browsers (except HtmlUnit) on different operating systems.
- To deploy your tests across multiple environments using Selenium Grid.
- To test your application against a new browser that supports JavaScript.
- To test web applications with complex AJAX-based scenarios.

## What is Selenium Webdriver?

Selenium Webdriver is an open-source collection of APIs which is used for testing web applications. The Selenium Webdriver tool is used for automating web application testing to verify that it works as expected or not. It mainly supports browsers like Firefox, Chrome, Safari and Internet Explorer. It also permits you to execute cross-browser testing.

## Selenium WebDriver- Architecture

Selenium WebDriver API provides communication facility between languages and browsers. The following image shows the architectural representation of Selenium WebDriver.



There are four basic components of WebDriver Architecture:

- Selenium Language Bindings
- JSON Wire Protocol
- Browser Drivers
- Real Browsers

### Selenium Language Bindings / Selenium Client Libraries

Selenium developers have built language bindings/Selenium Client Libraries in order to support multiple languages. For instance, if you want to use the browser driver in java, Ruby, Python, use the java bindings.

### JSON Wire Protocol

JSON stands for JavaScript Object Notation. It takes up the task of transferring information from the server to the client. JSON Wire Protocol is primarily responsible for transfer of data between HTTP servers. Generated Json is made available to browser drivers through http Protocol.

## Browser Drivers

Selenium uses drivers, specific to each browser in order to establish a secure connection with the browser without revealing the internal logic of browser's functionality. The browser driver is also specific to the language used for automation such as Java, C#, Ruby, Python, etc.

When we execute a test script using WebDriver, the following operations are performed internally.

- HTTP request is generated and sent to the browser driver for each Selenium command.
- The driver receives the HTTP request through HTTP server.
- HTTP Server decides all the steps to perform instructions which are executed on browser.
- Execution status is sent back to HTTP Server which is subsequently sent back to automation script.

## Real Browsers

Browsers supported by Selenium WebDriver:

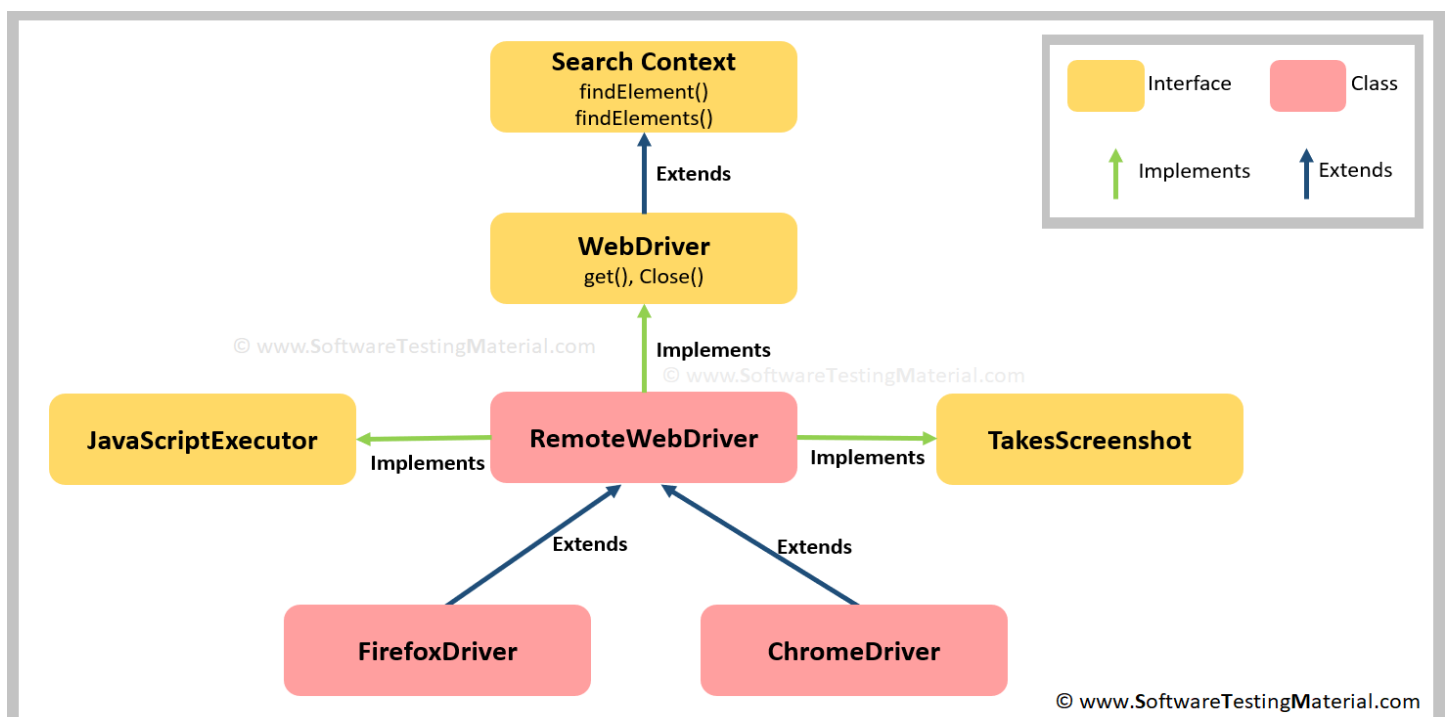
- Internet Explorer
- Mozilla Firefox
- Google Chrome
- Safari

## Q. `WebDriver driver = new FirefoxDriver ()` – Why we write in Selenium Scripts

We are going to discuss following points in this post.

1. Why not `WebDriver driver = new WebDriver ()`
2. Why we won't prefer `FirefoxDriver driver = new FirefoxDriver ()`
3. Why `WebDriver driver = new FirefoxDriver ()`
4. What is `RemoteWebDriver` and where we use it

Before going into the details let's see the below image.



Following are the few points based on the above image.

- Search Context is the super most interface in selenium, which is extended by another interface called WebDriver.
- All the abstract methods of Search Context and WebDriver interfaces are implemented in RemoteWebDriver class.
- All the browser related classes such as FirefoxDriver, ChromeDriver etc., extends the RemoteWebDriver class.

