Software Testing -

Software testing is the process of evaluating a software application or system to detect defects, errors, and other quality issues. The goal of software testing is to identify and report problems before the software is released to end-users, ensuring that it meets the requirements and performs as expected.

Types of Software Testing -

➢ **Manual Testing -** Manual testing is a type of software testing where testers manually execute test cases without using any automation tools or scripts.

➢ **Automation Testing -** Automation testing is a type of software testing where tests are automated using scripts or tools instead of being executed manually.

Why Automation Testing is preferred over Manual Testing -

➟ Efficiency
➟ Repetitiveness
➟ Consistency
➟ Scalability
➟ Test coverage
➟ Reliability
➟ Cost effectiveness

Where Automation Testing works best -

➟ Re-Testing
➟ Regression Testing

Where to avoid Automation Testing -

➟ Unstable Application.
➟ Tests Without Predictable Results.
➟ Test cases which require human intervention.

How Automation tools work -

➟ Identification of test cases to be automated.
➟ Creation of Test Scripts.
➟ Execution of Automated tests.
➟ Analysis and Reporting of test results.

What is Selenium -

Selenium is an open-source automated testing framework used for web application testing. It provides a suite of tools that can be used to automate web browsers.
Selenium was originally developed by Jason Huggins in 2004 at ThoughtWorks.

Advantages of Selenium -

➟ Open Source and Free.
➟ Cross-browser compatibility.
➟ Support Multiple OS (Windows, Linux, MacOS, etc).
➟ Support Multiple browsers (Chrome, Edge, Firefox, IE, etc).
➟ Support Multiple language (Java, Python, Ruby, C#, etc).
➟ Large community support and Integration of third party tools is allowed.

Disadvantages of Selenium -

➟ Support only web based application (AutoIT, Sikuli, RobotAPI, etc).
➟ Reporting of test result is not supported (TestNG, Extent reports, etc).
➟ Doesn't support excel files (ApachePOI).
➟ Limited Support for image, captcha, Graphs based testing.

Versions of Selenium -

➟ **Selenium 1.x -** IDE(Firefox), RC, Grid
➟ **Selenium 2.x -** IDE(Firefox), Webdriver, Grid
➟ **Selenium 3.x -** IDE(Firefox/chrome), Webdriver, Grid
➟ **Selenium 4.x -** IDE(Firefox/chrome/Edge), Webdriver, Grid

Components of Selenium Suite -

➢ **Selenium IDE -** A browser extension for Firefox, Edge and Chrome that provides a record-and-playback feature for creating test scripts.
It generate code in programming languages like C#, Java,Python, and Ruby, as well as Selenese (Selenium's own scripting language).

Advantages:
➟ Simple
➟ Easy Debugging
➟ Cross-browser Execution
➟ No programming knowledge is required

➢ **Selenium WebDriver -** A tool for programmatically controlling web browsers and running tests in a variety of programming languages, including Java, Python, Ruby, and C#.

Selenium WebDriver drives a browser natively, as a real user would, either locally or on remote machines.
It is also known as Selenium 2.0.

Supported Browsers: Chrome, Firefox, Internet Explorer, Edge, Safari, Opera, HtmlUnitDriver.
Supported OS: Microsoft Windows, MacOS, Linux.
Programming Languages: Java, Python, C#, JavaScript, Ruby, Kotlin.

Note: HTMLUnitDriver is the most lightweight and fastest implementation headless browser for WebDriver. It is based on
HtmlUnit. It is known as Headless Browser Driver. It is the same as Chrome, IE, or FireFox driver, but it does not have a GUI so one cannot see the test execution on screen.

Advantages:
➟ It is an open source, free and portable tool.
➟ It supports various operating systems.
➟ It supports various browsers.
➟ It supports various programming languages.
➟ It supports parallel test execution.
➟ It uses very less CPU and RAM consumption for script execution.
➟ It can be integrated with TestNG testing framework for testing our applications and generating reports.
➟ It can be integrated with Jenkins, Maven, GitHub and other tools for DevOps implementation.

Disadvantages :
➟ It only supports web based application and does not support windows based application.
➟ No centralized maintenance of object.
➟ It is difficult to test Image based application.
➟ No reliable support like Commercial Tools.
➟ Difficult to setup environment.
➟ It need outside support for report generation activity like dependence on TestNG or Jenkins.

➢ **Selenium RC -** Selenium Remote Control (RC) is an older version of Selenium that has been replaced by WebDriver.

➢ **Selenium Grid -** A tool for distributing tests across multiple machines and browsers to enable parallel testing.

If you want to scale by distributing and running tests on several machines and manage multiple environments from a central point, making it easy to run on a vast combination of browsers/OS, then you want to use Selenium Grid.
The Grid can minimize test runtime by executing multiple test scripts on any number of remote devices at once. This is called parallel testing.

Selenium WebDriver -

Webdriver is one of the core components of the Selenium suite.
Webdriver provides a programming interface for controlling web browsers and automating user actions on web pages.
Webdriver is an API as it provides a standardized way for developers to interact with web browsers such as Chrome, Firefox, and Safari, by providing a set of methods and functions that can be used to automate browser actions.

The Selenium Webdriver interface is implemented by different browser driver classes. These are as follows :

➟ **Firefox Browser** -----> FirefoxDriver class
➟ **Chrome Browser** -----> ChromeDriver class
➟ **Edge Browser** -----> EdgeDriver class
➟ **IE Browser** -----> InternetExplorerDriver class(This is deprecated as IE browser is deprecated)

Note - Browser driver classes implements all methods from Webdriver class along with their own methods.

