SELENIUM TUTORIAL

Selenium is one of the most widely used open source Web UI automation testing tools. It supports automation of websites across different browsers, platforms and programming languages. Our tutorials are designed for beginners with basic knowledge of programming language - Java. We will start with the basics of Selenium and then as the tutorial progresses we will move to the more advanced stuff.

BASIC SELENIUM WEBDRIVER TUTORIAL

CSS AND XPATH LOCATOR TUTORIAL

SELENIUM WEBDRIVER TUTORIAL

SELENIUM INTERVIEW QUESTIONS

- Top 100 Selenium Interview Questions with Answers

SELENIUM INTRODUCTION

Hello friends! this is our first tutorial of the "Selenium Automation" series. In this tutorial, we will study the basics of selenium, its components, features and limitations.

WHAT IS SELENIUM?

Selenium is an open source test automation suite used for automating web based applications. It supports automation across different browsers, platforms and programming languages. Using selenium, we can automate the functional tests and easily integrate them with Maven, Jenkins and other build automation and continuous integration tools.

COMPONENTS OF SELENIUM SUITE

Selenium Suite comprises of the following four components-

1. *Selenium IDE* - Selenium IDE is an add-on of Firefox browser that provides record and play back functionality. Its use is limited and test scripts generated are not very robust and portable.
2. *Selenium RC* - Selenium Remote Control(RC) is officially deprecated by Selenium. It used to require an additional server for running the automation scripts and had many limitation.
3. *Selenium WebDriver* - By far the most important component of Selenium Suite. It provides different drivers for different browsers and supports multiple programming languages.
4. *Selenium Grid* - Selenium Grid is also an important part of Selenium Suite. It helps in distributed running of selenium tests in parallel across multiple remote machines.

ADVANTAGES OF SELENIUM

Let's now see some of the advantages of Selenium-

1. Selenium is open source, there is no licensing cost for its usage.
2. Scripting can be done in most of the widely used programming languages like Java, C#, Ruby and Python
3. It supports most of the popular browsers like Chrome, FireFox, Internet Explorer, Opera and Safari.
4. Selenium IDE component of Selenium suite provides record and playback feature using which non-programmers can also write automation scripts.
5. Selenium Grid helps in parallel and distributed testing.

LIMITATIONS OF SELENIUM

Some of the limitation of Selenium are-

1. Selenium does not provide desktop application automation support.
2. Web Services - REST or SOAP cannot be automated using selenium.
3. Selenium webDriver requires programming language requirement for script creation.
4. For performing common tasks required in automation like logging, reading-writing to external files we have to rely on external libraries.

SELENIUM WEBDRIVER

Selenium Webdriver (also known as Selenium 2.0) is one of the most widely used tool for automating web applications. It automates the browsers by calling their native method directly unlike Selenium RC which injects javascript in browsers for automation. Hence, webdriver is much faster than Selenium RC and also, can handle scenarios like alerts, pop-ups, ajax requests, keyboard and mouse actions easily. It also supports most of the popular programming languages like - Java, C#, Python, Ruby etc. Since, webdriver directly calls the methods of different browsers hence we have separate driver for each browser. Some of the widely used drivers in selenium are - FirefoxDriver, ChromeDriver, InternetExplorerDriver, SafariDriver and HtmlUnitDriver(a special headless driver).

Going forward, we will be studying the Selenium WebDriver component exclusively in our next tutorials.

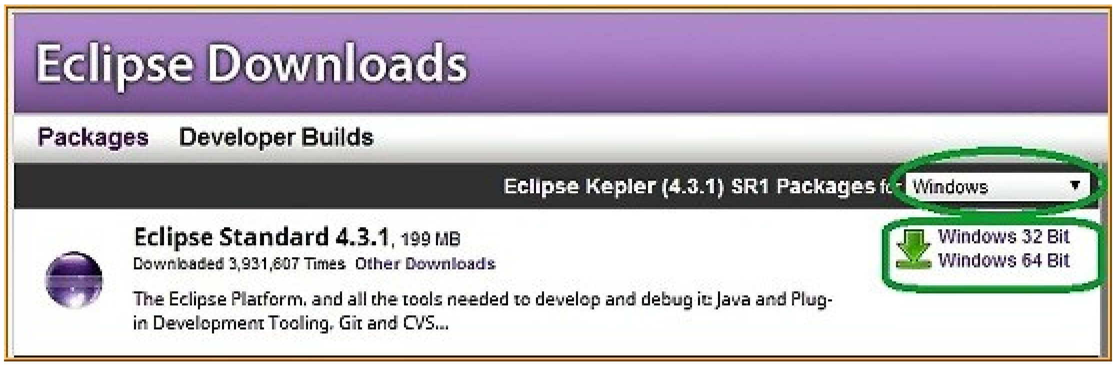
SELENIUM WEBDRIVER SETUP

SELENIUM WEBDRIVER WITH JAVA AND ECLIPSE SETUP

Following steps will guide you through setting up selenium webdriver on your machine-

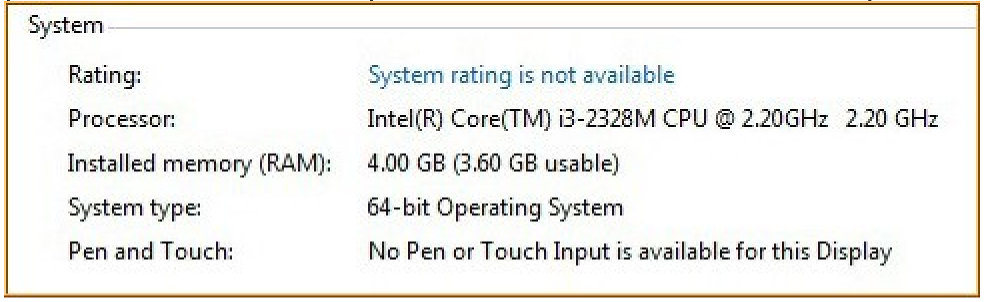
Step.1. Configuring Java on machine
Download latest version of Java Development Kit(JDK) from here. Follow the steps mentioned to install JDK. We are installing JDK as it will be required for developing and running our automation scripts which are nothing but java programs.

Step.2. Download eclipse or any Java IDE of your choice
Download the latest version of the Java IDE you would like to use, for Eclipse the download link is http://www.eclipse.org/downloads/. Select the appropriate version of Eclipse depending on your system type- 34 bit or 64 bit.



Eclipse Standard 4.3.1, 199 MB
Downloaded 3,931,607 Times Other Downloads
The Eclipse Platform, and all the tools needed to develop and debug it: Java and Plug-in Development Tooling, Git and CVS...

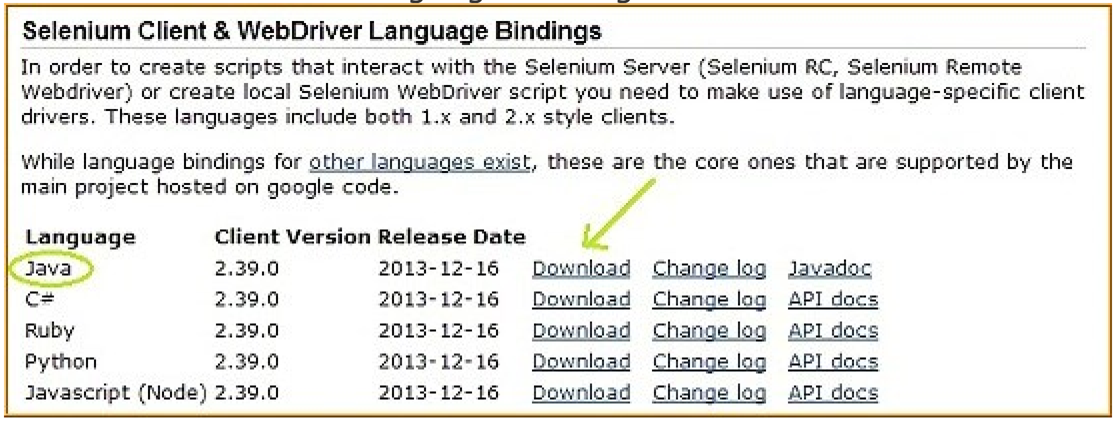
In order to check your system type- go to my computer and right click and select 'properties', under system section you will see a 'System type' parameter. This parameter will define your system type.



System

Rating:	System rating is not available
Processor:	Intel(R) Core(TM) i3-2328M CPU @ 2.20GHz 2.20 GHz
Installed memory (RAM):	4.00 GB (3.60 GB usable)
System type:	64-bit Operating System
Pen and Touch:	No Pen or Touch Input is available for this Display

Step.3. Download Selenium Webdriver jar from SeleniumHQ website
Go to SeleniumHQ website- http://docs.seleniumhq.org/download/ and under "Selenium Client & WebDriver Language Bindings" download the webdriver for



Selenium Client & WebDriver Language Bindings

In order to create scripts that interact with the Selenium Server (Selenium RC, Selenium Remote Webdriver) or create local Selenium WebDriver script you need to make use of language-specific client drivers. These languages include both 1.x and 2.x style clients.

While language bindings for other languages exist, these are the core ones that are supported by the main project hosted on google code.

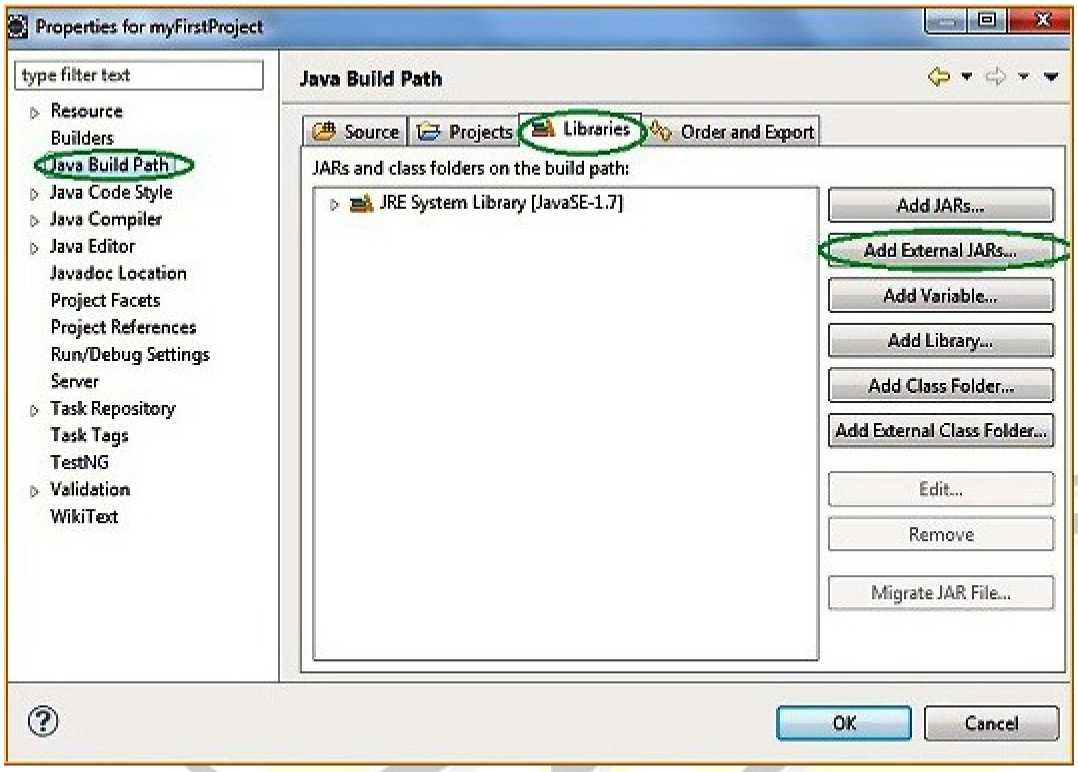
Language	Client Version	Release Date			
Java	2.39.0	2013-12-16	Download	Change log	Javadoc
C#	2.39.0	2013-12-16	Download	Change log	API docs
Ruby	2.39.0	2013-12-16	Download	Change log	API docs
Python	2.39.0	2013-12-16	Download	Change log	API docs
Javascript (Node)	2.39.0	2013-12-16	Download	Change log	API docs

java.
Unzip the package and place it on any directory as a library folder.

Step.4. Creating project and configuring selenium jars-

- Launch eclipse.exe
- Set your workspace to any location preferably other than C:(a workspace is a physical location where we store our project or group of related projects).
- Now create a new project- File->New->Project...->Java->Java Project
- Name you project and click Finish
- Now you will see an src folder under your project. Under this we need to create a package-Right Click src->New->Package (Basically these packages are used to group together related classes). Name your package e.g. 'myTestPackage'
- Inside this package create a new class and name it e.g. Test, your Test.java class will get created

Step.5. Adding selenium jars
Right Click your project on the left and click on properties. A "Properties for {project name}" dialog box will appear. Click on "Java Build Path" on the left and then click on Libraries tab on the right. In this tab click on "Add External Jars.." button.



Now browse to the location where selenium libraries are placed (library folder Step#3). Make sure to add both the libraries-selenium-java-2.39.0.jar and selenium-java-2.39.0-srcs.jar along with the libraries present in libs folder (selenium-2.39.0\libs). The selected libraries will appear, click OK to add these libraries to your project. You can verify the same in the "Referenced Libraries" section under your project in the "Package Explorer" section on the left.

Step.6. Creating first selenium webdriver project
Time to test the setup. Now, we will create our first selenium project, in which we will just open Firefox browser and launch a website. Following steps are required to launch the firefox browser.

1. Download geckodriver.exe from GeckoDriver Github Release Page. Make sure to download the right driver file based on your platform and OS version.
2. Set the **System Property** for "webdriver.gecko.driver" with the geckodriver.exe path - System.setProperty("webdriver.gecko.driver","geckodriver.exe path");

Code snippet to launch Firefox browser-

```
public class FirefoxBrowserLaunchDemo {

    public static void main(String[] args) {
        //Creating a driver object referencing WebDriver interface
        WebDriver driver;

        //Setting webdriver.gecko.driver property
        System.setProperty("webdriver.gecko.driver", "{path to geckodriver}\\geckodri
ver.exe");

        //Instantiating driver object and launching browser
        driver = new FirefoxDriver();

        //Using get() method to open a webpage
        driver.get("http://google.com");

        //Closing the browser
        driver.quit();

    }

}
```

To run the test, right click on Test.java file on the Package Explorer section, hover over "Run As" and select "Java Application". Firefox broswer will launch and open artoftesting.com.

LAUNCHING BROWSERS IN SELENIUM

In this post, we will study the Selenium WebDriver commands used to launch browsers in detail. We will also learn the different additional customization required for launching certain browsers like - Chrome and InternetExplorer.

UNDERSTANDING THE BROWSER LAUNCHING COMMAND

As we have studied in previous tutorials that Selenium WebDriver calls the native methods of the different browsers to automate them. Hence, in Selenium we have different WebDrivers for different browsers like - FirefoxDriver for Firefox browser, ChromeDriver for Google Chrome, InternetExplorerDriver for Internet Explorer etc. Now let's take an example of launching Firefox browser and understand the command in detail-

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

This is the java implementation of launching a browser in Selenium. Here, 'WebDriver' is an interface and we are creating a reference variable 'driver' of type WebDriver, instantiated using 'FireFoxDriver' class.
For those who are not very proficient in Java, an interface is like a contract that classes implementing it must follow. An interface contains a set of variables and methods without any body(no implementaion, only method name and signature). We cannot instantiate objects from interfaces. Hence, the below line of code is incorrect and throws compile time error saying "Cannot instantiate the type WebDriver".

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

For instantiation of driver object, we need classes like FirefoxDriver or ChromeDriver which have implemented the WebDriver interface. In other words, these driver classes have followed the contract of WebDriver by implementing all the methods of the WebDriver interface. Thus making all the different types of driver classes uniform, following the same protocol.
Please note that we can also create a reference variable of type FirefoxDriver like this-
FirefoxDriver driver = new FirefoxDriver();

But having a WebDriver reference object helps in multi-browser testing as the same driver object can be used to assign to any of the desired browser specific driver.

LAUNCHING FIREFOX BROWSER

Firefox is one of the most widely used browsers in automation. Following steps are required to launch the firefox browser.

1. Download geckodriver.exe from GeckoDriver Github Release Page. Make sure to download the right driver file based on your platform and OS version.
2. Set the **System Property** for "webdriver.gecko.driver" with the geckodriver.exe path - System.setProperty("webdriver.gecko.driver","geckodriver.exe path");

Code snippet to launch Chrome browser-

```
public class FirefoxBrowserLaunchDemo {

    public static void main(String[] args) {

        //Creating a driver object referencing WebDriver interface
        WebDriver driver;

        //Setting webdriver.gecko.driver property
        System.setProperty("webdriver.gecko.driver", pathToGeckoDriver + "\\geckodriv
er.exe");

        //Instantiating driver object and launching browser
        driver = new FirefoxDriver();

        //Using get() method to open a webpage
        driver.get("http://artoftesting.com");

        //Closing the browser
        driver.quit();

    }

}
```

LAUNCHING CHROME BROWSER

For running Chrome browser in Selenium, we need to set the webdriver.chrome.driver system property to point to a chromeDriver executable file-

1. Download the latest ChromeDriver binary from Chromium.org download page and place the executable on your local machine.
2. Set the webdriver.chrome.driver property to the chromeDriver.exe's location as- System.setProperty("webdriver.chrome.driver", "chromeDriver.exe path");

Code snippet to launch Chrome browser-

```
public class ChromeBrowserLaunchDemo {

    public static void main(String[] args) {

        //Creating a driver object referencing WebDriver interface
        WebDriver driver;

        //Setting the webdriver.chrome.driver property to its executable's location
        System.setProperty("webdriver.chrome.driver", "/lib/chromeDriver/chromedriver
.exe");

        //Instantiating driver object
        driver = new ChromeDriver();

        //Using get() method to open a webpage
        driver.get("http://artoftesting.com");

        //Closing the browser
        driver.quit();

    }

}
```

LAUNCHING INTERNET EXPLORER BROWSER

Like ChromeDriver, InternetExplore driver also requires setting up the "webdriver.ie.driver" property with the location of IEDriverServer.exe. The IEDriverServer.exe can be downloaded from here. Following code snippet can be used to launch IE browser-

```
public class IEBrowserLaunchDemo {

    public static void main(String[] args) {

        //Creating a driver object referencing WebDriver interface
        WebDriver driver;

        //Setting the webdriver.ie.driver property to its executable's location
        System.setProperty("webdriver.ie.driver", "/lib/IEDriverServer/IEDriverServer
.exe");

        //Instantiating driver object
        driver = new InternetExplorerDriver();

        //Using get() method to open a webpage
        driver.get("http://artoftesting.com");

        //Closing the browser
        driver.quit();

    }

}
```

LAUNCHING SAFARI BROWSER

Like Firefox browser, the Safari browser doesn't require any additional configuration and can be directly launched by instantiating with SafariDriver. The following code snippet can be used to launch the Safari browser-

```
public class SafariBrowserLaunchDemo {

    public static void main(String[] args) {

        //Creating a driver object referencing WebDriver interface
        WebDriver driver;

        //Instantiating driver object with SafariDriver
        driver = new SafariDriver();

        //Using get() method to open a webpage
        driver.get("http://artoftesting.com");

        //Closing the browser
        driver.quit();

    }

}
```

LOCATORS IN SELENIUM WEBDRIVER

Hello friends! Continuing with our "Selenium Automation" series, in this post we will study the different types of locators available in Selenium. Before studying the different locators, let's first see the need of locators in the automation process.
A simple automation process in Selenium can be presented as-

- Launching browser
- Opening the desired website to be automated
- Locating web elements like a textbox
- Performing operations on the located web elements like writing in the textbox
- Performing assertion like checking 'Success' message

Now, let's see a sample code snippet implementing the above process. We will be studying each line of code in the coming tutorials. For now just consider it as a black-box and focus on the operations performed(specified in the comments).

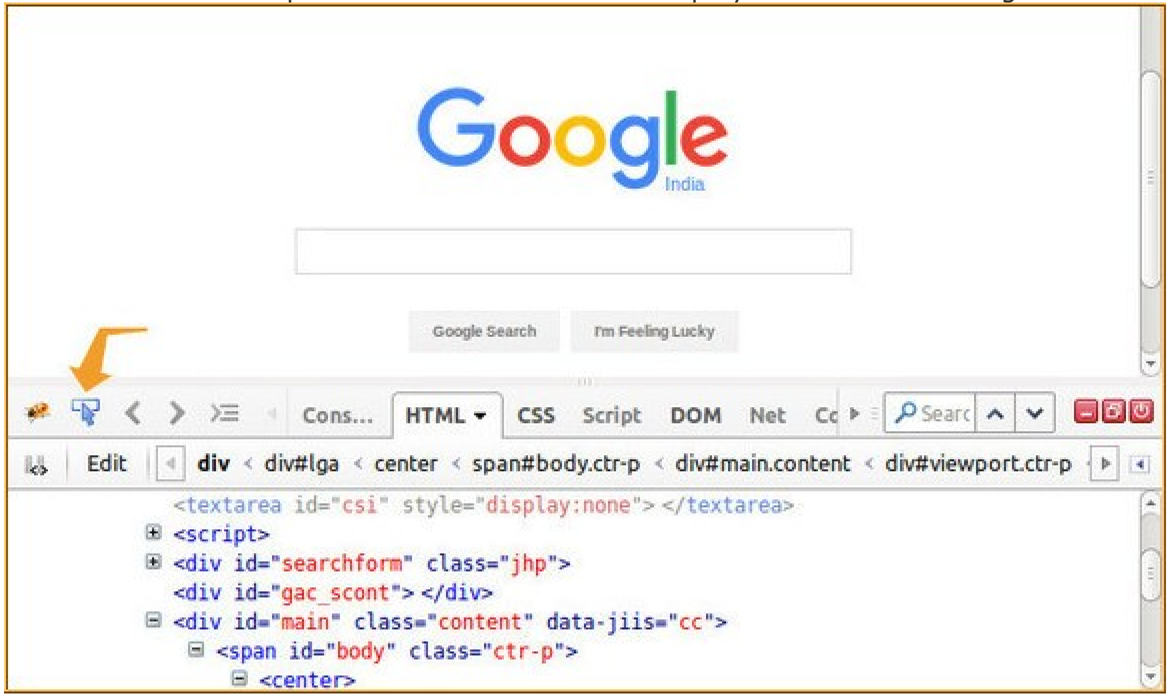
```
//Launching Firefox browser
WebDriver driver = new FirefoxDriver();

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

//Initializing webelement searchBox
WebElement searchBox = driver.findElement(By.name("q"));

//Writing a text "ArtOfTesting" in the search box
searchBox.sendKeys("ArtOfTesting");
```

So, here we see that in order to perform an operation on web element - searchBox, we first need to locate it. Here, **By.name("q")** is a locator which when passed to findElement method returns a searchBox web element.
Before going further with the different types of locators available in Selenium, let's first see, how to get the different attributes of an element which are used in the locators.

