[**What is Selenium and Its Hierarchy?        3**](#h.zu0gcz)

[WebDriver 4 Architecture -        3](#h.3jtnz0s)

[How Client Server Communication Works in Selenium?        4](#h.1yyy98l)

[How Browser Drivers Work in Selenium?        7](#h.4iylrwe)

[**1. What is Selenium and what are its components?        7**](#h.2y3w247)

[**2. What are the advantages of using Selenium?        8**](#h.1d96cc0)

[**3. What are different types of locators used for identifying web elements?        8**](#h.3x8tuzt)

[**4. Describe different types of web testing supported by Selenium.        9**](#h.2ce457m)

[**5. How do you handle dynamic web elements that change IDs or properties frequently?        9**](#h.rjefff)

[**6. What are some common challenges faced in Selenium automation and how do you address them?        9**](#h.3bj1y38)

[**7. Can you explain the concept of Page Object Model (POM) in Selenium?        9**](#h.1qoc8b1)

[**8. How do you handle data in your Selenium scripts?        10**](#h.4anzqyu)

[**9. What are some best practices for writing maintainable Selenium scripts?        10**](#h.2pta16n)

[**10. Can you explain how Selenium interacts with the web browser through drivers?        10**](#h.14ykbeg)

[**11. How can I handle dynamic web elements that change their IDs or other properties?        11**](#h.3oy7u29)

[**12. What are some common challenges faced when using Selenium?        11**](#h.243i4a2)

[**13. Explain the difference between findElement and findElements methods in Selenium.        11**](#h.j8sehv)

[**14. How do you handle synchronization issues in Selenium?        11**](#h.338fx5o)

[**15. What is Fluent Wait?        12**](#h.1idq7dh)

[**16. What is Implicit wait?        13**](#h.42ddq1a)

[**17. What is Explicit wait?        14**](#h.2hio093)

[**18. Describe your experience with handling popups and alerts in Selenium.        15**](#h.wnyagw)

[**19. Have you experience using different headless browsers for automation?        15**](#h.3gnlt4p)

[**20. How to handle popups and alerts in selenium?        16**](#h.1vsw3ci)

[**21. How to use a headless browser in selenium?        17**](#h.4fsjm0b)

[**22. What is Actions class and its use in Selenium?        18**](#h.2uxtw84)

[**23. What is the select class in Selenium and its use?        20**](#h.1a346fx)

[**ID - A webElement having a unique ID attribute, then it can be used to identify it.        24**](#h.2981zbj)

[**NAME - A webElement having a unique NAME attribute, but in most cases multiple elements can have the same name, so it’s less reliable.        24**](#h.odc9jc)

[**LinkText : Accessing links using their exact link text, This makes it easy to create such locators, but these are flaky because they can fail when text changes for links used.        25**](#h.38czs75)

[**Partial LinkText : Accessing links using a portion of their link text is done using the By.partialLinkText() method.        25**](#h.1nia2ey)

[**CSS : Used to identify webElements based on different properties such as:        25**](#h.47hxl2r)

[**XPath (XML Path Language)        26**](#h.2mn7vak)

[**XPath Axes:        28**](#h.11si5id)

[**How to find webelements in selenium with syntax?        30**](#h.20xfydz)

[**How to use JAVAScript Executor?        31**](#h.4kx3h1s)

[**JAVASCRIPT commands for scrolling in browser console tab:        33**](#h.302dr9l)

[**Switch use in Selenium in Detail?        34**](#h.1f7o1he)

[**Manage operation in selenium?        36**](#h.3z7bk57)

[**Browser Options Detailed Understanding in Selenium?        38**](#h.2eclud0)

[**How to Disable Popups in Chrome browser?        39**](#h.thw4kt)

[**How to disable location Popups in chrome browser?        40**](#h.3dhjn8m)

[**How to use incognito chrome browser for selenium script execution?        40**](#h.1smtxgf)

[**Basic Authentication in Selenium Tests:        41**](#h.49gfa85)

[**How to upload files using Selenium?        42**](#h.2olpkfy)

[**How to Take Screenshots in Selenium?        42**](#h.13qzunr)

[How to take element level screenshot in selenium?        43](#h.3nqndbk)

[How to take a full page screenshot using selenium?        43](#h.22vxnjd)

[**Selenium Specific Q&A with Examples        45**](#h.mnq6vlhx6hjk)

[1) How to handle Selenium WebDriver Exceptions?        45](#h.ho91lo91b8cs)

[2) There are four browser windows opened and you don’t have any idea where the required element is present. What will be your approach to find that element?        45](#h.l6rfe1xp87kr)

[3) How do you handle an alert pop-up in Selenium?        45](#h.ktarp7gz1te2)

[4) How do you retrieve the text displayed on an Alert?        46](#h.vecwmhggn2ss)

[5) How do you type text into the text box on an Alert?        46](#h.3lgmc2fxgx7g)

[6) Is Alert in Selenium an Interface or Class?        46](#h.dy24m7101j9u)

[7) How do you handle frames in Selenium?        46](#h.jue6p6sxbtht)

[8) Give an example of method overloading concept that you have used in Selenium?        46](#h.yarrqiem4f8q)

[9) How do you select a value from a drop-down field and what are the different methods available?        47](#h.5csr21td8zpw)

[10) When your XPath is matching more than one element, how do you handle it to locate the required element?        47](#h.6mdakn6xwl5o)

[11) How do you capture screen-shots in Selenium and what is the best place to have the screen-shot code?        47](#h.ike66e1isd8g)

[12) Write the code for connecting to Excel files and other operations?        48](#h.f956gedobf4r)

[13) How do you read and write into a PDF file?        48](#h.9xwarwkrie4e)

[14) What are the disadvantages of Selenium?        48](#h.5dzq1gw3vmh1)

[15) How do you debug your automation code when it is not working as expected?        48](#h.jfzogt5r5z55)

[16) What are the end methods you use for verifying whether the end result is achieved by our Selenium automation scripts?        49](#h.gv3v9nevuhre)

[17) How do you clear the cookies of a browser using Selenium, before starting the execution?        49](#h.skhgb3uolf0h)

[18) How do you implement collections in your framework?        49](#h.upwzx4hzrk6o)

[19) Give a scenario where inheritance is used in your framework?        49](#h.4fsticsiwkbq)

[20) Give a scenario where an interface is used in your framework?        49](#h.4c8ft4aadvxl)

[21) Write a code using JavascriptExecutor to scroll the web page?        50](#h.ahba7eyvxt6l)

[22) What is the use of property files in Selenium?        50](#h.rzjywmdy45nb)

[23) How do you handle multiple browsers selection in Selenium?        50](#h.qgsb7g3a1efy)

[24) What do you use for reporting in your Selenium Project?        50](#h.2waygn5k67v)

[25) How Cross Browser testing is handled in Selenium?        50](#h.x2656elsc50t)

[26) What are the types of WebDriver APIs available in Selenium?        51](#h.xwigkqt0ndz7)

[27) What is the super interface of WebDriver?        52](#h.xqvuoudv46cm)

[28) . Explain the line of code Webdriver driver = new FirefoxDriver(); ?        52](#h.vnktrdibc57y)

[29). We do create a reference variable ‘driver’ of type WebDriver, why?        52](#h.a6q2jipeoors)

[30) What is Page Object Model and Page Factory in Selenium?        52](#h.nt1lxbdgzfz2)

[31) How do you write a locator to identify paragraph elements that are the immediate child of a div element, or the descendent of a div element?        53](#h.wtngja5cpfcj)

[32) Why do we use headless drivers? How can you visually investigate test failure when using headless drivers?        53](#h.232t5ikkqn01)

[33) What will you do if there are failures in your suite execution and what is your approach?        54](#h.6nxv6rfsb5n)

[34) What expected conditions can be used in Explicit waits?        54](#h.672c2lnyx3lb)

[35) Tell me any difficulties you faced in developing automation scripts? Can you give any examples for any complex scenarios handled?        55](#h.i2dz2f8yids)

[36) What is the problem with Thread.Sleep in code?        56](#h.p17j43i9d5x4)

[37) Common exceptions which you have faced while working with Selenium?        56](#h.26lestgepq8)

## What is Selenium and Its Hierarchy?

### WebDriver 4 Architecture -

**Explanation:**

Selenium 4 is now W3C Compliant with Selenium 4. You may no longer be required to add ‘tweaks’ in the test script to make it work across different browsers, as everything (i.e., browsers & WebDriver APIs) runs in the W3C standard protocol.

W3C stands for the World Wide Web Consortium, an international community that develops and maintains standards and guidelines for the World Wide Web. The main aim of the W3C is to ensure the long-term growth and interoperability of the Web.

