SELENIUM & TESTNG

A must-have knowledge base, for every Test professional

"The de-facto library for UI automation. Every Test professional must have KNOWLEDGE about it."

Selenium Galaxy

Selenium is the de-facto API/Library for Web Automation now-a-days. But it doesn't work in isolation. There are other tools that complete the Selenium Galaxy of automation.

- Selenium Webdriver: web browser automation.
- IDE like Eclipse/IntelliJ: a text editor with additional support for developing, compiling and debugging applications.
- TestNG: a unit framework inspired from JUnit and NUnit.
- Build Automation like Maven/Ant/Gradle: automating the creation of a software build and the associated processes.
- Cucumber: A Behavior Driven Development (BDD) framework.
- Jenkins: for Continuous Integration.

Some other not-so-popular tools to add,

- AutoIT: to work on windows-based popups.
- Apache POI: to work on Office documents (read/write).
- ExtentReports/ReportNG: generate HTML logs & reports.

Selenium Positive & Negative aspects

"Selenium", the de-facto top automation testing library. The Positives are quite easy,

- Open-source testing tool.
- Capability to operate on almost every OS.
- Supports multiple languages.
- Independent of the language that the web application is using.
- Supports a range of browsers.
- Very dynamic developer & helping community and user base.
- Robust element locators.
- Convenient to implement frameworks that revolve around OOP.

- Support for integration of open-source frameworks like TestNG + Version control using Maven, Jenkins.
- Execute simultaneous tests leveraging various browsers on various machines.
- Supports Web and Mobile Web Applications.

But what about the negatives? Instead of calling it 'Negative', call it as 'Limitations',

- No dedicated official technical support.
- Supports only Web based applications.
- Dependence on third-party tools for complete benefit, unlike proprietary all-in-one tools.
- Limited support for Image-based Testing, Captcha and Bar code readers.
- Unstable new releases. It evolves with time.
- No implicit Test Tool integration for Test Management.
- No Built-in Reporting facility.
- Lot of challenges with IE browser.

Being an open-source tool, these limitations are okay to live with.

Type of locators in Selenium

The locator can be termed as an address that identifies a web element uniquely within the webpage.

ID | ClassName | Name | TagName | LinkText | PartialLinkText | Xpath | CSS Selector | DOM

What is the difference between "/" and "//" in Xpath?

- Single Slash "/": used to create Xpath with absolute path i.e. the xpath would be created to start selection from the document root node/start node.

- Double Slash "//": Double slash is used to create Xpath with relative path i.e. the xpath would be created to start selection from anywhere within the document.

Types of wait available in WebDriver

- **Implicit Wait**: used to provide a default waiting time (say 30 seconds) between each consecutive test step/command across the entire test script. Thus, the subsequent test step would only execute when the 30 seconds have elapsed after executing the previous test step/command.

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

- **Explicit Wait**: used to halt the execution till the time a particular condition is met or the maximum time has elapsed. Unlike Implicit waits, explicit waits are applied for a particular instance only.

  ```
  WebDriverWait wait = new WebDriverWait(WebDriver Reference, TimeOut);
  ```

- **Fluent wait**: used to wait for a condition, as well as the frequency with which we want to check the condition.

  ```
  Wait wait = new FluentWait(WebDriver reference).withTimeout(timeout,
  SECONDS).pollingEvery(timeout, SECONDS).ignoring(Exception.class);
  ```

How can we get a text of a web element?

getText() is used to retrieve the inner text of the specified web element. The command doesn’t require any parameter but returns a string value. It is also one of the extensively used commands for verification of messages, labels, errors etc. displayed on the web pages.

```
String text = driver.findElement(By.id(“Text”)).getText();
```

How to select value in a dropdown?

Using WebDriver’s Select class.

selectByValue:

```
Select selectByValue = new Select(driver.findElement(By.id(“SelectID_One”)));
selectByValue.selectByValue(“greenvalue”);
```

selectByVisibleText:

```
Select selectByVisibleText = new Select (driver.findElement(By.id(“SelectID_Two”)));
selectByVisibleText.selectByVisibleText(“Lime”);
```

selectByIndex:

```
Select selectByIndex = new Select(driver.findElement(By.id(“SelectID_Three”)));
selectByIndex.selectByIndex(2);
```

findElement() and findElements()

- **findElement():** used to find the first element in the current web page matching to the specified locator value. Take a note that only first matching element would be fetched.

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

- **findElements():** used to find all the elements in the current web page matching to the specified locator value. All the matching elements would be fetched and stored in the list of WebElements.

```
List <WebElement> elementList =
driver.findElements(By.xpath("//div[@id='example']//ul//li"));
```

driver.close() and driver.quit command

- **close():** closes the web browser window that the user is currently working on OR currently being accessed by the WebDriver. The command neither requires any parameter nor does it return any value.

- **quit():** Unlike close() method, quit() method closes down all the windows that the program has opened. Neither requires any parameter nor does is return any value.

How can we handle web-based pop-up?

WebDriver offers the users a very efficient way to handle these pop-ups using Alert interface. There are four methods that we would be using along with the Alert interface.

- void dismiss(): clicks on the "Cancel" button as soon as the pop-up window appears.
- void accept(): clicks on the "Ok" button as soon as the pop-up window appears.
- String getText(): returns the text displayed on the alert box.
- void sendKeys(String stringToSend): enters the specified string pattern into the alert box.

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

How can we handle windows-based pop up?

Selenium is an automation testing tool which supports only web application testing, that means, it doesn't support testing of windows-based applications. However, with some third-party intervention, this problem can be overcome. There are several third-party tools available for handling window-based pop-ups along with the selenium like AutoIT, Robot class etc.

How to retrieve CSS properties of an element?

Using getCssValue() method:

```
driver.findElement(By.id("id")).getCssValue("color");
driver.findElement(By.id("id")).getCssValue("font-size");
```

How to capture screenshot in WebDriver?

TakeScreenshot interface can be used to take screenshots in WebDriver. getScreenshotAs() method is used to save the screenshot.

```
import org.openqa.selenium.TakesScreenshot;
public void test() throws IOException
{
	File scrFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
	FileUtils.copyFile(scrFile, new File("C:\\CaptureScreenshot\\google.jpg"));
}
```

Navigation commands

- driver.navigate().to("https://www.softwaretestingstudio.in/");
- driver.navigate().refresh();
- driver.navigate().forward(); --- just like clicking on the Forward Button of any browser.
- driver.navigate().back(); --- just like clicking on the Back Button of any browser.

How to scroll down a page using JavaScript?

scrollBy() method is used to scroll down the webpage | executeScript("window.scrollBy(x-pixels,y-pixels)");

```
JavascriptExecutor js = (JavascriptExecutor) driver;
driver.get("https://www.softwaretestingstudio.com");
js.executeScript("window.scrollBy(0,1000)");
```

How to mouse hover over a web element?

Actions class utility is used to hover over a web element in Selenium WebDriver

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

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

Yes! Text can be set using JavaScriptExecutor

```
JavascriptExecutor jse = (JavascriptExecutor) driver;
jse.executeScript("document.getElementById('email').value="abc.efg@xyz.com");
```

How to upload a file in Selenium WebDriver?