Setup WebDriver in Eclipse -

To setup webdriver in eclipse we need the following :

Step-1 - Selenium Client Library (Collection of Jar files which contains classes and methods)
Step-2 - Browser specific drivers (Integrate selenium with specific browsers)
Step-3 - Browsers (Firefox/chrome/Edge)

Maven -

Maven is a popular build automation tool used primarily in Java and Java-related projects. It was created by the "Apache Software Foundation" and provides a structured approach to project management, dependency management, and the build process.

Features of Maven -

➟ Page Object Model(POM)
➟ Dependency Management
➟ Plugins
➟ Repositories
➟ Consistent Project Structure

Maven Repository link for Java-Selenium dependency configuration -
https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java/4.8.3
Add the Java-Selenium dependency code in the **pom.xml** file of Maven Project and it will automatically fetch all the Java-Selenium dependencies.

Note - Instead of using the Maven project we can do all this in a normal java project by manually downloading selenium libraries and browser drivers. Although it is a tedious task and preferably not used.
Note - Earlier DriverManager dependency also needed to be added along with Java-selenium dependency to get all the browser drivers at once. But with the Selenium-4 version it is not required to be added specifically, it will come along with Java-Selenium dependency.

Launching Browsers -

➢ **Edge Browser -**

```
System.setProperty(key,value)
WebDriver driver = new EdgeDriver();
```

➢ **Chrome Browser -**

```
System.setProperty(key,value)
WebDriver driver = new ChromeDriver();
```

➢ **Firefox Browser -**

```
System.setProperty(key,value)
WebDriver driver = new FirefoxDriver();
```

Note- key = webdriver.chrome.driver or **value =** path of chrome driver followed with ".exe" extension.

Locators -
A locator is a unique identifier used to locate a specific element on a web page or in a user interface using DOM code. Locators are used in automated testing to interact with elements such as buttons, links, input fields, and other UI components.

DOM - Document object model. Every tag/attribute will be present in the DOM that is created by the browser at run-time.

Types of Locators -
➢ **Normal Locators -** id, name, linkText, partialLinkText, ClassName, tagName.
➢ **Customized Locators -** css selector, xPath.

Note - Preferred sequence of using locators - id, name, linkText, partialLinkText, xPath, css selector, tagName, ClassName.

Id Locator -
An id locator uses the "id" attribute of a web element to locate it. It is a unique identifier assigned to a web element in the HTML code. IDs are commonly used as locators because they are guaranteed to be unique on a page.
Example- driver.findElement(By.**id**("inputUsername")).sendKeys("Himanshu");

Name Locator -
A name locator uses the "name" attribute of a web element to locate it. Name locators are often used for form elements, such as text boxes and radio buttons.
Example- driver.findElement(By.**name**("inputPassword")).sendKeys("Himanshu1234");

LinkText Locator -
A link text locator is used to locate hyperlinks on a webpage. Link text locators are useful when you want to click on a specific link on a page.
Example- driver.findElement(By.**linkText**("Forgot your password?")).click();

PartialLinkText Locator -
A partial link text locator is similar to the link text locator, but it matches a portion of the hyperlink text instead of the entire text.
Example- driver.findElement(By.**partialLinkText**("password?")).click();

ClassName Locator -
A class name locator uses the "class" attribute of a web element to locate it. Class name locators can be used when there are multiple elements with the same tag name.
Example- driver.findElement(By.**className**("signInBtn")).click();

TagName Locator -
A tag name locator uses the HTML tag name of a web element to locate it. Tag name locators are useful when there are multiple elements with the same class or name attributes.
Example- driver.findElements(By.**tagName**("a"));

CSS Selector -
CSS stands for "**Cascading Style Sheets**". A CSS selector locator uses a CSS to locate a web element on a webpage. CSS selectors are often used when the other types of locators cannot be used directly.

A CSS Selector is a customized locator that can be created using any of the combination:

➢ **Tag & ID -** tagName#IdValue OR #IdValue
Example- input#small-searchterms OR #small-searchterms

➢ **Tag & Class -** tagName.className OR .className
Example- input.search-box-text OR .search-box-text

➢ **Tag & Attribute -** tagName[Attribute='Value'] OR [Attribute='Value']
Example- input[placeholder='Search store'] OR [placeholder='Search store']

➢ **Tag, Class & Attribute -** tagName.className[Attribute='Value']
Example- input.search-box-text[name='q']

Xpath -

An Xpath locator uses an Xpath expression to locate a web element on a webpage. Xpath locators are often used when the other types of locators cannot be used or when a more complex search is needed.

Types of xpath -

➢ **Absolute xpath -** An absolute Xpath expression starts with the root node and traverses the entire document to locate the element.
The syntax for an absolute Xpath expression begins with a forward slash '/', which represents the root node of the document.
Example - /html/body/header/div/div/div[1]/div/a/img

➢ **Relative xpath -** A relative Xpath expression locates an element based on its relationship to another element in the document.
The syntax for a relative Xpath expression begins with a double forward slash '//', which represents that the xpath is not started from the root node of DOM but rather with a reference to the current node or one of its child nodes.
Example - //*[@id="logo"]/a/img

Note - Relative xpath is preferably used over Absolute xpath because it is quite hectic to maintain the code if Absolute xpath is changed. Even if a single node is added in DOM , a lot of xpath will be tampered.

How to write Relative xpath -

//tagName[@attribute='value'];
OR
//*[@attribute='value']; - In case tagName is unknown * can be used.