### How Client Server Communication Works in Selenium?

Following pointers would help to understand the communication between client and server using WebDriver protocol:

1. A command execution request is sent by the Selenium client (test script written in any language such as Java, Python, JavaScript, etc) to perform various actions on the browser such as navigating to a URL, interacting with the elements, executing code and so on.
2. The WebDriver client serializes the request into a standardized format specified by the WebDriver protocol. This format can be JSON or a similar format, depending on the specific implementation.
3. The serialized request is transmitted to the browser driver which acts as a bridge between the WebDriver client and the Web browser.
4. The browser driver processes the serialized request and then performs the necessary actions on the Web browser.
5. Browser driver generates a response of the command execution which includes relevant data or information, such as the status and the success or failure status.
6. The browser driver serialise the response into the standardized format by the WebDriver protocol and transmits it back to the client.
7. The client receives the response from the browser driver and deserializes the response. It extracts the relevant information, and the client can use this information to verify success / failure of the command execution.

Selenium Manager in Selenium version 4.6.0 is a big relief for the automation test engineers as it is not required to provide the executable driver path.

Selenium WebDriver Hierarchy

As an automation test engineer, we have been using Selenium WebDriver. Currently, while writing this blog on Selenium 4 WebDriver Hierarchy, Selenium’s latest version is 4.7.0. We know that by running the following line of code, the Chrome browser will be started, and we will be able to test the web page further using WebDriver methods.

WebDriver driver = new ChromeDriver();

1

WebDriver driver = new ChromeDriver();

However, very few automation test engineers know about the internal working of the WebDriver Interface. So, let’s dive deep into this and understand how Selenium WebDriver works.

Here is the pictorial representation of the Selenium WebDriver hierarchy.

**RemoteWebDriver** class because it is a fully implemented WebDriver Interface class extended by every BrowserDriver class within the Selenium framework.

RemoteWebDriver Class has the following constructors, which can be used to instantiate an instance of the class:

RemoteWebDriver(ICapabilities)

RemoteWebDriver(Uri, ICapabilities)

RemoteWebDriver(ICommandExecutor, ICapabilities)

RemoteWebDriver(Uri, ICapabilities, TimeSpan)

RemoteWebdriver class implements the following interfaces:

WebDriver

JavaScriptExecutor

TakesScreenshot

HasVirtualAuthenticator

PrintsPage

HasCapabilities

Interactive

**WebDriver** Interface is the core of the Selenium WebDriver as it has all the required methods and respective nested interfaces defined within it, which helps in simulating user actions inside the browser.

**How to use the WebDriver Interface?**

The WebDriver interface defines methods for interacting with a web page through a web browser. To use the WebDriver interface, you must first import the appropriate libraries and instantiate a WebDriver object.

### How Browser Drivers Work in Selenium?

Browser Drivers

Selenium provides drivers specific to each browser and without revealing the internal logic of browser functionality, the browser driver interacts with the respective browser by establishing a secure connection. These browser drivers are also specific to the language which is used for test case automation like C#, Python, Java, etc.

When a test script is executed with the help of WebDriver, the following tasks are performed in the background:

1. An HTTP request is generated and it is delivered to the browser driver for every Selenium Command
2. The HTTP request is received by the driver through an HTTP server
3. All the steps/instructions to be executed on the browser is decided by an HTTP server
4. The HTTP server then receives the execution status and in turn sends it back to the automation scripts

## 1. What is Selenium and what are its components?

Answer: Selenium is an open-source framework for automating web browser interactions. It consists of various components:

WebDriver: The core component that interacts with the browser through drivers like ChromeDriver (Chrome), FirefoxDriver (Firefox), etc.

Language Bindings: APIs for different programming languages like Java, Python, JavaScript, etc., to interact with WebDriver.

Selenium IDE: A record-and-playback tool for creating basic test scripts.

Grid: Distributed test execution across multiple machines can be achieved, it can be used along with TestNG for parallel testing.

## 2. What are the advantages of using Selenium?

Answer:

Reduced testing time and effort: Automates repetitive tasks, leading to faster testing cycles.

Improved test coverage: Enables testing complex scenarios and edge cases manually difficult.

Increased software quality: Catches more bugs and improves overall software reliability.

Cross-browser compatibility: Supports testing across different browsers and operating systems.

Open-source and free to use: Reduces costs compared to proprietary automation tools.

## 3. What are different types of locators used for identifying web elements?

Answer: Locators specify how to find specific elements on a web page. Some common types include:

ID: Unique identifier of an element (e.g., #login\_button).

Name: Attribute assigned to the element (e.g., username).

Class name: Class assigned to the element (e.g., error\_message).

XPath: Powerful path-based expression to locate elements based on their structure.

CSS selector: Similar to CSS selectors used for styling, targeting elements based on their attributes and relationships.

Tagname

Link Text

Partial Link Text

## 4. Describe different types of web testing supported by Selenium.

Answer: Functional Testing: Verifies core functionality of the web application.

Regression Testing: Ensures existing functionality remains intact after changes.

Smoke Testing: Basic tests to confirm core features work before deeper testing.

UI Testing: Validates user interface elements and their behavior.

Cross-browser Testing: Tests the application across different browsers and platforms.

## 5. How do you handle dynamic web elements that change IDs or properties frequently?

Answer: Relying on dynamic elements like IDs can be unreliable. Consider using locators based on other stable attributes like class names, unique text content, or even XPath expressions that consider element relationships rather than specific attributes. Additionally, wait mechanisms like WebDriverWait can ensure elements are present and stable before interacting with them.

## 6. What are some common challenges faced in Selenium automation and how do you address them?

Answer: Common challenges include:

Element not found: Use reliable locators, check for stale element references, and implement wait mechanisms.

Synchronization issues: Use explicit waits for specific conditions like page load or element visibility.

Popups and alerts: Implement pop-up handling strategies like dismissing or entering data.

Cross-browser compatibility issues: Leverage WebDriver capabilities for different browsers and adjust locators if needed.

## 7. Can you explain the concept of Page Object Model (POM) in Selenium?

Answer: POM is a design pattern for organizing test code by grouping elements and actions associated with specific web pages. This improves code modularity, reusability, and maintainability. It involves creating separate classes for each page, encapsulating elements and actions within that page.

## 8. How do you handle data in your Selenium scripts?

Answer: You can use various approaches:

Hardcoded data: Embed data directly in the script, suitable for simple cases.

Data files: Store data in external files like CSV or Excel, easily read and updated.

Database connections: Use database connections to retrieve test data dynamically.

Choose the approach based on your data complexity and test needs.

## 9. What are some best practices for writing maintainable Selenium scripts?

Use descriptive variable names and comments.

Break down complex actions into smaller functions.

Follow consistent coding conventions.

Log actions and errors for debugging purposes.

Use configuration files for environment settings.

## 10. Can you explain how Selenium interacts with the web browser through drivers?

Answer: Selenium drivers act as bridges between your programming language and the browser. They translate your commands into browser-specific actions, like clicking buttons or entering text. Different browsers require specific drivers, like ChromeDriver for Chrome and FirefoxDriver for Firefox. These drivers communicate with the browser's internal automation interfaces to perform the desired actions.

## 11. How can I handle dynamic web elements that change their IDs or other properties?

Answer: Relying solely on specific locators like IDs can be fragile. Consider using more robust approaches like:

XPath with relative paths based on element relationships.

CSS selectors based on element attributes and class names.

Page Object Model (POM) to encapsulate element identification logic in separate classes.

## 12. What are some common challenges faced when using Selenium?

Element identification issues: Locating elements dynamically or those with changing attributes can be tricky.

Synchronization: Ensuring elements are ready before interacting with them can be challenging.

Cross-browser compatibility issues: Some features might not work consistently across all browsers.

Debugging test failures: Identifying the root cause of failed tests can be time-consuming.

## 13. Explain the difference between findElement and findElements methods in Selenium.

findElement returns the first matching element for a given locator. If no element is found, it throws a NoSuchElementException.

findElements returns a list of all matching elements for a given locator. If no element is found, it returns an empty list.

## 14. How do you handle synchronization issues in Selenium?

Explicit waits (WebDriverWait) with Expected Conditions like elementToBeClickable and visibilityOfElementLocated.

Implicit waits set globally to wait for a certain amount of time before throwing an exception.

Fluent waits are used for polling and checking element availability

## 15. What is Fluent Wait?

Fluent wait is a powerful and customizable mechanism in Selenium that helps manage waiting for specific conditions during your automated tests. It's more flexible and efficient than traditional implicit waits and offers several advantages:

What is it?

Fluent wait defines a maximum timeout and a polling frequency for checking a specific condition.

Instead of waiting the entire timeout continuously, it attempts the condition at regular intervals until it becomes true or the timeout is reached.

This makes your tests more efficient and responsive, avoiding unnecessary delays even if the element/condition appears sooner.

Key features:

Configurable timeout: Set the maximum time to wait for the condition.

Polling frequency: Define how often to check for the condition (e.g., every 500 milliseconds).

Ignored exceptions: Specify exceptions to ignore during the wait (e.g., StaleElementReferenceException).

Custom condition: Define your own condition using a Lambda function to check for anything

beyond simple element presence.

Benefits:

Faster tests: Only waits for the actual condition instead of a fixed timeout.

Improved reliability: Reduces flaky tests caused by unpredictable element loading times.