Using sendkeys() or Robot class method. Locate the text box and set the file path using sendkeys() and click on submit button.

```
WebElement browse =driver.findElement(By.id("uploadfile"));
browse.sendKeys("D:\\STS\\UploadFile.txt");
```

How to login to any site if it is showing an Authentication Pop-Up for Username and Password?

To handle authentication pop-ups, verify its appearance and then handle them using an explicit wait command.

```
WebDriverWait wait = new WebDriverWait(driver, 10);
Alert alert = wait.until(ExpectedConditions.alertIsPresent());
alert.authenticateUsing(new UserAndPassword(<username>, <password>));
```

Explain how you can switch between frames?

Use driver.switchTo().frame() to switch between frames. By using,

- A number: It selects the number by its (zero-based) index
- A name or ID: Select a frame by its name or ID
- Previously found WebElement: Using its previously located WebElement select a frame

For Database Testing in Selenium Webdriver what API is required?

We need JDBC (Java Database Connectivity) API. It allows us to execute SQL statements.

Write a code to wait for a element to be visible on a page.

We can write a code such that we specify the XPath of the web element that needs to be visible on the page and then ask the WebDriver to wait for a specified time.

```
WebDriverWait wait=new WebDriverWait(driver, 20);
Element = wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath( "<xpath"")));
```

How to scroll down to a particular element?

To scroll down to a particular element on a web page, we can use the function scrollIntoView().

```
((JavascriptExecutor) driver).executeScript("arguments[0].scrollIntoView();", element);
```

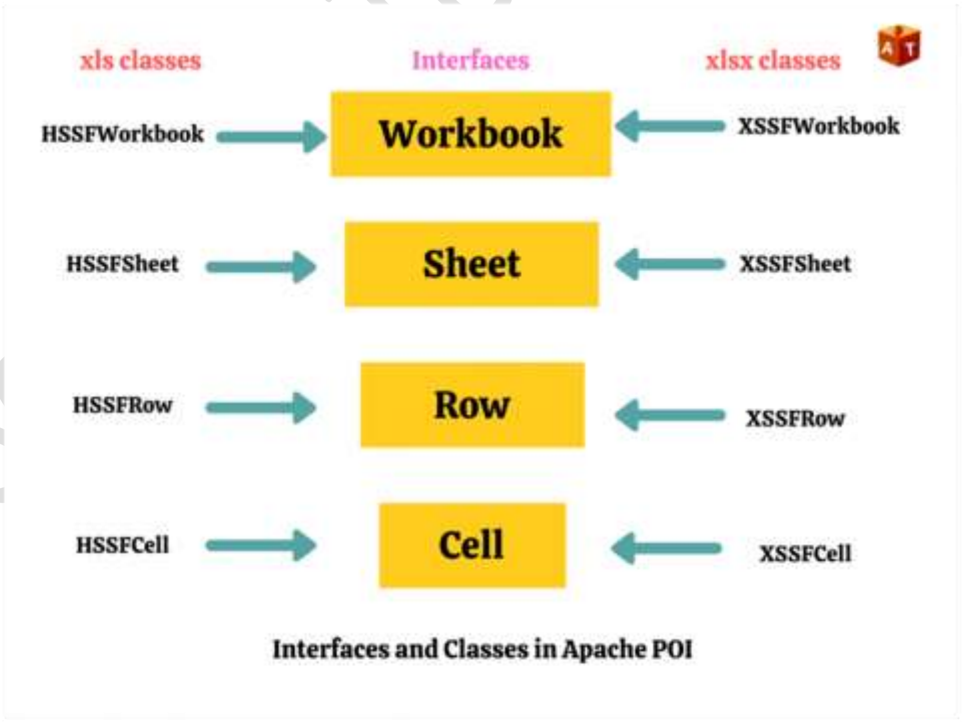
How to handle multiple windows in Selenium?

A window handle is a unique identifier that holds the address of all the windows. This is basically a pointer to a window, which returns the string value.

- get.windowhandle(): helps in getting the window handle of the current window.
- get.windowhandles(): helps in getting the handles of all the windows opened.
- set: helps to set the window handles which is in the form of a string.
- switch to: helps in switching between the windows.
- action: helps to perform certain actions on the windows.

It is always advisable to keep Test Data separate from Automation scripts. In Selenium, how do you read data from an Excel file using Java?

Apache POI is an Open-source Java library to manipulate file formats based on Microsoft Office. We use XSSF (XML SpreadSheet Format) to read an Excel (.xlsx) file format.



1. Since we have to read contents - Open existing excel using a File Input Stream.
2. Get Workbook >> & then specific Sheet.
3. Iterate through every row in excel file.
   3.a. Iterate through every cell within each row.
   3.b. Print the cell value.
4. Flush & Close the Input Stream.

```
File src = new File("excel file path");
FileInputStream fis = new FileInputStream(src);
XSSFWorkbook wb = new XSSFWorkbook(fis);
XSSFSheet sheet = wb.getSheet(sheetName);
XSSFRow row; XSSFCell cell;
Iterator rows = sheet.rowIterator();
while (rows.hasNext()) {
    row = (XSSFRow) rows.next();
    Iterator cells = row.cellIterator();
    while (cells.hasNext()) {
        cell = (XSSFCell) cells.next();
        System.out.println(cell.getStringCellValue());
    }
}
fis.flush();
fis.close();
```

Which is better to use in Selenium automation? Sleep or Wait?

Both Thread.sleep and Object.wait make the current thread wait for a specified amount of time. This is useful when the current thread needs to wait for other thread before it can proceed. For example, web element might not be clickable or enabled, or in the expected state.

Instead of using Thread.sleep, a better approach is to check for right state. Selenium provides a set of common ExpectedConditions that can be leveraged on top of webdriverwait. Unlike Thread.sleep (where locks are not released), in Object.wait locks are released as the thread goes to sleep. This approach helps make the test more stable and reliable.

What's 'DataProvider' in TestNG? Any examples...

What does the name 'DataProvider' suggest? Yeah! Something that provides data. Test Data is important in Software Testing, say you want to login to an application – it needs a username and password values – it's nothing but the Test data.

Method1 – Data Provider method – annotated with @DataProvider - Name 'LoginData' – returns an array of objects.

```
public class DataProviderClass {
   @DataProvider(name = "LoginData")
   public static Object[][] dataProviderMethod() {
      return new Object[][] { {"Username1", "Password@1" }, {"Username2", "Password@2"}
};
   }
}
```

Rows: # times test needs to be repeated.
Columns: # parameters in Test method

Method2 – Test method 'Login' – Executed using data provided by Method1-LoginData.

```
public class TestClass {
   @Test(dataProvider = "LoginData", dataProviderClass = DataProviderClass.class)
   public void Login(String username, String password) {
      System.out.println("Username: " + username + " and Password: " + password);
   }
}
```

Output –

```
Username: Username1 and Password: Password@1
Username: Username2 and Password: Password@2
```

Write data-driven tests (same test run multiple times with diff. Test data).

How do you read data from a Web Table using Selenium?

Simple, identify the table >> Go to row-2 >> then column-3 >> and then read data ;-) For dynamic table it's a bit different – just before going to specific cell we need to know the no of rows & columns rendered. Post that, steps will be same to fetch specific data.