USING FIREBUG OR DEVELOPER TOOL

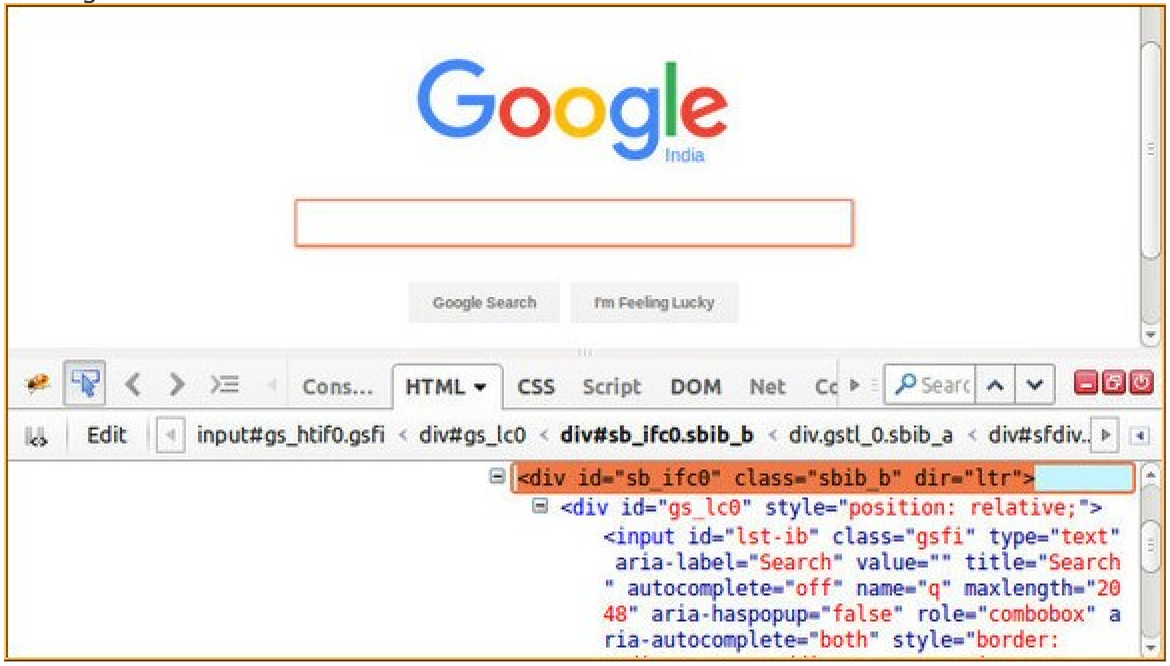
Locating web elements requires knowledge of their HTML attributes. For the HTML source code of specific elements, we can use a Mozilla Firefox plugin - firebug or use the inbuilt developer tools. Throughout the course of this tutorial we will use Firebug to locate elements. You can download firebug from here – Firebug addon for Mozilla.
Steps for finding element's HTML attributes-

- Launch the website to be automated e.g. - https://www.google.com
- Press F12 to launch firebug or developer tool.
- Click on the inspect-element icon as displayed in the image below.


Google
India
Google Search
I'm Feeling Lucky
Cons...
HTML
CSS
Script
DOM
Net
Search
Edit
div < div#lga < center < span#body.ctr-p < div#main.content < div#viewport.ctr-p

```
<textarea id="csi" style="display:none"> </textarea>
<script>
<div id="searchform" class="jhp">
<div id="gac_scont"> </div>
<div id="main" class="content" data-jiis="cc">
  <span id="body" class="ctr-p">
    <center>
```

- After clicking on the inspect-element icon, click on the web element to be located e.g. Google Search box. Once we click on the element, its HTML will get displayed in the firebug UI.


Google
India
Google Search
I'm Feeling Lucky
Cons... HTML CSS Script DOM Net
Edit input#gs_htif0.gsfi < div#gs_lc0 < div#sb_ifc0.sbib_b < div.gstl_0.sbib_a < div#sfdiv

```
<div id="sb_ifc0" class="sbib_b" dir="ltr">
  <div id="gs_lc0" style="position: relative;">
    <input id="lst-ib" class="gsfi" type="text"
     aria-label="Search" value="" title="Search
    " autocomplete="off" name="q" maxlength="20
    48" aria-haspopup="false" role="combobox" a
    ria-autocomplete="both" style="border:
```

- Here, we can see the different attributes of the web elements like - id, class, name, along with its tag like input, div etc. Now, we will be using these tags, attributes and their values to locate elements using different locators.

LOCATORS IN SELENIUM

There are a total of 8 locators in Selenium WebDriver-

1. **By Id** - Locates element using id attribute of the web element.

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

2. **By className** - Locates the web element using className attribute.

```
WebElement element = driver.findElement(By.className("elementsClass"));
```

3. **By tagName** - Locates the web element using its html tag like div, a, input etc.

```
WebElement element = driver.findElement(By.tagName("a"));
```

4. **By name** - Locates the web element using name attribute.

```
WebElement element = driver.findElement(By.name("male"));
```

5. **By linkText** - Locates the web element of link type using their text.

```
WebElement element = driver.findElement(By.linkText("Click Here"));
```

6. **By partialLinkText** - Locates the web element of link type with partial matching of text.

```
WebElement element = driver.findElement(By.partialLinkText("Click"));
```

7. **By cssSelector** - Locates the web element using css its CSS Selector patterns(explained in detailed here - CSS Locators).

```
WebElement element = driver.findElement(By.cssSelector("div#elementId"));
```

8. **By xpath** - Locates the web element using its XPaths(explained in detailed here XPath Locators).

```
WebElement element = driver.findElement(By.xpath("//div[@id='elementId']"));
```

Now you have successfully learnt how to locate elements in Selenium. As you can see locating element by id, className, tagName, name, linkText and partialLinkText is simple. We just have to select the right locator based on the uniqueness of the element e.g. we prefer using id because id of elements are generally unique. But there can be scenarios where we might not have id attributes of web elements, also other locators like name, className might not fetch the unique required web element. In those scenarios, we should use cssSelector and xpath locators. These locators are very powerful and help in creating robust locators for complex web elements. We have dedicated separate tutorials for these two locators in our coming posts- CSS Locatorsand XPath Locators. You are advised to have basic understanding of both the locators and choose either of the locators for detailed study.

SELENIUM WEBDRIVER COMMANDS

In this post, we will learn some of the basic selenium commands for performing operations like opening a URL, clicking on buttons, writing in textbox, closing the browser etc. For your practice, a dummy webpage with different types of web elements is available. Now, let's see the basic commands of Selenium WebDriver.

OPENING A URL

USING GET METHOD-

The driver.get() method is used to navigate to a web page by passing the string URL as parameter. Syntax-

```
driver.get("http://artoftesting.com");
```

USING NAVIGATE METHOD-

The driver.navigate().to() method does the task of opening a web page like driver.get() method. Syntax-

```
driver.navigate().to("http://artoftesting.com");
```

CLICKING ON WEB ELEMENTS

The click() method in Selenium is used to perform the click operation on web elements. In our previous tutorial Locators in Selenium WebDriver, we studied about locating the webElements in Selenium. The click() method is applied on the webElements identified, to perform the click operation.

```
//Clicking an element directly
driver.findElement(By.id("button1")).click();

//Or by first creating a WebElement and then applying click() operation
WebElement submitButton = driver.findElement(By.id("button2"));
submitButton.click();
```

WRITING IN A TEXTBOX

The sendKeys() method can be used for writing in a textbox or any element of text input type.

```
//Creating a textbox webElement
WebElement element = driver.findElement(By.name("q"));

//Using sendKeys to write in the textbox
element.sendKeys("ArtOfTesting!");
```

CLEARING TEXT IN A TEXTBOX

The clear() method can be used to clear the text written in a textbox or any web element of text input type.

```
//Clearing the text written in text fields
driver.findElement(By.name("q")).clear();
```

FETCHING TEXT WRITTEN OVER ANY WEB ELEMENT

In automation, many a times we need to fetch the text written over a web element for performing some assertions or debugging. For this, we have getText() method in selenium webDriver.

```
//Fetching the text written over web elements
```

```
driver.findElement(By.id("element123")).getText();
```

NAVIGATING BACKWARDS IN A BROWSER

Selenium provides navigate().back() command to move backwards in the browser's history.

```
//Navigating backwards in browser
driver.navigate().back();
```

NAVIGATING FORWARD IN A BROWSER

Selenium provides navigate().forward() command to move forward in a browser.

```
//Navigating forward in browser
driver.navigate().forward();
```

REFRESHING THE BROWSER

There are multiple ways to refresh a page in Selenium WebDriver-

- Using **driver.navigate().refresh()** command
- Using **sendKeys(Keys.F5)** on any textbox on the webpage
- Using **driver.get("URL")** with current URL
- Using **driver.navigate().to("URL")** with current URL

```
//Refreshing browser using navigate().refresh()
driver.navigate().refresh();

//By pressing F5 key on any textbox element
driver.findElement(By.id("id123")).sendKeys(Keys.F5);

//By opening the current URL using get() method
driver.get("http"//artoftesting.com");

//By opening the current URL using navigate() method
driver.navigate().to("http://artoftesting.com");
```

CLOSING BROWSER

Selenium provides two commands to close browsers close() and quite(). The driver.close() command is used to close the browser having focus. Whereas, the driver.quite command is used to close all the browser instances open.

```
//To close the current browser instance
driver.close();

//To close all the open browser instances
driver.quit();
```

SAMPLE SCRIPT

You can use the sample script below to automate the dummy webpage. The comments mentioned throughout the script will guide you through whole automation process. Also, note that the Thread.sleep() used in the sample script is to pause the automation in between events. This Thread.sleep() is not required and is only included to provide you some time to see the automation events.
Download this java file here seleniumBasicCommands.java (right click on 'seleniumBasicCommands.java' and click on 'save link as...' to save the sample selenium test script).

```
package myTestPackage;

import java.util.concurrent.TimeUnit;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.Select;
import org.testng.Assert;
import org.testng.annotations.Test;

public class seleniumBasicCommands {

public static void main(String Args[]) throws InterruptedException{

        //Create Firefox driver's instance
        WebDriver driver = new FirefoxDriver();

        //Set implicit wait of 10 seconds
        //This is required for managing waits in selenium webdriver
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

        //Launch sampleSiteForSelenium
        driver.get("http://www.artoftesting.com/sampleSiteForSelenium.html");

        //Fetch the text "This is sample text!" and print it on console
        //Use the id of the div to locate it and then fecth text using getText() meth
od
        String sampleText = driver.findElement(By.id("idOfDiv")).getText();
        System.out.println(sampleText);

        //Waiting for 3 seconds just for user to efficiently check automation
        //Its not mandatory though
        Thread.sleep(3000);

        //Using linkText locator to find the link and then using click() to click on
it
        driver.findElement(By.linkText("This is a link")).click();

        Thread.sleep(3000);

        //Finding textbox using id locator and then using send keys to write in it
        driver.findElement(By.id("fname")).sendKeys("Kuldeep Rana");

        Thread.sleep(3000);

        //Clear the text written in the textbox
        driver.findElement(By.id("fname")).clear();

        Thread.sleep(3000);

        //Clicking on button using click() command
        driver.findElement(By.id("idOfButton")).click();

        Thread.sleep(3000);

        //Find radio button by name and check it using click() function
        driver.findElement(By.name("male")).click();

        Thread.sleep(3000);

        //Find checkbox by cssSelector and check it using click() function
        driver.findElement(By.cssSelector("input.Automation")).click();

        Thread.sleep(3000);

        //Using Select class for for selecting value from dropdown
        Select dropdown = new Select(driver.findElement(By.id("testingDropdown")));
        dropdown.selectByVisibleText("Database Testing");
        Thread.sleep(50000);

        //Close the browser
        driver.close();

        }

}
```

IMPLICIT AND EXPLICIT WAITS IN SELENIUM

WHY ARE WAITS REQUIRED IN SELENIUM?

In UI automation, waits are required because certain elements get loaded on the page asynchronously, so after triggering an event a page may get loaded successfully but some of its element may still not get loaded. This causes elementNotFound exception while locating the element. In such cases we are left with using Thread.sleep() i.e. a static wait that will halt the test execution for some specified time and than perform the next step. As Thread.sleep() will wait for the specified time no matter if the elements gets visible before that time. So, using Thread.sleep() is never advisable in UI automation. To avoid this selenium provides different types of waits out of which Implicit and explicit waits are most commonly used.

1. Implicit Waits -

   An implicit wait when used is set to the WebDriver instance and is applied to all the web elements. In implicit wait the webdriver polls the DOM to check the availability of the webElement and waits till the maximum time specified before throwing NoSuchElementException.

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(20, TimeUnit.SECONDS);
```

In the above code snippet, the value 20 specified in implicit wait method is the maximum time in seconds till which webDriver will wait before throwing NoSuchElementException while locating a webElement.

2. Explicit Waits -

Unlike implicit waits, the explicit waits are applied to each and every webElement. In explicit wait, certain conditions are defined for which the webDriver instance waits before locating webElements or performing actions on them. Some of the most common conditions specified in explicit waits are- elementToBeClickable, presenceOfElementLocated etc.

```
WebDriverWait wait = new WebDriverWait(driver, 15);
wait.until(ExpectedConditions.presenceOfElementLocated(ElementLocator));
```

Here the webDriver instance will wait until the condition specified is met i.e. the presence Of Element located by the *ElementLocator* with the maximum wait time of 15 seconds after which if the condition is still not met than it will throw exception.

CSS LOCATORS IN SELENIUM TUTORIAL

In this tutorial, we will learn about CSS Selector and create CSS selectors manually using different HTML attributes of the web elements. For fetching the HTML information of the web elements we will use firebug or developer tool.

CSS AND CSS SELECTORS INTRODUCTION

CSS stands for Cascading Style Sheets, these are used for styling the different elements of an HTML webpage. In the .css files we can locate specific elements of a webpage(.html files) and then style them like - set their font size, width, height etc.
For locating the web elements to be styled, we use certain rules provided as CSS Selectors.
For example, the following statement first locates a web element satisfying the selector pattern - "div#searchBox" and then aligns the text inside it to center.

```
div#searchBox {text-align: center;}
```

In Selenium, we can use these CSS Selector rules/patterns for locating web elements and later perform different operations on them. For example-

```
//Locating searchBox element using CSS Selector
WebElement searchBox = driver.findElement(By.cssSelector("div#searchBox"));

//Performing click operation on the element
searchBox.sendKeys("ArtOfTesting");
```

Let's now see the different rules of CSS Selectors along with their syntax and usage example.

CSS SELECTORS

Below is the syntax and examples on how to locate the desired elements and use them in selenium scripts.

USING ID

CSS Selector Rule - #id
Example -
For the Sample HTML below-

```
<button id="submitButton1" type="button" class="btn">Submit</button>
```

CSS Locator - #submitButton1
Description - '#submitButton1' will select the element with id 'submitButton1'.

USING CLASS

CSS Selector Rule - .class
Example -
For the Sample HTML below-

```
<button id="submitButton1" type="button" class="btn">Submit</button>
```

CSS Locator - .btn
Description - '.btn' will select all the elements with class 'btn'.

USING TAG

CSS Selector Rule - tagName
Example -
For the Sample HTML below-

```
<input id="fname" type="text" name="firstName" class="textbox">
```

CSS Locator - input
Description - 'input' will select all the input type elements.

USING ATTRIBUTES AND THEIR VALUE

CSS Selector Rule - [attributeName='attributeValue']
Example -
For the Sample HTML below-

```
<input id="fname" type="text" name="firstName" class="textbox">
```

CSS Locator - [name='firstName']
Description - [name='firstName'] will select the elements with name attribute having value 'firstName'.

Now, using these basic rule of locating web elements, we can use them in conjunction to create more robust locators, selecting unique elements.

USING TAGS AND ID

CSS Selector Rule - tag#id
Example -
For the Sample HTML below-

```
<input id="fname" type="text" name="firstName" class="textbox">
```

CSS Locator - input#fname
Description - input#fname will select the 'input' element with id 'fname'.

USING TAGS AND CLASS

CSS Selector Rule - tag.class
Example -
For the Sample HTML below-

```
<input id="fname" type="text" name="firstName" class="textbox">
```

CSS Locator - input.textbox
Description - input.textbox will select the 'input' element with id 'textbox'.

USING TAGS AND ATTRIBUTES

CSS Selector Rule - tag[attributeName='attributeValue']
Example -
For the Sample HTML below-

```
<input id="fname" type="text" name="firstName" class="textbox">
```

CSS Locator - input[name='firstName']
Description - input[name='firstName'] will select the 'input' element with 'name' attribute having value 'firstName'.

LOCATING CHILD ELEMENTS (DIRECT CHILD ONLY)

CSS Selector Rule - parentLocator>childLocator
Example -
For the Sample HTML below-

```
<div id="buttonDiv" class="small">
<button id="submitButton1" type="button" class="btn">Submit</button>
</div>
```

CSS Locator - div#buttonDiv>button
Description - 'div#buttonDiv>button' will first go to div element with id 'buttonDiv' and then select its child element - 'button'.

LOCATING ELEMENTS INSIDE OTHER ELEMENTS (CHILD OR SUBCHILD)

CSS Selector Rule - locator1 locator2
Example -
For the Sample HTML below-

```
<div id="buttonDiv" class="small">
<button id="submitButton1" type="button" class="btn">Submit</button>
</div>
```

CSS Locator - div#buttonDiv button
Description - 'div#buttonDiv button' will first go to div element with id 'buttonDiv' and then select 'button' element inside it (which may be its child or sub child).

NTH CHILD

CSS Selector Rule - :nth-child(n)
Example -
For the Sample HTML below-

```
<ul id="testingTypes">
   <li>Automation Testing</li>
   <li>Performance Testing</li>
   <li>Manual Testing</li>
</ul>
```

CSS Locator - #testingTypes li:nth-child(2)
Description - '#testingTypes li:nth-child(2)' will select the element with id 'testingType' and then locate the 2nd child of type li i.e. 'Performance Testing' list item.

LOCATING SIBLINGS

CSS Selector Rule - locator1+locator2
Example -
For the Sample HTML below-

```
<ul id="testingTypes">
   <li id="automation">Automation Testing</li>
   <li>Performance Testing</li>
   <li>Manual Testing</li>
</ul>
```

CSS Locator - li#automation + li
Description - 'li#automation + li' will first go to li element with id 'automation' and then select its adjacent li i.e. 'Performance Testing' list item.

For handling dynamic elements having ids and other locators dynamically generated(not known beforehand). We can make use of the above locators by using different parent-sibling relationships of the dynamic elements. Apart from this, we can also use some special CSS locators using which we can match partial values of the attributes.

^ - STARTS WITH

CSS Selector Rule - [attribute^=attributeValue]
Example -
For the Sample HTML below-

```
<button id="user1_btn_263" type="button" class="btn">Submit</button>
```

CSS Locator - id^="user1"
Description - 'id^="user1"' will select the element whose id starts with "user1" value

$ - ENDS WITH

CSS Selector Rule - [attribute$=attributeValue]
Example -
For the Sample HTML below-

```
<button id="user1_btn_263" type="button" class="btn">Submit</button>
```

CSS Locator - id$="btn_263"
Description - 'id$="btn_263"' will select the element whose id ends with "btn_263" value

* - CONTAINS

CSS Selector Rule - [attribute*=attributeValue]
Example -
For the Sample HTML below-

```
<button id="user1_btn_263" type="button" class="btn">Submit</button>
```

CSS Locator - id*="btn"
Description - 'id*="btn"' will select the element whose id contains with "btn" value

XPATH TUTORIAL FOR SELENIUM WEBDRIVER

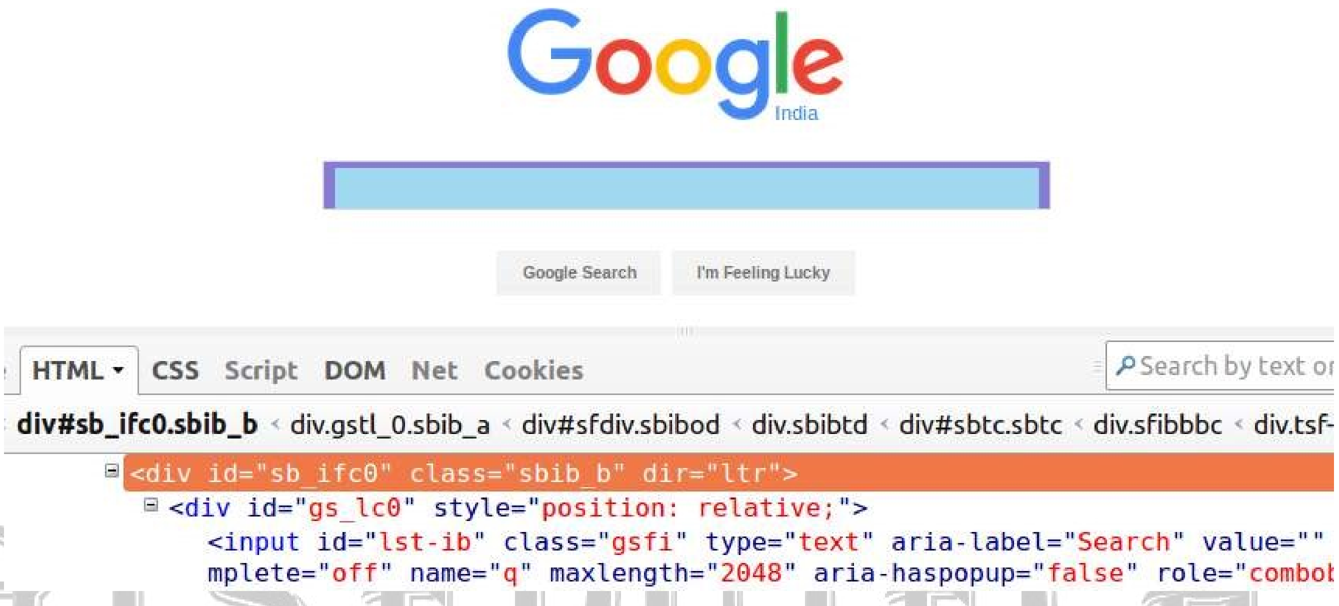
In this tutorial, we are going to study the XPath locators in Selenium WebDriver. We studied different types of locators used in Selenium WebDriver. Here, we will be studying how to create Xpath locators, the different types of Xpaths and the ways of finding dynamic elements using XPath.

WHAT IS AN XPATH?

An XPath can be defined as a query language used for navigating through the XML documents in order to locate different elements. The basic syntax of an XPath expression is-

```
//tag[@attributeName='attributeValues']
```

Now, let's understand the different elements in the Xpath expression syntax - tag, attribute and attributeValues using an example. Consider the below image of Google webpage with Firebug inspecting the Search-bar div.


Google
India
Google Search
I'm Feeling Lucky
HTML
CSS
Script
DOM
Net
Cookies
Search by text or
div#sb_ifc0.sbib_b < div.gstl_0.sbib_a < div#sfdiv.sbibod < div.sbibtd < div#sbtc.sbtc < div.sfibbbc < div.tsf-