How to generate xpath automatically -

➟ Using Developers tool
➟ Using Selectors hub extension

Different Methods of writing xpath -

➢ **Locating elements with index -**
Syntax - (//tagName[@attribute='value'])[index]

➢ **Locating Elements with InnerText(exact match) -**
Syntax - //tagName[text()='String']
//*[text()='String']

➢ **Locating Elements with InnerText(partial match) -**
Syntax - //tagName[contains(text(),'Substring')]
//*[contains(text(),'Substring')]

➢ **Locating Elements with prefix of InnerText is static(partial match) -**
Syntax - //tagName[starts-with(text(),'prefix of inner text')]
//*[starts-with(text(),'prefix of inner text')]

➢ **Locating Elements with visible text in any attributes -**
Syntax - //tagName[@attribute='visibleText']

➢ **Locating Elements with multiple attributes -**
Syntax - //tagName[@attribute1='value1'][@attribute2='value2']...
//*[@attribute1='value1'][@attribute2='value2']...
//tagName[@attribute1='value1' and @attribute2='value2']
//*[@attribute1='value1' and @attribute2='value2']
//tagName[@attribute1='value1' or @attribute2='value2']
//*[@attribute1='value1' or @attribute2='value2']

Note - 'and' operator will work when both of the attributes and values are correct but '**or'** operator will work if any of the attribute and value will be correct.

Xpath Axes -

Xpath axes are used to navigate the hierarchical structure of an XML or HTML DOM. An axis is a way to select a set of nodes relative to a context node.

➢ **Locating Parent Element -**

Syntax - <xpath of context element>/parent::parentTagName
<xpath of context element>/parent::*

➢ **Locating Child Element -**

Syntax - <xpath of context element>/child::childTagName
<xpath of context element>/childTagName

➢ **Locating Descendant Element -**

Syntax - <xpath of context element>/descendant::descendantTagName
<xpath of context element>/descendant::*
<xpath of context element>/child/grandchild
<xpath of context element>/*/grandchild

➢ **Locating Ancestor Element -**

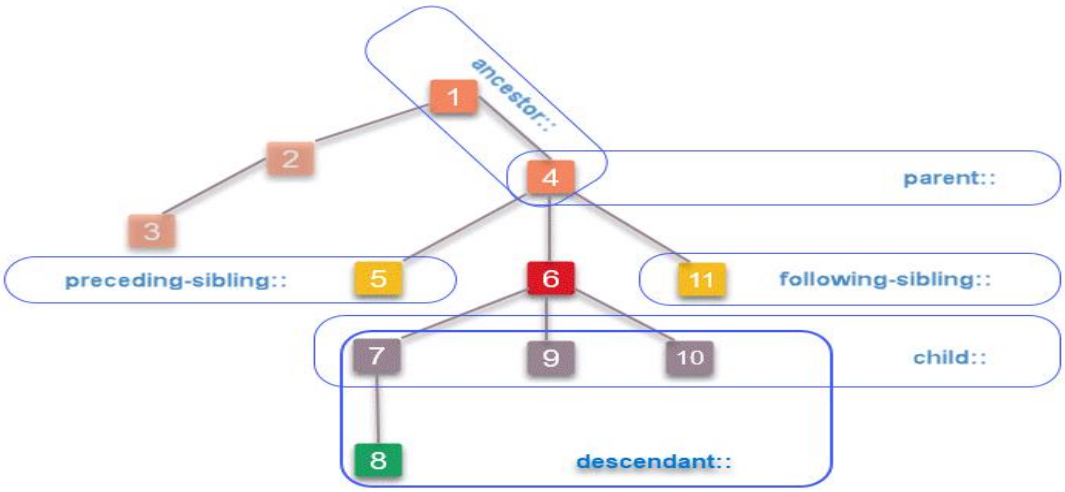
Syntax - <xpath of context element>/ancestor::ancestorTagName
<xpath of context element>/ancestor::*

➢ **Locating Following Sibling Element -**

Syntax - <xpath of context element>/following-sibling::siblingTagName
<xpath of context element>/following-sibling::*

➢ **Locating Preceding Sibling Element -**

Syntax - <xpath of context element>/preceding-sibling::siblingTagName
<xpath of context element>/preceding-sibling::*



WebDriver Methods -

Webdriver provides a number of methods that can be used to interact with web pages. Some of the webdriver methods are as follows:

➢ **get Methods -** These methods can be accessed through the WebDriver object.

➟ **get("URL") -** This method is used to navigate to the specified URL.
Example - driver.get("URL");

➟ **getTitle() -** It is used to retrieve the title of the current web page that is loaded in the browser.
Example - driver.getTitle();

➟ **getCurrentURL() -** It is used to retrieve the current URL of the web page that is loaded in the browser.
Example - driver.getCurrentUrl();

➟ **getPageSource() -** It is used to retrieve the source code of the web page that is loaded in the browser.
Example - driver.getPageSource();

➟ **getWindowHandle() -** It is used to retrieve the unique identifier of the current browser window.
Example - driver.getWindowHandle();

➟ **getWindowHandles() -** It is used to retrieve the unique identifiers of all the browser windows that are currently open.
Example - driver.getWindowHandles();

➢ **Conditional Methods -** These methods can be accessed through WebElements. These methods always return boolean values(true/false).