Identify the table,

```
WebElement Table = driver.findElement(By.ByClassName.className("dataTable"));
```

Read specific cell value,

```
WebElement cell = driver.findElement(By.xpath("//table/tbody/tr[2]/td[3]"));
System.out.println("Row 2 and Column 3: "+cell.getText()+".");
```

Fetch # of rows in a table,

```
List<WebElement> rows = Table.findElements(By.tagName("tr"));
rows_count = rows.size();
```

Fetch # of columns in a row,

```
List<WebElement> columns = rows.get(i).findElements(By.tagName("td"));
col_count = columns.size();
```

Iterate through the table,

```
for(int i= 0; i<rows_count; i++){
List<WebElement> columns = rows.get(i).findElements(By.tagName("td"));
        col_count = columns.size();
        for(int j=0; j< col_count; j++){
                cellText = columns.get(j).getText();
                System.out.print(cellText+"  ");
        }
        System.out.println("");
}
```

Why do we use PageFactory?

Why do we use PageFactory? Yeah! To initialize the web elements.
But why can't we use the simple,

```
        LoginPage loginPage = new LoginPage(driver);
```

Instead of using,

```
        LoginPage loginPage = PageFactory.initElements(driver, LoginPage.class);
```

This is one query that I forgot to study about. Use of PageFactory is NOT mandatory and it seems to be a stylistic choice. The difference is quite simple,

- Initialize elements to avoid NullPointerExceptions.
- Use of annotations for readability.

Why will you get NullPointerExceptions? Because you might forget to initialize a WebElement and use it. I.e.

```
private WebElement test;
public void tptest() {
        test.sendKeys("NullPointer"); //test element is not initialized yet.
}
```

PageFactory automatically initializes the Page element whenever a call is made. How? Simply by replacing a by b.

a. test.sendKeys("NullPointer");
b. driver.findElement(By.id("test")).sendKeys("NullPointer"); //or can use name instead of id.

This way, you won't get a NullPointerException. Though you might get 'ElementNotFound' one :P

Selenium assert or verify? Which one do you prefer?

Basic Difference: 'assert' terminates the execution whereas 'verify' marks the step as fail but continue the execution.

It is recommended to use a combination of assert and verify.

- assert: to check if the entry criteria are met. E.g., Login successful OR If the window is available - then further validate the fields - else stop. Just like a Sanity check :P to continue further execution.
- verify: to validate the individual steps within a test case.

'verify' helps in gauging the script stability, i.e., if the script is executing till the end. We can then analyze the individual step failures together.

Using 'assert' you go step-by-step in script fixing.

Note: There are arguments for/against both in the technical landscape :)

What all concepts/tools do you use to create Selenium logs & reports?

Before we list some options, let's look at Listeners first.

Listeners: As the name suggests, these are the spectators that watch over your test execution (or listening) and react (log/report) to various events like click, type, move, etc.

- WebDriver Listeners: listen the events triggered by webdriver like beforeClickOn, afterClickOn, beforeFindBy, afterFindBy, etc. and take actions.
- TestNG reporter: Reporter is a separate class in TestNG to log some text after the desired action.
- TestNG Listeners: TestNG provides many types of listeners but mostly used ones are ISuiteListener & ITestListener.
- ExtentReports: Extent Reports is a customizable HTML report which can be integrated into Selenium WebDriver using JUnit and TestNG frameworks.
- Log4j: Apache Log4j is a Java-based logging utility. One of several Java logging frameworks.
- ReportNG: a simple HTML/XML reporting plug-in for the TestNG framework.

How do you avoid OR resolve Selenium's StaleElementReferenceException?

The common reasons for this exception,

- Element has been deleted entirely.
- Element is no longer attached to the DOM.
- Webpage on which the element was part of has been refreshed.
- Element has been refreshed by a JavaScript or Ajax call.

How to avoid it? By adding appropriate waits. Initialize OR Find the WebElement after waiting for its visibility or when it enables.

How to resolve it? Simple, discard the current reference you have and replace it by locating the WebElement once again. And don't forget to add relevant wait conditions this time :-P

What is a WebDriver?

--public interface WebDriver extends SearchContext --

WebDriver is an interface which represents an "idealised web browser".

Why interface? Because every browser has their own logic to perform actions such as launch, close, load URL, handling web elements, etc. Same operations are performed in different ways by different browsers. It's difficult to manage (considering browser changes as well) if implemented at WebDriver level. Therefore, WebDriver is built as an interface which consist of all basic methods which could be performed on a browser. And then implemented by respective browser drivers.

Methods: close() | findElement(By by) | findElements(By by) | get(java.lang.String url) | getCurrentUrl() | getPageSource() | getTitle() | getWindowHandle() | getWindowHandles() | manage() | navigate() | quit() | switchTo()

Implementing Classes: ChromeDriver, EdgeDriver, EventFiringWebDriver, FirefoxDriver, InternetExplorerDriver, OperaDriver, RemoteWebDriver, SafariDriver

What's the sequence of TestNG annotation execution?

The hierarchical rep of execution order of TestNG annotations is S-T-C-M,

```
BeforeSuite
  > BeforeTest
      > BeforeClass

         > BeforeMethod
                 @Test – method1
         > AfterMethod
         > BeforeMethod
                 @Test – method2
         > AfterMethod

      > AfterClass
   > AfterTest
AfterSuite
```

This sequence will be clear if you look at sample TestNG xml file,

```
<?xml version="1.0" encoding="UTF-8"?>
<suite name="suite" verbose="1" >
  <test name="test" >
    <classes>
      <class name="class-1"/> --- each class has multiple methods @test-1 | @test-2 | @test-n
      <class name="class-2"/>
      <class name="class-3"/>
    </classes>
 </test>
</suite>
```

Selenium-Java: How to identify all broken links on a web page?

Broken links - links that don't work. Say page is no longer available, page was moved without a redirect, URL structure of a website was changed, etc.

Steps to identify broken links,

1. Identify all the links - usually image <img /> and anchor tags <a/> on a web page containing href attribute.
2. Iterate through all links checking for response code when sending a HTTP request. Based on the response we can figure out if the link is broken or not.

How? HttpURLConnection class from Java helps here - used to make HTTP requests to the webserver hosting the links.

```
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.connect();
String response = connection.getResponseMessage();
connection.disconnect();
```

Trust only those URLs that returned OK, rest should be verified once.

TestNG Soft-Asset & Hard-Assert

Assertion: a statement of fact. In testing terms, your actual validation [expected = actual] which marks the script as pass or fail.

TestNG provides us with two kinds of assertions,

HardAssert: instantly throws an exception when a assert statement fails and jumps to the next test in the test suite. "Assertion" class provides various assertion methods like assertNotEquals | assertTrue | assertFalse | assertNull | assertNotNull | etc.

But what if you want to continue the test case execution even if some of the assertions fail?

SoftAssert: does not throw an exception when an assert fails and would continue with the next step. It basically collects all the assertions throughout the @Test method. "SoftAssert" class also provides various assertion methods like assertEquals | assertTrue | etc.

Note: You need to instantiate a SoftAssert object within a @Test method, and call softAssert.assertAll() at the end in order to collect all assertions & mark the case as fail.

How to wait for a page to load in Selenium?

It is important to wait for a page to load in Selenium before interacting with the web elements in order to avoid NoSuchElementException.

a. Using document.readyState API for checking on whether a page has loaded.

```
((IJavaScriptExecutor)driver).ExecuteScript("return
document.readyState").Equals("complete"));
```

b. Using Explicit wait condition,

```
(new WebDriverWait(getDriver(),
10)).until(ExpectedConditions.visibilityOfElementLocated(By.cssSelector(".selector")));
```

How do you connect & run database queries via Test automation (Java)?