WebDriver defines common methods which all browser classes (such as Firefox, Chrome etc.,) use. All these class methods are derived from WebDriver interface.

All the abstract methods of both the interfaces are implemented in RemoteWebDriver class which is extended by browser classes such as Firefox Driver, Chrome Driver etc.

Let's see why can't we use the following statement.

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

We cannot write our code like this because we cannot create Object of an Interface. WebDriver is an interface.

But we can use any of the following statements in our script

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

or

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

Let's see both of them in detail.

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

The FirefoxDriver instance which gets created based on above statement will be only able to invoke and act on the methods implemented by FirefoxDriver and supported by Firefox Browser only. We know that FirefoxDriver is a class and it implements all the methods of WebDriver interface. Using this statement, we can run our scripts only on Firefox Browser.

To act with other browsers, we have to specifically create individual objects as below:

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

**InternetExplorerDriver driver = new InternetExplorerDriver ();**

We don't just run our scripts only on single browser. We use multiple browsers for Cross Browser Compatibility. We need the flexibility to use other browsers like ChromeDriver () to run on Chrome Browser and InternetExplorerDriver () to run on IE Browser and so on.

So, once you initiate a Firefox browser using FirefoxDriver driver = new FirefoxDriver (); same object cannot be used to initiate Chrome Browser (you have to rename it)

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

To solve this, we use "WebDriver driver = new FirefoxDriver ();"

Let's see this now.

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

We can create Object of a class FirefoxDriver by taking reference of an interface (WebDriver). In this case, we can call implemented methods of WebDriver interface.

As per the above statement, we are creating an instance of the WebDriver interface and casting it to FirefoxDriver Class.

All other Browser Drivers like ChromeDriver, InternetExplorerDriver, PhantomJSDriver, SafariDriver etc. implemented the WebDriver interface (actually the RemoteWebDriver class implements WebDriver Interface and the Browser Drivers extends RemoteWebDriver). Based on this statement, you can assign Firefox driver and run the script in Firefox browser (any browser depends on your choice).

If you define driver as a WebDriver, switching will be very easy. If we use this statement in our script then the WebDriver driver can implement any browser. Every browser driver class implements WebDriver interface and we can get all the methods. It helps you when you do test on multiple browsers.

Example:

```
WebDriver driver = new FirefoxDriver (); driver. quit (); driver = new ChromeDriver ();
```

WebDriver is an interface and all the methods which are declared in WebDriver interface are implemented by respective driver class. But if we do upcasting, we can run the scripts in any browser. i.e. running the same automation scripts in different browsers to achieve Runtime Polymorphism.

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

**Here, WebDriver is an interface, driver is a reference variable, FirefoxDriver () is a Constructor, new is a keyword, and new FirefoxDriver () is an Object.**

General information: Selenium WebDriver is an Interface which contains different methods (e.g., get (), getTitle (), close () etc.,).

All the third-party Browser vendors implement these methods in addition to their browser specific methods. This would in-turn help the end-users to use the exposed APIs to write a common code and implement the functionalities across all the available Browsers without any change. Selenium developers don't know how all these browsers work. So, Selenium developers just declare methods whatever they required and leave the implementation part to the browser developers.

#### **Difference between WebDriver and RemoteWebDriver:**

As per Java's interface concept, Interface contains only Method's signature and it doesn't contain the Method's definitions.

Check this post to know the difference between Method Declaration and Method Definition.

WebDriver is the Interface which contains all the Selenium Method signatures (E.g. findElement (), switchTo (), get () etc.) whereas definition for those methods are in RemoteWebDriver Class.

If we want to run our automation scripts on the local machine's browser then we can use any class (such as FirefoxDriver, IEDriver, ChromeDriver, HtmlUnitDriver) except RemoteWebDriver. WebDriver will start up a web browser on the computer where the code instantiates it.

If we want to run our automation scripts on the remote machine's browser then we use RemoteWebDriver. RemoteWebDriver requires the Selenium Standalone Server to be running but to use another drivers Selenium Standalone Server is not required.

If you want to work with Grid then you have to stick with RemoteWebDriver. The only requirement is that for a RemoteWebDriver to work, you would always have to have it pointing to the URL of a Grid.

If you are using any of the drivers other than RemoteWebDriver then the communication will happen on the local machine's browser.

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

driver will access Firefox on the local machine, directly.

If we use RemoteWebDriver then we have to mention where the Selenium Server is located and which web browser you want to use.

**Ex: WebDriver driver = new RemoteWebDriver (new URL Desired capabilities. Firefox ());**

We can use RemoteWebDriver the same way we would use WebDriver locally. The primary difference is that remote webdriver needs to be configured so that it can run your tests on a remote machine.

WebDriver is an interface that you should be using throughout your tests. RemoteWebDriver is a concrete implementation of that interface. In general, it's always a good idea to code against interfaces wherever possible.

## Locators in Selenium IDE: CSS Selector, DOM, XPath, Link Text, ID

### What are Locators?

Locator is a command that tells Selenium IDE which GUI elements (say Text Box, Buttons, Check Boxes etc) it needs to operate on. Identification of correct GUI elements is a prerequisite to creating an automation script. But accurate identification of GUI elements is more difficult than it sounds. Sometimes, you end up working with incorrect GUI elements or no elements at all! Hence, Selenium provides a number of Locators to precisely locate a GUI element.

The different types of Locators in Selenium IDE, “By” is abstract class and below are their static methods

- By.className
- By.cssSelector
- By.id
- By.name
- By.linkText
- By.partialLinkText
- By.tagName
- By.xpath

### Locating by ID and Name

This is the most common way of locating elements since ID's are supposed to be unique for each element.

```
driver.findElement(By.Id("Unique Id"))
```

```
driver.findElement(By.name("Unique Name"))
```

### XPath

In Selenium automation, if the elements are not found by the general locators like id, class, name, etc. then XPath is used to find an element on the web page.

There are two types of XPath

- **Absolute XPath**
- **Relative XPath**

## Absolute XPath

It is the direct way to find the element, but the disadvantage of the absolute XPath is that if there are any changes made in the path of the element then that XPath gets failed.