```
<div id="sb_ifc0" class="sbib_b" dir="ltr">
  <div id="gs_lc0" style="position: relative;">
    <input id="lst-ib" class="gsfi" type="text" aria-label="Search" value=""
    mplete="off" name="q" maxlength="2048" aria-haspopup="false" role="combob
```

- **'/' or '//'** - The single slash and double slash are used to create absolute and relative XPaths(explained later in this tutorial). Single slash is used to start the selection from root node. Whereas, the double slash is used to fetch the current node matching the selection. For now, we will be using '//' here.

- **Tag** - Tags in HTML begin with '<' and end with '>'. These are used to enclose different elements and provide information about processing of the elements. In the above image, 'div' and 'input' are tags.
- **Attribute** - Attributes define the properties that the HTML elements hold. In the above image, id, classes and dir are the attributes of the outer div.
- **AttrbuteValue** - AttributeValues as the name suggest, are the values of the attributes e.g. 'sb_ifc0' is the attribute value of 'id'.

Using the XPath syntax displayed above, we can create multiple XPath epressions for the google searchbar-div given in the image like- **//div[@id='sb_ifc0']**, **//div[@class='sbib_b']** or **//div[@dir='ltr']**. Any of these expressions can be used to fetch the desired element as long as the attributes chosen are unique.

WHAT ARE THE DIFFERENT TYPES OF XPATH?

There are two kinds of XPath expressions-

1. **Absolute XPath** - The XPath expressions created using absolute XPaths begins the selection from the root node. These expression either begin with the '/' or the root node and traverse the whole DOM to reach the element.
2. **Relative XPath** - The relative XPath expressions are a lot more compact and use forward double slashes '//'. These XPaths can select the elements at any location that matches the selection criteria and doesn't necessarily begin with root node.

So, which one of the two is better?- The relative XPaths are considered better because these expressions are easier to read and create; and also more robust. The problem with absolute XPaths is, even a slight change in the DOM from the path of root node to the desired element can make the XPath invalid.

FINDING DYNAMIC ELEMENTS USING XPATHS

Many a times in automation, we either don't have unique attributes of the elements that uniquely identify them or the elements are dynamically generated with the attribute's value not known beforehand. For cases like these, XPath provide different methods of locating elements like - using the text written over the elements; using element's index; using partially matching attribute value; by moving to sibling, child or parent of an element which can be uniquely identified etc.

USING TEXT()

Using text(), we can locate an element based on the text written over it e.g. XPath for the 'GoogleSearch' button - **//*[text()='Google Search']** (we used '*' here to match any tag with the desired text)

USING CONTAINS()

The contains(), we can match even the partially matching attributes values. This is particularly helpful for locating dynamic values whose some part remains constant e.g. XPath for the outer div in the above image having id as 'sb_ifc0' can be located even with partial id-'sb' using contains() - **//div[contains(@id,'sb')]**

USING ELEMENT'S INDEX

By providing the index position in the square brackets, we can move to the nth element satisfying the condition e.g. **//div[@id='elementid']/input[4]** will fetch the fourth input element inside the div element.

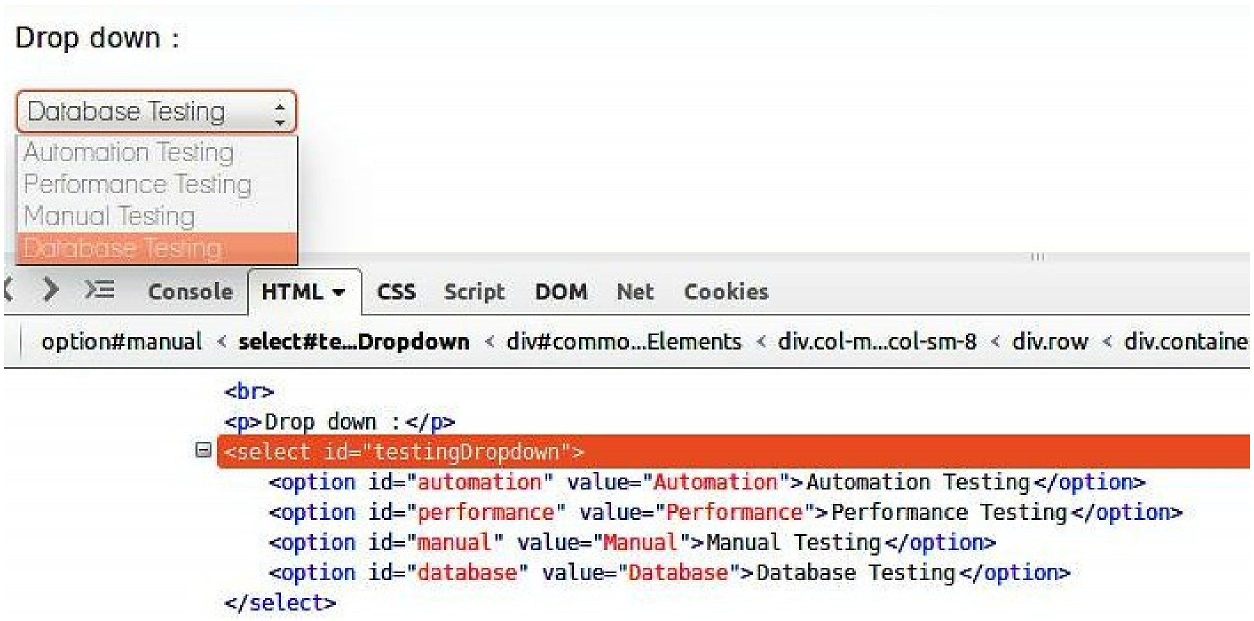
USING XPATH AXES

XPath axes helps in locating complex web elements by traversing them through sibling, child or parent of other elements which can be identified easily. Some of the widely used axes are-

- **child** - To select the child nodes of the reference node. Syntax - XpathForReferenceNode/child::tag
- **parent** - To select the parent node of the reference node. Syntax - XpathForReferenceNode/parent::tag
- **following** - To select all the nodes that come after the reference node. Syntax - XpathForReferenceNode/following::tag
- **preceding** - To select all the nodes that come before the reference node. Syntax - XpathForReferenceNode/preceding::tag
- **ancestor** - To select all the ancestor elements before the reference node. Syntax - XpathForReferenceNode/ancestor::tag

HANDLING DROP-DOWNS IN SELENIUM WEBDRIVER

In this tutorial, we are going to study the handling of dropdowns in Selenium WebDriver. For practicing, you can check the dummy page having a dropdown element.



```
<br>
<p>Drop down :</p>
<select id="testingDropdown">
    <option id="automation" value="Automation">Automation Testing</option>
    <option id="performance" value="Performance">Performance Testing</option>
    <option id="manual" value="Manual">Manual Testing</option>
    <option id="database" value="Database">Database Testing</option>
</select>
```

.

SELECT IN SELENIUM WEBDRIVER

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 testDropDown = driver.findElement(By.id("testingDropdown"));
Select dropdown = new Select(testDropDown);
```

SELECTING OPTIONS FROM DROPDOWN

There are three ways of selecting options from dropdown-

1. **selectByIndex** - To select an option based on its index, beginning with 0.

```
2. dropdown.selectByIndex(3);
```

3. **selectByValue** - To select an option based on its 'value' attribute.

```
4. dropdown.selectByValue("Database");
```

5. **selectByVisibleText** - To select an option based on the text over the option.

```
dropdown.selectByVisibleText("Database Testing");
```

DIFFERENT UTILITY METHODS IN THE SELECT CLASS

- **deselectAll()** - To deselect all the selected options.
- **deselectByIndex(int index)** - To deselect the option based on its index.
- **deselectByValue(String valueAttribute)** - To deselect the option its 'value' attribute.
- **deselectByVisibleText(String text)** - To deselect the option based on the text over the option.
- **getOptions()** - To return list of all the options(List<WebElement>).
- **getAllSelectedOptions()** - To return the list of all the selected options(List<WebElement>).
- **getFirstSelectedOption()** - To return the selected option or the first selected option in case of dropdowns allowing multi-select.
- **isMultiple()** - To return a boolean value, checking if the dropdown allows multiple option select or not.

RIGHT CLICK IN SELENIUM WEBDRIVER

Hello friends, quite often during automation we need to right click or context click an element. Later, this action is followed up by pressing the UP/DOWN arrow keys and ENTER key to select the desired context menu element (check our tutorial on pressing the non-text keys in selenium - Pressing ARROW KEYS, FUNCTION KEYS and other non-text keys in Selenium). For right clicking an element in Selenium, we make use of the Actions class. The Actions class provided by Selenium Webdriver is used to generate complex user gestures including right click, double click, drag and drop etc.

CODE SNIPPET TO RIGHT CLICK AN ELEMENT

```
Actions action = new Actions(driver);
WebElement element = driver.findElement(By.id("elementId"));
action.contextClick(element).perform();
```

Here, we are instantiating an object of Actions class. After that, we pass the WebElement to be right clicked as parameter to the contestClick() method present in the Actions class. Then, we call the perform() method to perform the generated action.

SAMPLE CODE TO RIGHT CLICK AN ELEMENT

For the demonstration of the right click action, we will be launching Sample Site for Selenium Learning. Then we will right-click a textbox after which it's context menu will get displayed, asserting that right click action is successfully performed.

```
package seleniumTutorials;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.interactions.Actions;

public class RightClick {

	public static void main(String[] args) throws InterruptedException{
		WebDriver driver = new FirefoxDriver();

		//Launching Sample site
		driver.get("http://artoftesting.com/sampleSiteForSelenium.html");

		//Right click in the TextBox
		Actions action = new Actions(driver);
		WebElement searchBox = driver.findElement(By.id("fname"));
		action.contextClick(searchBox).perform();

		//Thread.sleep just for user to notice the event
		Thread.sleep(3000);

		//Closing the driver instance
		driver.quit();
	}
}
```

DOUBLE CLICK IN SELENIUM WEBDRIVER

Hello friends! in this post, we will learn to double click an element using Selenium Webdriver with Java. For double clicking an element in Selenium we make use of the Actions class. The Actions class provided by Selenium Webdriver is used to generate complex user gestures including right click, double click, drag and drop etc.

CODE SNIPPET TO DOUBLE CLICK AN ELEMENT

```
Actions action = new Actions(driver);
WebElement element = driver.findElement(By.id("elementId"));
action.doubleClick(element).perform();
```

Here, we are instantiating an object of Actions class. After that, we pass the WebElement to be double clicked as parameter to the doubleClick() method present in the Actions class. Then, we call the perform() method to perform the generated action.

SAMPLE CODE TO DOUBLE CLICK AN ELEMENT

For the demonstration of the double click action, we will be launching Sample Site for Selenium Learning. Then we will double click the button on which the text "Double-click to generate alert box" is written. After that an alet box will appear, asserting that double-click action is successfully performed.

```
package seleniumTutorials;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.interactions.Actions;

public class DoubleClick {

        public static void main(String[] args) throws InterruptedException{
                WebDriver driver = new FirefoxDriver();

                //Launching sample website
                driver.get("http://artoftesting.com/sampleSiteForSelenium.html");
                driver.manage().window().maximize();

                //Double click the button to launch an alertbox
                Actions action = new Actions(driver);
                WebElement btn = driver.findElement(By.id("dblClkBtn"));
                action.doubleClick(btn).perform();

                //Thread.sleep just for user to notice the event
                Thread.sleep(3000);

                //Closing the driver instance
                driver.quit();
        }
}
```

MOUSEOVER IN SELENIUM WEBDRIVER

Hello friends! in this post, we will learn to automate the mouseover over an element using Selenium Webdriver with Java. For performing the mouse hover over an element in Selenium, we make use of the Actions class. The Actions class provided by Selenium Webdriver is used to generate complex user gestures including mouseover, right click, double click, drag and drop etc.

CODE SNIPPET TO MOUSEOVER

```
Actions action = new Actions(driver);
WebElement element = driver.findElement(By.id("elementId"));
action.moveToElement(element).perform();
```

Here, we are instantiating an object of Actions class. After that, we pass the WebElement to be mouse hovered as parameter to the moveToElement() method present in the Actions class. Then, we will call the perform() method to perform the generated action.

SAMPLE CODE TO MOUSE HOVER OVER AN ELEMENT

For the demonstration of the mouseover action, we will be launching Sample Site for Selenium Learning. Then we will mouse hover over the 'Submit' button. This will resut in a tooltip generation, asserting that the mouseover action is successfully performed.

```
package seleniumTutorials;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.interactions.Actions;

public class MouseOver {

        public static void main(String[] args) throws InterruptedException{
                WebDriver driver = new FirefoxDriver();

                //Launching sample website
                driver.get("http://artoftesting.com/sampleSiteForSelenium.html");
                driver.manage().window().maximize();

                //Mouseover on submit button
                Actions action = new Actions(driver);
                WebElement btn = driver.findElement(By.id("idOfButton"));
                action.moveToElement(btn).perform();

                //Thread.sleep just for user to notice the event
                Thread.sleep(3000);
                //Closing the driver instance
                driver.quit();
        }

}
```

DRAG AND DROP IN SELENIUM WEBDRIVER

Drag and Drop is one of the common scenarios in automation. In this tutorial, we are going to study the handling of drag and drop event in Selenium WebDriver using **Actions** class. For practicing, you can check the dummy page having a draggable web element.

ACTIONS IN SELENIUM WEBDRIVER

For performing a complex user gestures like drag and drop, we have a **Actions** class in Selenium WebDriver. Using the Actions class, we first build a sequence of composite events and then perform it using **Action** (an interface which represents a single user-interaction). The different methods of Actions class we will be using here are-

- **clickAndHold(WebElement element)** - Clicks a web element at the middle(without releasing).
- **moveToElement(WebElement element)** - Moves the mouse pointer to the middle of the web element without clicking.
- **release(WebElement element)** - Releases the left click (which is in pressed state).
- **build()** - Generates a composite action

CODE SNIPPET TO PERFORM DRAG AND DROP

Below is the code snippet for performing drag and drop operation-

```
//WebElement on which drag and drop operation needs to be performed
WebElement fromWebElement = driver.findElement(By Locator of fromWebElement);

//WebElement to which the above object is dropped
WebElement toWebElement = driver.findElement(By Locator of toWebElement);

//Creating object of Actions class to build composite actions
Actions builder = new Actions(driver);

//Building a drag and drop action
Action dragAndDrop = builder.clickAndHold(fromWebElement)
                      .moveToElement(toWebElement)
                      .release(toWebElement)
              .build();

//Performing the drag and drop action
dragAndDrop.perform();
```

HANDLING ALERTS IN SELENIUM WEBDRIVER

Alerts can be handled in selenium webdriver using the Alert class. Users can refer to the alert and confirm box in this sample webpage. Click on "Generate Alert Box" and "Generate Confirm Box" buttons to generate the alert and confirm boxes. The script below will first generate alert boxes by clicking on these buttons and use alert class methods to accept/dismiss them.

```
public void alertAutomation() throws InterruptedException{

        //Create firefox driver's instance
        WebDriver driver = new FirefoxDriver();
        //Set implicit wait of 10 seconds
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

        //Launch sampleSiteForSelenium
        driver.get("http://www.artoftesting.com/sampleSiteForSelenium.html");

        //Handling alert boxes
        //Click on generate alert button
        driver.findElement(By.cssSelector("div#AlertBox button")).click();

        Thread.sleep(3000);

        //Using Alert class to first switch to or focus to the alert box
        Alert alert = driver.switchTo().alert();

        //Using accept() method to accep the alert box
        alert.accept();

        //Handling confirm box
        //Click on Generate Confirm Box
        driver.findElement(By.cssSelector("div#ConfirmBox button")).click();

        Thread.sleep(3000);

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

        //Using dismiss() command to dismiss the confirm box
        //Similarly accept can be used to accept the confirm box
        confirmBox.dismiss();
}
```

SCROLLING IN SELENIUM WEBDRIVER

Scrolling a webpage is required in automation when the application requires scrolling down or up to display additional information e.g. most of the e-commerce sites display only 10-20 products at a time and then load more products as the user scrolls down. In this tutorial, we'll take example of an e-commerce website - flipkart and automate the scrolling of webpage in order to fetch more results.

In automation first we will launch filpkart.com, write a search term and then scroll down to fetch more results corresponding to that search term. Automating page scrolling will make use of "scrollBy" method of javascript. For executing the javascript method we will use Javascript executor. The scrollBy method takes two parameters one each for horizontal and vertical scroll in terms of pixels.

```
JavascriptExecutor js = (JavascriptExecutor)driver;
js.executeScript("scrollBy(0, 2500)");
```

Following test script automates flipkart's scroll down functionality to test display of new pages on the search result page as the user scrolls down.

```
@Test
public void testScroll() throws InterruptedException{

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

        //Write the search term - Buddha in search box
        WebElement searchBox = driver.findElement(By.id("fk-top-search-box"));
        searchBox.sendKeys("Buddha");

        //Click on searchButton
        WebElement searchButton = driver.findElement(By.className("search-bar-submit
"));
        searchButton.click();

        //Inserting an optional wait of 3 seconds just to notice scroll down event
        Thread.sleep(3000);

        //Scroll down the webpage by 2500 pixels
        JavascriptExecutor js = (JavascriptExecutor)driver;
        js.executeScript("scrollBy(0, 2500)");

        //Waiting till page:2 text is visible
        WebElement pageNumberdisplayer = (new WebDriverWait(driver, 10)).until
          (ExpectedConditions.presenceOfElementLocated(By.cssSelector("div.row")));

        //Verifying that page got scrolled  and "page-2" text is visible now
        //and more products become visible
        Assert.assertEquals(pageNumberdisplayer.getText(), "Page: 2");
}
```

NAVIGATE BACK AND FORWARD IN BROWSER

During automation we are at times required to move back to the previuos page or move forward to the next page in browser history. In this post we will learn to perform these operations using driver.navigate command.

NAVIGATE BACK IN BROWSER HISTORY

```
driver.navigate().back();
```

NAVIGATE FORWARD IN BROWSER HISTORY

```
driver.navigate().forward();
```

SAMPLE CODE FOR DEMONSTRATION

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

        //Setting system property required for launching chrome browser
        System.setProperty("webdriver.chrome.driver", "driverExecutables\\chromedriv
er.exe");

        //Launching chrome browser
        WebDriver driver = new ChromeDriver();

        //Navigating to the desired website
        driver.get("http://artoftesting.com/sampleSiteForSelenium.html");

        //Used for demo purpose only, not required
        Thread.sleep(4000);

        //Clicking a link
        WebElement artOfTestingLogo = driver.findElement(By.cssSelector("div.navbar-
header"));
        artOfTestingLogo.click();

        //Navigating back in browser
        driver.navigate().back();

        //Used for demo purpose only, not required
        Thread.sleep(4000);

        //Navigating forward in browser
        driver.navigate().forward();

}
```

REFRESH A WEBPAGE IN SELENIUM

There are multiple ways of refreshing a page in Selenium Webdriver. In this post, we will present all these ways and also specify which one is the best.

1. Using driver.navigate command- Selenium Webdriver provides inherent support for refreshing a webpage using its driver.navigate command. This is by far the most preferred and widely used way of refreshing a webpage.

```
2.  driver.navigate().refresh();
```

3. Opening current URL using driver.getCurrentUrl() with driver.get() command-

```
4.  driver.get(driver.getCurrentUrl());
```

5. Opening current URL using driver.getCurrentUrl() with driver.navigate() command-

```
6.  driver.navigate().to(driver.getCurrentUrl());
```

7. Pressing F5 key on any textbox using sendKeys command-

```
8.  driver.findElement(By textboxLocator).sendKeys(Keys.F5);
```

9. Passing ascii value of F5 key i.e. "\uE035" using sendKeys command-

```
10. driver.findElement(By textboxLocator).sendKeys("\uE035");
```

MAXIMIZE OR MINIMIZE A BROWSER

When we launch a browser using Selenium WebDriver, by default it is not in its maximized state. In this post, we will see how to maxmize and minimize a browser during automation.

MAXMIZE A BROWSER

During automation its one of the best practices to maximize the browser in the initial phases (or in the @BeforeTest method in case you are using TestNG as the testing framework). The following command can be used to maximize the browser window.

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

MINIMIZE A BROWSER

Unfortunately Selenium WebDriver doesn't provide any native supoort for minimizing a browser. If we don't want to see the browser in action during automation than we can **use Headless browsers like HTMLUnitDriver, PhantomJS** etc. But in case we specifically want to minimize a browser during automation than we can make use of the below statement. The following code snippet will position the browser to an area that is not within the viewable section of the machine we are working on. Thus making it appear as if the browser window got minimized.

```
driver.manage().window().setPosition(new Point(0, -1000));
```

SET BROWSER'S HEIGHT AND WIDTH USING SELENIUM

At times during automation we may be required to test an application with some specific browser window dimensions. In this post we will be studying how to resize a browser to an exact dimension.
In order to achieve this, we will use the **setSize()** method which takes the **Dimension** object as input. The Dimension object has width and height

integer fields. Hence, we can set the width and height of the browser by calling the setSize method with the required dimensions of the browser window.

CODE SNIPPET TO RESIZE A BROWSER

During automation its one of the best practices to maximize the browser in the initial phases (or in the BeforeTest method in case you are using TestNG as the testing framework). Selenium provides direct command to maximize the browser window. The following command is use to maximize the browser window.

```java
int width = 600;
int height = 400;
Dimension dimension = new Dimension(width, height);
driver.manage().window().setSize(dimension);
```

SCREENSHOTS IN SELENIUM WEBDRIVER

While doing manual testing, we always have the machine in front of us to check what happened when the test case failed. In automation, we rely on the assertion messages that we print in case of failure. In addition to that, we can also have screenshot of the browser in case of failure due to assertion or unavailability of any web element.

HOW TO TAKE SCREESHOT

The code to take the screenshot make use of **getScreenshotAs** method of **TakesScreenshot** interface. Following code will take screenshot of the web page opened by webDriver instance.

```
File scrFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
FileUtils.copyFile(scrFile, new File("D:\\testScreenShot.jpg"));
```

TAKE SCREENSHOT ON FAILURE

Now in order to take screenshot in case of test failure we will use @AfterMethod annotation of TestNG. In the @AfterMethod annotation we will use **ITestResult**interface's **getStatus()** method that returns the test result and in case of failure we can use the above commands to take screenshot. One more thing to mention here is in order to uniquely identify the screenshot file, we are naming it as the name of the test method appended with the test parameters (passed through a data-provider). For getting the test name and parameters we are using the getName() and getParameters() methods of ITestResult interface. In case you are not using any data-provider(like in case of this demo) then you can just have the getName() method to print the test method name.

```
@AfterMethod
public void takeScreenShotOnFailure(ITestResult testResult) throws IOException {
        if (testResult.getStatus() == ITestResult.FAILURE) {
                File scrFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.
FILE);
                FileUtils.copyFile(scrFile, new File("errorScreenshots\\" + testResu
lt.getName() + "-"
                                + Arrays.toString(testResult.getParameters()) +  ".j
pg"));
        }
}
```

DEMO WITH GOOGLE CALCULATOR

Following is the complete sample script for google calculator test that is intentionally made to fail by asserting result for 2+2 as 5. Just change the path of the screenshot file to the desired location and run the test script.