➟ **isDisplayed() -** It is used to check if an element is currently displayed on the web page or not. It returns a boolean value indicating whether the element is displayed or not.
Example - WebElement.isDisplayed();

➟ **isEnabled() -** It is used to check if an element is currently enabled on the web page or not. It returns a boolean value indicating whether the element is enabled or not.
Example - WebElement.isEnabled();

➟ **isSelected() -** It is used to check if an element is currently selected on the web page or not. It returns a boolean value indicating whether the element is selected or not.
Example - WebElement.isSelected();

➢ **Browser Methods -**

➟ **close() -** It is used to close the current browser window. This method does not end the WebDriver session, so you can still interact with other open browser windows or tabs.
Example - driver.close();

➟ **quit() -** It is used to close all open browser windows and tabs and end the WebDriver session. This method is typically used at the end of a test or script to clean up any resources used by the WebDriver.
Example - driver.quit();

➟ **maximize() -** It is used to maximize the browser.
Example - driver.manage().window().maximize();

➟ **minimize() -** It is used to minimize the browser.
Example - driver.manage().window().minimize();

➢ **Browser Navigation Methods -**

➟ **Navigate().to(URL) -** This command will navigate the browser to the specified URL, but it is different from 'driver.get()' in that it does not wait for the page to load before continuing. This can be useful in situations where you want to navigate to a new page and continue interacting with the previous page.
Example - driver.navigate().to(String URL);

➞ **Navigate().forward() -** This command will navigate the browser forward to the next page in the history.
Example - driver.navigate().forward();

➞ **Navigate().back() -** This command will navigate the browser back to the previous page in the history.
Example - driver.navigate().back();

➞ **Navigate().refresh() -** This command will refresh the current page in the browser.
Example - driver.navigate().refresh();

➢ **Other Important Webdriver Methods -**

➞ **By -** It is used with findElement and findElements methods to specify the locator strategy for locating web elements.
Example - driver.findElement(By.LocatorName("LocatorValue"));

➞ **findElement() -** It is used to locate a single web element on a web page.
Example - driver.findElement(By.LocatorName("LocatorValue"));

➞ **findElements() -** It is used to locate a single web element on a web page.
Example - driver.findElements(By.LocatorName("LocatorValue"));

Note- findElements will return a list of web elements of a particular type unlike findElement that returns a web element.

➞ **sendKeys("String") -** It is used to pass a value in a field/text-box.
Example - driver.findElement(By.LocatorName("LocatorValue")).sendKeys("Value");

➞ **clear() -** It is used to clear value from a field/text-box.
Example - driver.findElement(By.LocatorName("LocatorValue")).clear();

➞ **click() -** It is used to perform click action on an element.
Example - driver.findElement(By.LocatorName("LocatorValue")).click();

Synchronization -

It is a mechanism which involves more than one component to work parallel with each other.

Generally in Test Automation, we have two components
1. Application Under Test
2. Test Automation Tool

Both these components will have their own speed. We should write our scripts in such a way that both the components should move with same and desired speed, so that we will not encounter an Object or WebElement not found error.

Reasons for failure due to Synchronization -
1. When the application server or webserver is serving the page too slowly due to resource constraints.
2. When you are accessing the page on a very slow network.
3. Performance of the application in local and Remote machines.

WebDriverWait -

WebDriverWait is a class in the Selenium WebDriver library, which allows you to wait for a certain condition to occur before proceeding with the execution of your test script.

It is useful in solving synchronization problems in script, where you may need to wait for elements to appear, disappear, or change before you can perform an action or make an assertion.

Types of WebDriverWait -

Thread.sleep -

In Java, 'Thread.sleep()' is a method that pauses the execution of the current thread for a specified period of time. It is not recommended to be used in test automation.
Syntax - Thread.sleep(5000);

➢ **Advantages of using Thread.sleep -**

➟ **Easy to Use:** Thread.sleep is very easy to set up and use.

➢ **Disadvantages of using Thread.sleep -**

➟ **Fixed delays:** Thread.sleep() adds a fixed delay to the test script, regardless of whether the condition being waited for has actually been met. This can lead to longer test execution times and flaky tests.

➟ **Hard-coded waits:** By using a fixed delay, Thread.sleep() can make the test script more brittle and difficult to maintain, as any changes to the delay time or the conditions being waited for will require changes to the test code.

➟ **Limited flexibility:** Thread.sleep() does not provide any mechanism for dynamically adjusting the wait time based on the current state of the application.

Implicit Wait -

In Selenium, an Implicit Wait is a wait that is applied by default to all web elements. This means that when a web element is not immediately available, Selenium will wait for a certain period of time before throwing an exception.

The Implicit Wait is useful in scenarios where web pages load slowly or in cases where the elements on a page take time to appear. The wait time specified in the Implicit Wait applies to all elements on the page, and not just the ones that are being accessed through the WebDriver instance.

Note - Implicit Wait can be set at the WebDriver level and it applies to all the subsequent calls that are made to the WebDriver instance.

Syntax - driver.manage.timeouts.implicitlywait(TimeOut,Timeunit.SECONDS);

➢ **Advantages of using Implicit wait -**

➟ **Easy to Use:** Implicit Wait is very easy to set up and use. It can be set once for the entire session and does not require any additional code to be added to the test script.

➟ **Saves Time:** Implicit Wait can help to reduce the overall time taken to execute the test cases. Instead of waiting for a fixed amount of time, it waits only for the required time, which can help to speed up the test execution.

➟ **Consistency:** Implicit Wait applies to all the elements on a page, which ensures that the test script behaves consistently across different pages.

➢ **Disadvantages of using Implicit wait -**

➟ **Not Customizable:** Implicit Wait cannot be customized for specific elements or conditions, and it applies to all elements on the page. This can lead to longer wait times/exceptions for certain elements or conditions.