More accurate tests: Checks for specific conditions beyond just element presence.

Enhanced code readability: Makes wait logic clear and concise compared to nested loops.

How to use it:

Initialize a FluentWait object with your WebDriver instance and timeout.

Use methods like withTimeout, pollingEvery, and ignoring to configure the wait parameters.

Define your specific condition through a Lambda function in the until method.

The wait will keep checking the condition until it becomes true or the timeout is reached.

Example:  (IN JAVA)

WebDriver driver = new ChromeDriver();

FluentWait<WebDriver> wait = new FluentWait<>(driver)

  .withTimeout(Duration.ofSeconds(10))

  .pollingEvery(Duration.ofMillis(500))

  .ignoring(StaleElementReferenceException.class)

  .until(driver -> driver.findElement(By.id("submitButton")).isEnabled());

System.out.println("Submit button is now enabled!");

This example waits for up to 10 seconds, checking every 500 milliseconds, ignoring stale element exceptions, until the submit button is enabled.

## 16. What is Implicit wait?

A Simple Global Delay

What it is:

Implicit wait is a global setting that instructs Selenium WebDriver to wait for a certain amount of time before throwing a NoSuchElementException if an element is not immediately available.

It's a simple, built-in mechanism to handle potential synchronization issues during test execution.

How it works:

You set a timeout value (in seconds) using driver.manage().timeouts().implicitlyWait(time, TimeUnit.SECONDS).

Before each findElement or findElements call, WebDriver pauses for the specified time if the element isn't found immediately.

If the element still isn't found after the timeout, the exception is thrown.

Key points:

It applies to all subsequent findElement calls within the current WebDriver session.

It only waits for elements to be present, not necessarily visible or clickable.

It can slow down test execution if elements load quickly.

It can be overridden by explicit waits for specific conditions.

When to use it:

For simple scenarios where you expect minor delays in element loading.

To provide a basic level of synchronization for your tests.

When to avoid it:

When you need precise control over waiting for specific conditions like element visibility or clickability.

When you want to avoid unnecessary delays in test execution.

When you're dealing with dynamic elements or unpredictable loading times.

Best practices:

Use a moderate timeout value (e.g., 5-10 seconds).

Avoid using it alongside explicit waits as it can lead to unpredictable behavior.

Consider Fluent Wait for more granular control over waiting conditions.

Example:

WebDriver driver = new ChromeDriver();

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

// Subsequent findElement calls will wait up to 10 seconds

driver.findElement(By.id("myElement"));

## 17. What is Explicit wait?

Targeted Waiting for Specific Conditions

What it is:

Explicit wait is a dynamic and flexible mechanism in Selenium that allows you to wait for specific conditions to be met before proceeding with test script execution.

It provides more control and precision compared to implicit wait, ensuring actions are performed only when elements are ready.

How it works:

You create a WebDriverWait object, specifying a timeout and the WebDriver instance.

You use the until method to define the condition to wait for, often using Expected Conditions classes for common scenarios.

WebDriverWait polls the condition repeatedly until it's true or the timeout expires, throwing a TimeoutException if unsuccessful.

Key points:

Targets specific conditions, not just element presence (e.g., visibility, clickability, text presence).

Offers more control and flexibility compared to implicit wait.

Prevents errors caused by interacting with elements before they're ready.

Improves test reliability and accuracy.

When to use it:

When you need to wait for specific element states or conditions.

When dealing with dynamic content, AJAX requests, or unpredictable loading times.

When testing complex web applications with dynamic interactions.

Common Expected Conditions:

elementToBeClickable: Waits for an element to be visible and clickable.

visibilityOfElementLocated: Waits for an element to be visible.

presenceOfElementLocated: Waits for an element to be present in the DOM.

textToBePresentInElement: Waits for specific text to appear within an element.

titleIs: Waits for the page title to match a specific value.

Example:

WebDriver driver = new ChromeDriver();

WebDriverWait wait = new WebDriverWait(driver, 10);

// Wait for the submit button to be clickable

wait.until(ExpectedConditions.elementToBeClickable(By.id("submitButton"))).click();

## 18. Describe your experience with handling popups and alerts in Selenium.

Discuss handling strategies like switching to popup frames, dismissing buttons, sending text to input fields, and dealing with different types of alerts.

Mention preferred methods and libraries you've used for popup/alert handling.

## 19. Have you experience using different headless browsers for automation?

Discuss the advantages of headless browsers like Chrome Headless and Firefox Headless for faster execution and resource efficiency.

Mention how you've integrated headless browsers into your Selenium testing workflow.

## 20. How to handle popups and alerts in selenium?

Handling pop-ups and alerts in Selenium requires precise approaches depending on the type and purpose of the pop-up or alert. Here's a breakdown of commonly encountered scenarios and how to tackle them:

Types of Pop-ups and Alerts:

Modal (Dialog Box): Requires interaction (e.g., providing input, clicking buttons) before proceeding with the main page.

Non-Modal (Tooltip): Overlays the main page but allows interaction with underlying elements.

Alert: Presents a message (confirmation, warning, error) with buttons like "OK", "Cancel", or requiring input.

Handling Techniques:

Switching Focus:

Use driver.switchTo().alert() to switch focus to the alert or driver.switchTo().frame(frameName) for iframe-based pop-ups.

Use methods like getText(), accept(), dismiss(), or sendKeys(text) to interact with the pop-up or alert.

JavaScript Injection:

Execute JavaScript to directly manipulate the DOM and close the pop-up without switching focus. This can be helpful for non-modal pop-ups.

Window Handles:

Use getWindowHandles() to get a list of all open browser windows and switchTo().window(handle) to switch focus to the pop-up window if it has a separate handle.

Wait Strategies:

Use explicit waits like WebDriverWait with Expected Conditions like alertIsPresent() to ensure the pop-up appears before interacting with it.

Tips and Best Practices:

Identify the type of pop-up and its purpose before choosing a handling method.

Avoid generic "try-catch" blocks for pop-ups as they can mask specific errors.

Use wait strategies to avoid premature interaction with non-existent pop-ups.

Leverage tools like Selenium Grid for parallel testing with pop-ups appearing differently on different browsers.

Prioritize robust and maintainable solutions over quick fixes.

## 21. How to use a headless browser in selenium?

Key Points:

Headless browsers run without a visible UI, making them ideal for automated testing, web scraping, and performance optimization.

They offer faster execution, reduced resource consumption, and compatibility with environments without display capabilities (e.g., servers).

Setting Up:

Download WebDriver for Headless Mode:

Chrome: Download chromedriver for your OS and specify --headless option when creating the WebDriver instance.

Firefox: Use geckodriver and enable headless mode with options.setHeadless(true).

Creating a Headless WebDriver Instance: (JAVA)

System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");

ChromeOptions options = new ChromeOptions();

options.addArguments("--headless");

WebDriver driver = new ChromeDriver(options);

Using the Headless Browser:

Use the WebDriver instance as you would with a regular browser, interacting with elements and navigating pages.

Remember, you won't see visual feedback due to the headless nature.

Benefits:

Faster Execution: No rendering of UI elements, leading to quicker test runs.

Reduced Resource Usage: Less memory and CPU consumption.

Cross-Platform Compatibility: Run tests on systems without displays.

Testing Sensitive Data: Avoid visual exposure of sensitive information.

Additional Considerations:

Debugging: Use logging or browser extensions to inspect events in the headless context.

Troubleshooting: Headless mode might have unique issues; refer to documentation for troubleshooting.

Feature Compatibility: Some browser features might not work consistently in headless mode.

## 22. What is Actions class and its use in Selenium?

In Selenium WebDriver, the Actions class plays a crucial role in simulating complex user interactions with web elements through keyboard and mouse actions. It provides a fluent interface for building and performing a sequence of actions, making your automation scripts more realistic and robust.

Here's a breakdown of what the Actions class is and how you can use it:

What it is:

The Actions class is an API that enables you to create a series of user actions like clicks, drags, keyboard presses, and releases.

It builds a composite CompositeAction object containing all the specified actions.

You call the perform() method on the CompositeAction object to actually execute the sequence of actions.

Benefits of using the Actions class:

Simulates complex user interactions: It's more versatile than individual click or sendKeys methods for replicating realistic user behavior.

Improves test stability: Minimizes flakiness caused by inconsistent element interactions.

Enhances code readability: The fluent interface makes test scripts clearer and easier to maintain.

Here's how you use the Actions class:

Create an Actions object with your WebDriver instance: (JAVA)

WebDriver driver = new ChromeDriver();

Actions actions = new Actions(driver);

Build the sequence of actions using methods like:

click(WebElement element): Clicks on an element.

moveToElement(WebElement element): Moves the mouse cursor to an element.

sendKeys(CharSequence keys): Sends keyboard keys.

dragAndDrop(WebElement source, WebElement target): Drags and drops an element to another.

contextClick(WebElement element): Performs a right-click on an element.