```
public class ScreenshotDemo{

        String driverExecutablePath = "lib\\chromedriver.exe";
        WebDriver driver;

        @BeforeTest
        public void setup(){
                System.setProperty("webdriver.chrome.driver", driverExecutablePath);
                driver = new ChromeDriver();

                //Set implicit wait of 3 seconds
                driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);
        }


        @Test
        //Tests google calculator
        public void googleCalculator() throws IOException{

                //Launch google
                driver.get("http://www.google.co.in");

                //Write 2+2 in google textbox
                WebElement googleTextBox = driver.findElement(By.id("lst-ib"));
                googleTextBox.sendKeys("2+2");

                //Hit enter
                googleTextBox.sendKeys(Keys.ENTER);

                //Get result from calculator
                WebElement calculatorTextBox = driver.findElement(By.id("cwtltblr"))
;
                String result = calculatorTextBox.getText();

                //Intentionaly checking for wrong calculation of 2+2=5 in order to t
ake screenshot for failing test
                Assert.assertEquals(result, "5");
        }

        @AfterMethod
        public void takeScreenShotOnFailure(ITestResult testResult) throws IOExceptio
n {
                if (testResult.getStatus() == ITestResult.FAILURE) {
                        System.out.println(testResult.getStatus());
```

```
                            File scrFile = ((TakesScreenshot)driver).getScreenshotAs(Out
putType.FILE);
                            FileUtils.copyFile(scrFile, new File("errorScreenshots\\" +
testResult.getName() + "-"
                                              +  Arrays.toString(testResult.getParameters(
)) +  ".jpg"));
            }
         }

}
```

DESIRED CAPABILITIES IN SELENIUM WEBDRIVER

Hello friends! in this post we will be studying about Desired Capabilities. Many a times during automation, we may need to work on a particular session of browser or work with a browser having some specific configurable properties set or unset. Selenium web driver provides certain browser specific properties known as **Desired Capabilities**.

WHAT ARE DESIRED CAPABILITIES?

Desired Capabilities are a set of properties used to configure a particular instance of browser. We have both browser independent capabilties like browserName, version, acceptSslCerts etc as well browser specific capabilities like firefox_binary for Mozilla Firefox, chromeOptions for setting Chrome specific desired capabilties.

SETTING DESIRED CAPABILITIES IN SELENIUM

The Desired Capabilities class provides a setCapabilityMethod() to set the different capabilities in a browser like accepting SSL certificates, enabling javascript, querying the browser location is allowed or not etc. Code snippet to set a capability to a browser instance -

```
//Specifying desired capabilities for Chrome browser
DesiredCapabilities acceptSSLCertificate = DesiredCapabilities.chrome();

//Setting capability to accept SSL certificates
acceptSSLCertificate.setCapability(CapabilityType.ACCEPT_SSL_CERTS, true);

//Binding the capabilities to a new instance of chrome browser
WebDriver driver = new ChromeDriver(acceptSSLCertificate);
```

CAPABILITIES

Some of the capabilities common to all browsers are -

- **acceptSslCerts-**
  Constant Field - CapabilityType.ACCEPT_SSL_CERTS
  For enabling or a disabling a browser session to accept the SSL certificates by default.

- **applicationCacheEnabled-**
  Constant Field - CapabilityType.SUPPORTS_APPLICATION_CACHE
  For checking if the browser instance is capable of interacting with application cache.

- **cssSelectorsEnabled-**
  Constant Field - CapabilityType.SUPPORTS_FINDING_BY_CSS
  For checking if the use of CSS selector for locating web elements is enabled or not.

- **javascriptEnabled-**
  Constant Field - CapabilityType.SUPPORTS_JAVASCRIPT
  For checking if javascript execution is enabled or not in the browser instance.

- **takesScreenshot-**
  Constant Field - CapabilityType.TAKES_SCREENSHOT
  For checking if the take screenshot ability is enabled or not.

- **webStorageEnabled-**
  Constant Field - CapabilityType.SUPPORTS_WEB_STORAGE
  For checking if the browser instance is capable of interacting with storage objects.

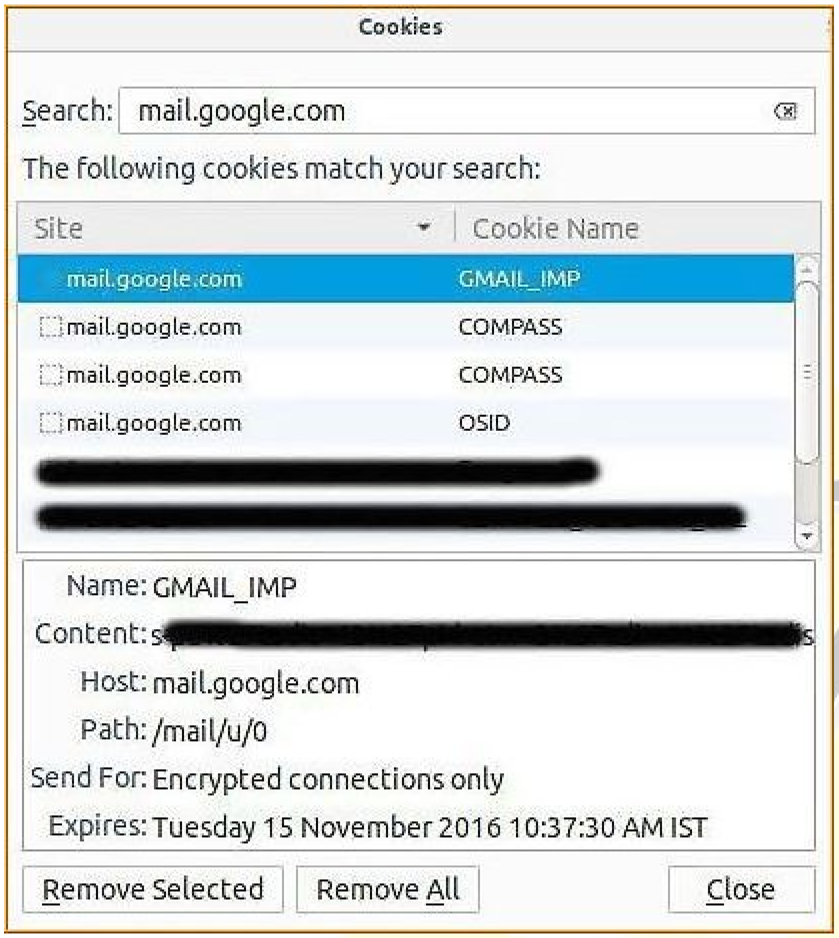
- **handlesAlert-**
  Constant Field - CapabilityType.SUPPORTS_ALERTS
  For checking if the browser instance is capable of handling window pop-ups.

HANDLING COOKIES IN SELENIUM WEBDRIVER

This post is dedicated to handling of cookies using Selenium webdriver. We will be studying about cookies, the need of automating cookie handling scenarios and different methods provided by Selenium to add, get and delete cookies.

WHAT ARE COOKIES?

Cookies are files stored in local computers containing information submitted by the websites visited by a user. The information is stored in key-value pairs and allows a particular website to customize its content as per the user.



Search: mail.google.com

The following cookies match your search:

Site	Cookie Name
mail.google.com	GMAIL_IMP
mail.google.com	COMPASS
mail.google.com	COMPASS
mail.google.com	OSID
[redacted]	
[redacted]	

Name: GMAIL_IMP
Content: [redacted]
Host: mail.google.com
Path: /mail/u/0
Send For: Encrypted connections only
Expires: Tuesday 15 November 2016 10:37:30 AM IST

NEED OF HANDLING COOKIES IN AUTOMATION

During automation we might need to handle cookies for following reasons-

1. There could be scenarios specific to cookies which needs to be automated and requires validating enteries in cookie files.
2. If we could create and load cookie files during automation, we can skip some scenarios like login to application for each test case as the same gets handled by cookies. E.g. in some application like GMail, we only need to login once, after that the login information gets stored in cookies and loaded each time we open gmail link. This saves the overall test execution time.

GET COOKIES IN SELENIUM

We can get Cookies in Selenium WebDriver in two ways-

1. **getCookieNamed(String cookieName)** - To get a particular Cookie by name

2. **getCookies()** - To get all the Cookies in the current domain.

```
//To fetch a Cookie by name
System.out.println(driver.manage().getCookieNamed(cookieName).getValue());

//To return all the cookies of the current domain
Set<Cookie> cookiesForCurrentURL = driver.manage().getCookies();
for (Cookie cookie : cookiesForCurrentURL) {
    System.out.println(" Cookie Name - " + cookie.getName()
                        + " Cookie Value - "  + cookie.getValue()));
}
```

ADD COOKIES IN SELENIUM

addCookie(Cookie cookie) - To add a particular Cookie in Selenium WebDriver.

```
Cookie cookie = new Cookie("cookieName", "cookieValue");
driver.manage().addCookie(cookie);
```

DELETE COOKIES IN SELENIUM

Delete Cookies in Selenium WebDriver can be done in three ways-

1. **deleteCookieNamed(String cookieName)** - To delete a particular Cookie by name.
2. **deleteCookie(Cookie cookie)** - To delete a particular Cookie.
3. **deleteAllCookies()** - To delete all the Cookies.

```
//Delete a particular Cookie by name
driver.manage().deleteCookieNamed(cookieName);

//Delete a particular Cookie
driver.manage().deleteCookie(cookie);

//Delete all the Cookies
driver.manage().deleteAllCookies();
```

JAVASCRIPTEXECUTOR IN SELENIUM WEBDRIVER

In this post, we will study - how to execute javascript code in Selenium webdriver and find the scenarios where we need to use the javascript code instead of the different methods of selenium.

NEED OF EXECUTING JAVASCRIPT CODE

JavaScript is a lightweight interpreted language for client side scripting. In automation, we might need JavaScript code execution for following reasons-

1. JavaScript executor is particularly used to handle scenarios including hidden web elements which cannot be located by Selenium WebDriver locators.
2. For handling test scenarios requiring explicit javascript code execution.
3. For performing operations like - "Scrolling a web page" which can be easily performed using Javascript.

JAVASCRIPTEXECUTOR IN SELENIUM WEBDRIVER

The JavaScriptExecutor is an interface in Selenium WebDriver that is implemented by FirefoxDriver, InternetExplorerDriver, ChromeDriver and other driver classes. Using JavaScriptExecutor, we can execute a JavaScript code with Selenium WebDriver.

METHODS OF JAVASCRIPTEXECUTOR

The JavaScriptExecutor provides two methods of javaScript code injection in browser-
1. executeScript() - To run the specified JavaScript code in the current window or frame.

```
JavascriptExecutor jsExecutor = (JavascriptExecutor)driver;
jsExecutor.executeScript("JavaScriptCode");
```

2. executeAsyncScript() - To run specified asynchronous JavaScript code in the current window or frame. As the javaScript runs asynchronously, it requires an explicit callback indicating the finishing of script execution.

```
JavascriptExecutor jsExecutor = (JavascriptExecutor)driver;
jsExecutor.executeAsyncScript("Async JavaScript Code");
```

SAMPLE SCRIPT FOR JAVASCRIPTEXECUTER

Scrolling a web page in Selenium WebDriver using JavaScriptExecuter-

```
@Test
public void testScroll() throws InterruptedException{

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

        //Write the search term - Buddha in search box
        WebElement searchBox = driver.findElement(By.id("fk-top-search-box"));
        searchBox.sendKeys("Buddha");

        //Click on searchButton
        WebElement searchButton = driver.findElement(By.className("search-bar-submit
"));
        searchButton.click();

        //Inserting an optional wait of 3 seconds just to notice scroll down event
        Thread.sleep(3000);

        //Scroll down the webpage by 2500 pixels
        JavascriptExecutor js = (JavascriptExecutor)driver;
        js.executeScript("scrollBy(0, 2500)");

        //Optional Wait
        Thread.sleep(3000);
}
```

PRESS KEYBOARD KEYS IN SELENIUM

During automation, we are often required to press enter, control, tab, arrow keys, function keys and other non-text keys as well from keyboard. In this post, we will find how to simulate pressing of these non-text keys using selenium webdriver in java. Here, we will be using **Keys** enum provided by Selenium webdriver for all the non-text keys.

PRESS ENTER/RETURN KEY IN SELENIUM

For pressing Enter key over a textbox we can pass Keys.ENTER or Keys.RETURN to the sendKeys method for that textbox.

```
WebElement textbox = driver.findElement(By.id("idOfElement"));
textbox.sendKeys(Keys.ENTER);
```

or

```
WebElement textbox = driver.findElement(By.id("idOfElement"));
textbox.sendKeys(Keys.RETURN);
```

Similarly, we can use Keys enum for different non-text keys and pass them to the sendKeys method. The following table has an entry for each of the non-text key present in a keyboard.

Keyboard's Key	Keys enum's value
Arrow Key - Down	Keys.ARROW_DOWN

Arrow Key - Up	Keys.ARROW_LEFT
Arrow Key - Left	Keys.ARROW_RIGHT
Arrow Key - Right	Keys.ARROW_UP
Backspace	Keys.BACK_SPACE
Ctrl Key	Keys.CONTROL
Alt key	Keys.ALT
DELETE	Keys.DELETE
Enter Key	Keys.ENTER
Shift Key	Keys.SHIFT
Spacebar	Keys.SPACE
Tab Key	Keys.TAB

Equals Key	Keys.EQUALS
Esc Key	Keys.ESCAPE
Home Key	Keys.HOME
Insert Key	Keys.INSERT
PgUp Key	Keys.PAGE_UP
PgDn Key	Keys.PAGE_DOWN
Function Key F1	Keys.F1
Function Key F2	Keys.F2
Function Key F3	Keys.F3
Function Key F4	Keys.F4
Function Key F5	Keys.F5

Function Key F6	Keys.F6
Function Key F7	Keys.F7
Function Key F8	Keys.F8
Function Key F9	Keys.F9
Function Key F10	Keys.F10
Function Key F11	Keys.F11
Function Key F12	Keys.F12

KEYBOARD ACTIONS IN SELENIUM WEBDRIVER

In our beginner's tutorials, we have seen the sendKeys() method which simulates the keyboard typing action on a textbox or input type element. But this method is not sufficient for handling complex keyboard actions. For this, Selenium has an **Actions**class which provides different methods for Keyboard interactions. In this tutorial, we wil be studying the three actions for handling keyboard action - keyDown(), keyUp() and sendKeys() along with their overloaded method implementations.

ACTIONS CLASS METHOD FOR KEYBOARD INTERACTION

1. **keyDown(Keys modifierKey)-** The keyDown(Keys modifierKey) method takes the modifier Keys as parameter (Shift, Alt and Control Keys - that modifies the purpose of other keys, hence the name). It is used to simulate the action of pressing a modifier key, without releasing. The expected values for the keyDown() method are - Keys.SHIFT, Keys.ALT and Keys.CONTROL only, passing key other than these results in IllegalArgumentException.

2. **keyDown(WebElement element, Keys modifierKey)-** This another implementation of keyDown() method in which the modifier key press action is performed on a WebElement.

3. **keyUp(Keys modifierKey)-** The keyUp() method is used to simulate the modifier key-up or key-release action. This method follows a preceeding key press action.

4. **keyUp(WebElement element, Keys modifierKey)-** This implementation of keyUp() method performs the key-release action on a web element.

5. **sendKeys(CharSequence KeysToSend)-** The sendKeys(CharSequence KeysToSend) method is used to send a sequence of keys to a currently focussed web element. Here, we need to note that it is different from the webElement.sendKeys() method. The Actions sendKeys(CharSequence KeysToSend) is particularly helpful when dealing with modifier keys as it doesn't release those keys when passed(resulting in correct behaviour) unlike the webElement.sendKeys() method.

6. **sendKeys(WebElement element, CharSequence KeysToSend)-** This implementation of sendKeys() method is used to send a sequence of keys to a web element.

CODE SNIPPET FOR KEYBOARD ACTIONS

```
//WebElement to which the keyboard actions are performed
WebElement textBoxElement = driver.findElement(By Locator of textBoxElement);

//Creating object of Actions class
Actions builder = new Actions(driver);

//Generating an action to type a text in CAPS
Action typeInCAPS = builder.keyDown(textBoxElement, Keys.SHIFT)
```

```
                        .sendKeys(textBoxElement, "artOfTesting")
                        .keyUp(textBoxElement, Keys.SHIFT)
                .build();

//Performing the typeInCAPS action
typeInCAPS.perform();
```

In this tutorial, we will be studying the advanced mouse interactions using **Actions**class. Using these methods,we can perform mouse operations like right click, double click, mouse hover, click and hold etc.

ACTIONS CLASS METHOD FOR MOUSE INTERACTIONS

1. **click()-**
   This method is used to click at the current mouse pointer position. It is particularly useful when used with other mouse and keyboard events, generating composite actions.

2. **click(WebElement webElement)-**
   This method is used to click at the middle of a web element passed as parameter to the click() method.

3. **clickAndHold()-**
   The clickAndHold() method is used to perform the click method without releasing the mouse button.

4. **clickAndHold(WebElement onElement)-**
   This method performs the click method without releasing the mouse button over a web element.

5. **contextClick()-**
   This method is used to perform the right click operation(context-click) at the current mouse position.

6. **contextClick(WebElement onElement)-**
   This method performs the right click operation at a particular web element.

7. **doubleClick()-**
   As the name suggest, this method performs double click operation at a current mouse position.

8. **doubleClick(WebElement onElement)-**
   Performs the double click operation at a particular web element.

9. **dragAndDrop(WebElement fromElement, WebElement toElement)-**
   This is a utility method to perform the dragAndDrop operation directly wherein, we can pass the source element and the target element as parameter.

10. **dragAndDropBy(WebElement fromElement, int xOffset, int yOffset)-**
    This method is a variation of dragAndDrop(fromElement, toElement) in which instead of passing the target element as parameter, we pass the x and y offsets. The method clicks the source web element and then releases at the x and y offsets.

11. **moveByOffset(int xOffset, int yOffset)-**
    This method is used to move the mouse pointer to a particular position based on the

x and y offsets passed as parameter.

12. **moveToElement(WebElement toElement)-**
This method is used to move the mouse pointer to a web element passed as parameter.

13. **moveToElement(WebElement toElement, int xOffset, int yOffset)-**
This method moves the mouse pointer by the given x and y offsets from the top-left corner of the specified web element.

14. **release()-**
This method releases the pressed left mouse button at the current mouse pointer position.

15. **release(WebElement onElement)-**
This method release the pressed left mouse button at a particular web element.

CODE SNIPPET FOR A MOUSE ACTION

```
//WebElement which needs to be right clicked
WebElement rtClickElement = driver.findElement(By Locator of rtClickElement);

//Generating a Action to perform context click or right click
Actions rightClickAction = new Actions(driver).contextClick(rtClickElement);

//Performing the right click Action generated
rightClickAction.build().perform();
```

HTMLUNITDRIVER - NON GUI BROWSER IN SELENIUM

WHAT IS HTMLUNITDRIVER?

HtmlUnitDriver is headless driver providing non-GUI implementation of Selenium WebDriver. It is based on HtmlUnit, fastest and light-weight browser implemented in Java.

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

ADVANTAGES OF HTMLUNITDRIVER

1. Test execution using HtmlUnitDriver is very fast. Since, it is the fastest implementation of Selenium WebDriver.
2. Being headless and fast, it is ideal choice for web scrapping.
3. Its browser - htmlUnit is Java based. Hence, it is platform independent.
4. It also supports JavaScript execution through in-built Rhino javascript engine.

LIMITATIONS OF HTMLUNITDRIVER

1. Being non-GUI makes it difficult to create scripts and debug issues while scripting.
2. The Rhino javascript engine of HtmlUnitDriver make it unsuitable to emulate other popular browser's javascript behaviour.

CODE SNIPPET FOR HTMLUNITDRIVER

The script creation in HtmlUnitDriver is similar to scripting in any other driver, only the driver instantiation part is different. The below script, when executed will run the test in non-GUI mode without opening any browser.

```
public class HtmlUnitDriverDemo {

        @Test
        public void htmlUnitDriverTest(){

                //Instantiating HtmlUnitDriver
                WebDriver driver = new HtmlUnitDriver();
                //Navigate to ArtOfTesting.com
                driver.get("http://www.artoftesting.com");

                //Printing page title
                System.out.println(driver.getTitle());

        }

}
```

PAGE OBJECT MODEL - POM

In this tutorial, we will study about "Page Object Model", a design pattern in UI automation testing. Before starting with Page Object Model, let's first know what are design patterns.

DESIGN PATTERN

A Design pattern is a generic solution to a common software design/architecture problem. Implementation of these design patterns leads to inclusion of best practices and best solution, evolved over the time by others while working with similar problems.

PAGE OBJECT MODEL IN SELENIUM

A Page Object Model is a design pattern that can be implemented using selenium webdriver. It essentially models the pages/screen of the application as objects called Page Objects, all the functions that can be performed in the specific page are encapsulated in the page object of that screen. In this way any change made in the UI will only affect that screens page object class thus abstracting the changes from the test classes.

ADVANTAGES OF USING PAGE OBJECT MODEL

- Increases code reusability - code to work with events of a page is written only once and used in different test cases
- Improves code maintainability - any UI change leads to updating the code in page object classes only leaving the test classes unaffected
- Makes code more readable and less brittle

CREATING A PAGE OBJECT MODEL IN JAVA

Here, we'll create an automation framework implementing Page Object Model using Selenium with Java.
Suppose we have to test a dummy application with only a login page and a home page. To start with we first need to create page objects for all available pages in our application - LoginPage.java and HomePage.java. Then we will create a test class that will create instance of these page objects and invoke there methods to create tests.
Let's take the scenario where in the login page user enters valid credentials and on clicking submit button, user is redirected to home page.

CONTENT OF LOGINPAGE.JAVA

```java
public class LoginPage {

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    //Using FindBy for locating elements
    @FindBy(id = "userName")
    private WebElement userName;

    @FindBy(id = "password")
    private WebElement password;

    @FindBy(id = "submitButton")
    private WebElement submit;

    /*Defining all the user actions that can be performed in the loginPage
    in the form of methods*/

    public void typeUserName(String text) {
        userName.sendKeys(text);
    }

    public void typePassword(String text) {
        password.sendKeys(text);
    }

    /*Take note of return type of this method, clicking submit will navigate
    user to Home page, so return type of this method is marked as HomePage.*/
    public HomePage clickSubmit() {
        submit.click();
        return new HomePage(driver);
    }

    public HomePage loginWithValidCredentials(String userName, String pwd) {
        typeUserName(userName);
        typePassword(pwd);
        return clickSubmit();
    }
}
```

CONTENT OF HOMEPAGEPAGE.JAVA

```java
public class HomePage {

    public HomePage(WebDriver driver) {
        this.driver = driver;
    }

    @FindBy(id = "userInfo")
    private WebElement userInfo;

    public void clickUserInfo(){
         userInfo.click();
    }

    public String showUserInfo(){
         String userData = clickUserInfo();
         return userData;
    }
}
```

CONTENT OF TEST CLASS - POMTEST.JAVA

```java
public class POMTest{

    //Create firefox driver's instance
    WebDriver driver = new FirefoxDriver();

    @Test
    public void verifyUserInfo() {

    //Creating instance of loginPage
    LoginPage loginPage = new LoginPage(driver);

    //Login to application
    HomePage homePage = loginPage.loginWithValidCredentials("user1","pwd1");

    //Fetch user info
    String userInfo = homePage.showUserInfo();

    //Asserting user info
    Assert.assertTrue(userInfo.equalsIgnoreCase("XYZ"),"Incorrect userInfo");
    }
}
```

This was just a demo for understanding of Page Object Model, an actual project would require several updations like creating abstract classes for page objects, creating base classes for test classes, creating helper and utility classes for database connectivity, for passing test data through config files etc.

PAGEFACTORY IN SELENIUM WEBDRIVER

Hello friends! in our previous tutorial on Page Object Model, we studied about POM design pattern, its advantages and implementation in Selenium WebDriver. In this tutorial, we will study about PageFactory, an enhanced implementation of Page Object Model.

WHAT IS PAGEFACTORY?

PageFactory is class provided by Selenium WebDriver to support the Page Object design pattern. It makes handling "Page Objects" easier and optimized by providing the following-

- @FindBy annotation
- initElements method

@FINDBY-

@FindBy annotation is used in PageFactory to locate and declare web elements using different locators. Here, we pass the attribute used for locating the web element along with its value to the @FindBy annotation as parameter and then declare the element. Example-

```
@FindBy(id="elementId") WebElement element;
```

In the above example, we have used 'id' attribute to locate the web element 'element'. Similarly, we can use the following locators with @FindBy annotations.

- className
- css
- name
- xpath
- tagName
- linkText
- partialLinkText

INITELEMENTS()-

The initElements is a static method of PageFactory class which is used in conjunction with @FindBy annotation. Using the initElements method we can initialize all the web elements located by @FindBy annotation. Thus, instantiating the Page classes easily.