The key characteristic of XPath is that it begins with the single forward slash (/), which means you can select the element from the root node.

```
Ex: /html/body/div[2]/div[1]/div/h4[1]/b/html[1]/body[1]/div[2]/div[1]/div[1]/h4[1]/b[1]
```

## Relative Xpath

Relative Xpath starts from the middle of HTML DOM structure. It starts with double forward slash (//). It can search elements anywhere on the webpage, means no need to write a long xpath and you can start from the middle of HTML DOM structure. Relative Xpath is always preferred as it is not a complete path from the root element.

```
Ex: //div[@class='featured-box cloumnsze1']//h4[1]//b[1]
```

## What are XPath axes.

XPath axes search different nodes in XML document from current context node. XPath Axes are the methods used to find dynamic elements, which otherwise not possible by normal XPath method having no ID, Classname, Name, etc.

Axes methods are used to find those elements, which dynamically change on refresh or any other operations. There are few axes methods commonly used in Selenium Webdriver like child, parent, ancestor, sibling, preceding, self, etc.

### 1) Basic XPath:

XPath expression select nodes or list of nodes on the basis of attributes like **ID, Name, Classname**, etc. from the XML document as illustrated below.

```
Xpath=//input[@name='uid']  
Xpath=//input[@type='text']  
Xpath= //label[@id='message23']  
Xpath= //input[@value='RESET']
```

### 2) Contains ():

Contains () is a method used in XPath expression. It is used when the value of any attribute changes dynamically, for example, login information.

we tried to identify the element by just using partial text value of the attribute.

In the below XPath expression partial value 'sub' is used in place of submit button.

Complete value of 'Type' is 'submit' but using only partial value 'sub'.

```
Xpath=//*[contains(@type,'sub')]
```

Complete value of 'name' is 'btnLogin' but using only partial value 'btn'.

```
Xpath=//*[contains(@name,'btn')]
```

### 3) Using OR & AND:

In OR expression, two conditions are used, whether 1st condition OR 2nd condition should be true.

```
Xpath=//*[@type='submit' or @name='btnReset']
```

**In AND expression**, two conditions are used, both conditions should be true to find the element. It fails to find element if any one condition is false

```
Xpath=//input[@type='submit' and @name='btnLogin']
```

### 4) Xpath Starts-with

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

In this method, the starting text of the attribute is matched to find the element whose attribute value changes dynamically.

You can also find elements whose attribute value is static (not changes).

**For example -:** Suppose the ID of particular element changes dynamically like:

```
Id=" message12"
```

```
Id=" message345"
```

```
Id=" message8769"
```

```
Xpath=//label[starts-with(@id,'message')]
```

### 5) XPath Text () Function

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

```
Xpath=//td[text()='UserID']
```

### 6) XPath axes methods:

#### a) Following:

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

#### b) Ancestor:

The ancestor axis selects all ancestors element (grandparent, parent, etc.) of the current node

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

#### c) Child:

```
Xpath=//*[@id='java_technologies']//child::li
```

#### d) Preceding:

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

**e) Following-sibling:** Select the following siblings of the context node. Siblings are at the same level of the current node It will find the element after the current node.

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



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

**g) Self:**

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

**h) Descendant:**

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

## Find Element and FindElements in Selenium WebDriver

### Find Element (Abstract Method)

- Returns the first most web element if there are multiple web elements found with the same locator.
- Throws exception NoSuchElementException if there are no elements matching the locator strategy.
- It will only find one web element.
- Not Applicable.

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

### Find Elements (Abstract Method)

- Returns a list of web elements.
- Returns an empty list if there are no web elements matching the locator strategy.
- It will find a collection of elements whose match the locator strategy.
- Each Web element is indexed with a number starting from 0 just like an array

```
List<WebElement> listOfElements = driver.  
findElements(By.xpath("//div"));
```

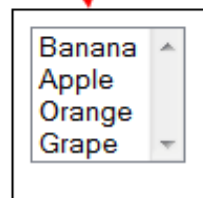
## How to Select Value from Drop Down using Selenium Webdriver.

The **Select Class in Selenium** is a method used to implement the HTML SELECT tag. The html select tag provides helper methods to select and deselect the elements. The Select class is an ordinary class so New keyword is used to create its object and it specifies the web element location.

page source

```
<select id="fruits" multiple="">  
  <option value="banana">Banana</option>  
  <option value="apple">Apple</option>  
  <option value="orange">Orange</option>  
  <option value="grape">Grape</option>  
</select>
```

HTML page



<pre>Select drpCountry = new Select(webElement); drpCountry. selectByIndex(0) drpCountry.selectByValue("apple"); drpCountry.selectByVisibleText("Grape");</pre>	Output: Banana Apple Grape
<pre>drpCountry .deselectAll()</pre>	Clears all selected entries. This is only valid when the drop-down element supports multiple selections
<pre>drpCountry..isMultiple() ;</pre>	Returns TRUE if the drop-down element allows multiple selections at a time; FALSE if otherwise

## Mouse Click & Keyboard Event: Action Class in Selenium Webdriver

### Action Class in Selenium

Action Class in Selenium is a built-in feature provided by the selenium for handling keyboard and mouse events. It includes various operations such as multiple events clicking by control key, drag and drop events and many more.

### Syntax:

```
Actions action = new Actions(driver);
```

To focus on element using WebDriver:

```
action.moveToElement(element).build().perform()
```

**click()** method is used here to click the element.

```
action.moveToElement(element).click().perform();
```

Instead of above **Click** method place the below any method.