We need to use APIs which helps to interact with database like JDBC [Java Database Connectivity] – a Java API used to connect and interact with Database.

Before building a connection, make sure to add connector dependency in your POM [Ex. MySQL Connector].

DriverManager – to establish a connection | Connection – to create a Statement object | Statement – to execute the query | Resultset – to store the query results.

a. Load & register the driver | Class.forName("com.mysql.jdbc.Driver");

b. Establish connection | Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/Employee","root", "root");

c. Create a Statement object | Statement st= con.createStatement();

d. Execute a query & get the Resultset | Resultset rs = st.executeQuery("Select * from Employee");

Note: PreparedStatement [instead of Statement] has methods to bind various object types, including files and arrays.

How to read config file?

.properties files are used in Java programs to maintain project configuration data, database config or project settings, URLs & credentials, etc. Each parameter is stored as a key-value pair on one line.

To read the config file, object of Properties class is used, provided by Java itself.

```
Properties CONFIG = new Properties();
FileInputStream fs = new FileInputStream(PropertyPath);
CONFIG.load(fs);
String driverPath = properties.getProperty("driverPath");
```

The benefit: If any information is changed from the properties file, you don't need to recompile the java class. We can configure things that are prone to change over a period without need of changing anything in code.

TestNG Listeners

Listener: interface that listen to a specific incident in the selenium script and behave accordingly, used for logging purposes and creating reports. Some of the popular TestNG listeners,

- ITestListener: listens to specific events (depending on its methods) and executes the code written inside the method. Can also log the events onto the reports. Methods: OnStart | onTestSuccess | onTestFailure | onTestSkipped | onTestFailedButWithinSuccessPercentage | onFinish

- IReporter: medium to generate custom reports, contains a method called generateReport() which is invoked when all the suites have run.

- ISuiteListener: works on the suite level, listens to the start and end of a suite execution. Methods: onStart | onFinish.

Note: 'ITestResult [interface that describes the result of the test] and 'ITestContext' [contains all the information about the test run] are generally used when implementing ITestListener.

Selenium Actions vs Java AWT Robot class

The basic & major difference - Selenium uses the WebDriver API and sends commands to a browser to perform actions (through the "JSON wire protocol"). Whereas Java AWT Robot uses native system events to control the mouse and keyboard.

Since Robot class uses the native system events, it will actually move the mouse cursor instead of just generating Mouse Event.

Actions: User-facing API that allow you to build a chain of actions and perform them which is based on the WebDriver API. It just mimics the keyboard & mouse actions on a browser instead of actually moving the cursor.

Note: It is always preferred to use Actions class rather than using the Robot class (Keyboard or Mouse directly). Say, you have parallel execution running – Actions object will be tied to specific driver instance, whereas Robot events will be performed on whatever window is open.

Selenium: How can we switch from one window to another?

- String currentWinHandle = driver.getWindowHandle(): to get the window handle of the current window.
- Set<String> allWinHandles = driver.getWindowHandles(): to get the window handles of all the open windows in a Set.
- driver.switchTo().window(winHandle): to switch to other window

Complete Code:

```
String parentWindow = driver.getWindowHandle();
Set<String> handles =  driver.getWindowHandles();
For (String windowHandle  : handles)
{
 if(!windowHandle.equals(parentWindow))
 {
    driver.switchTo().window(windowHandle);
    <!--Perform your operation here for new window-->
    driver.close(); //close child window
    driver.switchTo().window(parentWindow); //control back to parent window
  }
}
```

How to assert that an image is properly loaded?

a. **Using JavaScript executor (to check if image width is > 0)**

```
WebElement ImageFile = driver.findElement(By.xpath("//img[contains(@id,'Test
Image')]"));
Boolean ImagePresent = (Boolean) ((JavascriptExecutor)driver).executeScript("return
arguments[0].complete && typeof arguments[0].naturalWidth != \"undefined\" &&
arguments[0].naturalWidth > 0", ImageFile);
if (!ImagePresent)
        System.out.println("Image not displayed.");
else
      System.out.println("Image displayed.");
```

b. **Using isDisplayed method**

```
if (driver.findElement(By.xpath("//img[contains(@id,'Test Image')]")).isdisplayed())
        System.out.println("Image not displayed.");
else
        System.out.println("Image displayed.");
```

What is iframe?

The <iframe> tag (inline frame) defines a rectangular region within the HTML document in which the browser can display a separate document, including scrollbars and borders. An inline frame is used to embed another document within the current HTML document. Often used to insert content from another source, such as an advertisement, into a Web page. Example,

```
<html>
  <head>
    <title>HTML Iframes</title>
  </head>
  <body>
    <p>Document content goes here...</p>
    <iframe src = "/html/menu.htm" width = "555" height = "200">
      This text is present in an iFrame.
    </iframe>
    <p>Document content also go here...</p>
  </body>
</html>
```

Difference between relative and absolute path? Which one is fast?

- Absolute Xpath: It uses Complete path from the Root Element to the desire element.
- Relative Xpath: You can simply start by referencing the element you want and go from there.

Relative Xpaths are always preferred as they are not the complete paths from the root element. (//html//body). In future, if any web element is added/removed, then the absolute Xpath changes. So always use Relative Xpaths in your Automation.

Example,

```
<html>
 <body>
  <input type ="text" id="username">
 </body>
</html>
```

Absolute: html/body/input | **Relative**: //*[@id="username"]

How to travel through siblings in Selenium. Syntax for this.

Sibling, i.e., brother or sister. What does it mean in HTML DOM context? Elements at the same level within the DOM structure. We can use either preceding-sibling or following-sibling to traverse up & down at the same DOM level.

- //button[contains(.,'Arcade Reader')]/following-sibling::button[@name='settings']
- //button[contains(.,'Arcade Reader')]/preceding-sibling::button[@name='settings']

How to fetch data from property file?

'.properties' files are mainly used in Java programs to maintain project configuration data, database config or project settings etc. Each parameter in properties file are stored as a pair of strings, in key and value format, where each key is on one line. You can easily read properties from some file using object of type Properties.

```
File file = new File("D:/SoftwareTestingStudio/ReadData/src/datafile.properties");
FileInputStream fileInput = new FileInputStream(file);
Properties prop = new Properties();
prop.load(fileInput);
WebDriver driver = new FirefoxDriver();
driver.get(prop.getProperty("URL"));
driver.findElement(By.id("Email")).sendKeys(prop.getProperty("username"));
driver.findElement(By.id("Password")).sendKeys(prop.getProperty("password"));
driver.findElement(By.id("SignIn")).click();
```

What is the difference between @beforetest and @beforemethod?

- @BeforeTest: runs before each test class (can contain multiple @Test) declared inside testng.xml
- @BeforeMethod: runs before each @Test method declared within a class.

Say you have a class HomePageTest.Java where you have 3 @Test methods,

- check application Login
- check Homepage title
- check Homepage elements

Now you give this HomePageTest.java in your testng.xml.

- @BeforeTest: code will run once before HomePageTest.java and then before the next test mentioned in testng.xml
- @BeforeMethod: code will run before each @Test annotation within your HomePageTest.java file, i.e., 3 times in this case.

Difference between Xpath and CSS?