```
initElements(WebDriver driver, java.lang.Class pageObjectClass)
```

CODE SNIPPET FOR PAGEFACTORY

Below is a code snippet with a demo Page class. Here, the web elements are located by @FindBy annotation and the initElements method is invoked in the constructor of the PageFactoryDemoClass.
PS: The implementation of test classes will remain same (as stated in Page Object Model tutorial).

```
public class PageFactoryDemoClass {

    WebDriver driver;

    @FindBy(id="search")
    WebElement searchTextBox;

    @FindBy(name="searchBtn")
    WebElement searchButton;

    //Constructor invoking initElements method
    public PageFactoryDemoClass(WebDriver driver){
        this.driver = driver;

         //initializing all the web elements located by @FindBy
        PageFactory.initElements(driver, this);
    }

    //Sample method of the page class
    public void search(String searchTerm){
        searchTextBox.sendKeys(searchTerm);
        searchButton.click();
    }
}
```

DIFFERENCE BETWEEN ASSERT AND VERIFY

Both Assert and Verify statements are used in the test suites for adding validations to the test methods. Testing frameworks like TestNG and JUnit are used with Selenium to provide assertions.
The major difference between "Assert" and "Verify" commands is-
In case of "Assert" command, as soon as the validation fails the execution of that particular test method is stopped and the test method is marked as failed.
Whereas, in case of "Verify", the test method continues execution even after the failure of an assertion statement. Although the test method will still be marked as failed but the remaining statements of the test method will be executed normally. In TestNG, the "Verify" functionality is provided by means of "Soft Assertions" or "SoftAssert" class.

Now, let's get deeper into Assert and Verify/Soft Assert.

ASSERT

We use Assert when we have to validate critical functionality, failing of which makes the execution of further statements irrelevant. Hence, the test method is aborted as soon as failure occurs.
Example-

```
@Test
public void assertionTest(){

   //Assertion Passing
   Assert.assertTrue(1+2 == 3);
   System.out.println("Passing 1");

   //Assertion failing
   Assert.fail("Failing second assertion");
   System.out.println("Failing 2");
}
```

Output-

```
Passing 1
```

```
FAILED: assertionTest
java.lang.AssertionError: Failing second assertion
```

Here, we can observe that only the text "Passing 1" gets printed. The second assertion aborts the test method as it fails preventing further statement from getting executed.

VERIFY

At times, we might require the test method to continue execution even after the failure of the assertion statements. In TestNG, **Verify** is implemented using **SoftAssert** class.
In case of SoftAssert, all the statements in the test method are executed (including multiple assertions). Once, all the statements are executed, the test result is collated based on the assertion results and test is marked as passed or fail.
Example-

```
@Test
public void softAssertionTest(){

   //Creating softAssert object
   SoftAssert softAssert = new SoftAssert();

   //Assertion failing
   softAssert.fail("Failing first assertion");
   System.out.println("Failing 1");

   //Assertion failing
   softAssert.fail("Failing second assertion");
   System.out.println("Failing 2");

   //Collates the assertion results and marks test as pass or fail
   softAssert.assertAll();
}
```

Output-

```
Failing 1
Failing 2
FAILED: softAssertionTest
java.lang.AssertionError: The following asserts failed:
        Failing first assertion,
        Failing second assertion
```

Here, we can see that even though both the test methods are bound to fail, still the test continues to execute.

TESTNG INTRODUCTION

TestNG is a testing framework inspired from JUnit and NUnit but introducing some new functionality that make it more powerful and easier to use.

TestNG is an open source automated testing framework; where NG of TestNG means Next Generation. TestNG is similar to JUnit but it is much more powerful than JUnit but still it's inspired by JUnit. It is designed to be better than JUnit, especially when testing integrated classes. Pay special thanks to Cedric Beust who is the creator of TestNG.

TestNG eliminates most of the limitations of the older framework and gives the developer the ability to write more flexible and powerful tests with help of easy annotations, grouping, sequencing & parametrizing.

TestNG is a testing framework inspired from ***JUnit*** and ***NUnit*** but introducing some new functionality that make it more powerful and easier to use.
It is an open source automated testing framework; where **NG** OF TEST**NG** MEANS **N**EXT **G**ENERATION. TestNG is similar to JUnit but it is much more powerful than JUnit but still it's inspired by JUnit. It is designed to be better than JUnit, especially when testing integrated classes. Pay special thanks to *Cedric Beust who is the creator of TestNG*.
TestNG eliminates most of the limitations of the older framework and gives the developer the ability to write more flexible and powerful tests with help of easy annotations, grouping, sequencing & parametrizing.

BENEFITS OF TESTNG

There are number of benefits but from Selenium perspective, major advantages of TestNG are :

1. It gives the ability to produce ***HTML Reports*** of execution
2. ***Annotations*** made testers life easy
3. Test cases can be ***Grouped & Prioritized*** more easily
4. ***Parallel*** testing is possible
5. Generates ***Logs***
6. Data ***Parameterization*** is possible

TEST CASE WRITING

Writing a test in TestNG is quite simple and basically involves following steps:

Step 1 – Write the business logic of the test
Step 2 – Insert TestNG annotations in the code
Step 3 – Add the information about your test (e.g. the class names, methods names, groups names etc…) in a testng.xml file
Step 4 – Run TestNG

ANNOTATIONS IN TESTNG

@BeforeSuite: The annotated method will be run before all tests in this suite have run.
@AfterSuite: The annotated method will be run after all tests in this suite have run.
@BeforeTest: The annotated method will be run before any test method belonging to the classes inside the tag is run.
@AfterTest: The annotated method will be run after all the test methods belonging to the classes inside the tag have run.
@BeforeGroups: The list of groups that this configuration method will run before. This method is guaranteed to run shortly before the first test method that belongs to any of these groups is invoked.
@AfterGroups: The list of groups that this configuration method will run after. This method is guaranteed to run shortly after the last test method that belongs to any of these groups is invoked.
@BeforeClass: The annotated method will be run before the first test method in the current class is invoked.
@AfterClass: The annotated method will be run after all the test methods in the current class have been run.
@BeforeMethod: The annotated method will be run before each test method.
@AfterMethod: The annotated method will be run after each test method.
@Test: The annotated method is a part of a test case.

BENEFITS OF USING ANNOTATIONS

1. It identifies the methods it is interested in by looking up annotations. Hence method names are not restricted to any pattern or format.
2. We can pass additional parameters to annotations.
3. Annotations are strongly typed, so the compiler will flag any mistakes right away.
4. Test classes no longer need to extend anything (such as Test Case, for JUnit 3).

SELENIUM WEBDRIVER WITH TESTNG SAMPLE SCRIPT

Selenium WebDriver with TestNG in Java - in this example, we will test the Google Calculator feature using Selenium for UI automation and TestNG as testing framework. You can download this java file here calculatorTest.java (right click on 'calculatorTest.java' and click on 'save link as...' to save the sample selenium test script).

```
public class calculatorTest {

        @Test
        //Tests google calculator
        public void googleCalculator(){

                //Create firfox driver's instance
                WebDriver driver = new FirefoxDriver();

                //Set implicit wait of 10 seconds
                driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

                //Launch google
                driver.get("http://www.google.co.in");

                //Write 2+2 in google textbox
                WebElement googleTextBox = driver.findElement(By.id("gbqfq"));
                googleTextBox.sendKeys("2+2");
                //Click on searchButton
                WebElement  searchButton  =  driver.findElement(By.id("gbqfb"));
                searchButton.click();
                //Get result from calculator
                WebElement calculatorTextBox = driver.findElement(By.id("cwos"));
                String result = calculatorTextBox.getText();

                //Verify that result of 2+2 is 4
                Assert.assertEquals(result, "4");
        }

}
```

TESTNG ANNOTATIONS

In this tutorial, we will be studying all the annotations of TestNG along with the different attributes supported. **An annotation is a tag or metadata that provides additional information about a class, interface or method.** TestNG make use of these annotations to provide several features that aid in creation of robust testing framework. Here, we will refer to each TestNG annotation in detail and study their syntax and usage.

@TEST

The @Test is the most important and commonly used annotation of TestNG. It is used to mark a method as Test. So, any method over which we see @Test annotation, is considered as a TestNG test.

```
@Test
public void sampleTest() {
   //Any test logic
   System.out.println("Hi! ArtOfTesting here!");
}
```

Now, let's see some important attributes of @Test annotations-

1. description - The 'description' attribute is used to provide a description to the test method. It generally contains a one-liner test summary.

```
@Test(description = "Test summary")
```

2. **dataProvider** - This attribute helps in creating a data driven tests. It is used to specify the name of the data provider for the test.

```
@Test(dataProvider = "name of dataProvider")
```

3. **priority** - This attribute helps in prioritizing the test methods. The default priority starts with 0 and tests execute in ascending order.

```
@Test(priority = 2)
```

4. **enabled** - This attribute is used to specify whether the given test method will run with the suite or class or not.

```
@Test(enabled = false)
```

5. **groups** - Used to specify the groups, the test method belongs to.

```
@Test(groups = { "sanity", "regression" })
```

7. **dependsOnMethods** - Used to specify the methods on which the test method depends. The test method only runs after successful execution of the dependent tests.

```
@Test(dependsOnMethods = { "dependentTestMethodName" })
```

8. **dependsOnGroups** - Used to specify the groups on which the test method depends.

```
@Test(dependsOnGroups = { "dependentGroup" })
```

9. **alwaysRun** - When set as True, the test method runs even if the dependent methods fail.

```
@Test(alwaysRun=True)
```

10. **timeOut** - This is used to specify a timeout value for the test(in milli seconds). If test takes more than the timeout value specified, the test terminates and is marked as failure.

```
@Test (timeOut = 500)
```

@BEFORESUITE

The annotated method will run only once before all tests in this suite have run.

@AFTERSUITE

The annotated method will run only once after all tests in this suite have run.

@BEFORECLASS

The annotated method will run only once before the first test method in the current class is invoked.

@AFTERCLASS

The annotated method will run only once after all the test methods in the current class have been run.

@BEFORETEST

The annotated method will run before any test method belonging to the classes inside the <test> tag is run.

@AFTERTEST

The annotated method will run after all the test methods belonging to the classes inside the <test> tag have run.

@DATAPROVIDER

Using @DataProvider we can create a data driven framework in which data is passed to the associated test method and multiple iteration of the test runs for the different test data values passed from the @DataProvider method. The method annotated with @DataProvider annotation return a 2D array or object.

```
//Data provider returning 2D array of 3*2 matrix
 @DataProvider(name = "dataProvider1")
   public Object[][] dataProviderMethod1() {
      return new Object[][] {{"kuldeep","rana"}, {"k1","r1"},{"k2","r2"}};
   }

   //This method is bound to the above data provider
   //The test case will run 3 times with different set of values
   @Test(dataProvider = "dataProvider1")
   public void sampleTest(String str1, String str2) {
      System.out.println(str1 + " " + str2);
   }
```

@PARAMETER

The @Parameter tag is used to pass parameters to the test scripts. The value of the @Parameter tag can be passed through testng.xml file. Sample testng.xml with parameter tag-

```
@Test (timeOut = 500)<suite name="sampleTestSuite">
   <test name="sampleTest">
      <parameter name="sampleParamName" value="sampleParamValue"/>
      <classes>
         <class name="TestFile" />
      </classes>
   </test>
</suite>
```

Sample Test Script with @Parameter annotation-

```
public class TestFile {
   @Test
   @Parameters("sampleParamName")
   public void parameterTest(String paramValue) {
      System.out.println("sampleParamName = " + sampleParamName);
   }
```

@LISTENER

TestNG provides us different kind of listners using which we can perform some action in case an event has triggered. Usually testNG listeners are used for configuring reports and logging.

```
@Listeners(PackageName.CustomizedListenerClassName.class)

public class TestClass {
    WebDriver driver= new FirefoxDriver();@Test
    public void testMethod(){
    //test logic
    }
}
```

@FACTORY

The @Factory annotation helps in dynamic execution of test cases. Using @Factory annotation, we can pass parameters to the whole test class at run time. The parameters passed can be used by one or more test methods of that class.
In the below example, the test class TestFactory will run the test method in TestClass with different set of parameters - 'k1' and 'k2'.

```
public class TestClass{
    private String str;

    //Constructor
    public TestClass(String str) {
        this.str = str;
    }

    @Test
    public void TestMethod() {
        System.out.println(str);
    }
}
```

```
public class TestFactory{
    //Because of @Factory, the test method in class TestClass
    //will run twice with data "k1" and "k2"
    @Factory
    public Object[] factoryMethod() {
        return new Object[] { new TestClass("K1"), new TestClass("k2") };
    }
}
```

DATA DRIVEN TESTING USING TESTNG

In this tutorial, we will be studying about data driven testing. We will refer to the @DataProvider annotation of TestNG using which we can pass data to the test methods and create a data driven testing framework.

WHAT IS DATA DRIVEN TESTING?

Data driven testing is a test automation technique in which the test data and the test logic are kept separated. The test data drives the testing by getting iteratively loaded to the test script. Hence, instead of having hard-coded input, we have new data each time the script loads the data from the test data source.

DATA DRIVEN TESTING USING @DATAPROVIDER

Data driven testing can be carried out through TestNG using its @DataProvider annotation. A method with @DataProvider annotation over it returns a 2D array of object where the rows determines the number of iterations and columns determine the number of input parameters passed to the Test method with each iteration.
This annotation takes only the name of the data provider as its parameter which is used to bind the data provider to the Test method. If no name is provided, the method name of the data provider is taken as the data provider's name.

```
@DataProvider(name = "nameOfDataProvider")
public Object[][] dataProviderMethodName() {
        //Data generation or fetching logic from any external source
        //returning 2d array of object

        return new Object[][] {{"k1","r1",1},{"k2","r2",2}};
}
```

After the creation of data provider method, we can associate a Test method with data provider using 'dataProvider' attribute of @Test annotation. For successful binding of data provider with Test method, the number and data type of parameters of the test method must match the ones returned by the data provider method.

```
@Test(dataProvider = "nameOfDataProvider")
public void sampleTest(String testData1, String testData2, int testData3) {
        System.out.println(testData1 + " " + testData2 + " " + testData3);
}
```

CODE SNIPPET FOR DATA DRIVEN TESTING IN TESTNG

```
@DataProvider(name = "dataProvider1")
public Object[][] dataProviderMethod1() {
        return new Object[][] {{"k1","r1"},{"k2","r2"},{"k3","r3"}};
}

//The test case will run 3 times with different set of values
@Test(dataProvider = "dataProvider1")
public void sampleTest(String str1, String str2) {
        System.out.println(str1 + " " + str2);
}
```

The above test "sampleTest" will run 3 times with different set of test data - {"k1","r1"},{"k2","r2"},{"k3","r3"} received from 'dataProvider1' dataProvider method.

RUNNING TESTS IN PARALLEL USING TESTNG

Quite often in automation testing, we want to reduce the test execution time to get the test results as fast as possible. When we run the selenium tests using as testNG suite, by default it runs the tests serially. But testNG provides an inherent support to run the tests in parallel.
Before starting with parallel execution, lets get a brief insight into TestNG.xml file (those familiar with testng.xml file can skip this section and move to next paragraph - "Running tests in parallel").
Testng.xml file eases test executions and grouping of multiple different tests from same or different classes. This XML file has different tags for suite, test, classes and/or packages with suite being the root tag.
Sample testNG.xml file-

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="ArtOfTestingTestSuite" verbose="1" >
  <test name="parallelExecutionTests" >
    <classes>
       <class name="testsInParallel" />
           <class name="testsInParallel" />
    </classes>
  </test>
</suite>
```

On right clicking on the testng.xml file and running as TestNG suite, testNG will find the tests with @Test annotation in testsInParallel.java class and start test execution.

RUNNING TESTS IN PARALLEL

In order to run the tests in parallel just add these two key value pairs in suite-

- parallel="{methods/tests/classes}"
- thread-count="{number of thread you want to run simultaneously}".

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
```

```
<suite name="ArtOfTestingTestSuite" verbose="1" parallel="methods" thread-count="5">
  <test name="parallelExecutionTestsA" >
    <classes>
       <class name="testsInParallelA1" />
           <class name="testsInParallelA2" />
    </classes>
  </test>
  <test name="parallelExecutionTestsB" >
    <classes>
       <class name="testsInParallelB1" />
           <class name="testsInParallelB2" />
    </classes>
  </test>
</suite>
```

Let's see the purpose of the three options that can be set as value for the "parallel" parameter-

- **methods** - method value for parallel attribute will run all the tests independently on separate threads(maximum available threads). In the above example all the test methods listed under all the four classes will run in parallel in maximum 5 threads.
- **tests** - test value will run the methods specified in the test tag in the same thread. In the above example, all the test methods under the "parallelExecutionTestsA" test will run in same thread and all the test methods under the "parallelExecutionTestsB" will run in same thread. So, there will be two threads each running the test methods under parallelExecutionTestsA and parallelExecutionTestsB tests in parallel but independent of each other.
- **class** - class value will run the all the test methods stated in a class in same thread and with each class's method running in different thread. So, there would be four thread running the tests in the four classes- testsInParallelA1,testsInParallelA2,testsInParallelB1,testsInParallelB2.

MULTI BROWSER TESTING IN SELENIUM

In this tutorial, we will be studying about multi browser testing and its implementation in Selenium. For this we will integrate Selenium with TestNG and use @Parameter annotation of TestNG to parameterize the test script with different values of browser. Also, note that multi browser testing can be carried out using Selenium Grid as well. But here, we will be using TestNG and Selenium WebDriver only for implementing multi browser testing.

WHAT IS MULTI BROWSER TESTING?

Multi browser testing or cross browser testing is a type of testing in which the application under test is tested with multiple supported browsers. The need for multi browser testing arises from the fact that different browsers have different UI implementations. So, one cannot ensure that application running on Chrome will also run on IE without any issues.

MULTI BROWSER TESTING IN SELENIUM AND TESTNG

Multi browser testing can be carried out in Selenium by parameterizing the browser variable. For parameterizing the browser variable we can use the @Parameter annotation of TestNG. Using the @Parameter annotation, we can pass different values of browser to the test scripts from the testng.xml file. The value of the browser parameter can then be used to instantiate the corressponding driver class of Selenium WebDriver. As the browser value is used across all the test methods so, it is better to use the browser variable in @BeforeTest method.
The beforeTest method with @Parameter of TestNG will look like-

```
@Parameters("browser")
@BeforeTest
public void setBrowser(String browser)
{
   if (browser.equalsIgnoreCase("Firefox")) {
      driver = new FirefoxDriver();

   }
```

```
    else if (browser.equalsIgnoreCase("Chrome")) {
       System.setProperty("webdriver.chrome.driver",
          + pathToChromeDriverBinary + "chromedriver.exe");
       driver = new ChromeDriver();

    }
    else {
       throw new IllegalArgumentException("Invalid browser value!!");
    }
    driver.get(URL);
    driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
    driver.manage().window().maximize();
}
```

As stated earlier, the value of the browser variable will be passed to the test scripts through testng.xml file. This testng.xml file will have a parameter tag containing the browser variable and its value. Here, we will create two tests, one each for Firefox and Chrome browser with different browser values. Refer to the below snippet of testng.xml file to understrand the test parameterization concept. Also note that having **parallel="tests"** in the Suite tag makes the two tests run in parallel.

```
<suite name="MultiBrowserSuite" parallel="tests" thread-count="2">
   <test name="TestFirefox">
      <parameter name="browser" value="Firefox"/>
      <classes>
         <class name="sampleTestPackage.MultiBrowserTest"/>
      </classes>
   </test>
   <test name="TestChrome">
      <parameter name="browser" value="Chrome"/>
      <classes>
         <class name="sampleTestPackage.MultiBrowserTest"/>
      </classes>
   </test>
</suite>
```

On executing the test using testng.xml, the test will run in parallel for all the specified browsers(Firefox and Chrome in case of our example).

RERUN FAILED TESTS IN TESTNG

In this post, we will learn to rerun failed test cases using TestNG. We will explore the two approaches to achieve this, namely - using the testng-failed.xml file and by implementing the testNG IRetryAnalyzer.

RERUN FAILED TESTS USING TESTNG-FAILED.XML

WHEN TO USE?

Sometimes after some bug fixes, as a test automation engineer we are required to run only the failed tests reported by test automation suite. Running only the failed tests validate the bug fixes quickly.

HOW TO ACHIEVE?

Running only the failed tests is fairly simple as TestNG provides inherent support for this. Whenever a test suite is run using the testng.xml file then after the test execution, a **testng-failed.xml file gets created in the test-output folder**. Later on, we can run this file just like we run the testng.xml file. As this file only keeps track of the failed tests, so running this file, runs the failed tests only.

RETRY FAILED TESTS AUTOMATICALLY USING IRETRYANALYZER

WHEN TO USE?

At times, the test execution report comes up with some failures that are not because of the issues in the application. The underlying cause of these issues might be related to the test environment setup or some occasional server issue. In order to make sure that the failure reported in the test report are genuine and not just one-off cases, we can retry running the failed test cases to eliminate false negative tests results in our test reports.

HOW TO ACHIEVE?

For retrying the failure test runs automatically during the test run itself, we need to implement the **IRetryAnalyzer** interface provided by TestNG. The IRetryAnalyzer interface provide methods to control retrying the test runs. Here, we will override the retry() method of IRetryAnalyzer to make sure the test runs in case of failure with specifed number of retry limit. The comments in the code snippet make it self-explainatory.

CODE SNIPPET

```
package com.artoftesting.test;
import org.testng.IRetryAnalyzer;
import org.testng.ITestResult;

public class RetryAnalyzer implements IRetryAnalyzer {

        //Counter to keep track of retry attempts
        int retryAttemptsCounter = 0;

        //The max limit to retry running of failed test cases
        //Set the value to the number of times we want to retry
        int maxRetryLimit = 1;

        //Method to attempt retries for failure tests
        public boolean retry(ITestResult result) {
                if (!result.isSuccess()) {
                        if(retryAttemptsCounter < maxRetryLimit){
                                retryAttemptsCounter++;
                                return true;
                        }
                }
                return false;
        }
}
```

For the demo, I am creating a dummy test method below and intentionally failing it using the assert.fail() method. Here, we will set the @Test annotation's **retryAnalyzer attribute with RetryAnalyzer.class** that we created above.

