1. WebDriver Basics

- **Q1: What is WebDriver?**
  - **A: WebDriver** is a core interface in Selenium that enables automation of browser actions. It directly interacts with browsers via dedicated drivers (like ChromeDriver or GeckoDriver) using the W3C WebDriver protocol, making it faster and more stable than older methods. It's used for actions like navigation, element interaction, and handling windows.
- **Q2: Why do we write WebDriver driver = new ChromeDriver();?**
  - **A:** This syntax is an example of **programming to an interface**. WebDriver is the interface that defines all browser interaction methods, and ChromeDriver is a class that implements this interface. By using the WebDriver interface as the variable type, you can easily switch to another browser (e.g., new FirefoxDriver()) without changing the rest of your code.
- **Q3: What is RemoteWebDriver?**
  - **A: RemoteWebDriver** is a class that allows you to run Selenium tests on a remote machine. It communicates with the remote browser driver using a protocol, making it essential for **Selenium Grid** setups or cloud-based testing services like BrowserStack or Sauce Labs. This enables distributed, cross-platform, and parallel test execution.

2. Locators and Elements

- **Q4: Difference between findElement() and findElements()?**
  - **A:** findElement() returns the **first matching element** found and throws a NoSuchElementException if no element is found. findElements() returns a **list of all matching elements** and returns an empty list if none are found, making it useful for verifying the presence of zero or more elements.
- **Q5: How to handle dynamic elements?**
  - **A:** Use flexible locators like XPath functions (contains(), starts-with()) or CSS wildcards (^, $, *). You should also implement **Explicit Waits** (WebDriverWait) to wait for the element to become stable and visible. With Selenium 4, you can also use **Relative Locators**.
- **Q6: What is SearchContext?**
  - **A:** SearchContext is the **top-level interface** in Selenium, from which both WebDriver and WebElement inherit. It defines the findElement() and findElements() methods, which is why these methods are available on both the driver object (to search the entire page) and a web element (to search within that element's context).
- **Q7: Difference between getText() and getAttribute("value")?**
  - **A:** getText() retrieves the **visible inner text** of an element, while getAttribute("value") retrieves the text from the **value attribute** of an element. This is especially important for input fields, where getText() often returns an empty string, but getAttribute("value") will return the user-entered text.

3. Waits and Synchronization

- **Q8: Types of waits in Selenium?**
  - **A:**

        - **Implicit Wait:** A global setting that tells WebDriver to poll the DOM for a specified amount of time when searching for an element. It's applied to every findElement() call.
        - **Explicit Wait:** A specific wait applied to a single element or condition. It uses WebDriverWait to wait for a specific condition to be met (e.g., element to be clickable, element to be visible).
        - **Fluent Wait:** An advanced form of explicit wait that allows you to specify a polling interval and the types of exceptions to ignore during the wait.
- **Q9: Difference between Thread.sleep() and Implicit Wait?**
    - **A:** Thread.sleep() is a Java method that forces a **fixed, unconditional pause** in the execution, which can be inefficient. An **Implicit Wait** is a Selenium-specific command that waits for a **maximum amount of time** but proceeds as soon as the element is found, making it more dynamic and efficient.

4. Page Object Model (POM) & Page Factory

- **Q10: What is POM?**
    - **A: Page Object Model (POM)** is a design pattern in test automation where each web page is represented as a separate Java class. This class contains the locators for all the elements on that page and methods that perform actions on those elements. It improves code **maintainability** and **reusability**.
- **Q11: What is Page Factory?**
    - **A: Page Factory** is a built-in Selenium class that provides an alternative way to implement the POM pattern. It uses the @FindBy annotation to initialize elements, often with **lazy loading**, which means elements are only located and loaded into memory when they are first used.

5. Advanced Interactions

- **Q12: What is the Actions class?**
    - **A:** The **Actions class** is used to perform advanced user interactions that cannot be done with simple click() or sendKeys() methods. This includes actions like **drag-and-drop**, mouse hovers, right-clicks, and keyboard combinations.
- **Q13: build().perform() vs perform()?**
    - **A:** The perform() method is used to **execute a single action** immediately. build().perform() is used to **chain multiple actions together** into a single compound action and then execute them. It's essential when a sequence of user actions (like a drag-and-drop) needs to be performed in a specific order.
- **Q14: How to handle multiple windows?**
    - **A:** When a new browser window or tab opens, it gets a unique **window handle**. To interact with it, you must first switch control using driver.switchTo().window(handle). You can get all available window handles using driver.getWindowHandles(), which returns a Set<String>. You can iterate through this set to find the new window and switch to it. Remember to close the new window and switch back to the original to continue your test.