Execute the built sequence of actions using:

actions.build().perform();

Examples:

1. Simulate right-clicking an element and selecting an option from a context menu:

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

actions.contextClick(element).moveToElement(driver.findElement(By.xpath("//\*[@id='contextMenu']/li[2]"))).click().build().perform();

1. Drag and drop an element to another location:

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

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

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

Tips and Best Practices:

Use explicit waits with actions to ensure elements are ready before interacting.

Break down complex actions into smaller steps for better control and debugging.

Leverage advanced methods like keyDown and keyUp for specific keyboard interactions.

Consider alternative frameworks like Robot Class (Java) for specific edge cases.

## 23. What is the select class in Selenium and its use?

The Select class in Selenium WebDriver provides functionalities to work with HTML <select> elements, commonly known as dropdown menus. It offers methods for selecting and deselecting options as well as retrieving information about the dropdown.

Key functionalities of the Select class:

Selecting options:

selectByVisibleText(String text): Chooses the option based on its displayed text.

selectByValue(String value): Selects the option based on its attribute value.

selectByIndex(int index): Selects the option based on its 0-based index.

Deselecting options:

deselectByIndex(int index): Deselects the option based on its index.

deselectByValue(String value): Deselects the option based on its value.

deselectAll(): Deselects all options in the dropdown.

Retrieving information:

getFirstSelectedOption(): Returns the first selected option element.

getAllSelectedOptions(): Returns a list of all selected option elements.

getOptions(): Returns a list of all option elements in the dropdown.

isMultiple(): Checks if the dropdown allows multiple selections.

Benefits of using the Select class:

Provides consistent methods for interacting with various types of dropdowns.

Simplifies selecting and deselecting options compared to manipulating DOM directly.

Improves code readability and maintainability.

Using the Select class:

Locate the <select> element using a By locator: (JAVA)

WebElement selectElement = driver.findElement(By.id("myDropdown"));

Create a Select object with the element:

Select select = new Select(selectElement);

Use the appropriate methods for selection, deselection, or information retrieval:

// Select the option with text "Option 2":

select.selectByVisibleText("Option 2");

// Get the value of the first selected option:

String selectedValue = select.getFirstSelectedOption().getAttribute("value");

// Check if multiple selections are allowed:

boolean isMultiple = select.isMultiple();

Tips and Best Practices:

Verify if the dropdown allows multiple selections before using selectAll() or deselectAll().

Handle potential exceptions like NoSuchElementException when interacting with the dropdown.

Consider using explicit waits to ensure the dropdown is loaded before interacting with it.

For dynamic dropdowns with changing options, adapt your scripts to dynamically identify and interact with the desired options.

24. Why do we use WebDriver driver, instead of ChromeDriver driver?

We generally do it this way because usually 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.?

These are extended classes of the WebDriver interface.

If your main tests and other classes define the commonly shared driver as simply WebDriver instead of specifically being tied to ChromeDriver, then the same tests can be run without change to the test code itself simply by initializing the shared driver object with a different driver extended class.

25. How to do mobile emulation tests in browser using Selenium?

Map<String, String> mobileEmulation = new HashMap<>();

mobileEmulation.put("deviceName", "Nexus 5");

ChromeOptions chromeOptions = new ChromeOptions();

chromeOptions.setExperimentalOption("mobileEmulation", mobileEmulation);

WebDriver driver = new ChromeDriver(chromeOptions);

26. When and where FindElements is better than FindElement?

**findElement**: A command to uniquely identify a web element within the web page.

Throws NoSuchElementException if the element is not found

WebElement elementName = driver.findElement(By.LocatorStrategy("LocatorValue"));

**findElements**: A command to identify a list of web elements within the web page.

Returns an empty list if no matching element is found

List<WebElement> elementName = driver.findElements(By.LocatorStrategy("LocatorValue"));

27. How to handle window based popups?

Yes, it is possible to handle Windows based pop-ups in Selenium webdriver. Sometimes on clicking a link or a button, another window gets opened. It can be a pop up with information or an advertisement.

The methods getWindowHandles and getWindowHandle are used to handle child windows. The getWindowHandles method stores all the handle ids of the opened windows in the form of Set data structure.

public class FirstAssign {

   public static void main(String[] args) {

      System.setProperty("webdriver.chrome.driver", "chromedriver");

      WebDriver driver = new ChromeDriver();

      //implicit wait

      driver.manage().timeouts().implicitlyWait(15, TimeUnit.SECONDS);

      //url launch

      driver.get("https://secure.indeed.com/account/login");

      driver.findElement(By.id("login-google-button")).click();

      //hold window handles

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

      // iterate handles

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

      //child window handle id

      String c = i.next();

      //parent window handle id

      String p = i.next();

      // child window switch

      driver.switchTo().window(c);

      System.out.println("Page title of child window: "+ driver.getTitle());

      // switch to parent window

      driver.switchTo().window(p);

      System.out.println("Page title of parent window: "+ driver.getTitle());

      //browser quit

      driver.quit();

   }

28. How to handle shadow DOM elements in selenium?

Shadow DOM is a functionality that allows the web browser to render DOM elements without putting them into the main document DOM tree. This creates a barrier between what the developer and the browser can reach; the developer cannot access the Shadow DOM the same way they would with nested elements, while the browser can render and modify that code the same way it would with nested elements

There are some bits of Shadow DOM terminology to be aware of:

Shadow host: The regular DOM node to which the Shadow DOM is attached.

Shadow tree: The DOM tree inside the Shadow DOM.

Shadow boundary: The place where the Shadow DOM ends and the regular DOM begins.

Shadow root: The root node of the Shadow tree.

What are the ways to handle shadow DOM elements?

When we try to find the Shadow DOM elements using Selenium locators, we get NoSuchElementException as it is not directly accessible to the DOM.

Selenium WebDriver’s version 4.0.0 and above, the **getShadowRoot**() method was introduced and helped locate Shadow root elements.

In case Shadow root is not found, it will throw **NoSuchShadowRootException**

**How to Use getShadowRoot() →**

All selenium locators?

Selenium is a open source library/toolset for automating web browser interactions within a web application.

Now these interactions can be created using Locators. These locators are used to identify webElements within a web page. These elements can vary from images, buttons, dropDowns, calendars, input fields etc.

Now what are different types of locators in Selenium?

## ID - A webElement having a unique ID attribute, then it can be used to identify it.

Example - driver.findElement(By.id("depart-from"));

## NAME - A webElement having a unique NAME attribute, but in most cases multiple elements can have the same name, so it’s less reliable.

Example - driver.findElement(By.name("signInButton"));

## LinkText : Accessing links using their exact link text, This makes it easy to create such locators, but these are flaky because they can fail when text changes for links used.

Example: driver.findElement(By.linkText("click here")).click();

## Partial LinkText : Accessing links using a portion of their link text is done using the By.partialLinkText() method.

Example: driver.findElement(By.partialLinkText("here")).click();

TagName : Using the HTML tagname directly to identify the webElement. It is easy to implement but can be challenging to find a unique tagname each time.

Example: driver.findElement(By.tagName (“htmltagname”));

// Click on the textbox and send value

driver.findElement(By.tagName("input")).sendKeys("JAVA");

## CSS : Used to identify webElements based on different properties such as:

ID

Class

Attribute

1. CSS selector finding webElements using ID

Majorly there are 3 ways to find elements using ID in CSS selectors:-

We can use “#” notation to select the “id” attribute of an element

We can use tagName and “id” attribute of an element

We can use syntax like - <tagname>[id=’<id value>’]

Examples:

driver.findElement(By.cssSelector("a#offers")) // Using tagName & ID

driver.findElement(By.cssSelector("#offers")); // Using only ID

driver.findElement(By.cssSelector("a[id='offers']")); // Using a fixed syntax

2) CSS selector finding webElements using ClassName

Majorly there are 3 ways to find elements using ClassName in CSS selectors:-

We can use “.” notation to select the “class” attribute of an element

We can use tagName and “id” attribute of an element

We can use syntax like - <tagname>[class=’<class value>’]

Examples:

driver.findElement(By.cssSelector("a.Navbar\_logo")); // Using tagName & className

driver.findElement(By.cssSelector(".Navbar\_logo"));; // Using only className

driver.findElement(By.cssSelector("a[class='Navbar\_logo']")); // Using a fixed syntax

3) CSS selector finding webElements using Attribute

Majorly there are 1 way to find elements using Attribute in CSS selectors:-

driver.findElement(By.cssSelector(“<tagname>[href=’<href value>’]”));

Examples:

driver.findElement(By.cssSelector("a[href='/login']"));

Note: Other attributes like placeholder, text etc. can also be used!

## XPath (XML Path Language)

It enables testers to navigate any DOM XML structure, which can be used on both HTML and XML documents

Majorly there are 2 ways to create XPath:

Absolute XPath

Relative XPath

Note 1: CSS selectors are considered to be faster to work as compared to XPath, but I didn’t find any such issues & I have primarily been using XPath selectors.

Note 2: XPath selectors work in both forward and backward tracing of elements in DOM.