```
@Test(retryAnalyzer = RetryAnalyzer.class)
    public void intentionallyFailingTest(){
        System.out.println("Executing Test");
        Assert.fail("Failing Test");
    }
```

TEST OUTPUT

```
===============================================
Test suite
Total tests run: 2, Failures: 1, Skips: 1
===============================================
```

Here, we can observe that the number of runs for the method are displayed as 2 with one failure and one skipped. The first run of the test method when failed will be marked as Skipped and then the test will run again due to the logic specified in the RetryAnalyzer.

Now, there is just one problem, we need to set the **retryAnalyzer** attribute in each of the @Test annotation. To deal with this we can implement **IAnnotationTransformer**interface. This interface provides the capability to alter the testNG annotation at runtime. So, we will create a class implementing the IAnnotationTransformer and make it set the RetryAnalyzer for @Test annotations.

```
package com.artoftesting.test;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

import org.testng.IAnnotationTransformer;
import org.testng.IRetryAnalyzer;
import org.testng.annotations.ITestAnnotation;

public class FailureRetryListener implements IAnnotationTransformer {

        //Overriding the transform method to set the RetryAnalyzer
        public void transform(ITestAnnotation testAnnotation, Class testClass,
                        Constructor testConstructor, Method testMethod)   {
                IRetryAnalyzer retry = testAnnotation.getRetryAnalyzer();
```

```
                 if (retry == null)
                         testAnnotation.setRetryAnalyzer(RetryAnalyzer.class);
        }
}
```

Having created a listener, we can specify it in the testNG.xml file like this-

```
<listeners>
        <listener class-name="com.artoftesting.test.FailureRetryListener"/>
</listeners>
```

Now, we don't have to set the **retryAnalyzer** in each @Test annotation. Having the listener specified in the testng.xml file will make it work for all the tests.

PS: If you want to add retry functionality to only limited set of test method than just set the @Test annotations with **retryAnalyzer attribute with RetryAnalyzer.class**, there is no need to implement IAnnotationTransformer and adding the listener in the testng.xml file.

PRIORITY IN TESTNG

In automation, many times we are required to configure our test suite to run test methods in a specific order or we have to give precedence to certain test methods over others. TestNG allows us to handle scenarios like these by providing a **priority**attribute within @Test annotation. By seeting the value of this priority attribute we can order the test methods as our need.

PRIORITY PARAMETER

We can assign a priority value to a test method like this-

```
@Test(priority=1)
```

The tests with lower priority value will get executed first. Example-

DEFAULT PRIORITY

The default priority of test when not specified is integer value 0. So, if we have one test case with priority 1 and one without any priority value then the test without any priority value will get executed first (as default value will be 0 and tests with lower priority are executed first).

CODE SNIPPET

```
@Test(priority = 1)
public void testMethodA() {
   System.out.println("Executing - testMethodA");
}

@Test
public void testMethodB() {
   System.out.println("Executing - testMethodB");
}

@Test(priority = 2)
public void testMethodC() {
   System.out.println("Executing - testMethodC");
}
```

OUTPUT

```
Executing - testMethodB
Executing - testMethodA
Executing - testMethodC
```

Here, we can see that testMethodB got executed first as it had default priority of 0. Since the other tests had priority value as 1 and 2 hence, the execution order was testMethodB then testMethodA and then testMethodC.

PS: If we want to give a test method, priority higher than the default priority then we can simply assign a negative value to the priority attribute of that test method.

```
@Test(priority = -1)
public void testMethod() {
   System.out.println("Priority higher than default");
}
```

DEPENDENCY IN TESTNG

As a rule of thumb each test method must be independent from other i.e. result or execution of one test method must not affect the other test methods. But at times in automation, we may require one test method to be dependent on other. We may want to maintain dependecy between the test methods in order to save time or for scenarios where the call to run a particular test depends on the test result of a some other tests. TestNG as a testing framework provides us two attributes **dependsOnMethod** and **dependsOnGroup** within @Test annotation to achieve dependency between the tests. Now let's explore these attributes.

DEPENDSONMETHOD

Using 'dependsOnMethod' attribute within @Test annotation, we can specify the name of parent test method on which the test should be dependent. Some points to remember about dependsOnMethod-

- On execution of the whole test class or test suite, the parentTest method will run before the dependentTest.
- If we run the dependentTest alone even then the parentTest method will also run that too before the dependentTest.
- In case, the parentTest method gets failed then the dependentTest method will not run and will be marked as skipped.

```
@Test(dependsOnMethods={"parentTest"})
```

CODE SNIPPET

In the below code snippet, we have two test methods - parentTest() and dependentTest(). The dependentTest method is made dependent on the parentTest method using **dependsOnMethods** attribute.

```
public class TutorialExample {

    @Test
    public void parentTest() {
      System.out.println("Running parent test.");
    }

    @Test(dependsOnMethods={"parentTest"})
    public void dependentTest() {
        System.out.println("Running dependent test.");
    }
}
```

OUTPUT

On running the dependentTest() alone, the parentTest() method will also run and that too before the dependentTest(). This is done to maintain the dependency.

```
Running parent test.
Running dependent test.
PASSED: parentTest
PASSED: dependentTest

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
===============================================
```

DEPENDSONGROUP

The **dependsOnGroup** attribute is used to make a test method dependent on a collection of tests belonging to a particular groups. Thus ensuring that the dependent test will run after all the tests of the parent group are executed. Some points to remember about dependsOnMethod-

- On execution of the whole test class or test suite, the tests belonging to the parent group will run before the dependentTest.
- If we run the dependentTest alone even then all the tests belonging to the parent group will also run that too before the dependentTest.
- In case, anyone or all of the tests of the parent group method gets failed then the dependentTest method will not run and will be marked as skipped.

```
@Test(dependsOnGroups = "groupA")
```

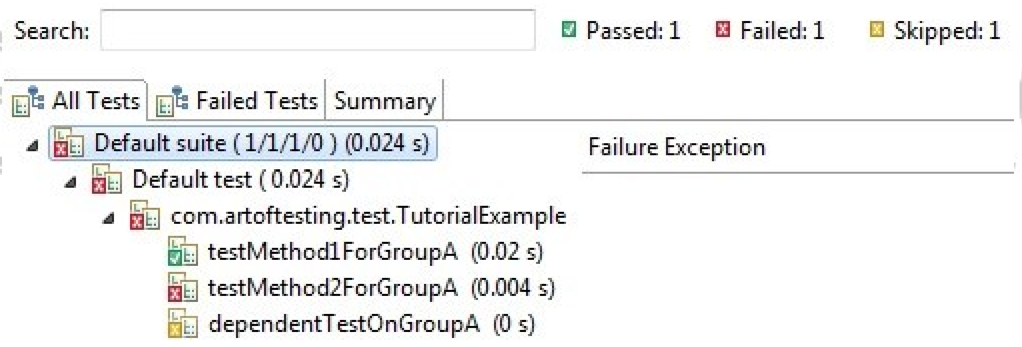
CODE SNIPPET

In the below code snippet, we have two test methods - testMethod1ForGroupA() and testMethod2ForGroupA() belonging to group **groupA**. Then we have a dependentTest method - **dependentTestOnGroupA** that is made dependent on the methods of **groupA** using **dependsOnGroups** attribute. For the sake of demo, we are intentionally failing the **testMethod2ForGroupA**.

```
public class TutorialExample {

    @Test(groups = "groupA")
    public void testMethod1ForGroupA() {
        System.out.println("Running test method1 of groupA");
    }

    @Test(groups = "groupA")
    public void testMethod2ForGroupA() {
        System.out.println("Running test method2 of groupA");
        Assert.fail();
    }

    @Test(dependsOnGroups = "groupA")
    public void dependentTestOnGroupA() {
        System.out.println("Running the dependent test");
    }
}
```

TESTNG RESULT

On running the dependentTestOnGroupA() alone or running the whole test class, we can observe the following output.



```
Default suite (1/1/1/0 ) (0.024 s)
  Default test ( 0.024 s)
    com.artoftesting.test.TutorialExample
      testMethod1ForGroupA (0.02 s)
      testMethod2ForGroupA (0.004 s)
      dependentTestOnGroupA (0 s)
```

Here, we can observe that TestNG attemps to run all the 3 tests and because of the dependecy, runs **testMethod1ForGroupA** and **testMethod1ForGroupB** first. Now, since **testMethod1ForGroupB** gets failed, so, the dependentTest i.e. **dependentTestOnGroupA** is not run and marked as skipped.

SOFT ASSERTION IN TESTNG

Assertions in our test suites are required to validate the actual result with expected result. The **Assert** class provided by TestNG provides a form of hard assertion wherein as soon as an assertion fails the execution of that particular test method stops and the test method is marked as failure.
But many times we might require the test method to continue execution even after the failure of the first assertion statement. Requirement like these arise

when we have multiple assertions in our test method or we want to execute some other line of codes after the assertion statement. For handling these cases, TestNG provides a **SoftAssert**class.

```
SoftAssert softAssert = new SoftAssert();
```

When we use SoftAssert, in case of assertion failure assertionException is not thrown instead all the statements are executed and later the test collates the result of all the assertions and marks the test case as passed or failed based on the assertion result.
In the below code snippet, we are using three assertion within the test method. We are intentionally failing the first two assertions still the whole test method will get executed. The last statement **softAssert.assertAll()** is very important, it will collate the result of all the assertions and in case of any failure mark, the test as failed.
PS: If we don't use softAssert.assertAll(), then the test case will be marked as passed even in case of assertion failure.

CODE SNIPPET

```
@Test
public void softAssertionTest(){

   //Creating softAssert object
   SoftAssert softAssert = new SoftAssert();

   //Assertion failing
   softAssert.fail("Failing first assertion");
   System.out.println("Failing 1");

   //Assertion failing
   softAssert.fail("Failing second assertion");
   System.out.println("Failing 2");

   //Assertion passing
   softAssert.assertEquals(1, 1, "Passing third assertion");
   System.out.println("Passing 3");

   //Collates the assertion results and marks test as pass or fail
   softAssert.assertAll();
}
```

OUTPUT

```
Failing 1
Failing 2
Passing 3
```

FAILURE EXCEPTION

```
FAILED: softAssertionTest
java.lang.AssertionError: The following asserts failed:
        Failing first assertion,
        Failing second assertion
        at org.testng.asserts.SoftAssert.assertAll(SoftAssert.java:43)
.....
```

Here, we can observe that the console output contains all the print statements even though the assertions are failing. Also, we can see that the assertion messages imply that two out of the three assertions failed - **Failing first assertion, Failing second assertion**.

TIMEOUT IN TESTNG

The automation test suites have the tendency to take too much time in case the elements are not readily available for interaction. Also, in certain tests we might have to wait for some asynchronous event to occur in order to proceed with the test execution. In these cases, we may want to limit the test execution time by specifying an upper limit of timeout exceeding which the test method is marked as failure.
TestNG provides us **timeOut** attribute for handling these requirements.

TIMEOUT

Time **timeOut** attribute within the @Test annotation method is assigned a value specifying the number of milliseconds. In case the test method exceeds the timeout value, the test method is marked as failure with **ThreadTimeoutException**.

```
@Test(timeOut = 1000)
```

CODE SNIPPET

In the below code snippet we have specified a timeout of 1000ms. Inside the test methods we can see that a Thread.sleep() of 3seconds is introduced. On test execution, we can notice in the output that the test fails with ThreadTimeoutException as the timeout is 1 second and the test takes liitle over 3 seconds to execute.

```
@Test(timeOut = 1000)
public void timeOutTest() throws InterruptedException {
   Thread.sleep(3000);
   //Test logic
}
```

OUTPUT

```
FAILED: timeOutTest
org.testng.internal.thread.ThreadTimeoutException:
Method org.testng.internal.TestNGMethod.timeOutTest()
didn't finish within the time-out 1000
```

TESTNG INTERVIEW QUESTIONS

Ques.1. What is testNG?

Ans. TestNG(NG for Next Generation) is a testing framework that can be integrated with selenium or any other automation tool to provide multiple capabilities like assertions, reporting, parallel test execution etc.

Ques.2. What are some advantages of testNG?

Ans. Following are the advantages of testNG-

1. TestNG provides different assertions that helps in checking the expected and actual results.
2. It provides parallel execution of test methods.
3. We can define dependency of one test method over other in TestNG.
4. We can assign priority to test methods in selenium.
5. It allows grouping of test methods into test groups.
6. It allows data driven testing using @DataProvider annotation.
7. It has inherent support for reporting.
8. It has support for parameterizing test cases using @Parameters annotation.

Ques.3. What is the use of testng.xml file?

Ans. The testng.xml file is used for configuring the whole test suite. In testng.xml file, we can create test suite, create test groups, mark tests for parallel execution, add listeners and pass parameters to test scripts. We can also use this testng.xml file for triggering the test suite from command prompt/terminal or Jenkins.

Ques.4. What are some commonly used TestNG annotations?

Ans. The commonly used TestNG annotations are-

- @Test- @Test annotation marks a method as Test method.
- @BeforeSuite- The annotated method will run only once before all tests in this suite have run.
- @AfterSuite-The annotated method will run only once after all tests in this suite have run.
- @BeforeClass-The annotated method will run only once before the first test method in the current class is invoked.
- @AfterClass-The annotated method will run only once after all the test methods in the current class have been run.
- @BeforeTest-The annotated method will run before any test method belonging to the classes inside the <test> tag is run.
- @AfterTest-The annotated method will run after all the test methods belonging to the classes inside the <test> tag have run.

- @DataProvider-The @DataProvider annotation is used to pass test data to the test method. The test method will run as per the number of rows of data passed via data provider method.

Ques.5. What are some common assertions provided by testNG?

Ans. Some of the common assertions provided by testNG are-

1. assertEquals(String actual, String expected, String message) - (and other overloaded data types in parameter)
2. assertNotEquals(double data1, double data2, String message) - (and other overloaded data types in parameter)
3. assertFalse(boolean condition, String message)
4. assertTrue(boolean condition, String message)
5. assertNotNull(Object object)
6. fail(boolean condition, String message)
7. true(String message)

Ques.6. How can we disable or prevent a test case from running?

Ans. By setting "enabled" attribute as false, we can disable a test method from running.

```
@Test(enabled = false)
public void testMethod() {
  //Test logic
}
```

Ques.7. How can we make one test method dependent on other using TestNG?

Ans. Using dependsOnMethods parameter inside @Test annotation in testNG we can make one test method run only after successful execution of dependent test method.

```
@Test(dependsOnMethods = { "preTests" })
```

Ques.8. How can we set priority of test cases in TestNG?

Ans. We can define priority of test cases using "priority" parameter in @Test annotation. The tests with lower priority value will get executed first. Example-

```
@Test(priority=1)
```

Ques.9. What is the default priority of test cases in TestNG?

Ans. The default priority of test when not specified is integer value 0. So, if we have one test case with priority 1 and one without any priority then the test without any priority value will get executed first (as default value will be 0 and tests with lower priority are executed first).

Ques.10. How can we run a Test method multiple times in a loop(without using any data provider)?

Ans. Using invocationCount parameter and setting its value to an integer value, makes the test method to run n number of times in a loop.

```
@Test(invocationCount = 10)
public void invocationCountTest(){
   //Test logic
}
```

Ques.11. What is threadPoolSize? How can we use it?

Ans. The **threadPoolSize** attribute specifies the number of thread to be assigned to the test method. This is used in conjunction with **invocationCount** attribute. The number of threads will get divided with the number of iterations of the test method specified in the invocationCount attribute.

```
@Test(threadPoolSize = 5, invocationCount = 10)
public void threadPoolTest(){
   //Test logic
}
```

Ques.12. What is the difference between soft assertion and hard assertion in TestNG?

Ans. Soft assertions (SoftAssert) allows us to have multiple assertions within a test method, even when an assertion fails the test method contiues with the remaining test execution. The result of all the assertions can be collated at the end using softAssert.assertAll() method.

```
@Test
public void softAssertionTest(){
   SoftAssert softAssert= new SoftAssert();

   //Assertion failing
   softAssert.fail();
   System.out.println("Failing");

   //Assertion passing
   softAssert.assertEquals(1, 1);
   System.out.println("Passing");

   //Collates test results and marks them pass or fail
   softAssert.assertAll();
}
```

Here, even though the first assertion fails still the test will continue with execution and print the message below the second assertion. Hard assertions on the other hand are the usual assertions prodived by TestNG. In case of hard assertion in case of any failure, the test execution stops, preventing execution of any further steps within the test method.

Ques.13. How to fail a testNG test if it doesn't get executed within a specified time?

Ans. We can use **timeOut** attribute of @Test annotation. The value assigned to this timeOut attribute will act as an upperbound, if test doesn't get executed within this time frame then it will fail with timeOut exception.

```
@Test(timeOut = 1000)
public void timeOutTest() throws InterruptedException {
   //Sleep for 2sec so that test will fail
   Thread.sleep(2000);
   System.out.println("Will throw Timeout exception!");
}
```

Ques.14. How can we skip a test case conditionally?

Ans. Using SkipException, we can conditionally skip a test case. On throwing the skipException, the test method me marked as skipped in the test execution report and any statement after throwing the exception will not get executed.

```
@Test
public void testMethod(){
  if(conditionToCheckForSkippingTest)
        throw new SkipException("Skipping the test");
  //test logic
}
```

Ques.15. How can we make sure a test method runs even if the test methods or groups on which it depends fail or get skipped?

Ans. Using "alwaysRun" attribute of @Test annotation, we can make sure the test method will run even if the test methods or groups on which it depends fail or get skipped.

```
@Test
public void parentTest() {
  Assert.fail("Failed test");
}

@Test(dependsOnMethods={"parentTest"}, alwaysRun=true)
public void dependentTest() {
  System.out.println("Running even if parent test failed");
}
```

Here, even though the parentTest failed, the dependentTest will not get skipped instead it will executed because of "alwaysRun=true". In case, we remove the "alwaysRun=true" attribute from @Test then the report will show one failure and one skipped test, without trying to run the dependentTest method.

Ques.16. How can we pass parameter to test script using testNG?

Ans. Using @Parameter annotation and 'parameter' tag in testng.xml we can pass paramters to test scripts.
Sample testng.xml -

```
<suite name="sampleTestSuite">
   <test name="sampleTest">
      <parameter name="sampleParamName" value="sampleParamValue"/>
      <classes>
         <class name="TestFile" />
      </classes>
   </test>
</suite>
```

Sample test script-

```
public class TestFile {
   @Test
   @Parameters("sampleParamName")
   public void parameterTest(String paramValue) {
      System.out.println("Value of sampleParamName is - " + sampleParamName);
   }
```

Ques.17. How can we create data driven framework using testNG?

Ans. Using @DataProvider we can create a data driven framework in which data is passed to the associated test method and multiple iteration of the test runs for the different test data values passed from the @DataProvider method. The method annotated with @DataProvider annotation return a 2D array of object.

```
//Data provider returning 2D array of 3*2 matrix
 @DataProvider(name = "dataProvider1")
   public Object[][] dataProviderMethod1() {
      return new Object[][] {{"kuldeep","rana"}, {"k1","r1"},{"k2","r2"}};
   }

   //This method is bound to the above data provider returning 2D array of 3*2 matrix
   //The test case will run 3 times with different set of values
   @Test(dataProvider = "dataProvider1")
   public void sampleTest(String s1, String s2) {
        System.out.println(s1 + " " + s2);
    }
```

Ques.18. What is the use of @Listener annotation in TestNG?

Ans. TestNG provides us different kind of listeners using which we can perform some action in case an event has triggered. Usually testNG listeners are used for configuring reports and logging. One of the most widely used lisetner in testNG is ITestListener interface. It has methods like onTestSuccess, onTestFailure, onTestSkipped etc. We need to implement this interface creating a listner class of our own. After that using the *@Listener annotation we can use specify that for a particular test class our customized listener class should be used.*

```
@Listeners(PackageName.CustomizedListenerClassName.class)

public class TestClass {
    WebDriver driver= new FirefoxDriver();@Test
    public void testMethod(){
    //test logic
    }
}
```

Ques.19. What is the use of @Factory annotation in TestNG?

Ans. @Factory annotation helps in dynamic execution of test cases. Using @Factory annotation we can pass parameters to the whole test class at run time. The parameters passed can be used by one or more test methods of that class.
Example - there are two classes TestClass and the TestFactory class. Because of the @Factory annotation the test methods in class TestClass will run twice with the data "k1" and "k2"

```
public class TestClass{
    private String str;

    //Constructor
    public TestClass(String str) {
        this.str = str;
    }
    @Test
    public void TestMethod() {
        System.out.println(str);
    }
}

public class TestFactory{
    //The test methods in class TestClass will run twice with data "k1" and "k2"
    @Factory
    public Object[] factoryMethod() {
        return new Object[] { new TestClass("K1"), new TestClass("k2") };
    }
}
```

Ques.20. What is difference between @Factory and @DataProvider annotation?

Ans. @Factory method creates instances of test class and run all the test methods in that class with different set of data. Whereas, @DataProvider is bound to individual test methods and run the specific methods multiple times.

Ques.21. How can we run test cases in parallel using TestNG?

Ans. In order to run the tests in parallel just add these two key value pairs in suite-

- parallel="{methods/tests/classes}"
- thread-count="{number of thread you want to run simultaneously}".

```
<suite name="ArtOfTestingTestSuite" parallel="methods" thread-count="5">
```

UNDERSTANDING WEBDRIVER DRIVER = NEW FIREFOXDRIVER();

Hello Friends! while doing UI automation using Selenium WebDriver, the very first line of code that we see to launch a browser is-

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

Depending on the type of browser we want to launch the browser specific driver changes to ChromeDriver(), InternetExplorerDriver() etc. But have you ever noticed that why do we need to create a reference variable of type WebDriver? What if we directly instantiate the browser specific driver like-

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

Is the above line of code incorrect? If not than what's the benefit of creating WebDriver type reference variable. In this post, we will answer all these questions in detail. But first of all let's see what is WebDriver.

WHAT IS WEBDRIVER?

WebDriver is an interface provided by Selenium WebDriver. As we know that interfaces in Java are the collection of constants and abstract methods(methods without any implementation). The WebDriver interface serves as a contract that each browser specific implementation like ChromeDriver, FireFoxDriver must follow. The WebDriver interface declares methods like get(), navigate(), close(), sendKeys() etc. and the developers of the browser specific drivers implement these methods to get the stuff automated.
Take for example the ChromeDriver, it is developed by the guys from Chromium team, the developers of the Selenium project don't have to worry about the implementation details of these drivers.

IS THIS CORRECT - WEBDRIVER DRIVER = NEW WEBDRIVER();?

No, the above statement is incorrect as WebDriver is an interface and in java we cannot instantiate an interface. Also, logically speaking, saying we want to launch WebDriver doesn't make any sense as we haven't specified any particualr browser that needs to be launched.

IS THIS CORRECT - FIREFOXDRIVER DRIVER = NEW FIREFOXDRIVER();?

Yes, it is perfectly correct. FirefoxDriver is an implementing class of WebDriver interface and the above statement will launch the Firefox browser. Now, the question arises, why do we need to create reference variable of type WebDriver?

BENEFIT OF WEBDRIVER DRIVER = NEW FIREFOXDRIVER();

Having a reference variable of type WebDriver allows us to assign the driver object to different browser specific drivers. Thus allowing multi-browser testing by assigning the driver object to any of the desired browser. For details check our tutorial on Multi browser testing using Selenium.

CHECK IF AN ELEMENT IS PRESENT ON THE WEBPAGE

During automation, we quite frequently need to work with dynamic elements or elements which are not always available in the DOM and who's presence can be used as a medium of assertion to pass or fail a test case. Cases like these require us to check for the presence of element on the webpage.

USING DRIVER.FINDELEMENTS()

In order to check if an element is present on a webpage we make use of driver.findElements() method. As we know that driver.findElements() method returns a list of webElements located by the "By Locator" passed as parameter. If element is found then it returns a list of non-zero webElements else it returns a size 0 list. Thus, checking the size of list can be used to check for the presence and absence of element.