➟ **Hidden Waits:** Implicit Wait can lead to hidden waits, which means that the test script may wait for an element to appear without explicitly waiting for it, leading to flaky tests and longer execution times.

➟ **Increases the Risk of NoSuchElementException:** In some cases, if the Implicit Wait time is set too high, it can increase the risk of NoSuchElementException, as Selenium may wait longer than necessary for an element to appear.

Explicit Wait -

In Selenium, Explicit Wait is a type of wait that allows the test script to wait for a specific condition to be met before proceeding with the next step.

Unlike Implicit Wait, which waits for a fixed amount of time before throwing an exception, Explicit Wait waits for a specific condition to be met before proceeding. Explicit Wait is useful in scenarios where the web page elements may take different amounts of time to load or appear, and the test script needs to wait until a specific element appears or a specific condition is met.

Note - Declaration of Explicit Wait is done at the WebDriver level and it applies to all the subsequent calls that are made to the WebDriver instance but usage of Explicit wait is done for different elements individually.

Syntax-
Declaration- WebDriverWait wait = new WebDriverWait(WebDriver Reference,TimeOut);
Usage- wait.until(ExpectedConditions.visibilityOf(WebElement));

➢ **Advantages of using Explicit wait -**

➞ **Improved Reliability:** Explicit Wait can help to improve the reliability of the test script by waiting for a specific condition to be met before proceeding, which reduces the risk of flaky tests and false positives.

➞ **Customizable:** Explicit Wait can be customized for specific elements or conditions, which provides greater flexibility and control over the wait time and can help to reduce the overall execution time.

➞ **Saves Time:** Explicit Wait can help to reduce the overall time taken to execute the test cases by waiting only for the required time, instead of waiting for a fixed amount of time.
➞ **Explicit Error Messages:** When an Explicit Wait condition is not met, it throws an exception with a clear error message, which helps in identifying the root cause of the issue.

➞ **Finding Element:** Finding Elements is inclusive in Explicit wait. So code is somehow reduced.

➢ **Disadvantages of using Explicit wait -**

➞ **Requires More Code:** Explicit Wait requires more code to be added to the test script as compared to Implicit Wait, which can make the test script longer and more complex.

➞ **Learning Curve:** Using Explicit Wait requires a good understanding of the different conditions that can be waited for, which can have a steeper learning curve as compared to Implicit Wait.

➞ **Increases Complexity:** As Explicit Wait conditions can be customized for specific elements or conditions, it can increase the complexity of the test script.

Fluent Wait -

Fluent Wait is a type of wait in Selenium that allows you to wait for a certain condition to occur within a specified amount of time. It is called "fluent" because it allows you to chain different wait conditions together in a single expression.

Syntax-
Declaration-

```
Wait<WebDriver> wait = new FluentWait<>(driver)
        .withTimeout(Duration.ofSeconds(30))
        .pollingEvery(Duration.ofSeconds(5))
        .ignoring(NoSuchElementException.class);
```

Usage-

```
mywait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(Locator)));
```

Here, FluentWait will try to find the element every 5 seconds, and it will continue to try for a maximum of 30 seconds. If the element is found before the timeout period expires, the wait will end and the element will be returned. If the element is not found within the timeout period, a TimeoutException will be thrown.

➢ **Advantages of using Fluent wait -**

➞ **Customizable:** FluentWait is highly Customizable, allowing you to specify the maximum time to wait, the frequency of checks, and the conditions to ignore, making it more flexible than other wait types.
➞ **Readable:** The fluent syntax of FluentWait allows for code that is easy to read and understand, making it more readable than other wait types.
➞ **Robust:** FluentWait can handle a variety of exceptions and conditions, making it more robust than other wait types.
➞ **Efficient:** FluentWait uses a polling mechanism to check for conditions, which is more efficient than a simple sleep-based wait.

➢ **Disadvantages of using Fluent wait -**

➞ **Complexity:** FluentWait can be more complex to use than other wait types, as it requires you to chain together multiple methods to set up the wait conditions.
➞ **Overhead:** The polling mechanism used by FluentWait can put additional load on the application under test, particularly if the polling interval is set too low.
➞ **Overreliance:** Overreliance on FluentWait can lead to poorly written tests that are not resilient to changes in the application or environment.

Page Lode Timeouts -

This sets the time to wait for a page to load completely before throwing an error.
This timeout is applicable only to driver.manage() and driver.navigate.to() methods.

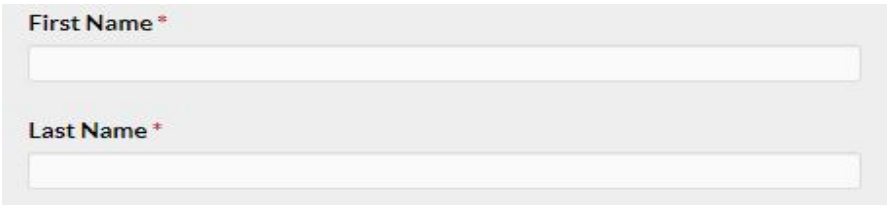
Syntax-

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

How to Handle Text Box -

A textbox is a graphical user interface (GUI) element that allows users to enter or clear a text/alphanumeric value.

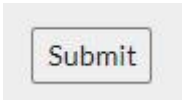
➞ **Step-1: isDisplayed() -** determines if an element is displayed or not.
➞ **Step-2: isEnabled() -** determines if an element is enabled or not.
➞ **Step-3: sendKeys() -** to enter the user input value.
➞ **Step-4: clear() -** clear the values present in the text box.