Absolute XPath : Begins from the root of the HTML document and specifies the complete path to the element. It’s not as flexible and can break if the page structure changes.

Example: html/body/form/input[3]

driver.findElement(By.xpath("//body")).sendKeys(Keys.SPACE);

Relative XPath : Starts from a specific element and navigates through the DOM hierarchy to locate the desired element. It’s much easier to customize and use. Double slash is used to create relative xpath.

Example: //form/input[3]

driver.findElement(By.xpath("//body")).sendKeys(Keys.SPACE);

Single slash ‘/’ anywhere in xpath signifies to look for the element immediately inside the parent element.

Double slash ‘//’ signifies to look for any child or nested-­‐ child element inside the parent element.

Syntax: //tag[@attribute='value']

Understanding Xpath Creation:

Using Text of the element to build xpath:

Syntax: //div[@class='homepage-­‐hero']//a[text()='Enroll now']

Using Contains Keyword to find the elements:

Syntax: //tag[contains(attribute, ‘value’)]

Using Starts-­‐With Keyword  to find the elements:

Syntax: //tag[starts-­‐with(attribute, ‘value’)]

## XPath Axes:

Parent Keyword - used to retrieve the parent node of the current node selected

Syntax: xpath-­‐to-­‐some-­‐element//parent::<tag>

Example: //[@id='rt-feature']//parent::div

Preceding Sibling Keyword - will return the immediate element of the particular element

Syntax: xpath-­‐to-­‐some-­‐element//preceding-­‐sibling::<tag>

Example: //[@type='submit']//preceding::input

Following Sibling Keyword - will return the preceding element of the particular element

Syntax: xpath-­‐to-­‐some-­‐element//following-­‐sibling::<tag>

Example: //[@type='text']//following::input

Descendant Keyword - will return the descendant elements of the particular element. It returns all the webElements which are present with that particular selector.

Syntax: xpath-­‐to-­‐some-­‐element//descendant::<tag>

Example: //[@id='rt-feature']//descendant::a

All WebElement Operations

WebElements are the building blocks of web pages in Selenium. Interacting with them effectively is crucial for automating web applications. Here's a breakdown of the key WebElement operations:

**1. Accessing and Identifying Elements:**

**Finding elements:**

findElement(By byLocator): Locates a single element using a specific locator (ID, Name, XPath, CSS selector, etc.).

findElements(By byLocator): Returns a list of elements matching the chosen locator.

**Element properties:**

getAttribute(String attributeName): Retrieves the value of a specific attribute of the element.

getText(): Gets the visible text content of the element.

isEnabled(): Checks if the element is enabled for interaction.

isDisplayed(): Checks if the element is visible on the page.

getSize(): Returns the dimensions (width and height) of the element.

getLocation(): Gets the coordinates of the element on the page.

**2. Interactions with Elements:**

**Basic actions:**

click(): Clicks on the element.

sendKeys(String keys): Sends keyboard keys to the element.

clear(): Clears the text input field of the element.

**Advanced actions:**

submit(): Submits the form containing the element.

hover(WebElement element): Moves the mouse cursor over the element.

dragAndDrop(WebElement source, WebElement target): Drags an element and drops it on another.

doubleClick(): Performs a double-click on the element.

contextClick(): Performs a right-click on the element.

**3. Element Manipulation:**

**JavaScript interactions:**

You can directly execute JavaScript code to manipulate elements beyond browser-supported actions.

**Frame switching:**

switchTo().frame(WebElement frameLocator): Switches the focus to a specific frame within the page.

**Alert handling:**

switchTo().alert(): Switches focus to an alert or confirmation dialog.

accept(), dismiss(), sendKeys(String text): Interact with the alert.

**4. Waits and Synchronization:**

Implicit wait: Sets a global timeout for element searches, throwing an exception if not found within the waiting period.

Explicit wait: Waits for a specific condition (element visibility, clickability, etc.) to be met before proceeding.

Fluent wait: Provides a flexible way to define custom waiting conditions with a timeout limit.

**5. Conditional Operations:**

isSelected(): Checks if a checkbox or radio button is selected.

## How to find webelements in selenium with syntax?

finding elements is the essential first step for interacting with web pages. While several findElement methods exist, it's crucial to understand the "properties" of these methods, meaning the different ways you can specify how to locate an element. These properties are called "locators". Here's a comprehensive breakdown of the available locator types:

1. By.id:

Target elements with unique ID attributes. This is the most efficient and reliable method if IDs are unique and consistent across page updates.

Example: driver.findElement(By.id("submitButton"));

2. By.name:

Locate elements with unique name attributes. Less reliable than IDs as elements can share the same name.

Example: driver.findElement(By.name("username"));

3. By.className:

Find elements with specific class names. Useful for elements sharing the same class but having unique content or other distinguishing features.

Example: driver.findElements(By.className("error-message"));

4. By.tagName:

Target elements by their HTML tag names. A broad approach, useful for generic scenarios when other locators are unavailable.

Example: driver.findElements(By.tagName("input"));

5. By.linkText:

Locate elements by the exact text displayed on links. Can be fragile if link text changes frequently.

Example: driver.findElement(By.linkText("Click Here"));

6. By.partialLinkText:

Find elements containing the specified substring within their displayed link text. More flexible than By.linkText but less precise.

Example: driver.findElement(By.partialLinkText("Learn More"));

7. By.cssSelector:

Target elements using CSS selectors, offering powerful and flexible selection based on various attribute combinations, tag names, and class names.

Example: driver.findElement(By.cssSelector("#main-content input[type='submit']));

8. By.xpath:

Locate elements using XPath expressions. Highly flexible and powerful, allowing complex element selection based on various criteria and relationships.

Example: driver.findElement(By.xpath("//div[@class='product']/a[contains(text(), 'Buy Now')]"));

9. Composite Locators:

You can combine these locators within the same By object to build even more specific element identification strategies.

Example: driver.findElement(By.cssSelector("#myForm input[name='password'][type='password']"));

## How to use JAVAScript Executor?

JavaScript can be a powerful tool for interacting with web elements and manipulating the browser in ways not directly exposed through native WebDriver commands. Here's a breakdown of the key JavaScript interaction methods:

**1. JavaScriptExecutor interface:**

Acts as a bridge between your Selenium scripting language and the browser's JavaScript engine.

Provides two important methods:

executeScript(String script, Object... args): Executes a JavaScript string as code within the browser context, optionally with arguments.

executeAsyncScript(String script, Object... args): Similar to executeScript, but waits for the asynchronous script to finish before returning a result.

**2. Common JavaScript interactions:**

Element manipulation:

You can directly set element attributes, style properties, and inner HTML content using JavaScript within executeScript.

DOM manipulation:

Access and interact with any DOM element or API by writing custom JavaScript code within executeScript.

Custom element actions:

Trigger any user-defined JavaScript functions attached to elements on the page.

Dynamic content manipulation:

Access and interact with dynamically loaded content that might not be readily available through WebDriver commands.

**3. Examples of JavaScript interactions:**

Change the background color of an element:

String script = "document.getElementById('myElement').style.backgroundColor = 'red';";

driver.executeScript(script);

Get the inner text of an element hidden by CSS:

String script = "return document.getElementsByClassName('hidden-element')[0].innerText;";

String text = (String) driver.executeScript(script);

Click an element located dynamically after some AJAX call:

String script = "return document.querySelector('.new-button').click();";

driver.executeAsyncScript(script);

More Examples:

WebDriver driver = new ChromeDriver();

// ... (navigate to a page)

// Scroll to an element with ID "myElement"

((JavascriptExecutor) driver).executeScript("arguments[0].scrollIntoView(true);", driver.findElement(By.id("myElement")));

// Change the background color of an element

((JavascriptExecutor) driver).executeScript("arguments[0].style.backgroundColor = 'red';", driver.findElement(By.className("myClass")));

// Click a button generated dynamically

((JavascriptExecutor) driver).executeAsyncScript("arguments[0].click();", driver.findElement(By.cssSelector("button.new-button")));

## JAVASCRIPT commands for scrolling in browser console tab:

Here are different ways to scroll in the browser using JavaScript:

**1. scrollTo() Method:**

Scrolls the entire viewport to a specific position.

window.scrollTo(x-coordinate, y-coordinate);

Example:

window.scrollTo(0, 500); // Scrolls to 500 pixels from the top

**2. scrollBy() Method:**

Scrolls the viewport relative to its current position.

Syntax:

window.scrollBy(x-offset, y-offset);

Example:

window.scrollBy(0, 200); // Scrolls down 200 pixels

**3. scrollIntoView() Method (for Elements):**

Scrolls a specific element into view.

Syntax:

element.scrollIntoView(options);

**4.Element Selection:**

document.getElementById(id): Returns the element with the specified ID.

document.getElementsByClassName(className): Returns a collection of elements with the specified class name.

document.getElementsByTagName(tagName): Returns a collection of elements with the specified tag name.

**5. Element Selection:**

document.getElementById(id): Returns the element with the specified ID.

document.getElementsByClassName(className): Returns a collection of elements with the specified class name.

document.getElementsByTagName(tagName): Returns a collection of elements with the specified tag name.