- The primary difference - with XPath we can traverse both forward and backward whereas CSS selector only moves forward.
- CSS selectors perform far better than Xpath.
- Xpath engines are different in each browser, hence make them inconsistent.
- IE does not have a native Xpath engine, therefore selenium injects its own Xpath engine for compatibility of its API. Thus, losing the advantage of using native browser features.

However, there are some situations where, you need to use Xpath, like while searching for a parent element or searching element by its text.

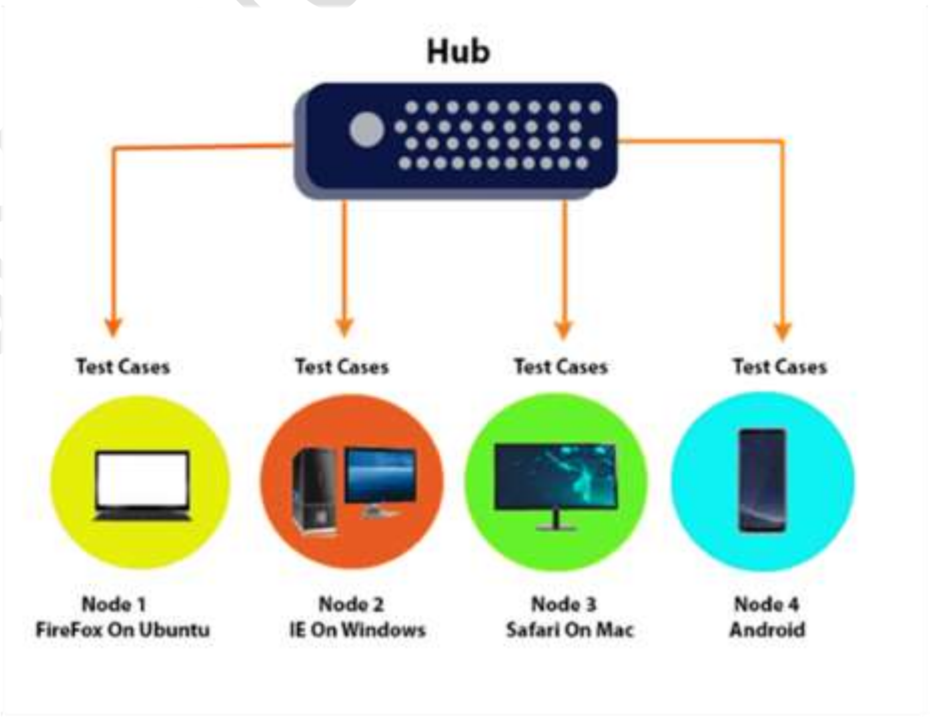
How to execute testng.xml from command prompt?

cd C:\Selenium\SampleTestNG (Project path)
java -cp C:\Selenium\SampleTestNG\lib\;C:\Selenium\SampleTestNG\bin org.testng.TestNG testng.xml*

For 'java -cp' argument - provide the classpath i.e. path to classes/libraries that the program requires to run.

Explain Selenium Grid concept and commands used for it.

Selenium Grid allows us to execute our tests in multiple machines (physical / virtual) and multiple browsers with different versions, which dramatically speeds up test execution and helps in reducing total amount of time required for test execution.



A grid consists of a single hub, and one or more nodes which can be configured by specifying command line parameters OR by specifying a JSON config file.

- **Hub**: the central point which will receive all the test requests along with information on which browser, platform (i.e. WINDOWS, LINUX, etc) and where the test should be run. Based on the request received, it will distribute them to the registered nodes. To start a hub with default parameters,

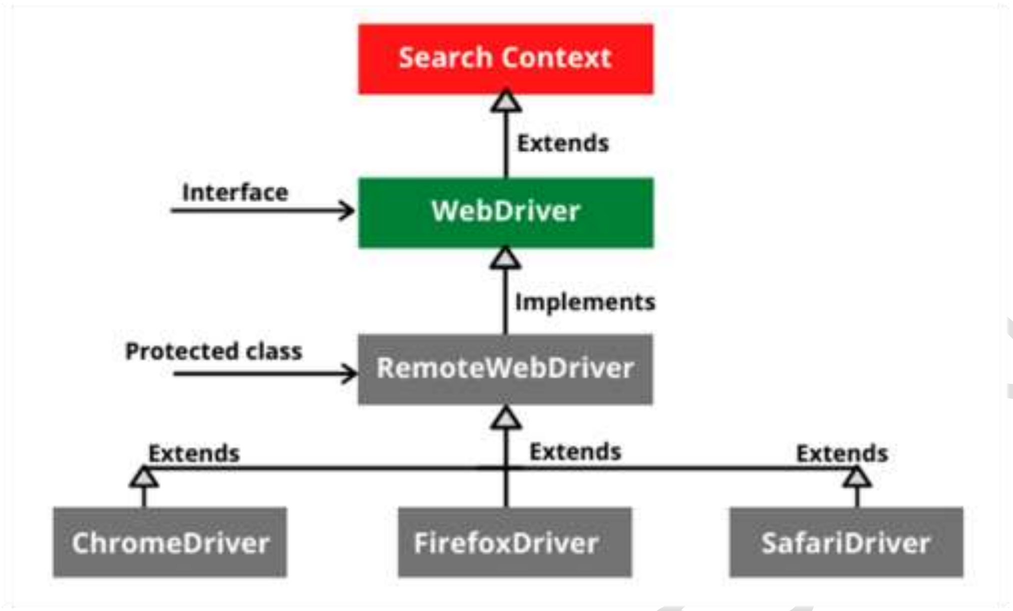
java -jar selenium-server-standalone-2.48.2.jar -role hub

After starting the hub, we can view the status of the hub by opening any browser window and navigating to: http://localhost:4444/grid/console.

- **Nodes** are where our tests will run, each Node is machine (can be a physical machine / virtual machine) that we register with the Hub (browser and configuration details).

java -jar selenium-server-standalone-2.48.2.jar -role node -hub http://localhost:4444/grid/register

Explain web driver hierarchy.



1. SearchContext (interface): The topmost interface of Webdriver which contains only two abstract method findElement() and findElements(). These two methods don't have a method body.

2. WebDriver (interface) extends SearchContext interface - many abstract methods like close(), get(java.lang.String URL), quite(), navigate(), switchTo() and others.

3. RemoteWebDriver (fully implemented class) - all abstract methods of WebDriver and SearchContext interface implemented. Also two other interfaces JavascriptExecutor and TakesScreenshot abstract methods are implemented in RemoteWebDriver class.

4. Finally, browser-specific driver classes like FirefoxDriver, ChromeDriver, IEDriver, SafariDriver, etc.

Why we are writing Web Driver driver = new ChromeDriver(). Why can't we write ChromeDriver() cd = new ChromeDriver();

We generally do it this way because we want to be able to run our tests on multiple browsers. If we declare the driver as a specific driver type, we are then anchored to only that driver. This is not a problem if you only ever need to test on say Chrome for example. But what if you later want your tests to also be able to work with IE, Opera, Firefox, etc.?

First,

- WebDriver is an interface.
- FirefoxDriver()/ChromeDriver() is inheriting RemoteDriver class which implements the WebDriver interface.

We cannot create the WebDriver object, because WebDriver is an interface and not a class.