How to Handle Button -

A button is a graphical user interface (GUI) element that allows users to take actions, and make choices, with a single tap.

➞ **Step-1: isDisplayed() -** determines if an element is displayed or not.
➞ **Step-2: isEnabled() -** determines if an element is enabled or not.
➞ **Step-3: click() -** to click the button.



How to Handle Radio Button -

A radio button is a graphical user interface (GUI) element that allow users to make a single selection from a list of options.

➞ **Step-1: isDisplayed() -** determines if an element is displayed or not.
➞ **Step-2: isEnabled() -** determines if an element is enabled or not.
➞ **Step-3: isSelected() -** determines if an element is selected or not.
➞ **Step-4: click() -** to click the button.

How to Handle CheckBox -

A checkbox is a graphical user interface (GUI) element that allows users to select one or more options from a list of choices. It is a small box that can be either checked (selected) or unchecked (deselected).

➞ **Step-1: isDisplayed() -** determines if an element is displayed or not.
➞ **Step-2: isEnabled() -** determines if an element is enabled or not.
➞ **Step-3: isSelected() -** determines if an element is selected or not.
➞ **Step-4: click() -** to click the button.

Course Interested

☐ Selenium WebDriver ☐ UFT ☐ TestNG ☐ Core Java ☐ Functional Testing ☐ Others

How to Handle Images -

An image is a graphical user interface (GUI) element that is provided as an image on a webpage and it may or may not have the ability to perform actions.

➢ **Images without functionality -**
➥ **Step-1: isDisplayed() -** determines if an element is displayed or not.
➥ **Step-2: getAttribute() -** get the value of an element.

➢ **Images with functionality -**
➥ **Step-1: isDisplayed() -** determines if an element is displayed or not.
➥ **Step-2: getAttribute() -** get the value of an element.
➥ **Step-3: click() -** to click the an element.

How to Handle Text/Error Message -

A text/error message is a graphical user interface (GUI) element that is basically a text message or an error message that needs to be validated.

➥ **Step-1: isDisplayed() -** determines if an element is displayed or not.
➥ **Step-2: getText() -** get the value of the element.

How to Handle DropDown -

Dropdowns are graphical user interface (GUI) elements that allow users to select one option from a list of available options that "drop down" when the user clicks or hovers over the dropdown menu.

Types of Dropdown -
➥ Dropdown having select tag in DOM.
➥ Dropdown not having select tag in DOM - BootStrap Dropdown.
➥ AutoSuggestive dropdown - Dynamic Dropdown.

➢ **Dropdown with Select tag in DOM -**
To handle such dropdowns there is a separate class in Java called Select class. Select class has certain methods to handle these dropdowns.

➥ **Step-1: Creating object for select class -** Select dropdown = new Select(WebElement);
➥ **Step-2: isDisplayed() -** determines if an element is displayed or not.
➥ **Step-3: isEnabled() -** determines if an element is enabled or not.
➥ **Step-4: selectByVisibleText() -** to select the user input value based on visible text.
➥ **Step-4: selectByValue -** to select the user input value based on value.
➥ **Step-4: selectByIndex() -** to select the user input value based on index number.

➢ **Dropdown without Select tag in DOM -**
To handle such dropdowns we use primitive ways of locating elements.

➥ **Step-1:** Fetching all the options of dropdown in a list.
➥ **Step-2:** Selecting an element from the list.

➢ **Auto-Suggestive Dropdowns -**
Auto-suggestive dropdowns, also known as autocomplete dropdowns, are user interface elements that provide suggestions to users as they type in a search term or query. These dropdowns are commonly used in search bars or form fields to help users quickly find what they are looking for without having to type out the full search term.

➥ **Step-1:** Fetching all the options of dropdown in a list.
➥ **Step-2:** Selecting an element from the list.

How to Handle Window Authentication -

Authentication alerts are popups or messages that appear on web pages to prompt users to authenticate themselves. Authentication is the process of verifying the identity of a user to grant access to protected resources or information.

Authentications can be handled by manipulating the URL of test application.