6. Exceptions & Troubleshooting

- **Q15: Common Selenium Exceptions?**
    - **A:**

        - **NoSuchElementException**: The element could not be found using the provided locator.
        - **StaleElementReferenceException**: The element you are trying to interact with is no longer attached to the DOM (e.g., the page was reloaded or the element was dynamically removed).
        - **TimeoutException**: An Explicit Wait condition was not met within the specified time.
        - **ElementClickInterceptedException**: Another element is covering the one you are trying to click.
- **Q16: Script worked yesterday but fails today?**
    - **A:** This is often due to changes in the application's **DOM**. Common causes include: a locator being changed, an element's ID becoming dynamic, a new pop-up or modal window appearing, or a change in network latency. The solution is to debug, update your locators, add appropriate waits, and make your tests more robust.

7. Selenium 4 Features

- **Q17: Selenium 3 vs Selenium 4 – Key Differences**
    - **A:** Selenium 4 introduced significant updates, including:
        - **W3C WebDriver Standardization:** This makes cross-browser testing more consistent and eliminates the need for the older JSON Wire Protocol.
        - **Relative Locators:** New methods (above(), below(), near(), etc.) to locate elements based on their position relative to other elements.
        - **Chrome DevTools Protocol (CDP) Support:** Allows advanced browser control and access to network, performance, and console logs directly through Selenium.
- **Q18: What are Relative Locators?**
    - **A:** Relative Locators (also known as "Friendly Locators") are a new feature in Selenium 4 that allows you to find an element by its proximity to a known element. This is useful for elements that don't have stable locators but have a fixed position relative to another element. The methods include above(), below(), toLeftOf(), toRightOf(), and near().

8. TestNG

- **Q19: Common TestNG Annotations**
    - **A:**
        - **Configuration Annotations:** @BeforeSuite, @BeforeTest, @BeforeClass, @BeforeMethod (for setup logic) and @AfterMethod, @AfterClass, @AfterTest, @AfterSuite (for teardown logic).
        - **Test Annotation:** @Test marks a method as a test case.
        - **Data Annotations:** @DataProvider for data-driven testing and @Parameters to pass data from a testng.xml file.
- **Q20: What is Parallel Execution in TestNG?**
    - **A: Parallel Execution** is a feature of TestNG that allows you to run multiple test methods, classes, or test suites simultaneously. This significantly **reduces the overall test execution time** for large test suites and is configured in the testng.xml file using the parallel attribute.

9. Utilities

- **Q21: How to capture screenshots?**
    - **A:** To capture a screenshot in Selenium, you first cast the WebDriver instance to TakesScreenshot, and then call getScreenshotAs(OutputType.FILE) to save the screenshot as a file. Finally, you use FileUtils.copyFile (from the Apache Commons IO library) to copy the temporary file to a permanent location.
- **Q22: How to handle file uploads?**
    - **A:** You can handle file uploads by using the sendKeys() method on the <input type='file'> element. Instead of interacting with the operating system's file dialog, you simply send the absolute path of the file you want to upload to the element.
- **Q23: How to handle JavaScript alerts?**
    - **A:** To handle a JavaScript alert, you must first switch to the alert context using driver.switchTo().alert(). Once you have an Alert object, you can use methods like accept() (for 'OK'), dismiss() (for 'Cancel'), getText(), and sendKeys() to interact with it.

10. Advanced Questions (5 Years Experience Level)

- **Q24: How do you handle dynamic dropdowns?**
    - **A:** For dynamic dropdowns that populate as you type, you first locate the input field and use sendKeys() to enter the partial text. Then, use an **Explicit Wait** to wait for the suggestion list to appear. Finally, find the list of suggestion elements, iterate through them, and click the desired option.
- **Q25: How do you avoid StaleElementReferenceException?**
    - **A:** This exception occurs when the DOM changes. To avoid it, always **re-locate the element** right before performing an action on it. For simple cases, you can use a custom explicit wait that handles the stale exception internally, or use the ExpectedConditions.refreshed() method in Selenium 4.
- **Q26: How do you find broken links on a page?**
    - **A:** First, get a list of all <a> tags on the page. Then, iterate through this list and extract the href attribute from each tag. For each href, send an HTTP request (using a library like HttpURLConnection or Apache HttpClient) and check if the **HTTP status code is 200** (OK). Any other status code, especially 404 (Not Found), indicates a broken link.
- **Q27: Can Selenium handle Windows-based popups?**
    - **A: No, Selenium cannot handle Windows-based popups**, as it is designed only to interact with web browsers. For OS-level popups (like file explorers or security prompts), you would need to use other tools like **AutoIT**, **Sikuli**, or Java's **Robot class**.
- **Q28: How do you find hidden elements?**
    - **A:** Hidden elements often have a display: none or visibility: hidden CSS property. You can find them using standard locators, but to interact with them, you need to use **JavaScript Executor** to either make them visible or to directly perform an action on them.
- **Q29: How do you perform cross-browser testing in Selenium?**
    - **A:**
        1. **Browser Initialization:** Use a modular approach to initialize the correct driver based on a parameter (e.g., from a properties file or command line). **WebDriverManager** can simplify driver setup.
        2. **Parameterization:** Pass the browser name from an external source like testng.xml or Maven (-Dbrowser=chrome).
        3. **Parallel Execution:** Use TestNG's parallel execution feature in testng.xml to run tests on different browsers at the same time.