```
List<WebElement> dynamicElement = driver.findElements(By.id("id"));
if(dynamicElement.size() != 0){
        //If list size is non-zero, element is present
        System.out.println("Element present");
}
else{
        //Else if size is 0, then element is not present
        System.out.println("Element not present");
}
```

Please note that this method will wait till the implicit wait specified for the driver while finding the element. So, in case we know that element will be at once present on the webpage then in order to speed up the process, we can set the implcit wait to 0 first, check for the presence of element and then revert the value of implicit wait to its default value.

```
//Set implict wait to 0
driver.manage().timeouts().implicitlyWait(0, TimeUnit.SECONDS);

//Check for element's presence
List<WebElement> dynamicElement = driver.findElements(By.id("id"));
if(dynamicElement.size() != 0)
        System.out.println("Element present");
else
        System.out.println("Element not present");

//Revert back to default value of implicit wait
driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);
```

WHY NOT USE DRIVER.FINDELEMENT()

You may ask, why can't we directly use the driver.findElement() method here? The reason for not using findElement() method is, in case the element is not present then findElement() method will throw NoSuchElementException exception which we need to catch first and then in the catch block, we have to write the code to take further steps after knowing that the element is not present. Like this-

```
try{
        driver.findElements(By.id("id"));
        //Since, no exception, so element is present
        System.out.println("Element present");
}
catch(NoSuchElementException e){
        //Element is not present
        System.out.println("Element not present");
}
```

But having business logic in catch block is highly disregarded as it is not an optimized way and make use of additional resources. Bottom line here is - our code should not rely on any exception to perform the normal flow.

WAIT FOR PAGE TO LOAD

In this tutorial, we will learn how to wait for a page to load before interacting with the weblements present on that page in order to avoid NoSuchElementException in Selenium.
Before going deeper into implementation of wait till page load in Selenium you need to understand the following points-

- An element that triggers page load waits till the DOM gets loaded before returing control to the driver. For example - if we submit a form by clicking the Submit button then the next statement after the submit button click operation will be attemted by webDriver only after the page gets loaded completely. So, in most cases we don't even have to wait for the page to load.
- We may have to introduce a wait time when dealing with elements that are ajax based or dynamic elements that load even after the page load.
- In certain cases, we deal with elements that may be visible on page load or after some trigger action but take some time to be available for interaction. For example a dropdown with dynamic values may be available on the DOM throughout but have its values populated only after we perform some action based on which the corresponding

values get dynamically populated. So, in this case if we try to select a particular value, we must wait for some time for the element to be available for interaction.

WAIT FOR PAGE TO LOAD IMPLEMENTATION

Selenium Webdriver doesn't provide any inherent support for wait till page load implementation. But we can make use of **Explicit Waits** to achieve the desired outcome. For this, we need to identify the element on the webpage that's the last one to load or become available for intercation. Now, we need to use the explicit wait with the appropriate Expected condition like "ElementToBeClickable" for page load implementation. Thus, making sure that the driver waits till all the web elements get loaded successfully before attempting to interact with them.

```
WebDriverWait wait = new WebDriverWait(driver, 20);
wait.until(ExpectedConditions.elementToBeClickable(lastElementToLoad));
```

OPEN A NEW TAB IN SELENIUM

In this tutorial, we will learn how to open a new tab in Selenium webdriver with Java. Although there are multiple ways of opening a new tab in Selenium like using Robot class, using Actions class, passing Keys.Control+"t" in sendKeys method to any element. But Action class and sendKeys method doesn't work with some browser/drivers versions. So, in this post we will see how to use Robot class to open a new tab as it is mpst stable option to perform this action.

ROBOT CLASS TO OPEN TAB

As we know that Robot class in Selenium is used for simulating keyboard and mouse events. So, in order to open a new tab we can simulate keyboard event

of pressing Control Key followed by 't' key of keyboard. After the new tab gets opened, we need to switch focus to it otherwise the driver will try to perform the operation on the parent tab only.
For switching focus, we will be using getWindowHandles() to get the handle of the new tab and then switch focus to it.

```
//Launch the first URL
driver.get("http://www.google.com");

//Use robot class to press Ctrl+t keys
Robot robot = new Robot();
robot.keyPress(KeyEvent.VK_CONTROL);
robot.keyPress(KeyEvent.VK_T);
robot.keyRelease(KeyEvent.VK_CONTROL);
robot.keyRelease(KeyEvent.VK_T);

//Switch focus to new tab
ArrayList<String> tabs = new ArrayList<String> (driver.getWindowHandles());
driver.switchTo().window(tabs.get(1));

//Launch URL in the new tab
driver.get("http://google.com");
```

GET ALL LINKS ON A WEBPAGE

Hello friends! at times during automation we are required to fetch all the links present on a webpage. Also, this is one of the most frequent requirements of web-scrapping. In this tutorial, we will learn to fetch all the links present on a webpage by using **tagname** locator.
If you have basic understanding of HTML, you must be aware of the fact that all hyperlinks are of type **anchor** tag or 'a'.

```
<a href="selenium-introduction.html">Selenium Introduction</a>
```

HOW TO FETCH ALL THE LINKS ON A WEBPAGE?

- Navigate to the desired webpage

- Get list of WebElements with tagname 'a' using driver.findElements()-

```
• List<WebElement> allLinks = driver.findElements(By.tagName("a"));
```

- Traverse through the list using for-each loop
- Print the link text using getText() along with its address using getAttribute("href")

```
• System.out.println(link.getText() + " - " + link.getAttribute("href"));
```

SAMPLE CODE TO MOUSE HOVER OVER AN ELEMENT

```
package seleniumTutorials;

import java.util.List;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class GetAllLinks {

        public static void main(String[] args){
                WebDriver driver = new FirefoxDriver();

                //Launching sample website
                driver.get("http://artoftesting.com/sampleSiteForSelenium.html");
                driver.manage().window().maximize();

                //Get list of web-elements with tagName  - a
                List<WebElement> allLinks = driver.findElements(By.tagName("a"));

                //Traversing through the list and printing its text along with link 
address
                for(WebElement link:allLinks){
                        System.out.println(link.getText() + " - " + link.getAttribut
e("href"));
                }

                //Commenting driver.quit() for user to verify the links printed
                //driver.quit();
        }

}
```

EXCEPTIONS IN SELENIUM

In this post, we will study the most commonly encountered exceptions in Selenium WebDriver and the root cause of these exceptions.

NOSUCHELEMENTEXCEPTION

Reason - In case no element could be located from the locator provided.
Resolution - Check the correctness of the locators for the elements and make sure that the element is present when interacted with.

ELEMENTNOTVISIBLEEXCEPTION

Reason - In case element is present in the dom but is not visible.
Resolution - Make sure that the element is in the visible area when interacted

with. Some common methods to acheive this are - maximizing the browser window, scrolling to the element.

NOALERTPRESENTEXCEPTION

Reason - In case we try to switch to an alert but the targetted alert is not present.
Resolution - Make sure that alert is present when it is interacted with.

NOSUCHFRAMEEXCEPTION

Reason - In case we try to switch to a frame but the targetted frame is not present.
Resolution - Check the frame locators and make sure frame is present on the webpage.

NOSUCHWINDOWEXCEPTION

Reason - In case we try to switch to a window but the targetted window is not present.
Resolution - Get the list of window handles using driver.getWindowHandles() and switch to one of the handles present at that particualr time.

UNEXPECTEDALERTPRESENTEXCEPTION

Reason - In case an unexpected alert blocks normal interaction of the driver.
Resolution - Accept or dismiss the alert to continue interacting with the dom.

TIMEOUTEXCEPTION

Reason - In case a command execution gets timeout.
Resolution - This may be a valid exception unless we have set very low timeout values in implicit and explicit waits.

INVALIDELEMENTSTATEEXCEPTION

Reason - In case the state of an element is not appropriate for the desired action.
Resolution - Make sure that the element is available to perform the desired operation by waiting for the desired ExpectedCondition in explicit wait.

NOSUCHATTRIBUTEEXCEPTION

Reason - In case we are trying to fetch an attribute's value but the attribute is not correct.
Resolution - Just make sure the attribute we want to fetch from an element is actually present in th element or not.

WEBDRIVEREXCEPTION

Reason - In case there is some issue with driver instance preventing it from getting launched.
Resolution - Check the driver's instantiation and the dependecies required to instantiate the driver object.

GECKODRIVER EXCEPTION WHILE LAUNCHING FIREFOX

Hello friends, with Selenium Webdriver versions after release Selenium 3.0 you might have faced the issue of Illegal State Exception while launching Firefox browser. In this tutorial, we will present the resolution for this exception and also its root cause.

EXCEPTION DETAIL

Exception in thread "main" java.lang.IllegalStateException: The path to the driver executable must be set by the webdriver.gecko.driver system property; for more information, see https://github.com/mozilla/geckodriver. The latest version can be downloaded from https://github.com/mozilla/geckodriver/releases

STEPS TO REPRODUCE

Launch Firefox browser with Selenium 3.0 and above versions directly without setting "webdriver.gecko.driver" path.

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

RESOLUTION

1. Download geckodriver.exe from GeckoDriver Github Release Page. Make sure to download the right driver file based on your platform and OS version.
2. Set the **System Property** for "webdriver.gecko.driver" with the geckodriver.exe path.
3. System.setProperty("webdriver.gecko.driver","geckodriver.exe path"); (This is similar to the way we use to set system property for chromedriver.exe).

CODE SNIPPET

```
System.setProperty("webdriver.gecko.driver", pathToGeckoDriver + "\\geckodriver.exe")
;
WebDriver driver = new FirefoxDriver();
```

DIFFERENCE B/W DRIVER.CLOSE() AND DRIVER.QUIT()

Selenium webdriver provides two methods for closing a browser window - driver.close() and driver.quit(). Some people incorrectly use them interchangeably but the two methods are different. In this post, we will study the difference between the two and also see where to use them effectively.

DRIVER.CLOSE()

The driver.close() command is used to close the current browser window having focus. In case there is only one browser open then calling driver.close() quits the whole browser session.
Usability
It is best to use driver.close() when we are dealing with multiple browser tabs or windows e.g. when we click on a link that opens another tab. In this case

after performing required action in the new tab, if we want to close the tab we can call the driver.close() method.

```
//Closing the tab
driver.close();
```

DRIVER.QUIT()

The driver.quit() is used to quit the whole browser session along with all the associated browser windows, tabs and pop-ups.
Usability
It is best to use driver.quit() when we no longer want to interact with the driver object along with any associated window, tab or pop-up. Generally it is one of the last statements of the automation scripts. In case, we are working with Selenium with TestNG or JUnit, we call driver.quit() in the @AfterSuite method of aur suite. Thus, closing it at the end of whole suite.

```
@AfterSuite
public void tearDown() {
   driver.quit();}
```

DIFFERENCE B/W FINDELEMENT() AND FINDELEMENTS()

Selenium webdriver provides two methods for finding webelements on a webpage - driver.findElement() and driver.findElements(). In this post, we will study the difference between the two and also see where to use them effectively.

#	findElement	findElements

Definition	driver.findElement() is used to find a webElement on a webpage.	driver.findElements() is used to find a List of webElements matching the locator passed as parameter.
Syntax	WebElement element = driver.findElement(By locator);	List elements = driver.findElements(By locator);
For multiple matches	In case the same locator matches multiple webElements then findElement method returns the first web element found.	In case of multiple matches the findElements method returns a list of webElements. For interacting with a particular element we have find the particular element by its index e.g. elements.get(0).click(); will perform the click operation on the first element of the 'elements' list.
If no element is found	In case the locator passed to findElement() method leads to no element then NoSuchElementException is thrown.	In case the locator passed to findElements() method leads to no element then a List of 0 size is returned instead of an exception.

Please note that because of the fact that in case no element is found driver.findElements() return a list of size 0 (unlike findElement() that throws exception), it is generally used to check for the presence of an element. We can just check the size of the list returned by findElements() method and if its non-zero, we can interact with the first element of the list using 0 index.

CHECK A CHECKBOX IF NOT ALREADY CHECKED

In this post, we will learn how to check if a checkbox is in checked or unchecked state. Also, we will see how to check/set a checkbox if its not already checked.

VERIFY IF A CHECKBOX IS CHECKED OR NOT

In order to check if a checkbox is checked or unchecked, we can used the isSelected() method over the checkbox element. The isSelected() method returns a boolean value of true if the checkbox is checked false otherwise.

CODE SNIPPET

```
WebElement checkbox = driver.findElement(By.id("checkboxId"));
```

```
System.out.println("The checkbox is selection state is - " + checkbox.isSelected());
```

CHECK A CHECKBOX IF NOT ALREADY CHECKED

During automation, if we want to check a checkbox with click() method then the checkbox gets unchecked if its already checked. So, if you want to make sure that you are only checking the checkbox then we should first get the checkbox state using isSelected() method and then click on it only if it is in unchecked state.

CODE SNIPPET

```
WebElement checkbox = driver.findElement(By.id("checkboxId"));

//If the checkbox is unchecked then isSelected() will return false
//and NOT of false is true, hence we can click on checkbox
if(!checkbox.isSelected())
        checkbox.click();
```

MYSQL AUTOMATION IN JAVA

WHY DO WE NEED DATABASE AUTOMATION?

- To get test data - If we automate the database, we can directly fetch the test data from database and then work on them in test automation script
- To verify result - In automation we can verify the front end result with backend entry in the database
- To delete test data created - In automation it is good practice to delete the test data created, using database automation, we directly fire the delete query to delete the test data created
- To update certain data - As per the need of test script, the test data can be updated using update query

CONNECTING TO MYSQL DATABASE IN JAVA

Database automation in mySQL database involves the following steps-

- Loading the required JDBC Driver class, which in our case is com.mysql.jdbc.Driver. You can download the jar from here and add it to your classpath or add it as maven dependency in you pom.xml file in case you are using maven project.

```
Class.forName("com.mysql.jdbc.Driver");
```

- Creating a connection to the database-

```
Connection conn = DriverManager.getConnection("DatabaseURL","UserName", "Password");
```

- Executing SQL queries-

```
Statement st = conn.createStatement();
String Sql = "select * from [tableName] where <condition>";
ResultSet rs = st.executeQuery(Sql);
```

- Fetching data from result set-

```
while (rs.next()) {
  System.out.println(rs.getString(<requiredField>));
}
```

SAMPLE CODE TO CONNECT TO A MYSQL DATABASE

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class connectingToMySQLDBExample {

    private static String userName;
    private static String password;
    private static String dbURL;
    private static Connection connection;

    public static void main(String[] args) {
        try {
                userName = "username";
                password = "password";
                dbURL = "jdbc:mysql://artoftesting.com/testDB";
                try {
                        Class.forName("com.mysql.jdbc.Driver");
                }
               catch (ClassNotFoundException e) {
                        System.out.println("MySQL JDBC driver not found.");
                        e.printStackTrace();
                }
                try {
                        connection = DriverManager.getConnection(dbURL, userName, pa
ssword);
                        Statement st = connection.createStatement();
                        String sqlStr = "select * from testTable";
                        ResultSet rs = st.executeQuery(sqlStr);
                        while (rs.next()) {
                                System.out.println(rs.getString("name"));
                        }

                        } catch (SQLException e) {
                                System.out.println("Connection to MySQL db failed");
                         e.printStackTrace();
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

SELENIUM INTERVIEW QUESTIONS

Prepare for selenium interview with our comprehensive list of over 100 interview questions. These interview questions are designed for both beginners and professionals. We will start with fairly simple questions and move to the more advanced level as the post progresses.

Ques.1. What is Selenium?

Ans. Selenium is a robust test automation suite that is used for automating web based applications. It supports multiple browsers, programming languages and platforms.

Ques.2. What are different forms of selenium?

Ans. Selenium comes in four forms-

1. *Selenium WebDriver* - Selenium WebDriver is used to automate web applications using browser's native methods.
2. *Selenium IDE* - A firefox plugin that works on record and play back principle.
3. *Selenium RC* - Selenium Remote Control(RC) is officially deprecated by selenium and it used to work on javascript to automate the web applications.
4. *Selenium Grid* - Allows selenium tests to run in parallel across multiple machines.

Ques.3. What are some advantages of selenium?

Ans. Following are the advantages of selenium-

1. Selenium is open source and free to use without any licensing cost.
2. It supports multiple languages like Java, ruby, python etc.
3. It supports multi browser testing.
4. It has good amount of resources and helping community over the internet.
5. Using selenium IDE component, non-programmers can also write automation scripts
6. Using selenium grid component, distributed testing can be carried out on remote machines possible.

Ques.4. What are some limitations of selenium?

Ans. Following are the limitations of selenium-

1. We cannot test desktop application using selenium.
2. We cannot test web services using selenium.
3. For creating robust scripts in selenium webdriver, programming langauge knowledge is required.
4. We have to rely on external libraries and tools for performing tasks like - logging(log4J), testing framework-(testNG, JUnit), reading from external files(POI for excels) etc.

Ques.5. Which all browsers/drivers are supported by Selenium Webdriver?

Ans. Some commonly used browsers supported by selenium are-

1. Google Chrome - ChromeDriver
2. Firefox - FireFoxDriver
3. Internet Explorer - InternetExplorerDriver
4. Safari - SafariDriver
5. HtmlUnit (Headless browser) - HtmlUnitDriver
6. Android - Selendroid/Appium
7. IOS - ios-driver/Appium

Ques.6. Can we test APIs or web services using Selenium webdriver?

Ans. No selenium webdriver uses browser's native method to automate the web applications. Since web services are headless, so we cannot automate web services using selenium webdriver.

Ques.7. What are the testing type supported by Selenium WebDriver?

Ans. Selenium webdriver can be used for performing automated functional and regression testing.

Ques.8. What are various ways of locating an element in selenium?

Ans. The different locators in selenium are-

1. Id
2. XPath
3. cssSelector
4. className
5. tagName
6. name
7. linkText
8. partialLinkText

Ques.9. What is an XPath?

Ans. Xpath or XML path is a query language for selecting nodes from XML documents. XPath is one of the locators supported by selenium webdriver.

Ques.10. What is an absolute XPath?

Ans. An absolute XPath is a way of locating an element using an XML expression beginning from root node i.e. html node in case of web pages. The main disadvantage of absolute xpath is that even with slightest change in the UI or any element the whole absolute XPath fails.
Example - html/body/div/div[2]/div/div/div/div[1]/div/input

Ques.11. What is a relative XPath?

Ans. A relative XPath is a way of locating an element using an XML expression beginning from anywhere in the HTML document. There are different ways of creating relative XPaths which are used for creating robust XPaths (unaffected by changes in other UI elements).
Example - //input[@id='username']

Ques.12. What is the difference between single slash(/) and double slash(//) in XPath?

Ans. In XPath a single slash is used for creating XPaths with absolute paths beginning from root node.
Whereas double slash is used for creating relative XPaths.

Ques.13. How can we inspect the web element attributes in order to use them in different locators?

Ans. Using Firebug or developer tools we can inspect the specific web elements.
Firebug is a plugin of firefox that provides various development tools for debugging applications. From automation perspective, firebug is used

specifically for inspecting web-elements in order to use their attributes like id, class, name etc. in different locators.

Ques.14. How can we locate an element by only partially matching its attributes value in Xpath?

Ans. Using contains() method we can locate an element by partially matching its attribute's value. This is particularly helpful in the scenarios where the attributes have dynamic values with certain constant part.

```
xPath expression = //*[contains(@name,'user')]
```

The above statement will match the all the values of name attribute containing the word 'user' in them.

Ques.15. How can we locate elements using their text in XPath?

Ans. Using the text() method -

```
xPathExpression = //*[text()='username']
```

Ques.16. How can we move to parent of an element using XPath?

Ans. Using '..' expression in XPath we can move to parent of an element.

Ques.17. How can we move to nth child element using XPath?

Ans. There are two ways of navigating to the nth element using XPath-

- Using square brackets with index position- Example - div[2] will find the second div element.
- Using position()- Example - div[position()=3] will find the third div element.

Ques.18. What is the syntax of finding elements by class using CSS Selector?

Ans. By .className we can select all the element belonging to a particluar class e.g. '.red' will select all elements having class 'red'.

Ques.19. What is the syntax of finding elements by id using CSS Selector?

Ans. By #idValue we can select all the element belonging to a particluar class e.g. '#userId' will select the element having id - userId.

Ques.20. How can we select elements by their attribute value using CSS Selector?

Ans. Using [attribute=value] we can select all the element belonging to a particluar class e.g. '[type=small]' will select the element having attribute type of value 'small'.

Ques.21. How can we move to nth child element using css selector?

Ans. Using :nth-child(n) we can move to the nth child element e.g. div:nth-child(2) will locate 2nd div element of its parent.

Ques.22. What is fundamental difference between XPath and css selector?

Ans. The fundamental difference between XPath and css selector is using XPaths we can traverse up in the document i.e. we can move to parent elements. Whereas using CSS selector we can only move downwards in the document.

Ques.23. How can we launch different browsers in selenium webdriver?

Ans. By creating an instance of driver of a particular browser-

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

Ques.24. What is the use of driver.get("URL") and driver.navigate().to("URL") command? Is there any difference between the two?

Ans. Both *driver.get("URL")* and *driver.navigate().to("URL")* commands are used to navigate to a URL passed as parameter. There is no difference between the two commands.

Ques.25. How can we type text in a textbox element using selenium?

Ans. Using sendKeys() method we can type text in a textbox-

```
WebElement searchTextBox = driver.findElement(By.id("search"));
searchTextBox.sendKeys("searchTerm");
```

Ques.26. How can we clear a text written in a textbox?

Ans. Using clear() method we can delete the text written in a textbox.

```
driver.findElement(By.id("elementLocator")).clear();
```

Ques.27. How to check a checkBox in selenium?

Ans. The same click() method used for clicking buttons or radio buttons can be used for checking checkbox as well.

Ques.28. How can we submit a form in selenium?

Ans. Using submit() method we can submit a form in selenium.

```
driver.findElement(By.id("form1")).submit();
```

Also, the click() method can be used for the same purpose.

Ques.29. Explain the difference between close and quit command.

Ans. *driver.close()* - Used to close the current browser having focus
driver.quit() - Used to close all the browser instances

Ques.30. How to switch between multiple windows in selenium?

Ans. Selenium has *driver.getWindowHandles()* and *driver.switchTo().window("{windowHandleName}")* commands to work with multiple windows. The getWindowHandles() command returns a list of ids corresponding to each window and on passing a particular window handle to driver.switchTo().window("{windowHandleName}") command we can switch control/focus to that particular window.