**doubleClick ()**: Double clicks on Element

**contextClick()** : Performs a context-click (right click) on an element

```
action.moveToElement(element).contextClick().build().perform();
action.contextClick(element).build().perform();
```

**clickAndHold()**: Clicks at the present mouse location (without releasing)

**dragAndDrop(source, target)**:

Invokes click-and-hold at the source location and moves to the location of the target element before releasing the mouse.

source – element to grab,

target – element to release

**dragAndDropBy(source,xOffset,yOffset)** :

Performs click-and-hold at the source location, shifts by a given offset, then frees the mouse. xOffset – to shift horizontally, yOffset – to shift vertically

**moveByOffset(x-offset, y-offset):** Shifts the mouse from its current position (or 0,0) by the given offset. x-offset – Sets the horizontal offset (negative value – shifting the mouse to the left), y-offset – Sets the vertical offset (negative value – shifting the mouse to the up)

**moveToElement(toElement):**

It shifts the mouse to the center of the element

**release():** Releases the depressed left mouse button at the existing mouse location

keyUp(modifier \_key) Performs a key release.

modifier\_key - any of the modifier keys (Keys.ALT, Keys.SHIFT, or Keys.CONTROL)

```
public static void main(String[] args) {
    String baseUrl = "http://www.facebook.com/";
    WebDriver driver = new FirefoxDriver();

    driver.get(baseUrl);
    WebElement txtUsername = driver.findElement(By.id("email"));

    Actions builder = new Actions(driver);
    Action seriesOfActions = builder
        .moveToElement(txtUsername)
        .click()
        .keyDown(txtUsername, Keys.SHIFT)
        .sendKeys(txtUsername, "hello")
        .keyUp(txtUsername, Keys.SHIFT)
        .doubleClick(txtUsername)
        .contextClick()
        .build();

    seriesOfActions.perform();
}
```

this will type "hello" in uppercase

this will highlight the text "HELLO"

this will bring up the context menu

## Alert & Popup Window Handling in Selenium WebDriver

### What is Alert?

Alert is a small message box which displays on-screen notification to give the user some kind of information or ask for permission to perform certain kind of operation. It may be also used for warning purpose.

#### 1) Simple Alert

This simple alert displays some information or warning on the screen.

#### 2) Prompt Alert.

This Prompt Alert asks some input from the user and selenium webdriver can enter the text using sendkeys(" input.... ").

#### 3) Confirmation Alert.

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

### How to handle Alert in Selenium WebDriver

**Alert interface** provides the below few methods which are widely used in Selenium Webdriver.

1) void dismiss() // To click on the 'Cancel' button of the alert.

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

2) void accept() // To click on the 'OK' button of the alert.

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

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

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

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

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

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

when we have multiple windows in any web application, the activity may need to switch control among several windows from one to other in order to complete the operation. After completion of the operation, it has to return to the main window i.e. parent window.

### **Driver.getWindowHandles();**

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

### **Driver.getWindowHandle();**

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

```
String MainWindow = driver.getWindowHandle();
// To handle all new opened window.
Set<String> s1 = driver.getWindowHandles();
Iterator<String> i1 = s1.iterator();
while (i1.hasNext()) {
    String ChildWindow = i1.next();
    if (!MainWindow.equalsIgnoreCase(ChildWindow)) {
        // Switching to Child window
        driver.switchTo().window(ChildWindow);
        driver.findElement(By.name("emailid")).sendKeys("gaurav.3n@gmail.com");
        driver.findElement(By.name("btnLogin")).click();
        // Closing the Child Window.
        driver.close();
    }
}
// Switching to Parent window i.e Main Window.
driver.switchTo().window(MainWindow);
```

# Handling iFrames in Selenium Webdriver: switchTo()

## iFrame in Selenium Webdriver

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

We can identify the iframes using methods given below:

- Right click on the element, If you find the option like 'This Frame' then it is an iframe.
- Right click on the page and click 'View Page Source' and Search with the 'iframe', if you can find any tag name with the 'iframe' then it is meaning to say the page consisting an iframe.

We can even identify total number of iframes by using below snippet.

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

## How to switch over the elements in iframes using Web Driver commands:

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

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

### Switch to the frame by index:

Index is one of the attributes for the Iframe through which we can switch to it.

Index of the iframe starts with '0'.

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

- `driver.switchTo().frame(0);`
- `driver.switchTo().frame(1);`
- 

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

Name and ID are attributes of iframe through which we can switch to the it.

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

## How to switch back to the Main Frame

- We have to come out of the iframe.
- To move back to the parent frame, you can either use `switchTo().parentFrame()` or if you want to get back to the main (or most parent) frame, you can use `switchTo().defaultContent();`
- `driver.switchTo().parentFrame();`
- `driver.switchTo().defaultContent();`

## Take Screenshot in Selenium WebDriver

Screenshots are desirable for bug analysis. Selenium can automatically take screenshots during execution. You need to type cast WebDriver instance to TakesScreenshot.

```
TakesScreenshot scrShot=((TakesScreenshot)driver); //Convert web driver object to TakeScreenshot File
File SrcFile=scrShot.getScreenshotAs(OutputType.FILE); //Call getScreenshotAs method to create image file
FileUtils.copyFile(SrcFile, DestFilePath); //Copy file at destination
```

## JavaScriptExecutor in Selenium WebDriver

- Sometimes web controls don't react well against selenium commands and we may face issues with the above statement (click()). To overcome such kind of situation, we use JavaScriptExecutor interface.
- JavaScriptExecutor is an Interface that helps to executor java Script through Selenium Webdriver. JavaScriptExecutor provides two methods "executeScript" & "executeAsyncScript" to run javascript on the selected window or current page.

```
//Creating the JavascriptExecutor interface object by Type casting
JavascriptExecutor js = (JavascriptExecutor)driver;
// Perform Click on LOGIN button using JavascriptExecutor
js.executeScript("arguments[0].click();", WebElement);
```

**Implicit Wait:** During Implicit wait if the Web Driver cannot find Web Element immediately because of its availability, it will keep polling (around 4 seconds) to get the element. If the element is not available within the specified Time, then exception NoSuchElementException will be raised(occur). The default setting is zero.

Once we set a time, the Web Driver waits for the period of the WebDriver object instance.

```
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("http://url_that_delays_loading");
WebElement myDynamicElement = driver.findElement(By.id("myDynamicElement"));
```

**Explicit Wait:** There can be instance when a particular element takes more than a minute to load. In that case you definitely not like to set a huge time to Implicit wait, as if you do this your browser will be going to wait for the same time for every element.