It is possible to create an object for an interface and instantiate it using any of the classes that implements the interface like this: WebDriver driver = new FirefoxDriver(); OR WebDriver driver = new ChromeDriver(); By using above code, your scripts are now flexible and can use any WebDriver object which is required to invoke particular browser.

If your main tests and other classes define the reference variable of type WebDriver, it allows us to assign the driver object to different browser specific drivers (without change to the test code itself). Thus, allowing multi-browser testing by assigning the driver object to any of the desired browser.

CacheLookUp concept in PageFactory?

When PageFactory initializes a WebElement which is decorated with @FindBy or @FindAll annotation, it creates a Java proxy object, i.e. delay load elements and avoid FindElementBy call during Page Object initialization.

Proxy object has to resolve the actual WebElement to make any call,

- Find the element every time you need it (Which is time consuming because of a FindElement REST call to WebDriver)

- Cache the element from the very first FindElement call to WebDriver and return it in subsequent calls. Note: Not a good idea for elements which are dynamic because referring to an older cached version will result in Stale Element exception.

If you use @CacheLookUp annotation, you can instruct Selenium to NOT make a FindElement call to Browser's WebDriver every time and rather used the earlier cached version.

How to check background color in Selenium?

Color can be determined using the CSS value of the element,

- driver.findElement(By.xpath("..blah..")).getCssValue("background-color").toString();
- driver.findElement(By.xpath("..blah..")).getCssValue("color").toString();

How to handle Proxy in Selenium?

Sometimes when you try to access a secure application you might get proxy issues. Until we do not set proxy, we cannot access the application. To handle proxy setting in Selenium - we have a separate class called Proxy.

- Create object of proxy class and set HTTP proxy or FTP proxy based on requirement.

```
Proxy p=new Proxy();
p.setHttpProxy("localhost:7777");
```

- Use DesiredCapability class to customize capability of browser and pass the proxy object.

```
DesiredCapabilities cap=new DesiredCapabilities();
cap.setCapability(CapabilityType.PROXY, p);
```

- While initiating browser pass capability object

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

How to open chrome browser in Incognito Mode?

Add argument "-incognito" in the DesiredCapabilities,

```
ChromeOptions options = new ChromeOptions();
options.addArguments("--incognito");
DesiredCapabilities capabilities = DesiredCapabilities.chrome();
capabilities.setCapability(ChromeOptions.CAPABILITY, options);
```

If we are using Actions class then what is the usage of Robot class?

There is a difference in terms of how do these two works.

- Selenium Actions uses the WebDriver API and sends commands to a browser to perform actions (through the "JSON wire protocol").
- Java AWT Robot uses native system events to control the mouse and keyboard.

If you are doing browser automation, ideally, you don't ever use things like Robot since usually the functionality provided by selenium is more than enough. If there is a technical glitch using actions class in certain environments, then we can use Robot class.

How to handle untrusted certificates in Selenium?

Firefox,

- Create a firefox Profile
- Set the setAcceptUntrustedCertificates() method as true and setAssumeUntrustedCertificateIssuer() method to false

```
FirefoxProfile profile=new FirefoxProfile();
profile.setAcceptUntrustedCertificates(true);
WebDriver driver=new FirefoxDriver(profile);
driver.get("STS URL");
```

Chrome & IE,

Take the help of 'Desired capabilities; to get and accept the SSL certificate error on run time.

- Create An object for Desiredcapabilities class
- Set the ACCEPT_SSL_CERTS as true
- Open the chrome browser with the capability

```
DesiredCapabilities cap=DesiredCapabilities.chrome();
cap.setCapability(CapabilityType.ACCEPT_SSL_CERTS, true);
System.setProperty("webdriver.chrome.driver","Chrome driver path");
WebDriver driver=new ChromeDriver(cap);
```

Selenium: How do you handle a dynamic drop-down list?

The most common example - Only after city selection is made > the city options are loaded in the next drop-down list.

Need to handle using the text value to click on, e.g.

```
dropdown.click();
List<WebElement> options = dropdown.findElements(By.tagName("li"));
for (WebElement option : options)
{
  if (option.getText().equals(searchText))
  {
    option.click(); // click the desired option
    break;
  }
}
```

How to handle dynamic elements in Selenium?

Should use JavaScript functions like “starts-with” or “contains” in our element locators to separate the dynamic part of locator from static part.

For example, XPath: //input[contains(@id, '_name')]

How do you handle a flash which comes up in the website in between like it is a flash file. How do you check whether it is a flash or not a flash in selenium?

Need to use some third-party libraries such as Sikuli to automate flash objects.

In a Drop-down, there are several options, and I need to select HYD, how to achieve using Webdriver?

For handling dropdowns, Selenium already provides Select class that has some predefined method which help is a lot while working with Dropdown.

Select city=new Select(driver.findElement(By.id(“city”)))

- City.selectByIndex(4)

Index starts from 0. Select a value depending on the index, i.e., index as 4, it will select 5th value.

- city.selectByValue(“3”)

Select a value depending on the ‘value’ tag of the given element.

- city.selectByVisibleText("HYD");

Select the value from the visible text. Straight forward – whatever text we are passing it will simply select that value.

- Get All option from dropdown

```
List<WebElement> dropdown=city.getOptions();
for(int i=0;i<dropdown.size();i++)
{
        System.out.println(dropdown.get(i).getText());
}
```

Common Exceptions in Selenium WebDriver

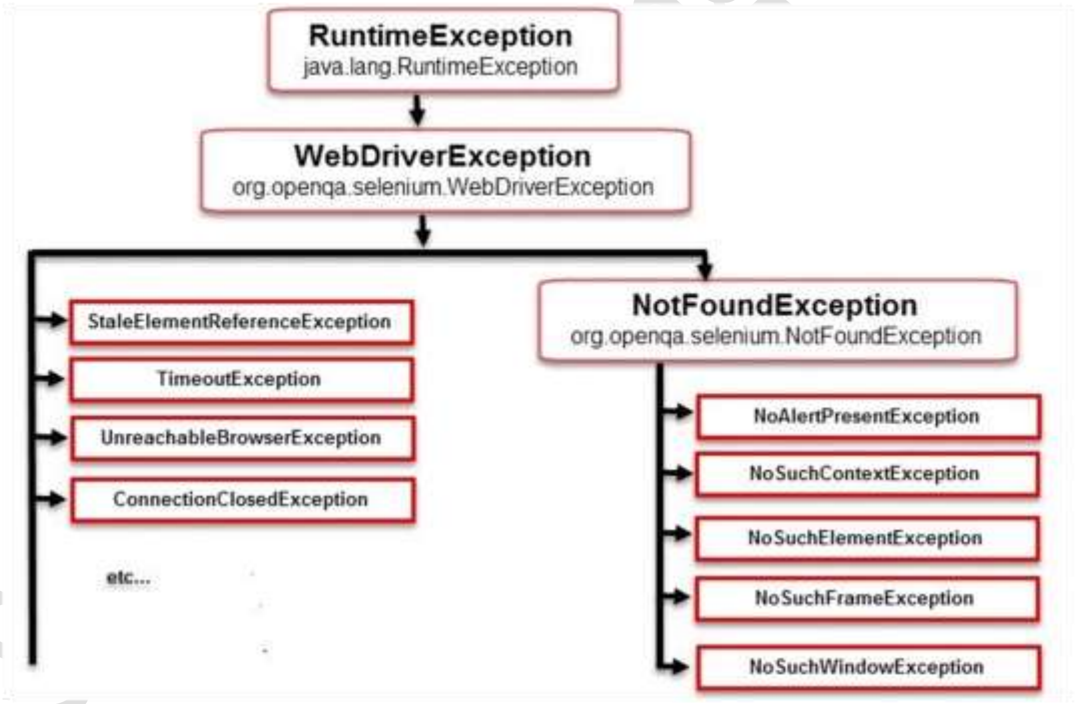
- ElementNotVisibleException: Although an element is present in the DOM, it is not visible (cannot be interacted with). E.g. Hidden Elements – defined in HTML using type="hidden".