```
for (String windowHandle : driver.getWindowHandles()) {
     driver.switchTo().window(handle);
}
```

Ques.31. What is the difference between driver.getWindowHandle() and driver.getWindowHandles() in selenium?

Ans. driver.getWindowHandle() returns a handle of the current page (a unique identifier)
Whereas driver.getWindowHandles() returns a set of handles of the all the pages available.

Ques.32. How can we move to a particular frame in selenium?

Ans. The *driver.switchTo()* commands can be used for switching to frames.

```
driver.switchTo().frame("{frameIndex/frameId/frameName}");
```

For locating a frame we can either use the index (starting from 0), its name or Id.

Ques.33. Can we move back and forward in browser using selenium?

Ans. Yes, using *driver.navigate().back()* and *driver.navigate().forward()* commands we can move backward and forward in a browser.

Ques.34. Is there a way to refresh browser using selenium?

Ans. There a multiple ways to refresh a page in selenium-

- Using *driver.navigate().refresh()* command
- Using *sendKeys(Keys.F5)* on any textbox on the webpage
- Using driver.get("URL") on the current URL or using *driver.getCurrentUrl()*
- Using driver.navigate().to("URL") on the current URL or *driver.navigate().to(driver.getCurrentUrl());*

Ques.35. How can we maximize browser window in selenium?

Ans. We can maximize browser window in selenium using following command-

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

Ques.36. How can we fetch a text written over an element?

Ans. Using *getText()* method we can fetch the text over an element.

```
String text = driver.findElement("elementLocator").getText();
```

Ques.37. How can we find the value of different attributes like name, class, value of an element?

Ans. Using *getAttribute("{attributeName}")* method we can find the value of different attrbutes of an element e.g.-

```
String valueAttribute =
driver.findElement(By.id("elementLocator")).getAttribute("value");
```

Ques.38. How to delete cookies in selenium?

Ans. Using deleteAllCookies() method-

```
driver.manage().deleteAllCookies();
```

Ques.39. What is an implicit wait in selenium?

Ans. An implicit wait is a type of wait which waits for a specified time while locating an element before throwing NoSuchElementException. By default selenium tries to find elements immediately when required without any wait. So, it is good to use implicit wait. This wait is applied to all the elements of the current driver instance.

```
driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
```

Ques.40. What is an explicit wait in selenium?

Ans. An explicit wait is a type of wait which is applied to a particular web element untill the expected condition specified is met.

```
WebDriverWait wait = new WebDriverWait(driver, 10);

WebElement element = wait.until(ExpectedConditions.elementToBeClickable(By.id("elemen
tId")));
```

Ques.41. What are some expected conditions that can be used in Explicit waits?

Ans. Some of the commonly used expected conditions of an element that can be used with expicit waits are-

- elementToBeClickable(WebElement element or By locator)
- stalenessOf(WebElement element)
- visibilityOf(WebElement element)
- visibilityOfElementLocated(By locator)
- invisibilityOfElementLocated(By locator)
- attributeContains(WebElement element, String attribute, String value)
- alertIsPresent()
- titleContains(String title)
- titleIs(String title)
- textToBePresentInElementLocated(By, String)

Ques.42. What is fluent wait in selenium?

Ans. A fluent wait is a type of wait in which we can also specify polling interval(intervals after which driver will try to find the element) along with the maximum timeout value.

```
Wait wait = new FluentWait(driver)

    .withTimeout(20, SECONDS)

    .pollingEvery(5, SECONDS)

    .ignoring(NoSuchElementException.class);

  WebElement textBox = wait.until(new Function<webdriver,webElement>() {

    public WebElement apply(WebDriver driver) {

    return driver.findElement(By.id("textBoxId"));

    }
}
);
```

Ques.43. What are the different keyboard operations that can be performed in selenium?

Ans. The different keyboard operations that can be performed in selenium are-

1. **.sendKeys("sequence of characters")** - Used for passing character sequence to an input or textbox element.
2. **.pressKey("non-text keys")** - Used for keys like control, function keys etc that are non-text.
3. **.releaseKey("non-text keys")** - Used in conjuntion with keypress event to simulate releasing a key from keyboard event.

Ques.44. What are the different mouse actions that can be performed?

Ans. The different mouse evenets supported in selenium are

1. click(WebElement element)
2. doubleClick(WebElement element)
3. contextClick(WebElement element)
4. mouseDown(WebElement element)

5. mouseUp(WebElement element)
6. mouseMove(WebElement element)
7. mouseMove(WebElement element, long xOffset, long yOffset)

Ques.45. Write the code to double click an element in selenium?

Ans. Code to double click an element in selenium-

```
Actions action = new Actions(driver);
WebElement element=driver.findElement(By.id("elementId"));
action.doubleClick(element).perform();
```

Ques.46. Write the code to right click an element in selenium?

Code to right click an element in selenium-

```
Actions action = new Actions(driver);
WebElement element=driver.findElement(By.id("elementId"));
action.contextClick(element).perform();
```

Ques.47. How to mouse hover an element in selenium?

Ans. Code to mouse hover over an element in selenium-

```
Actions action = new Actions(driver);
WebElement element=driver.findElement(By.id("elementId"));
action.moveToElement(element).perform();
```

Ques.48. How to fetch the current page URL in selenium?

Ans. Using getCurrentURL() command we can fetch the current page URL-

```
driver.getCurrentUrl();
```

Ques.49. How can we fetch title of the page in selenium?

Ans. Using driver.getTitle(); we can fetch the page title in selenium. This method returns a string containing the title of the webpage.

Ques.50. How can we fetch the page source in selenium?

Ans. Using driver.getPageSource(); we can fetch the page source in selenium. This method returns a string containing the page source.

Ques.51. How to verify tooltip text using selenium?

Ans. Tooltips webelements have an attribute of type 'title'. By fetching the value of 'title' attribute we can verify the tooltip text in selenium.

```
String toolTipText = element.getAttribute("title");
```

Ques.52. How to locate a link using its text in selenium?

Ans. Using linkText() and partialLinkText() we can locate a link. The difference between the two is linkText matches the complete string passed as parameter to the link texts. Whereas partialLinkText matches the string parameter partially with the link texts.

```
WebElement link1 = driver.findElement(By.linkText("artOfTesting"));
WebElement link2 = driver.findElement(By.partialLinkText("artOf"));
```

Ques.53. What are DesiredCapabilities in selenium webdriver?

Ans. Desired capabilities are a set of key-value pairs that are used for storing or configuring browser specific properties like its version, platform etc in the browser instances.

Ques.54. How can we find all the links on a web page?

Ans. All the links are of anchor tag 'a'. So by locating elements of tagName 'a' we can find all the links on a webpage.

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

Ques.55. What are some commonly encountered exceptions in selenium?

Ans. Some of the commonly seen exception in selenium are-

- NoSuchElementException - When no element could be located from the locator provided.
- ElementNotVisibleException - When element is present in the dom but is not visible.
- NoAlertPresentException - When we try to switch to an alert but the targetted alert is not present.
- NoSuchFrameException - When we try to switch to a frame but the targetted frame is not present.
- NoSuchWindowException - When we try to switch to a window but the targetted window is not present.
- UnexpectedAlertPresentException - When an unexpected alert blocks normal interaction of the driver.
- TimeoutException - When a command execution gets timeout.
- InvalidElementStateException - When the state of an element is not appropriate for the desired action.
- NoSuchAttributeException - When we are trying to fetch an attribute's value but the attribute is not correct
- WebDriverException - When there is some issue with driver instance preventing it from getting launched.

Ques.56. How can we capture screenshots in selenium?

Ans. Using getScreenshotAs method of TakesScreenshot interface we can take the screenshots in selenium.

```
File scrFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
FileUtils.copyFile(scrFile, new File("D:\\testScreenShot.jpg"));
```

Ques.57. How to handle dropdowns in selenium?

Ans. Using Select class-

```
Select countriesDropDown = new Select(driver.findElement(By.id("countries")));
dropdown.selectByVisibleText("India");

//or using index of the option starting from 0
dropdown.selectByIndex(1);

//or using its value attribute
```

```
dropdown.selectByValue("Ind");
```

Ques.58. How to check which option in the dropdown is selected?

Ans. Using isSelected() method we can check the state of a dropdown's option.

```
Select countriesDropDown = new Select(driver.findElement(By.id("countries")));
dropdown.selectByVisibleText("India");

//returns true or false value
System.out.println(driver.findElement(By.id("India")).isSelected());
```

Ques.59. How can we check if an element is getting displayed on a web page?

Ans. Using isDisplayed method we can check if an element is getting displayed on a web page.

```
driver.findElement(By locator).isDisplayed();
```

Ques.60. How can we check if an element is enabled for interaction on a web page?

Ans. Using isEnabled method we can check if an element is enabled or not.

```
driver.findElement(By locator).isEnabled();
```

Ques.61. What is the difference between driver.findElement() and driver.findElements() commands?

Ans. The difference between driver.findElement() and driver.findElements() commands is-

- findElement() returns a single WebElement (found first) based on the locator passed as parameter. Whereas findElements() returns a list of WebElements, all satisfying the locator value passed.
- Syntax of findElement()- WebElement textbox = driver.findElement(By.id("textBoxLocator"));

Syntax of findElements()-
List <WebElement> elements = element.findElements(By.id(“value”));

- Another difference between the two is- if no element is found then findElement() throws NoSuchElementException whereas findElements() returns a list of 0 elements.

Ques.62. Explain the difference between implicit wait and explicit wait.?

Ans. An implicit wait, while finding an element waits for a specified time before throwing NoSuchElementException in case element is not found. The timeout value remains valid throughout the webDriver's instance and for all the elements.

```
driver.manage().timeouts().implicitlyWait(180, TimeUnit.SECONDS);
```

Whereas, Explicit wait is applied to a specified element only-

```
WebDriverWait wait = new WebDriverWait(driver, 5);
wait.until(ExpectedConditions.presenceOfElementLocated(ElementLocator));
```

It is advisable to use explicit waits over implicit waits because higher timeout value of implicit wait set due to an element that takes time to be visible gets applied to all the elements. Thus increasing overall execution time of the script. On the other hand, we can apply different timeouts to different element in case of explicit waits.

Ques.63. How can we handle window UI elements and window POP ups using selenium?

Ans. Selenium is used for automating Web based application only(or browsers only). For handling window GUI elements we can use AutoIT. AutoIT is a freeware used for automating window GUI. The AutoIt scripts follow simple BASIC lanaguage like syntax and can be easily integrated with selenium tests.

Ques.64. What is Robot API?

Ans. Robot API is used for handling Keyboard or mouse events. It is generally used to upload files to the server in selenium automation.

```
Robot robot = new Robot();

//Simulate enter key action
robot.keyPress(KeyEvent.VK_ENTER);
```

Ques.65. How to do file upload in selenium?

Ans. File upload action can be performed in multiple ways-

1. Using element.sendKeys("path of file") on the webElement of input tag and type file i.e. the elements should be like -

```
2. <input type="file" name="fileUpload">
```

3. Using Robot API.
4. Using AutoIT API.

Ques.66. How to handle HTTPS website in selenium? or How to accept the SSL untrusted connection?

Ans. Using profiles in firefox we can handle accept the SSL untrusted connection certificate. Profiles are basically set of user preferences stored in a file.

```
FirefoxProfile profile = new FirefoxProfile();
profile.setAcceptUntrustedCertificates(true);
profile.setAssumeUntrustedCertificateIssuer(false);
WebDriver driver = new FirefoxDriver(profile);
```

Ques.67 How to do drag and drop in selenium?

Using Action class, drag and drop can be performed in selenium. Sample code-

```
Actions builder = new Actions(driver);
Action dragAndDrop = builder.clickAndHold(SourceElement)
```

```
.moveToElement(TargetElement)
.release(TargetElement)
.build();
dragAndDrop.perform();
```

Ques.68. How to execute javascript in selenium?

Ans. JavaScript can be executed in selenium using JavaScriptExecuter. Sample code for javascript execution-

```
WebDriver driver = new FireFoxDriver();
if (driver instanceof JavascriptExecutor) {
        ((JavascriptExecutor)driver).executeScript("{JavaScript Code}");
}
```

Ques.69. How to handle alerts in selenium?

Ans. In order to accept or dismiss an alert box the alert class is used. This requires first switching to the alert box and than using accept() or dismiss() command as the case may be.

```
Alert alert = driver.switchTo().alert();
//To accept the alert
alert.accept();

Alert alert = driver.switchTo().alert();
//To cancel the alert box
alert.dismiss();
```

Ques.70. What is HtmlUnitDriver?

Ans. HtmlUnitDriver is the fastest WebDriver. Unlike other drivers (FireFoxDriver, ChromeDriver etc), the HtmlUnitDriver is non-GUI, while running no browser gets launched.

Ques.71. How to handle hidden elements in Selenium webDriver?

Ans. Using javaScript executor we can handle hidden elements-
(JavascriptExecutor(driver))
.executeScript("document.getElementsByClassName(ElementLocator).click()
;");

Ques.72. What is Page Object Model or POM?

Ans. Page Object Model(POM) is a design pattern in selenium. A design pattern is a solution or a set of standards that are used for solving commonly occuring software problems.
Now coming to POM - POM helps to create a framework for maintaining selenium scripts. In POM for each page of the application a class is created having the web elements belonging to the page and methods handling the events in that page. The test scripts are maintained in seperate files and the methods of the page object files are called from the test scripts file.

Ques.73. What are the advantages of POM?

Ans. The advantages are POM are-

1. Using POM we can create an Object Repository, a set of web elements in seperate files along with their associated functions. Thereby keeping code clean.
2. For any change in UI(or web elements) only page object files are required to be updated leaving test files unchanged.
3. It makes code reusable and maintable.

Ques.74. What is Page Factory?

Ans. Page factory is an implementation of Page Object Model in selenium. It provides @FindBy annotation to find web elements and PageFactory.initElements() method to initialize all web elements defined with @FindBy annotation.

```
public class SamplePage {

    WebDriver driver;

    @FindBy(id="search")
    WebElement searchTextBox;

    @FindBy(name="searchBtn")
    WebElement searchButton;

    //Constructor
    public samplePage(WebDriver driver){
        this.driver = driver;
        //initElements method to initialize all elements
        PageFactory.initElements(driver, this);
    }

    //Sample method
    public void search(String searchTerm){
        searchTextBox.sendKeys(searchTerm);
        searchButton.click();
    }
}
```

Ques.75. What is an Object repository?

Ans. An object repository is centralized location of all the object or WebElements of the test scripts. In selenium we can create object repository using Page Object Model and Page Factory design patterns.

Ques.76. What is a data driven framework?

Ans. A data driven framework is one in which the test data is put in external files like csv, excel etc separated from test logic written in test script files. The test data drives the test cases, i.e. the test methods run for each set of test data values. TestNG provides inherent support for data driven testing using @dataProvider annotation.

Ques.77. What is a keyword driven framework?

Ans. A keyword driven framework is one in which the actions are associated with keywords and kept in external files e.g. an action of launching a browser will be associated with keyword - launchBrowser(), action to write in a textbox with keyword - writeInTextBox(webElement, textToWrite) etc. The code to perform the action based on a keyword specified in external file is implemented in the framework itself.
In this way the test steps can be written in a file by even a person of non-programming background once all the identified actions are implemented.

Ques.78. What is a hybrid framework?

Ans. A hybrid framework is a combination of one or more frameworks. Normally it is associated with combination of data driven and keyword driven frameworks where both the test data and test actions are kept in external files(in the form of table).

Ques.79. What is selenium Grid?

Ans. Selenium grid is a tool that helps in distributed running of test scripts across different machines having different browsers, browser version, platforms etc in parallel. In selenium grid there is hub that is a central server managing all the distributed machines known as nodes.

Ques.80. What are some advantages of selenium grid?

Ans. The advantages of selenium grid are-

1. It allows running test cases in parallel thereby saving test execution time.
2. Multi browser testing is possible using selenium grid by running the test on machines having different browsers.
3. It is allows multi-platform testing by configuring nodes having different operating systems.

Ques.81. What is a hub in selenium grid?

Ans. A hub is server or a central point in selenium grid that controls the test executions on the different machines.

Ques.82. What is a node in selenium grid?

Ans. Nodes are the machines which are attached to the selenium grid hub and have selenium instances running the test scripts. Unlike hub there can be multiple nodes in selenium grid.

Ques.83. Explain the line of code Webdriver driver = new FirefoxDriver();.

Ans. In the line of code ***Webdriver driver = new FirefoxDriver();*** 'WebDriver' is an interface and we are creating an object of type WebDriver instantiating an object of FirefoxDriver class.

Ques.84 What is the purpose of creating a reference variable- 'driver' of type WebDriver instead of directly creating a FireFoxDriver object or any other driver's reference in the statement Webdriver driver = new FirefoxDriver();?

Ans. By creating a reference variable of type WebDriver we can use the same variable to work with multiple browsers like ChromeDriver, IEDriver etc.

Ques.85. What is testNG?

Ans. TestNG(NG for Next Generation) is a testing framework that can be integrated with selenium or any other automation tool to provide multiple capabilities like assertions, reporting, parallel test execution etc.

Ques.86. What are some advantages of testNG?

Ans. Following are the advantages of testNG-

1. TestNG provides different assertions that helps in checking the expected and actual results.
2. It provides parallel execution of test methods.
3. We can define dependency of one test method over other in TestNG.
4. We can assign priority to test methods in selenium.
5. It allows grouping of test methods into test groups.
6. It allows data driven testing using @DataProvider annotation.
7. It has inherent support for reporting.
8. It has support for parameterizing test cases using @Parameters annotation.

Ques.87. What is the use of testng.xml file?

Ans. testng.xml file is used for configuring the whole test suite. In testng.xml file we can create test suite, create test groups, mark tests for parallel execution, add listeners and pass parameters to test scripts. Later this testng.xml file can be used for triggering the test suite.

Ques.88. How can we pass parameter to test script using testNG?

Ans. Using @Parameter annotation and 'parameter' tag in testng.xml we can pass parameters to the test script.
Sample testng.xml -

```
<suite name="sampleTestSuite">
   <test name="sampleTest">
      <parameter name="sampleParamName" value="sampleParamValue"/>
      <classes>
         <class name="TestFile" />
      </classes>
   </test>
</suite>
```

Sample test script-

```
public class TestFile {
   @Test
   @Parameters("sampleParamName")
   public void parameterTest(String paramValue) {
      System.out.println("Value of sampleParamName is - " + sampleParamName);
   }
```

Ques.89. How can we create data driven framework using testNG?

Ans. Using @DataProvider we can create a data driven framework in which data is passed to the associated test method and multiple iteration of the test runs for the different test data values passed from the @DataProvider method. The method annotated with @DataProvider annotation return a 2D array of object.

```
//Data provider returning 2D array of 3*2 matrix
 @DataProvider(name = "dataProvider1")
   public Object[][] dataProviderMethod1() {
      return new Object[][] {{"kuldeep","rana"}, {"k1","r1"},{"k2","r2"}};
   }

   //This method is bound to the above data provider returning 2D array of 3*2 matrix
   //The test case will run 3 times with different set of values
   @Test(dataProvider = "dataProvider1")
   public void sampleTest(String s1, String s2) {
      System.out.println(s1 + " " + s2);
   }
```

Ques.90. What is the use of @Listener annotation in TestNG?

Ans. TestNG provides us different kind of listeners using which we can perform some action in case an event has triggered. Usually testNG listeners are used for configuring reports and logging. One of the most widely used lisetner in testNG is ITestListener interface. It has methods like onTestSuccess, onTestFailure, onTestSkipped etc. We need to implement this interface creating a listener class of our own. After that using the *@Listener* annotation, we can use specify that for a particular test class, our customized listener class should be used.

```
@Listeners(PackageName.CustomizedListenerClassName.class)

public class TestClass {
    WebDriver driver= new FirefoxDriver();

    @Test
    public void testMethod(){
    //test logic
    }
}
```

Ques.91. What is the use of @Factory annotation in TestNG?

Ans. @Factory annotation helps in dynamic execution of test cases. Using @Factory annotation we can pass parameters to the whole test class at run time. The parameters passed can be used by one or more test methods of that class.

Example - there are two classes TestClass and the TestFactory class. Because of the @Factory annotation the test methods in class TestClass will run twice with the data "k1" and "k2"

```
public class TestClass{
    private String str;

    //Constructor
    public TestClass(String str) {
        this.str = str;
    }

    @Test
    public void TestMethod() {
        System.out.println(str);
    }
}

public class TestFactory{
    //The test methods in class TestClass will run twice with data "k1" and "k2"
    @Factory
    public Object[] factoryMethod() {
        return new Object[] { new TestClass("K1"), new TestClass("k2") };
    }
}
```

Ques.92. What is difference between @Factory and @DataProvider annotation?

Ans. @Factory method creates instances of test class and run all the test methods in that class with different set of data. Whereas, @DataProvider is bound to individual test methods and run the specific methods multiple times.

Ques.93. How can we make one test method dependent on other using TestNG?

Ans. Using dependsOnMethods parameter inside @Test annotation in testNG we can make one test method run only after successful execution of dependent test method.

```
 @Test(dependsOnMethods = { "preTests" })
```

Ques.94. How can we set priority of test cases in TestNG?

Ans. Using priority parameter in @Test annotation in TestNG we can define priority of test cases. The default priority of test when not specified is integer value 0. Example-

```
@Test(priority=1)
```

Ques.95. What are commonly used TestNG annotations?

Ans. The commonly used TestNG annotations are-

- @Test- @Test annotation marks a method as Test method.
- @BeforeSuite- The annotated method will run only once before all tests in this suite have run.
- @AfterSuite-The annotated method will run only once after all tests in this suite have run.
- @BeforeClass-The annotated method will run only once before the first test method in the current class is invoked.
- @AfterClass-The annotated method will run only once after all the test methods in the current class have been run.
- @BeforeTest-The annotated method will run before any test method belonging to the classes inside the <test> tag is run.
- @AfterTest-The annotated method will run after all the test methods belonging to the classes inside the <test> tag have run.

Ques.96. What are some common assertions provided by testNG?

Ans. Some of the common assertions provided by testNG are-

1. assertEquals(String actual, String expected, String message) - (and other overloaded data type in parameters)
2. assertNotEquals(double data1, double data2, String message) - (and other overloaded data type in parameters)
3. assertFalse(boolean condition, String message)
4. assertTrue(boolean condition, String message)
5. assertNotNull(Object object)
6. fail(boolean condition, String message)
7. true(String message)

Ques.97. How can we run test cases in parallel using TestNG?

Ans. In order to run the tests in parallel just add these two key value pairs in suite-

- parallel="{methods/tests/classes}"
- thread-count="{number of thread you want to run simultaneously}".

```
<suite name="ArtOfTestingTestSuite" parallel="methods" thread-count="5">
```

Check Running Selenium Tests in parallel for details.

Ques.98. Name an API used for reading and writing data to excel files.

Ans. Apache POI API and JXL(Java Excel API) can be used for reading, writing and updating excel files.

Ques.99. Name an API used for logging in Java.

Ans. Log4j is an open source API widely used for logging in Java. It supports multiple levels of logging like - ALL, DEBUG, INFO, WARN, ERROR, TRACE and FATAL.

Ques.100. What is the use of logging in automation?

Ans. Logging helps in debugging the tests when required and also provides a storage of test's runtime behaviour.