To avoid that situation, you can simply put a separate time on the required element only. By following this your browser implicit wait time would be short for every element and it would be large for specific element.

```
WebDriverWait wait = new WebDriverWait(driver, 10);
WebElement element = wait.until(ExpectedConditions.elementToBeClickable(By.id("someid")));
```

**Fluent Wait:** Let's say you have an element which sometime appears in just 1 second and some time it takes minutes to appear. In that case it is better to use fluent wait, as this will try to find element again and again until it finds it or until the final timer runs out.

```
Wait wait = new FluentWait(driver)
    .withTimeout(30, SECONDS)
    .pollingEvery(5, SECONDS)
    .ignoring(NoSuchElementException.class);
WebElement foo = wait.until(new Function() {
    public WebElement apply(WebDriver driver) {
        return driver.findElement(By.id("foo"));
    }
});
```

## Select(class.getOption)

**getOptions:** is used to get the selected option from the dropdown list.

**Select class :** Models a SELECT tag, providing helper methods to select and deselect options. Select is a class which is provided by Selenium to perform multiple operations on DropDown object and Multiple Select object

**getOptions() :** List<WebElement> -This gets the all options belonging to the Select tag. It takes no parameter and returns List<WebElements>. , Command – oSelect.getOptions();

Sometimes you may like to count the element in the dropdown and multiple select box, so that you can use the loop on Select element.

Code – To get the Count of the total elements inside SELECT.

```
Select oSelect = new Select(driver.findElement(By.id("yy_date_8")));
List <WebElement> elementCount = oSelect.getOptions();
System.out.println(elementCount.size());}
```

All of the above methods work on both Dropdown and Multiple select box.

# Advanced Selenium

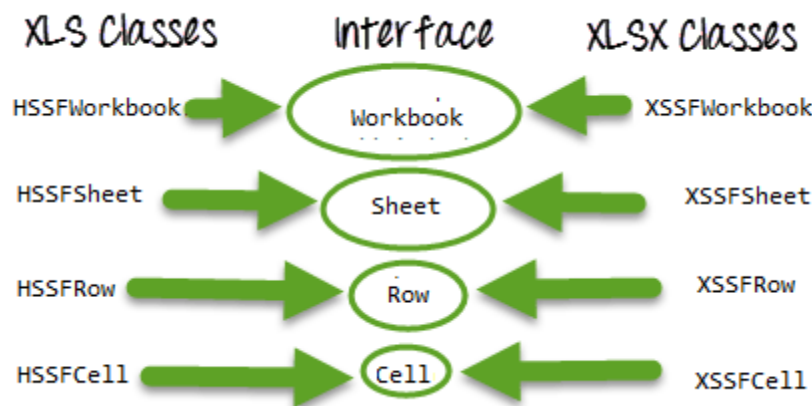
## Read & Write Data from Excel File in Selenium Webdriver: POI & JXL

File IO is a critical part of any software process. We frequently create a file, open it & update something or delete it in our Computers. Same is the case with Selenium Automation. We need a process to manipulate files with Selenium.

### Apache POI in Selenium

The **Apache POI in Selenium** is a widely used API for selenium data driven testing. It is a POI library written in Java that gives users an API for manipulating Microsoft documents like .xls and .xlsx. Users can easily create, modify and read/write into excel files. POI stands for “Poor Obfuscation Implementation.”

## Classes and Interfaces in POI:



Following is a list of different Java Interfaces and classes in **POI** for reading **XLS** and **XLSX** file-

- **Workbook:** XSSFWorkbook and HSSFWorkbook classes implement this interface.
- **XSSFWorkbook:** Is a class representation of XLSX file.
- **HSSFWorkbook:** Is a class representation of XLS file.
- **Sheet:** XSSFSheet and HSSFSheet classes implement this interface.
- **XSSFSheet:** Is a class representing a sheet in an XLSX file.
- **HSSFSheet:** Is a class representing a sheet in an XLS file.
- **Row:** XSSFRow and HSSFRow classes implement this interface.
- **XSSFRow:** Is a class representing a row in the sheet of XLSX file.
- **HSSFRow:** Is a class representing a row in the sheet of XLS file.
- **Cell:** XSSFCell and HSSFCell classes implement this interface.
- **XSSFCell:** Is a class representing a cell in a row of XLSX file.
- **HSSFCell:** Is a class representing a cell in a row of XLS file.



## How to read excel files using Apache POI

```
public void ReadExcel() {FileInputStream fis = new FileInputStream("D:\\Test.xlsx");

    XSSFWorkbook workbook = new XSSFWorkbook(fis);

    XSSFSheet sheet = workbook.getSheetAt(0);

    // I have added test data in the cell A1 as "SoftwareTestingMaterial.com"

    // Cell A1 = row 0 and column 0. It reads first row as 0 and Column A as 0.

    Row row = sheet.getRow(0);

    Cell cell = row.getCell(0);

    System.out.println(cell);

    System.out.println(sheet.getRow(0).getCell(0));

    // String cellval = cell.getStringCellValue();

    // System.out.println(cellval);

}
```

## Page Object Model (POM) & Page Factory

**Page Object Model (POM)** is a design pattern, popularly used in test automation that creates **Object Repository** for web UI elements. The advantage of the model is that it reduces code duplication and improves test maintenance.

Under this model, for each web page in the application, there should be a corresponding **Page Class**. This Page class will identify the WebElements of that web page and also contains **Page methods** which perform operations on those WebElements. Name of these methods should be given as per the task they are performing, i.e., if a loader is waiting for the payment gateway to appear, POM method name can be `waitForPaymentScreenDisplay()`.

## What is Page Factory in Selenium?

**Page Factory in Selenium** is an inbuilt Page Object Model 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."

Here as well, we follow the concept of separation of Page Object Repository and Test Methods. Additionally, with the help of PageFactory class, we use annotations **@FindBy** to find WebElement. We use `initElements` method to initialize web elements

WebElements are identify by  
@FindBy Annotation

Static initElements method of  
PageFactory class for  
initializing WebElement

```
@FindBy(xpath="//table//tr[@class='heading3']")
WebElement homePageUserName;

public Guru99HomePage(WebDriver driver){
    this.driver = driver;
    //This initElements method will create all WebElements
    PageFactory.initElements(driver, this);
}
```

@FindBy can accept **tagName**, **partialLinkText**, **name**, **linkText**, **id**, **css**, **className**, **xpath** as attributes

## Data-driven Framework:

Data-driven test automation framework is focused on separating the test scripts logic and the test data from each other. It allows us to create test automation scripts by passing different sets of test data.