- ElementNotSelectableException: Although an element is present in the DOM, it may be disabled (cannot be clicked/selected).

- InvalidSelectorException: Selector used to find an element does not return a WebElement. Say XPath expression is used which is either syntactically invalid or does not select WebElement.

- NoSuchElementException: WebDriver is unable to identify the elements during run time, i.e. FindBy method can't find the element.

- NoSuchFrameException: WebDriver is switching to an invalid frame, which is not available.


RuntimeException
java.lang.RuntimeException
WebDriverException
org.openqa.selenium.WebDriverException
StaleElementReferenceException
TimeoutException
UnreachableBrowserException
ConnectionClosedException
etc...
NotFoundException
org.openqa.selenium.NotFoundException
NoAlertPresentException
NoSuchContextException
NoSuchElementException
NoSuchFrameException
NoSuchWindowException

- NoAlertPresentException: WebDriver is switching to an invalid alert, which is not available.

- NoSuchWindowException: WebDriver is switching to an invalid window, which is not available.

- StaleElementReferenceException: The referenced element is no longer present on the DOM page (reference to an element is now Stale). E.g. The Element belongs to a different frame than the current one OR the user has navigated away to another page.

- SessionNotFoundException: The WebDriver is performing the action immediately after 'quitting' the browser.

- TimeoutException: The command did not complete in enough time. E.g. the element didn't display in the specified time. Encountered when working with waits.

- WebDriverException: The WebDriver is performing the action immediately after 'closing' the browser.

What is Selenium "WebElement"?

WebElement is a class that represents a DOM element. E.g., input element, select element, div element, hyperlink element, etc. WebElements can be found by searching from the document root using a WebDriver instance [absolute XPath], or by searching under another WebElement [relative XPath].

WebDriver API provides built-in methods to find the WebElements which are based on different properties like ID, Name, Class, XPath, CSS Selectors, link Text, etc.

How to handle keyboard operations in Selenium using Robot class?

Robot class is a part of the Java API awt package and NOT org.openqa.selenium.

Unlike Selenium which uses the WebDriver API to invoke commands to a browser to perform actions, Robot class uses native system events to control the mouse and keyboard.

E.g., need to handle a window-popup or a combination of modifier keys such as Alt, Shift, etc.

It differs from Selenium which uses the WebDriver API and invokes commands to a browser to perform actions.

Steps: import java.awt.Robot >> Robot robot = new Robot(); >> robot.<method_name>(parameter);

Keyboard Methods,

- **keyPress(int keycode):** press a given key. say Alphabet A = KeyEvent.VK_A I i.e., keyPress(KeyEvent.VK_A);
  KeyEvent: a low-level event. In Java AWT, low-level events are events that indicate direct communication like a keypress, key release.
- **keyRelease(int keycode):** releases a given key. E.g., the Shift key pressed using the keyPress(KeyEvent.VK_SHIFT) method needs to release using the keyRelease (KeyEvent.VK_SHIFT) method.

What is Selenium - Actions class? And why do we need it?

Why? Because there are complex interactions like Drag-n-Drop and Double-click etc. which cannot be done by simple WebElement commands. To handle those, we have the Actions class in Selenium.

"The user-facing API for emulating complex user gestures. Use this class rather than using the Keyboard or Mouse directly. Implements the builder pattern: Builds a Composite Action containing all actions specified by the method calls".

Build method: build the sequence if actions using the build() method and get the composite action. And finally, perform the actions sequence using perform() method.

E.g.
Actions actions = new Actions(webdriver object);
actions.keyDown(element,Keys.SHIFT).sendKeys(TextToBeConvertAndSendInUpperCase).keyUp(Keys.SHIFT).build().perform();

Note: Actions vs Action | "Action is an Interface", used to represent the single user interaction to perform the series of action items build by "Actions class".

What are the different methods available in Selenium - Actions class?

Keyboard Events:

- keyDown(modifier key): Performs a modifier key press.
- sendKeys(keys to send): Sends keys to the active web element.
- keyUp(modifier key): Performs a modifier key release.

Mouse Events

- click(): Clicks at the current mouse location.
- doubleClick(): Performs a double-click at the current mouse location.
- contextClick() : Performs a context-click at middle of the given element.
- clickAndHold(): Clicks (without releasing) in the middle of the given element.
- dragAndDrop(source, target): Click-and-hold at the location of the source element, moves to the location of the target element
- dragAndDropBy(source, xOffset, yOffset): Click-and-hold at the location of the source element, moves by a given offset
- moveByOffset(x-offset, y-offset): Moves the mouse from its current position (or 0,0) by the given offset
- moveToElement(toElement): Moves the mouse to the middle of the element
- release(): Releases the depressed left mouse button at the current mouse location

Can we prioritize Test Cases in Selenium test automation? If Yes, how?

Yes. How? Using TestNG – use the priority argument along with @Test annotation.

```
@Test (priority = 0)
public void method1()
{
        //test code
}
@Test (priority = 1)
public void method2()
{
        //test code
}
```

Few pointers,

- Priority can only be passed for @Test annotated methods.
- Lower the priority number; higher is the priority. i.e., 0 will be executed before 1.
- If two or more methods have the same priorities in TestNG, then their running test sequence is alphabetic.
- The test methods with no priority assigned have a default priority equal to 0.

What is the use of JavaScriptExecutor?

JavascriptExecutor is the Selenium interface which is implemented by all the driver classes.

Why? While you execute your Selenium script - at times because of cross domain policies browsers enforce, your script execution may fail unexpectedly. Sometimes web controls don't react well against Selenium commands. Say an element is not interactable, or not visible on webpage (scroll), etc.

Methods,

- executeScript(): executes JavaScript in the context of the currently selected frame or window.
- executeAsyncScript(): executes an asynchronous piece of JavaScript in the context of the currently selected frame or window.

Syntax,

```
JavascriptExecutor js = (JavascriptExecutor) driver;
js.executeScript(Script, Arguments);
```

Best Practise: If you see too many issues due to element not interactable,

1. Catch the exception, take screenshot for your reference, generate a warning in logs.
2. Implement direct click using Javascript executor in catch block.
3. By generating warning, you can check if you are not missing out on an actual issue.

How do you manage browser-driver compatibility, i.e., compatible versions? Say browser gets an auto-update and your scripts start failing due to older driver version.

Problem: To run test automation, we need to have browser drivers, .exe files like chromedriver.exe and geckodriver.exe [windows] OR binary files like chromedriver and geckodriver [Linux], which allows Selenium WebDriver to handle browsers. Also, we need to set the path of these files in our script OR need to add the location to the classpath. But what if some browser version is auto-updated? These steps become cumbersome as we need to carry them out every time the versions change.