## Switch use in Selenium in Detail?

In Selenium, switching is a crucial skill for automating web applications that involve multiple windows, frames, or alerts. While navigating a single page is straightforward, handling additional contexts requires understanding various switching techniques. This guide will delve into the details of switching in Selenium, covering different scenarios and approaches.

**Key Concepts:**

Window handles: Unique identifiers representing open browser windows.

Window contexts: The active frame/window where Selenium interacts with elements.

Switching methods: Techniques to change the active window context.

Scenarios Requiring Switching:

**Multiple browser windows**: Websites opening new tabs/windows (account login, online chat).

**Frames**: Content embedded within the main page (ads, comments sections).

**Alerts and prompts**: Dialog boxes requiring interaction (confirmation popups, login prompts).

Switching Techniques:

**1. Window Handling:**

getWindowHandles(): Retrieves a list of all open window handles.

switchTo().window(handle): Switches the active window context to the specified handle.

close(): Closes the current window.

quit(): Closes all open browser windows and shuts down the Selenium session.

**2. Frame Switching:**

switchTo().frame(element): Switches the active context to the frame identified by the element.

switchTo().frame(index): Switches the active context to the frame at the specified index.

switchTo().parentFrame(): Switches the active context back to the parent frame.

switchTo().defaultContent(): Switches the active context back to the main page.

**3. Handling Alerts and Prompts:**

switchTo().alert(): Switches the active context to the current alert/prompt.

accept(): Confirms the alert/prompt.

dismiss(): Cancels the alert/prompt.

sendKeys(text): Enters text into the alert/prompt input field.

**Best Practices:**

Identify switching needs early in your test script.

Use explicit waits to ensure elements or contexts are available before switching.

Store window handles to easily switch back later.

Use descriptive variable names for handles and contexts to improve code readability.

Avoid excessive switching; optimize your script for efficiency.

## Manage operation in selenium?

In Selenium, driver.manage() provides access to various WebDriver options and capabilities that can be configured to manage the browser session and improve test execution. Here's a breakdown of its key uses and properties:

**1. Timeouts:**

timeouts().implicitlyWait(timeout): Sets a global implicit wait for element searches, waiting for the specified timeout before throwing an exception if an element is not found.

timeouts().pageLoadTimeout(timeout): Sets the maximum time to wait for a page to load before throwing an exception.

timeouts().scriptTimeout(timeout): Sets the maximum time to wait for asynchronous JavaScript to execute before throwing an exception.

**2. Browser window management:**

window().maximize(): Maximizes the browser window.

window().minimize(): Minimizes the browser window.

window().fullscreen(): Toggles the browser window to fullscreen mode.

window().setSize(width, height): Resizes the browser window to the specified dimensions.

**3. Logs and performance:**

logs().get(loggingType): Retrieves logs from the browser session based on the specified logging type (e.g., browser, driver).

timeouts().setPerformancePreference(performancePreference): Sets the browser's performance preference (e.g., SPEED, QUALITY).

**4. Advanced options:**

timeouts().setNetworkConnection(connection): Simulates different network connection speeds (e.g., SLOW, FAST).

timeouts().setProxy(proxy): Sets a proxy server for the browser session.

timeouts().setAcceptSslCertificates(acceptSslCertificates): Controls whether the driver accepts invalid SSL certificates.

**5. Usage examples:**

// Set an implicit wait of 10 seconds

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

// Maximize the browser window

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

// Get logs from the browser console

List<LogEntry> logs = driver.manage().logs().get(LogType.BROWSER);

// Set the browser performance to prioritize speed

driver.manage().timeouts().setPerformancePreference(PerformancePreference.SPEED);

**Remember:**

Use driver.manage() sparingly and only configure options necessary for your specific test scenario.

Avoid using implicit waits excessively, as it can slow down test execution.

Carefully consider the impact of each option on your test environment and browser compatibility.

***More details on Driver.Manage uses:***

Methods:

timeouts().implicitlyWait(time, unit): Sets a default waiting time for elements to appear before throwing an exception.

timeouts().pageLoadTimeout(time, unit): Sets a maximum time for a page to fully load.

timeouts().setScriptTimeout(time, unit): Sets a maximum time for asynchronous scripts to execute.

window().maximize(): Maximizes the browser window.

window().fullscreen(): Toggles fullscreen mode.

window().setSize(width, height): Resizes the window.

window().getPosition(): Retrieves the window's current position.

window().setPosition(x, y): Sets the window's position.

timeouts().setPerformancePreference(preference): Sets the browser's performance preference (e.g., PerformancePreference.SPEED, PerformancePreference.QUALITY).

driver.manage().timeouts().setPerformancePreference(PerformancePreference.SPEED);

timeouts().setNetworkConnection(connection): Simulates network conditions (e.g., ConnectionType.SLOW, ConnectionType.FAST);

timeouts().setProxy(proxy): Sets a proxy server for the browser session.

timeouts().setAcceptSslCertificates(accept): Controls whether to accept invalid SSL certificates.

## Browser Options Detailed Understanding in Selenium?

Browser options in Selenium play a crucial role in customizing browser behavior during your automation scripts. They allow you to configure various settings, enabling diverse testing scenarios and enhanced control over the testing environment. Here's a detailed breakdown of their use and functionalities:

1. Purpose:

Influence how the browser launches and operates.

Modify settings like headless mode, extensions, languages, and preferences.

Set up specific testing conditions for improved realism and reliability.

2. Accessing Options:

WebDriver instances (e.g., ChromeDriver, FirefoxDriver) provide options objects specific to the browser being used.

Examples:

ChromeOptions options = new ChromeOptions();

FirefoxOptions options = new FirefoxOptions();

3. Common Options:

Headless mode: Run the browser without a visible GUI, suitable for headless testing and server-side execution.

options.setHeadless(true);

Extensions: Load and enable desired browser extensions for specific functionalities.

options.addArguments("--load-extension=/path/to/extension.crx");

Languages: Set the preferred browser language for localization testing.

options.addArguments("--lang=fr");

Preferences: Modify various browser preferences like download location, notifications, and security settings.

options.setExperimentalOption("prefs", {"download.default\_directory": "/path/to/downloads"});

4. Advanced Options:

Network conditions: Simulate network speeds and latencies for performance testing.

options.setProxy(new Proxy(Proxy.ProxyType.HTTP, new InetSocketAddress("localhost", 8080)));

Logging: Configure additional logging levels for debugging and analysis.

options.setLogLevel(LogLevel.FINE);

Capabilities: Combine browser options with capabilities for more extensive control, including platform, version, and mobile device emulation.

DesiredCapabilities capabilities = new DesiredCapabilities(); capabilities.setCapability(ChromeOptions.CAPABILITY, options);

5. Best Practices:

Only configure options necessary for your test scenario.

Avoid relying on excessive options to prevent unnecessary complexity.

Consider the impact of options on test execution time and browser compatibility.

Refer to the official Selenium documentation for detailed information on specific options and their usage.

## How to Disable Popups in Chrome browser?

Create a ChromeOptions object:

ChromeOptions options = new ChromeOptions();

Add arguments to disable popups:

options.addArguments("--disable-popup-blocking");

Optional: Disable notifications for further control:

options.addArguments("--disable-notifications");

Create the WebDriver instance with the options:

WebDriver driver = new ChromeDriver(options);

Explanation:

--disable-popup-blocking: This argument prevents Chrome from displaying any popup windows during your Selenium test.

--disable-notifications: This optional argument disables browser notifications, which can also interrupt test execution.

## How to disable location Popups in chrome browser?

Within your Selenium test script, create a ChromeOptions object.