The test data set is kept in the external files or resources such as MS Excel Sheets, MS Access Tables, SQL Database, XML files, etc.,

The test scripts connect to the external resources to get the test data. By using this framework, we could easily make the test scripts work properly for different sets of test data.

This framework significantly reduces the number of test scripts compared to module-based framework.

This framework gives more test coverage with reusable tests and flexibility in the execution of tests only when required and by changing only the input test data and reliable in terms of no impact on tests by changing the test data but it has its own drawbacks such as testers who work on this framework needs to have the hands-on programming knowledge to develop test scripts

## Hybrid Driven Testing Framework:

Hybrid Test automation framework is the combination of two or more frameworks mentioned above. It attempts to leverage the strengths and benefits of other frameworks for the particular test environment it manages. Most of the teams are building this hybrid driven framework in the current market.

## Explain Test Automation Framework

Test Automation Framework such as programming language used, Type of framework used,

Test Base Class (Initializing WebDriver, Implicit Waits),

How we separate Element locators and tests (Page Objects, Page Factory),

Utility functions file, Property files, TestNG annotations,

How we parameterize tests using Excel files,

How we capture error screenshots, Generating reports (Extent Reports), Emailing reports, Version Control System used and Continuous Integration Tool used.

**Language:** In our Selenium Project we are using Java language.

**Type of Framework:** In our project, we are using Data-driven Framework by using Page Object Model design pattern with Page Factory.

### **POM (Page Object Model):**

As per the Page Object Model, we have maintained a class for every web page. Each web page has a separate class and that class holds the functionality and members of that web page. Separate classes for every individual test.

### **Packages:**

We have separate packages for *Pages* and *Tests*. All the *web page* related classes come under the **Pages** package and all the *tests* related classes come under **Tests** package.

For example, *Home Page* and *Login Page* have separate classes to store element locators. For the *login test*, there would be a separate class which calls the methods from the *Home Page* class and *Login Page* class.

### **Test Base Class:**

Test Base class (TestBase.java) deals with all the common functions used by all the pages. This class is responsible for loading the configurations from properties files, Initializing the WebDriver, Implicit Waits, Extent Reports, and also to create the object of FileInputStream which is responsible for pointing towards the file from which the data should be read.

### **Utility Class (AKA Functions Class):**

Utility class (TestUtil.java) stores and handles the functions (The code which is repetitive in nature such as waits, actions, capturing screenshots, accessing excels, sending email, etc.,) which can be commonly used across the entire framework.

The reason behind creating a utility class is to achieve reusability. This class extends the TestBase class to inherit the properties of TestBase in TestUtil.

### **Properties file:**

This file (***config.properties***) stores the information that remains static throughout the framework such as browser-specific information, application URL, screenshots path, etc.

All the details which change as per the environment and authorization such as URL, Login Credentials are kept in the ***config.properties*** file. Keeping these details in a separate file makes it easy to maintain.

**Screenshots:** Screenshots will be captured and stored in a separate folder and also the screenshots of failed test cases will be added to the extent reports.

**Test Data:** All the historical test data will be kept in an excel sheet (***controller.xlsx***). By using '***controller.xlsx***', we pass test data and handle data-driven testing. We use Apache POI to handle excel sheets.

**TestNG:** Using TestNG for Assertions, Grouping, and Parallel execution.

## Maven:

Using Maven for build, execution, and dependency purpose. Integrating the TestNG dependency in the POM.xml file and running this POM.xml file using Jenkins or command prompt.

**Version Control Tool:** We use Git as a repository to store our test scripts.

## Jenkins:

By using Jenkins CI (Continuous Integration) Tool, we execute test cases on a daily basis and also for nightly execution based on the schedule. Test Results will be sent to the peers using Jenkins.

## Extent Reports:

For the reporting purpose, we are using Extent Reports. It generates beautiful HTML reports. We use the extent reports for maintaining logs and also to include the screenshots of failed test cases in the Extent Report.

# What is Maven?

**Maven** is an automation and management tool/ project management tool developed by Apache Software Foundation. It is written in Java Language to build projects

It allows developers to create projects, dependency, and documentation using Project Object Model and plugins. It has a similar development process as ANT, but it is more advanced than ANT.

Maven can also build any number of projects into desired output such as jar, war, metadata.

The main feature of Maven is that it can download the project dependency libraries automatically.

## Build Life Cycle:

Basic maven phases are used as below.

- **clean:** deletes all artifacts and targets which are created already.
- **compile:** used to compile the source code of the project.
- **test:** test the compiled code and these tests do not require to be packaged or deployed.
- **package:** package is used to convert your project into a jar or war etc.
- **install:** install the package into the local repository for use of another project.

Build the Project:

The project can be built by both using IDE and command prompt.

Using IDE you have to right click on POM- *Run As-Maven* Build

There are three built-in build lifecycles: default, clean and site. The **default** lifecycle handles your project deployment, the **clean** lifecycle handles project cleaning, while the **site** lifecycle handles the creation of your project's web site.

## A Build Lifecycle is Made Up of Phases

Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

For example, the default lifecycle comprises of the following phases (for a complete list of the lifecycle phases, refer to the Lifecycle Reference):

- **validate** - validate the project is correct and all necessary information is available
- **compile** - compile the source code of the project
- **test** - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- **package** - take the compiled code and package it in its distributable format, such as a JAR.
- **verify** - run any checks on results of integration tests to ensure quality criteria are met
- **install** - install the package into the local repository, for use as a dependency in other projects locally
- **deploy** - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

## Usual Command Line Calls

You should select the phase that matches your outcome. If you want your jar, run **package**. If you want to run the unit tests, run **test**.

If you are uncertain what you want, the preferred phase to call is

### mvn verify

This command executes each default lifecycle phase in order (**validate**, **compile**, **package**, etc.), before executing **verify**. You only need to call the last build phase to be executed, in this case, **verify**. In most cases the effect is the same as **package**. However, in case there are integration-tests, these will be executed as well. And during the **verify** phase some additional checks can be done, e.g., if your code written according to the predefined check style rules.

In a build environment, use the following call to cleanly build and deploy artifacts into the shared repository.

### mvn clean deploy

The same command can be used in a multi-module scenario (i.e. a project with one or more subprojects). Maven traverses into every subproject and executes **clean**, then executes **deploy** (including all of the prior build phase steps).