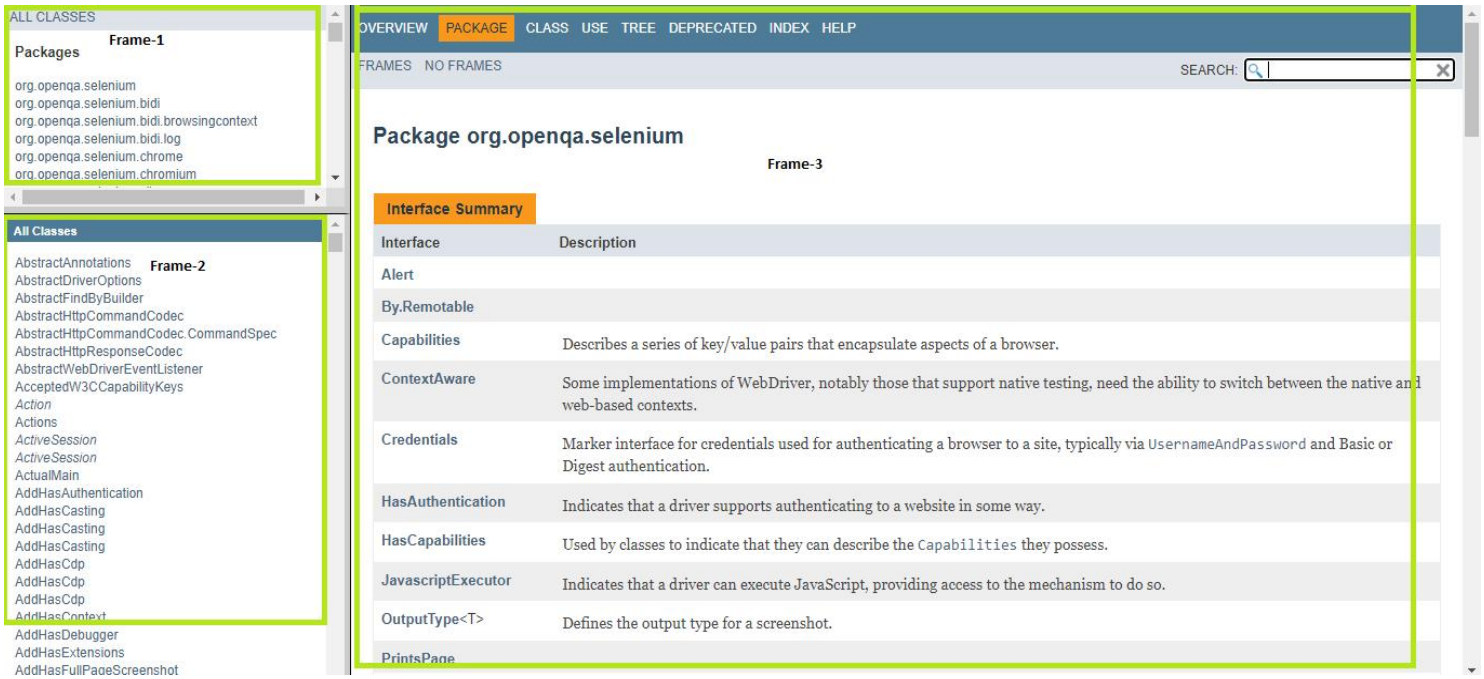
➢ **Syntax:** https://username:password@URL

How to Handle iFrames -

Frames and iframes are HTML elements that allow web developers to divide a webpage into separate sections, each of which can display different content.

Frames are used to create separate regions within a webpage, each with its own independent HTML document. Each frame is a separate HTML document, and changes made to one frame do not affect the other frames on the page.

➥ **Step-1: switchTo() -** To give you access to switch frames and windows.
➥ **Step-2: defaultContent() -** Used to switch the control back to default content in the window.
➥ **Step-3: frame(name) -** Switch to particular frame using name.
➥ **Step-4: frame(index) -** Switch to particular frame using index.



How to Handle Mouse Events -

Mouse actions in Selenium refer to the interaction with web elements using the mouse, such as clicking, double-clicking, right-clicking, hovering, dragging and dropping, and more.

Mouse actions can be performed using the "**Actions**" class provided by Selenium WebDriver.

➥ **Step-1:** Creating object of actions class and pass driver in it.
```
Actions act = new Actions(driver);
```

➥ **Step-2:** Perform mouse actions using actions class methods.
```
act.moveToElement(WebElement).moveToElement(WebElement).click().build().perform();
act.contextClick(WebElement)..build().perform();
```

Important Methods in Actions class -
➥ **build() -** Method in Actions class is used to create chain of action or operation you want to perform.
➥ **perform() -** Method in Actions Class is used to execute chain of action which are build using Action build method.

Note- .build().perform() is used at the end of the statement to build and perform the action. Action will not be performed without .build().perform().

Mouse Actions that can be performed using Actions class are -
➞ **Mouse Hover -** "moveToElement(WebElement)" method.
➞ **Right click -** "contextClick(WebElement)" method.
➞ **Double click -** "doubleClick(WebElement)" method.
➞ **Drag and Drop -** "dragAndDrop(sourceElement, targetElement)" method.
➞ **Slider -** "element.getLocation()" then "dragAndDropBy(element , x-axis , y-axis)" methods.

How to Handle Keyboard Events -

Keyboard actions in Selenium refer to the interaction with web elements using the keyboard, such as ctrl, space, tab, enter, etc keys and perform actions by combination of keys.

Keyboard actions can be performed using the "Actions" class provided by Selenium WebDriver.

➞ **Step-1:** Creating object of actions class and pass driver in it.

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

➞ **Step-2:** Perform mouse actions using actions class methods.

```
act.keyDown(Keys.CONTROL).sendKeys("a").keyUp(Keys.CONTROL).perform();      - Select all
act.keyDown(Keys.CONTROL).sendKeys("c").keyUp(Keys.CONTROL).perform();      - Copy
act.keyDown(Keys.CONTROL).sendKeys("v").keyUp(Keys.CONTROL).perform();      - Paste
act.keyDown(Keys.TAB).keyUp(Keys.TAB).perform();                            - Tab
```

Keyboard Actions that can be performed using Actions class are -
➞ **Press button -** "keyDown(Keys.NameOfKey)" method.
➞ **Release pressed button -** "keyUp(Keys.NameOfKey)" method.