Add the argument --disable-geolocation to disable geolocation requests:

ChromeOptions options = new ChromeOptions();

options.addArguments("--disable-geolocation");

WebDriver driver = new ChromeDriver(options);

## How to use incognito chrome browser for selenium script execution?

Create a ChromeOptions object:

ChromeOptions options = new ChromeOptions();

Add the argument to enable incognito mode:

options.addArguments("--incognito");

Create the WebDriver instance using the options:

WebDriver driver = new ChromeDriver(options);

Proceed with your Selenium test script as usual:

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

// ... perform your test actions …

***More Use of Chrome Options in Selenium Tests:***

Languages: Set preferred language:

options.addArguments("--lang=fr");

Network conditions: Simulate network speeds and latencies:

options.setProxy(new Proxy(Proxy.ProxyType.HTTP, new InetSocketAddress("localhost", 8080)));

Capabilities: Combine with DesiredCapabilities for broader control:

DesiredCapabilities capabilities = new DesiredCapabilities();

capabilities.setCapability(ChromeOptions.CAPABILITY, options);

Argument list:

options.addArguments("--disable-popup-blocking", "--disable-notifications");

# Basic Authentication in Selenium Tests:

1. Encoding Credentials in URL:

Include username and password in the URL itself:

String url = "http://username:password@example.com/protected-page";

driver.get(url);

1. Handling Authentication with JavaScript:

Use executeScript to create custom authentication logic:

Java

((JavascriptExecutor) driver).executeScript("window.prompt = function(msg, def) { return 'username:password'; };");

driver.get("[http://example.com/protected-page](https://www.google.com/url?q=https://www.google.com/url?q%3Dhttp://example.com/protected-page%26amp;sa%3DD%26amp;source%3Deditors%26amp;ust%3D1718978910038796%26amp;usg%3DAOvVaw2khnatk-Gwzs2u4P4v-VTq&sa=D&source=docs&ust=1718978910157863&usg=AOvVaw3KfOMb3MKT5xJOMpWYhM3W)");

# How to upload files using Selenium?

1. Locate the File Input Element:

Use appropriate By methods to find the file input element:

WebElement uploadElement = driver.findElement(By.id("upload-file-input"));

2. Specify the File Path:

Provide the full path to the file you want to upload:

String filePath = "C:\\path\\to\\your\\file.txt";

3. Send Keys to the Element:

Use the sendKeys method to simulate typing the file path into the input field:

uploadElement.sendKeys(filePath);

# How to Take Screenshots in Selenium?

The TakeScreenshot interface in Selenium provides powerful capabilities for capturing screenshots during your tests. Let's explore its detailed use and features in Java:

1. Capturing Screenshots:

Get the TakeScreenshot instance:

TakesScreenshot screenshot = (TakesScreenshot) driver;

Capture the screenshot:

File screenshotFile = screenshot.getScreenshotAs(OutputType.FILE);

OutputType choices:

FILE: Saves the screenshot as a PNG file.

BYTES: Returns the screenshot as a byte array.

BASE64: Returns the screenshot encoded in Base64 format.

2. Advanced Usage:

Specify location: Provide a custom path for saving the screenshot file:

File screenshotFile = new File("C:/screenshots/test\_1.png");

screenshot.getScreenshotAs(OutputType.FILE, screenshotFile);

## How to take element level screenshot in selenium?

WebElement elementLogo = driver.findElement(By.className("has-logo-image"));

                 File src = elementLogo.getScreenshotAs(OutputType.FILE);

                 File dest = new File(System.getProperty("user.dir") +    "/screenshots/elementLogo.png");

## How to take a full page screenshot using selenium?

There is a method called getFullPageScreenshotAs() for Firefox. It allows us to take a full page screenshots and store it in the specified location.

Here, instead of typecasting it to the ‘TakeScreenshot‘ interface. We have to typecast it to the ‘FirefoxDriver‘ instance. Since it works only on Firefox browser.

driver = new FirefoxDriver();

 driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));

  File src = ((FirefoxDriver)driver).getFullPageScreenshotAs(OutputType.FILE);

             FileHandler.copy(src, new File("FullPageScreenshot.png"));

  File src = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);

             FileHandler.copy(src, new File("PageScreenshot.png"));

Situational Q&A for QA Automation AutomationFramework Questions

Q1) How do you refactor an existing automation framework?

Ans: 1) Assess the current framework's weaknesses and maintenance challenges

2) Identify redundant or outdated code and refactor it to improve readability

3) Prioritise modularisation and design patterns for a more scalable framework

4) Implement version control and code review practices to ensure code quality

5) Introduce continuous integration for automated tests to detect integration issues

Q2) During test execution, you encounter a scenario where a test consistently fails due to intermittent issues, making it a "flaky test." How would you troubleshoot and address this issue to ensure test stability?

Ans: 1) Analyse test logs and identify patterns in failures

2) Review the environment setup and test data to identify inconsistencies

3) Introduce explicit waits or enhance synchronisation to handle timing issues

4) Isolate the test case to identify if it's due to application issues or test script problems

5) Collaborate with developers to address any underlying application issues causing flakiness.

Note - You can also provide answers specific to automation libraries used in organisation.

Q3) There's a critical bug discovered in the production environment. How would you prioritise creating automated tests to cover this specific scenario to prevent similar issues?

Ans: 1) Analyse the critical bug and identify test scenarios to replicate the issue

2) Prioritise automated test creation to cover affected functionality

3) Perform root cause analysis to identify similar areas prone to issues

4) Integrate automated tests into the regression suite for continuous coverage

5) Ensure collaboration between development and testing to prevent future occurrences

Q4) How would you approach automating cross-browser testing to cover various user scenarios?

Ans: 1) Utilise cross-browser testing tools like Selenium Grid, BrowserStack etc. for multi-browser testing

2) Develop test scripts that validate UI elements and functionalities across targeted browsers

3) Leverage parameterisation to run test suites across multiple browsers in parallel for efficient validation

4) Will also make sure there are no browser specific dependencies in tests

# Selenium Specific Q&A with Examples

## 1) How to handle Selenium WebDriver Exceptions?

We can handle selenium exceptions by using try catch block methods of Java.

try{

driver.findElement(by.id("button")).click();

}

catch(NoSuchElementException e){

System.out.println("Element not present");

}

## 2) There are four browser windows open and you don’t have any idea where the required element is present. What will be your approach to find that element?

– use getWindowHandles() method to get Window handles of all browser windows

– use switchTo() method to switch to each browser window using the handle id

– Find the element in each browser window and close the window if not present

## 3) How do you handle an alert pop-up in Selenium?

We can use the following methods to handle an alert in Selenium:

- dismiss()

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

- accept()

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

## 4) How do you retrieve the text displayed on an Alert?

String text = driver.switchTo().alert().getText();

## 5) How do you type text into the text box on an Alert?

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

## 6) Is Alert in Selenium an Interface or Class?

Alert is an interface in Selenium.

## 7) How do you handle frames in Selenium?

We can switch to frames by following methods:

– By Index

driver.switchTo().frame(0);

– By Name or Id

driver.switchTo().frame(“id of the element”);

– By Web Element

driver.switchTo().frame(WebElement);

## 8) Give an example of method overloading concept that you have used in Selenium?

Implicit Wait in Selenium uses method overloading as we can provide different Timestamp or TimeUnit like SECONDS, MINUTES, etc.

## 9) How do you select a value from a drop-down field and what are the different methods available?

We can select values from drop-down using methods of Select class. Following are the methods:

– selectByVisibleText

– selectByValue

– selectByIndex

Select elements = new Select(driver.findElement(By.id("button"));

elements.selectByVisibleText("Selenium");

elements.selectByIndex(1);

## 10) When your XPath is matching more than one element, how do you handle it to locate the required element?

We can use index of the element to locate it or we can use different Xpath axes methods to locate the element like Following, Ancestor, Child, Preceding or Following-sibling

## 11) How do you capture screen-shots in Selenium and what is the best place to have the screen-shot code?

//Convert web driver object to TakeScreenshot

TakesScreenshot scrShot =((TakesScreenshot)webdriver);

//Call getScreenshotAs method to create image file

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

//Move image file to new destination

File DestFile=new File(fileWithPath);

//Copy file at destination

FileUtils.copyFile(SrcFile, DestFile);

## 12) Write the code for connecting to Excel files and other operations?

XSSFWorkbook srcBook = new XSSFWorkbook("Demo.xlsx");

XSSFSheet sourceSheet = srcBook.getSheetAt(0);

int rownum=rowcounter;

XSSFRow sourceRow = sourceSheet.getRow(rownum);

XSSFCell cell1=sourceRow.getCell(0);

(Understand about Excel integration before learning the above code)

## 13) How do you read and write into a PDF file?

BufferedInputStream file = new BufferedInputStream("Path of PDF file");

PDFParser pdf = new PDFParser(file);

pdf.parse();

String text = new PDFTestStripper().getText(pdf.getPDDocument());

## 14) What are the disadvantages of Selenium?

– It supports only web applications and cannot automate desktop applications

– No default reporting mechanism

– No default object repository

– Cannot automate captcha

## 15) How do you debug your automation code when it is not working as expected?

– Add breakpoints on the lines of code where it is not working

– Run code in debugging mode

– Use different actions like F7(Step Into), F8(Step Over), F9(Step Out) to debug the problem

## 16) What are the end methods you use for verifying whether the end result is achieved by our Selenium automation scripts?

We can use different assertion methods available in different test frameworks like TestNG or Junit.

## 17) How do you clear the cookies of a browser using Selenium, before starting the execution?

driver.manage().deleteAllCookies();

## 18) How do you implement collections in your framework?

Collections can be used in a framework in situations where you have to store a large number of objects. For example, findElements() method returns a list of all matching elements.

## 19) Give a scenario where inheritance is used in your framework?

We create a Base Class in the Framework to initialize WebDriver interface, WebDriver waits, Property files, Excels, etc., in the Base Class. We extend the Base Class in other classes such as Tests and Utility Class.

## 20) Give a scenario where an interface is used in your framework?

WebDriver is an interface and when we create an instance of the driver object to use its different methods.

## 21) Write a code using JavascriptExecutor to scroll the web page?

//This will scroll the web page till the end.

js.executeScript("window.scrollTo(0, document.body.scrollHeight)");

## 22) What is the use of property files in Selenium?

Property files can be used to store the different web elements of an application or to store all the different application framework configurations.