Solution: Hence, we use the "WebDriverManager" class in Selenium, to download the binary file (or .exe files) of the driver automatically based on the browser version.

```
WebDriverManager.chromedriver().setup();
driver = new ChromeDriver();
```

- automates the management of WebDriver binaries.
- downloads the appropriate driver binaries, if not already present, into the local cache.
- downloads the latest version of the browser binary, unless otherwise specified.
- Eliminates the need to store driver binaries locally. We also need not maintain various versions of the binary driver files for different browsers.

All we have to do is to add its dependency through Maven or Gradle to download all the necessary drivers.

Selenium: How to run only failed cases?

Method-1: TestNG - IRetryAnalyzer

Create a separate JAVA class implementing TestNG - IRetryAnalyzer interface. And need to override 'retry' method [to define the retry count] taking < ITestResult result > as argument. Change your @Test annotations with @Test(retryAnalyzer = RetryAnalyzer.class).

Method-2: Using testng-failed.xml file

After every execution a testing-failed.xml file is created which keeps track of all the failed tests. We can run this file just like we run the testng.xml file.

How to run parallel tests using TestNG?

Process of running the test scripts parallelly rather than one after the other.

- Reduces Time: Running the tests in parallel reduces the overall execution time.
- Allow Multi-Threaded Tests: We can allow multiple threads to run simultaneously.

How? with the help of keyword "parallel", and assign any of the four values – Methods/Tests/Classes/Instances.

```
<suite name = "Parallel Testing Suite">
  <test name = "Parallel Tests" parallel = "methods">
    <classes>
      <class name = "ParallelTest" />
    </classes>
  </test>
</suite>
```

The TestNG has a default value of thread = 5 for parallel testing which we can modify,

```
<test name = "Parallel Tests" parallel = "methods" thread-count = "2">
```

Note: You need to be aware of dependent test scripts, to avoid any conflicts while parallel execution.

What is Headless testing? And how can we achieve it via Selenium?

Problem: When we run the Selenium tests on any of the browsers, we generally face some challenges such as slow rendering on the browser, interference of other applications running on the system, etc.

Solution: Headless browser helps in the execution of the Selenium Headless Browser tests in a Non-UI mode. Almost all modern browsers provide the capability to run them in headless mode.

HTMLUnitDriver

HTMLUnitDriver is an implementation of Selenium WebDriver [similar to FirefoxDriver, ChromeDriver, etc.] based on HtmlUnit, which is a Java-based implementation of a web browser without a GUI.

- Add the driver dependency
- Initialize: *HTMLUnitDriver unitDriver=HtmlUnitDriver(true)* **OR** *HTMLUnitDriver driver = new HTMLUnitDriver(BrowserVersion.Firefox_68)* **OR** *HtmlUnitDriver unitDriver = new HtmlUnitDriver(BrowserVersion.FIREFOX_68,true);*

Headless Chrome

Selenium provides a class ‘ChromeOptions’ where we can specify certain configurations to change the default behaviour of Chrome – one configuration is the “headless” mode.

```
ChromeOptions options = new ChromeOptions()
options.addArgument("headless");
ChromeDriver driver = new ChromeDriver(options);
```

Note: Debugging will not be feasible, as the only way to check what’s running on the browser is to grab the screenshots and validate the output.

TestNG: How can we group test cases? Say, Sanity, Regression, etc.?

Using GROUPS attribute in TestNG, we can assign the test methods to different groups.

```
@Test(groups = {"Sanity"})
public void testMethod1()
{
  //Test logic
}

@Test(groups = {"Regression"})
public void testMethod2()
{
  //Test logic
}
```

TestNG: What are some common assertions provided by TestNG?

- assertEquals (String actual, String expected, String message) and other overloaded data types in parameter
- assertNotEquals (double data1, double data2, String message) and other overloaded data types in parameter
- assertFalse (boolean condition, String message)
- assertTrue (boolean condition, String message)
- assertNotNull (Object object)
- fail (boolean condition, String message)
- true (String message)

TestNG: How can we make one test method dependent on other using TestNG?

Using dependsOnMethods parameter inside @Test annotation, i.e., test method will run only after the successful execution of dependent test method.

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

TestNG: How to run a Test method multiple times in a loop?

Using invocationCount parameter and setting its value to an integer value.

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

TestNG: Any idea about threadPoolSize?

The threadPoolSize attribute specifies the number of threads 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
}
```

What's new with Selenium v-4?

W3C compliance of WebDriver APIs

In Selenium's earlier versions (i.e., Selenium 3), the JSON Wire Protocol was responsible for communication between the web browser and the test code. This led to the additional overhead of encoding and decoding the API requests using the W3C protocol.

A significant change under the hood for WebDriver is the complete W3C compliance of the WebDriver APIs. This standardization will eliminate the need for encoding and decoding the API requests by the JSON wire protocol for communication between browsers and test scripts. This means the WebDriver will now interact directly with the target browser. This standardization will result in more stable cross browser tests. In a nutshell, JSON wire protocol is deprecated in Selenium.

Support for browsers
New Selenium 4 IDE
Improved Selenium Grid
Detailed Documentation
WebDriver became W3C
Better Analysis
Se
Selenium 4.0

Enhanced Selenium Grid

No longer be any need to set up and start hubs and nodes separately. Teams can now deploy the grid in three modes: Standalone mode, Hub and Node, fully distributed. The Grid will now support IPV6 addresses, and users can communicate with the Grid using the HTTPS protocol. It would be much easier to use configuration files as users can configure the Grid using human-understandable TOML (Tom's Obvious, Minimal Language) language.

The new Selenium Grid comes with Docker support. This will enable developers or testers to spin up the containers rather than setting up heavy virtual machines. Moreover, it is redesigned in a way that will allow QAs to deploy the grid on Kubernetes for better scaling.

Upgraded Selenium IDE

With Selenium 4, the IDE is revived and now it's add-on is available for major web-browsers like Firefox and Chrome. The add-on for Selenium IDE is now also available on the MS store.

Relative Locators in Selenium 4

Selenium 4 brings an easy way of locating elements with the inclusion of relative locators like To left of, To right of, Above, Below, etc. The introduction of this new method in Selenium 4 helps locate web elements based on the visual location relative to other DOM elements.

Support for Chrome Debugging Protocol

Selenium 4 comes with native support for Chrome DevTools Protocol. This means QAs can now use Chrome development properties like Fetch, Network, Profiler, Performance, Application cache, and more. QAs can also leverage the APIs offered by Chrome DevTools to simulate poor network conditions and perform geolocation testing.

Better Window/Tab Management

Selenium 4 provides a new API newWindow that lets you create a new window (or tab) and automatically switches to it. Since the new window or tab is created in the same session, it avoids creating a new WebDriver object.

Deprecation of Desired Capabilities

Desired Capabilities were primarily used in the test scripts to define the test environment (browser name, version, operating system) for execution on the Selenium Grid. In Selenium 4, capabilities objects are replaced with Options. This means testers now need to create an Options object, set test requirements, and pass the object to the Driver constructor.

Modifications in the Actions Class

Actions class in Selenium is primarily used to simulate input actions from mouse and keyboard on specific web elements. In Selenium 4, several new methods have been added to the Actions class: click(WebElement) | clickAndHold(WebElement) | contextClick(WebElement) | doubleClick(WebElement) | release()