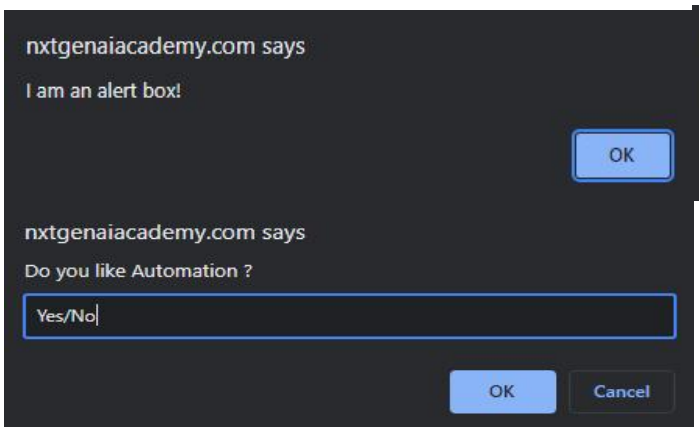
How to Handle Alert Popups -

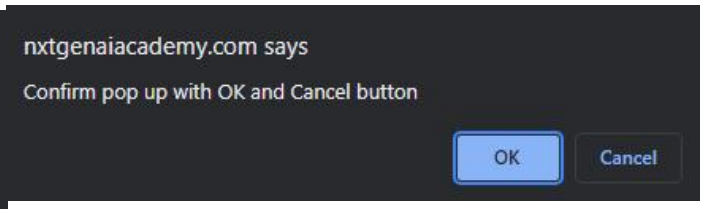
Alert popups on web pages are small messages that appear in a separate window or dialog box when a certain event occurs on a website. These pop ups are designed to alert users to important information or to prompt them to take some action, such as confirming a form submission or warning about potential errors or security issues.

Alerts are not a part of HTML DOM and cannot be located with elements. So, a different approach is used for handling alert popups.

Types of Alerts/Popups -
➞ Alert with OK button
➞ Alert with OK & Cancel button
➞ Alert with Input box along with OK & Cancel button
➞ Alert with No Elements

➞ **Step-1: switchTo().alert() -** To give you access to alert popup.
➞ **Step-2: accept() -** To click on OK button in the popup.
➞ **Step-3: dismiss() -** To click on Cancel button in the popup.





How to Handle multiple Browser Windows -

To handle multiple browser windows using Java and Selenium, you need to identify and switch between the different window handles. Each window opened by the WebDriver has a unique window handle, which allows you to interact with them individually.

➢ **getWindowHandle() -** This method is used to get the window handle of the currently focused browser window.
➢ **getWindowHandles() -** This method is used to get the window handles of all the browser windows opened by the WebDriver. It returns a set of strings representing the window handles. You can use this set to iterate through each window handle and switch to different windows.
➥ **Step-1: getWindowHandles() -** To give you the id's of different window to switch your control.
➥ **Step-2:** User Iterator class to iterate over the windows.

```
Iterator<String> iterator = windowsId.iterator();
iterator.next();
```

➥ **Step-3: switchTo().window(windowId) -** This will switch control to a different window.

How to Handle multiple Browser Tabs -

Same steps and mechanism that discussed above for handling multiple windows will be used for handling multiple tabs.

How to Handle Web Tables -

Web tables are HTML elements that display data in a tabular format, typically consisting of rows and columns. They are commonly used to present structured information such as product listings, financial data, schedules, and more on web pages.

➢ **Static Web tables -** Static web tables are tables in which the data is pre-defined and does not change dynamically. The content of static web tables is typically hardcoded in the HTML markup.
➢ **Dynamic Web tables -** Dynamic web tables are tables in which the data is generated or modified dynamically, usually through JavaScript or AJAX requests. The content of dynamic web tables can change without a full page reload, making them more challenging to handle compared to static tables.

➥ **Step-1:** Create webdriver instance and Launch Application
➥ **Step-2:** Identify web table using proper locator
➥ **Step-3:** Retrieve the rows webelement using TR tag
➥ **Step-4:** Calculate the number of rows
➥ **Step-5:** Use for loop to iterate till last row
➥ **Step-6:** Retrieve the column webelement using TD tag
➥ **Step-7:** Calculate the number of columns
➥ **Step-8:** Use for each loop to iterate till last column of each row
➥ **Step-9:** Include filter criteria as per requirement
➥ **Step-10:** Close the application

How to Capture Screenshot -

➢ **Why Is It Important To Capture Screenshots In Selenium Testing?**
➟ Analyze the bug.
➟ Trace and examine the reason behind the test failure.
➟ Understand the end to end test case flow.
➟ Track test execution.
➟ Analyze the behavior of the test in different browsers/environments.

➢ **When To Capture Screenshots In Selenium Testing?**
➟ Capture screenshot for each and every test case.
➟ Capturing screenshots only when there is a failure.
➟ Capturing screenshots in specific checkpoints.
➟ Capturing screenshots in different browsers.
➟ Capturing screenshots of specific elements in a webpage.

➥ **Step-1:** TakesScreenshot ts = (TakesScreenshot)driver;
➥ **Step-2:** File file = ts.getScreenshotAs(OutputType.FILE);
➥ **Step-3:** FileUtils.copyFile(file,new File ("C:\\Users\\Himanshu Pathak\\Desktop\\ss.png"));
Note - Ensure to add the Apache Common io jar files.

Java Script Executor -

In Selenium WebDriver, the JavascriptExecutor interface allows you to execute JavaScript code within the context of the current browser window or web page. This feature is useful for performing actions that cannot be easily achieved using the standard Selenium WebDriver commands. With JavascriptExecutor, you can interact with elements, manipulate the DOM, scroll, handle asynchronous operations, and perform other advanced operations.

➞ **Step-1:** JavascriptExecutor jsExecutor = (JavascriptExecutor) driver;
➞ **Step-2:** js.executeScript("arguments[0].setAttribute('value','VALUE')", WebElement); — Send Keys using JS executor
➞ **Step-3:** js.executeScript("arguments[0].click();",WebElement); — Click using JS executor