## 23) How do you handle multiple browsers selection in Selenium?

We can select different browsers in Selenium using the TestNG framework.

## 24) What do you use for reporting in your Selenium Project?

We can use the default TestNG or Cucumber report. We can also use different reporting libraries like Extent reports.

## 25) How Cross Browser testing is handled in Selenium?

@BeforeTest

@Parameters("browser")

public void setup(String browser) throws Exception{

//Check if parameter passed from TestNG is 'firefox'

if(browser.equalsIgnoreCase("firefox")){

//create firefox instance

driver = new FirefoxDriver();

}

//Check if parameter passed as 'chrome'

else if(browser.equalsIgnoreCase("chrome")){

//set path to chromedriver.exe

//create chrome instance

driver = new ChromeDriver();

}

testng.xml:

<?xml version="1.0" encoding="UTF-8"?>

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

<suite name="TestSuite" thread-count="2" parallel="tests" >

<test name="ChromeTest">

<parameter name="browser" value="Chrome" />

<classes>

<class name="crossbrowsertests">

</class>

</classes>

</test>

<test name="FirefoxTest">

<parameter name="browser" value="Firefox" />

<classes>

<class name="crossbrowsertests">

</class>

</classes>

</test>

</suite>

## 26) What are the types of WebDriver APIs available in Selenium?

•   Firefox Driver

•   Gecko Driver

•   InternetExplorer Driver

•   Chrome Driver

•   HTMLUNIT Driver

•   Opera Driver

•   Safari Driver

•   Android Driver

•   iPhone Driver

•   EventFiringWebDriver

## 27) What is the super interface of WebDriver?

SearchContext.

## 28) . Explain the line of code Webdriver driver = new FirefoxDriver(); ?

 ‘WebDriver‘ is an interface and we are creating an object reference of type WebDriver instantiating an object of FirefoxDriver class.

## 29). We do create a reference variable ‘driver’ of type WebDriver, why?

WebDriver driver = new FirefoxDriver();

instead of creating

FirefoxDriver driver = new FirefoxDriver();

**What is the purpose of doing this?**

first we create a reference variable driver of type WebDriver then we could use the same driver variable to work with any browser of our choice such as IEDriver, SafariDriver etc.,

//FirefoxDriver driver = new FirefoxDriver();

ChromeDriver driver = new ChromeDriver();

driver.get(“http://www.google.com”);

WebDriver driver = new FirefoxDriver();

## 30) What is Page Object Model and Page Factory in Selenium?

Page Object Model in Selenium is a design pattern where web pages are represented using classes. Variables in the class can then be used to hold references to elements on the web page, and methods can be used to implement behaviors. This allows an elegant way of implementing test routines that are both readable and easier to maintain and extend in the future.

clickLoginButton();

setCredentials(user\_name, user\_password);

submitLoginForm();

Page Factory is used to initialize every WebElement variable with a reference to a corresponding element on the actual web page using configured “locators”. Annotations, such as @FindBy, can be used to define strategies for looking up elements, along with the necessary information for identifying them:

@FindBy(how=How.NAME, using="username")

private WebElement user\_name;

## 31) How do you write a locator to identify paragraph elements that are the immediate child of a div element, or the descendent of a div element?

Immediate child:

An immediate child in XPath is indicated using “/”, while on CSS, it is indicated using “>”. For example, with XPath:

//div/p

… and with CSS:

div > p

Descendent:

To find paragraph elements that are descendent to any div element (i.e. the paragraph element appears in the subtree rooted at the div element), we can use “//” in XPath, and just a whitespace in CSS:

//div//p

div p

## 32) Why do we use headless drivers? How can you visually investigate test failure when using headless drivers?

Headless drivers are typically used in continuous integration (CI) setups. Headless drivers, such as PhantomJS, provide all standard web browser functionalities, but run in the command-line. These drivers are based on command-line tools and don’t produce screen output, making them ideal for completely automated setups.

To be able to visually investigate test failures, the developer needs to implement mechanisms to capture screenshots, otherwise rely on command line output.

## 33) What will you do if there are failures in your suite execution and what is your approach?

 While executing the automation scripts, test cases may fail for several reasons. To optimize our next runs, we need to re-run only failed test cases. How to execute failed Test cases? What is the best approach? In TestNg class, we can easily re-run the test cases using two methods as explained below:

Method 1:  By using testng-failed.xml file in test-output folder.

Method 2:  By implementing TestNG IRetryAnalyzer.

## 34) What expected conditions can be used in Explicit waits?

Some of the commonly used expected conditions of an element that can be used with explicit waits are: -

* elementToBeClickable(WebElement element or By locator)
* stalenessOf(WebElement element)
* visibilityOf(WebElement element)
* visibilityOfElementLocated(By locator)
* invisibilityOfElementLocated(By locator)
* attributeContains(WebElement element, String attribute, String value)
* alertIsPresent()
* titleContains(String title)
* titleIs(String title)
* textToBePresentInElementLocated(By, String)

## 35) Tell me any difficulties you faced in developing automation scripts? Can you give any examples for any complex scenarios handled?

  Sync issue or Timeout

Example: After clicking on some button one alert should present and we have to handle it via code but due to many issues the alert might come after a few seconds, in that case, the script will fail. We need to handle this kind of scenario using an explicit wait. This is just one example like this. We have many examples which show without a smart locator we cannot build stable scripts.

Smart locators

As we all know that locators are the core part of any scripting and we need to keep on enhancing our XPath and CSS for script stability, because if XPath and CSS are not proper then chances are very high that script might fail in upcoming releases.

We should always write dynamic or custom XPath or class, which can make our script more stable.

Cross browser testing

While designing scripts we always focus on one browser and we design our script for that browser only, but when it comes to real execution of the script then we have to make sure that our script should run in all browsers which is known as Cross Browser Testing (Chrome, FF, IE at least). I had been struggling with this topic , because few locators will work in one browser but not on the other. In order to avoid failure once the script is developed we need to run them on the different browsers and analyze the result. If it is failing on another browser then we need to change locator strategy.

Pop up handling

In many applications, you will find random pop that keeps coming and their behavior is not persistent, so we also have to take care of these unwanted pop up which stops our execution.

When Code Review is not done

Many companies do not follow proper code review which can create many issues in the future.

## 36) What is the problem with Thread.Sleep in code?

1) It is a Static wait: If given a wait of 5000 Milliseconds(5 seconds) and an element just takes just 1-2 seconds to load, script will still wait for another 3 seconds which is bad as it is unnecessarily increasing the execution time. So thread.sleep() increases the execution time in cases where elements are loaded in no due time.

2) When using Thread.sleep(), we have to mention wait time in advance, there is no guarantee that the element will be displayed in that specific wait time, there may be cases when it will take more than 5 seconds to load and again the script would fail.

3) You need to write a sleep() method whenever we need to make the webdriver wait. So if you want to wait for two web elements, you need to write Thread.sleep() twice just before you locate web elements.

4) It is not good programming practice. Instead you can use implicit or explicit waits.

## 37) Common exceptions which you have faced while working with Selenium?

 Element Finding:

NoSuchElementException: "Can't find that element you're looking for!" (Trying to click a non-existent button)

ElementNotVisibleException: "Hey, that element's hiding! I can't see it." (Clicking a button hidden behind a pop-up)

StaleElementReferenceException: "Element went poof! It's not there anymore." (Interacting with a removed element)

 Window and Session:

NoSuchWindowException: "Lost in the browser maze, which window are we in?" (Switching to a closed tab)

SessionNotCreatedException: "Tried to start the browser, but it wouldn't budge." (Browser initialization failure)

Alerts and Interactions:

NoAlertPresentException: "No pop-up here, move along!" (Trying to handle a non-existent alert)

UnhandledAlertException: "Surprise! There's an alert, and I don't know what to do with it." (Unexpected alert)

ElementNotInteractableException: "That element's a tease, I can't click it!" (Element disabled or invisible)

ElementNotSelectableException: "This checkbox wants nothing to do with you." (Element not meant for selection)

 Timeouts and Network:

TimeoutException: "Waited too long, element still playing hide-and-seek." (Implicit wait times out)

WebDriverTimeoutException: "Browser took too long to do something, time's up!" (General WebDriver wait timeout)

ConnectionClosedException: "The line's gone dead, the browser disconnected." (Network issue)

 Driver and Runtime:

WebDriverException: "Uh oh, something went wrong in the WebDriver realm." (Broad exception for WebDriver issues)

UnhandledKeyEventException: "Those keystrokes went nowhere, the element doesn't like them." (Unsupported keypress)

InvalidSelectorException: "Your locator's a mess, the element doesn't understand it." (Malformed CSS or XPath selector)

 Bonus:

InsecureCertificateException: "Security alert! This website's not safe to play with." (Unsecured website connection)

ElementClickInterceptedException: "Another element stole the show, I couldn't click your target." (Click intercepted by another element)