## A Build Phase is Made Up of Plugin Goals

However, even though a build phase is responsible for a specific step in the build lifecycle, the manner in which it carries out those responsibilities may vary. And this is done by declaring the plugin goals bound to those build phases.

A plugin goal represents a specific task (finer than a build phase) which contributes to the building and managing of a project. It may be bound to zero or more build phases. A goal not bound to any build phase could be executed outside of the build lifecycle by direct invocation. The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked. For example, consider the command below. The **clean** and **package** arguments are build phases, while the **dependency:copy-dependencies** is a goal (of a plugin).

### mvn clean dependency:copy-dependencies package

If this were to be executed, the `clean` phase will be executed first (meaning it will run all preceding phases of the clean lifecycle, plus the `clean` phase itself), and then the `dependency:copy-dependencies` goal, before finally executing the `package` phase (and all its preceding build phases of the default lifecycle).

Moreover, if a goal is bound to one or more build phases, that goal will be called in all those phases.

Furthermore, a build phase can also have zero or more goals bound to it. If a build phase has no goals bound to it, that build phase will not execute. But if it has one or more goals bound to it, it will execute all those goals.

### **What is the command to check the maven version?**

`mvn -version`

### **What is the purpose of mvn clean command?**

The command removes the target directory before the starting of a build process

### **What is the command to package maven project?**

1. `mvn -package`

## Web Table

A **Web Table** in Selenium is a WebElement used for the tabular representation of data or information. The data or information displayed can be either static or dynamic. Web table and its elements can be accessed using WebElement functions and locators in Selenium. A typical example of a web table would be product specifications displayed on an eCommerce platform.

### Reading a HTML Web Table

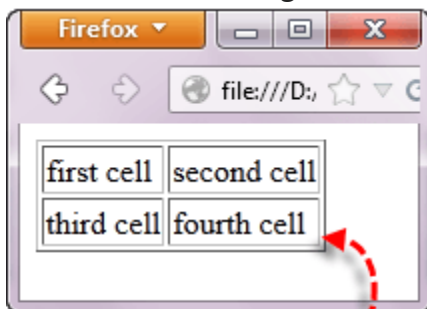
There are times when we need to access elements (usually texts) that are within HTML tables. However, it is very seldom for a web designer to provide an id or name attribute to a certain cell in the table. Therefore, we cannot use the usual methods such as “By.id()”, “By.name()”, or “By.cssSelector()”. In this case, the most reliable option is to access them using the “By.xpath()” method.

### How to write XPath for Table in Selenium

Consider the HTML code below for handling web tables in Selenium.

```
<html>
<head>
  <title>Sample</title>
</head>
<body>
  <table border="1">
    <tbody>
      <tr>
        <td>first cell</td>
        <td>second cell</td>
      </tr>
      <tr>
        <td>third cell</td>
        <td>fourth cell</td>
      </tr>
    </tbody>
  </table>
</body>
</html>
```

We will use [XPath](#) to get the inner text of the cell containing the text “fourth cell.”



*we will try to  
access this cell*

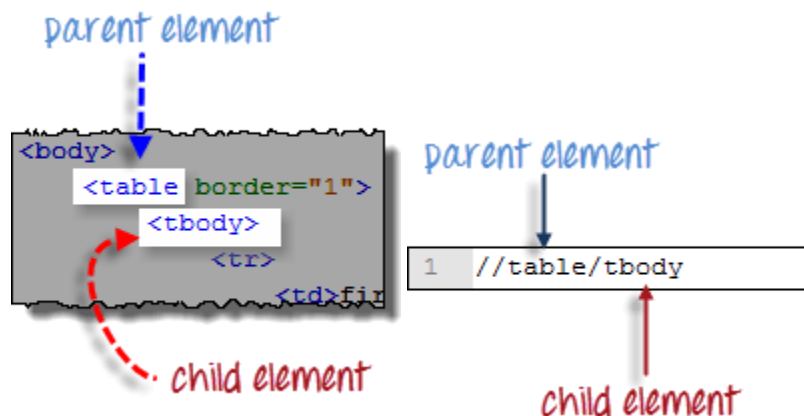
#### Step 1 – Set the Parent Element (table)

**XPath locators in WebDriver always start with a double forward slash “//” and then followed by the parent element.** Since we are dealing with web tables in Selenium, the parent element should always be the <table> tag. The first portion of our Selenium XPath table locator should, therefore, start with “//table”.

```
//table
```

## Step 2 – Add the child elements

The element immediately under <table> is <tbody> so we can say that <tbody> is the “child” of <table>. And also, <table> is the “parent” of <tbody>. All child elements in XPath are placed to the right of their parent element, separated with one forward slash “/” like the code shown below.



## Step 3 – Add Predicates

The <tbody> element contains two <tr> tags. We can now say that these two <tr> tags are “children” of <tbody>. Consequently, we can say that <tbody> is the parent of both the <tr> elements.

Another thing we can conclude is that the two <tr> elements are siblings. **Siblings refer to child elements having the same parent.**

To get to the <td> we wish to access (the one with the text “fourth cell”), we must first access the **second** <tr> and not the first. If we simply write “//table/tbody/tr”, then we will be accessing the first <tr> tag.

So, how do we access the second <tr> then? The answer to this is to use **Predicates**.

**Predicates are numbers or HTML attributes enclosed in a pair of square brackets “[ ]” that distinguish a child element from its siblings.** Since the <tr> we need to access is the second one, we shall use “[2]” as the predicate.



```
1 //table/tbody/tr[2]
```

The `[2]` predicate denotes that we are accessing the 2nd `<tr>` of the parent `<tbody>`

If we won't use any predicate, XPath will access the first sibling. Therefore, we can access the first `<tr>` using either of these XPath codes.

```
//table/tbody/tr
```

this will automatically access the first `<tr>` because no predicate was used

```
//table/tbody/tr[1]
```

this will access the first `<tr>` because the predicate `[1]` explicitly says it

#### Step 4 – Add the Succeeding Child Elements Using the Appropriate Predicates

The next element we need to access is the second `<td>`. Applying the principles we have learned from steps 2 and 3, we will finalize our XPath code to be like the one shown below.

```
//table/tbody/tr[2]/td[2]
```

Now that we have the correct XPath locator, we can already access the cell that we wanted to and obtain its inner text using the code below. It assumes that you have saved the HTML code above as “newhtml.html” within your C Drive.

```
public static void main(String[] args) {  
    String baseUrl = "file:///C:/newhtml.html";  
    WebDriver driver = new FirefoxDriver();  
  
    driver.get(baseUrl);  
    String innerText = driver.findElement(  
        By.xpath("//table/tbody/tr[2]/td[2]")).getText();  
    System.out.println(innerText);  
    driver.quit();  
}
```

- By.xpath() is commonly used to access elements of WebTable in Selenium.
- If the element is written deep within the HTML code such that the number to use for the predicate is very difficult to determine, we can use that element's unique attribute instead for Selenium get table element.
- Attributes are used as predicates by prefixing them with the @ symbol.
- Use Inspect Element for Accessing WebTable in Selenium

## Selenium Exceptions:

An exception is an error that happens at the time of execution of a program. However, while running a program, programming languages generates an exception that should be handled to avoid your program to crash.

The exception indicates that, although the event can occur, this type of event happens infrequently. When the method is not able to handle the Exception, it is thrown to its caller function. Eventually, when an exception is thrown out of the main function, the program is terminated abruptly.

### Common Exceptions in Selenium Web driver

<b><i>Exception name</i></b>	<b><i>Description</i></b>
<b>ElementNotVisibleException</b>	This type of Selenium exception occurs when an existing element in DOM has a feature set as hidden.
<b>ElementNotSelectableException</b>	This Selenium exception occurs when an element is presented in the DOM, but you can be able to select. Therefore, it is not possible to interact.
<b>NoSuchElementException</b>	This Exception occurs if an element could not be found.
<b>NoSuchFrameException</b>	This Exception occurs if the frame target to be switched to does not exist.
<b>NoAlertPresentException</b>	This Exception occurs when you switch to no presented alert.
<b>NoSuchWindowException</b>	This Exception occurs if the window target to be switch does not exist.
<b>StaleElementReferenceException</b>	This Selenium exception occurs happens when the web element is detached from the current DOM.
<b>SessionNotFoundException</b>	The WebDriver is acting after you quit the browser.
<b>TimeoutException</b>	Thrown when there is not enough time for a command to be completed. For Example, the element searched wasn't found in the specified time.
<b>WebDriverException</b>	This Exception takes place when the WebDriver is acting right after you close the browser.
<b>ConnectionClosedException</b>	This type of Exception takes place when there is a disconnection in the driver.
<b>ElementClickInterceptedException</b>	The command may not be completed as the element receiving the events is concealing the element which was requested clicked.
<b>ElementNotInteractableException</b>	This Selenium exception is thrown when any element is presented in the DOM. However, it is impossible to interact with such an element.
<b>ErrorResponseException</b>	This happens while interacting with the Firefox extension or the remote driver server.
<b>ErrorHandler.UnknownServerException</b>	Exception is used as a placeholder in case if the server returns an error without a stacktrace.
<b>ImeActivationFailedException</b>	This expectation will occur when IME engine activation has failed.
<b>ImeNotAvailableException</b>	It takes place when IME support is unavailable.
<b>InsecureCertificateException</b>	Navigation made the user agent to hit a certificate warning. This can cause by an invalid or expired TLS certificate.
<b>InvalidArgumentException</b>	It occurs when an argument does not belong to the expected type.
<b>InvalidCookieDomainException</b>	This happens when you try to add a cookie under a different domain instead of current URL.
<b>InvalidCoordinatesException</b>	This type of Exception matches an interacting operation that is not valid.

<b>InvalidElementStateException</b>	It occurs when command can't be finished when the element is invalid.
<b>InvalidSessionIdException</b>	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.
<b>InvalidSwitchToTargetException</b>	This occurs when the frame or window target to be switched does not exist.
<b>JavascriptException</b>	This issue occurs while executing JavaScript given by the user.
<b>JsonException</b>	It occurs when you afford to get the session when the session is not created.
<b>NoSuchAttributeException</b>	This kind of Exception occurs when the attribute of an element could not be found.
<b>MoveTargetOutOfBoundsException</b>	It takes place if the target provided to the ActionChains move() methodology is not valid. For Example, out of the document.
<b>NoSuchContextException</b>	ContextAware does mobile device testing.
<b>NoSuchCookieException</b>	This Exception occurs when no cookie matching with the given pathname found for all the associated cookies of the currently browsing document.
<b>NotFoundException</b>	This Exception is a subclass of WebDriverException. This will occur when an element on the DOM does not exist.
<b>RemoteDriverServerException</b>	This Selenium exception is thrown when the server is not responding because of the problem that the capabilities described are not proper.
<b>ScreenshotException</b>	It is not possible to capture a screen.
<b>SessionNotCreatedException</b>	It happens when a new session could not be successfully created.
<b>UnableToSetCookieException</b>	This occurs if a driver is unable to set a cookie.
<b>UnexpectedTagNameException</b>	Happens if a support class did not get a web element as expected.
<b>UnhandledAlertException</b>	This expectation occurs when there is an alert, but WebDriver is not able to perform Alert operation.
<b>UnexpectedAlertPresentException</b>	It occurs when there is the appearance of an unexpected alert.
<b>UnknownMethodException</b>	This Exception happens when the requested command matches with a known URL but and not matching with a methodology for a specific URL.
<b>UnreachableBrowserException</b>	This Exception occurs only when the browser is not able to be opened or crashed because of some reason.
<b>UnsupportedCommandException</b>	This occurs when remote WebDriver does n't send valid commands as expected.

- An exception is an error that happens at the time of execution of a program.
- Try-catch: This method can **catch** Exceptions, which uses a combination of the **try** and **catch** keywords.
- Multiple **catches** help you to handle every type of Exception separately with a separate block of code.
- **Throw** keyword is used to throw Exception to handle it in the run time.
- **printStackTrace():** This function prints stack trace, name of the Exception, and other useful description
- **toString():** This function returns a text message describing the exception name and description.
- **getMessage():** Helps to displays the description of the Exception.