4. **Cloud/Grid:** For comprehensive cross-browser and cross-platform testing, use **Selenium Grid** or cloud-based services like BrowserStack, which offer a wide range of browser and OS combinations.

- **Q30: driver.get() vs driver.navigate().to() in selenium?**
    - **A:** Both methods are used to navigate to a URL. driver.get() is a direct, simple command that waits for the page to load. driver.navigate().to() is part of the Navigation interface, which also provides methods like back(), forward(), and refresh(). While both methods are functionally similar in modern Selenium, Maps().to() is often preferred when you need to use other navigation features.

TestNG

1. Basics & Annotations

- **Q30: What is TestNG, and why is it used in automation?**
    - **A: TestNG** is a powerful testing framework for Java. It's used in automation because it offers a structured way to write tests using **annotations** for setup and teardown, supports **data-driven testing** (@DataProvider), allows **parallel execution** to speed up tests, and generates detailed **reports**.
- **Q31: What are the major TestNG annotations and their execution order?**
    - **A:** The main annotations and their execution order are:
        1. @BeforeSuite
        2. @BeforeTest
        3. @BeforeClass
        4. @BeforeMethod
        5. @Test
        6. @AfterMethod
        7. @AfterClass
        8. @AfterTest
        9. @AfterSuite
- **Q32: Difference between @BeforeMethod and @BeforeTest.**
    - **A:** @BeforeMethod runs **before each @Test method** within a class. It's ideal for setups that are unique to each test case, like logging in. @BeforeTest runs **once before all the @Test methods** inside a <test> tag in testng.xml, making it suitable for broader setups like initializing a database connection.
- **Q33: Can you skip a test case in TestNG? How?**
    - **A:** Yes, you can skip a test case in several ways:
        1. Set enabled=false on the @Test annotation.
        2. Use the exclude tag in testng.xml to prevent a specific group or method from running.
        3. Dynamically skip a test at runtime by throwing a SkipException.

2. Parameterization

- **Q34: How do you pass parameters from testng.xml to your test?**
    - **A:** You define a <parameter> tag inside your <test> or <suite> in testng.xml and use the @Parameters annotation in your test method to accept the parameter. The parameter name in the XML must match the name in the annotation.
- **Q35: What is @DataProvider, and how is it different from @Parameters?**

- **A:** @DataProvider is a method that provides multiple sets of test data to a test method, enabling **data-driven testing**. @Parameters is used to get a single set of data (or parameters) from the testng.xml file. @DataProvider is more flexible and is the preferred method for running the same test with different data inputs.

3. Test Control & Grouping

- **Q36: How do you group test cases in TestNG?**
  - **A:** You can assign a test method to a group by using the groups attribute on the @Test annotation, like @Test(groups={"smoke", "regression"}).
- **Q37: What is dependsOnMethods in TestNG?**
  - **A:** The dependsOnMethods attribute forces one test method to **depend on the successful execution of another**. If the test it depends on fails or is skipped, the dependent test will also be skipped. This is useful for controlling the execution order when a test case requires a preceding setup test to pass.

Advanced Questions

Advanced WebDriver & Architecture

- **Q38: Explain WebDriver's architecture and the W3C WebDriver protocol.**
  - **A:** WebDriver operates on a **client-server model**. Your test code acts as the **client**, which sends commands (as JSON over HTTP) to a **browser-specific driver** (the server). This driver then executes the commands on the real browser. The **W3C WebDriver protocol** is the standardized set of commands that ensures consistency across different browsers.
- **Q39: What are Desired Capabilities and Chrome Options?**
  - **A: Desired Capabilities** (Selenium 3) and **Browser Options** (Selenium 4) are used to **configure browser behavior** during automation. You can use them to set things like headless mode, window size, browser-specific flags, and SSL handling preferences.

Advanced Synchronization

- **Q40: What are Custom Expected Conditions and when would you create one?**
  - **A:** A custom expected condition is a user-defined wait condition that you can use with WebDriverWait. You would create one when the built-in ExpectedConditions methods don't meet your needs, such as waiting for an element's text to contain a specific string or for a particular element to be removed from the DOM.
- **Q41: How do you handle AJAX-heavy applications effectively?**
  - **A:** The key is to use explicit waits to handle the asynchronous nature of AJAX. You can wait for specific elements to appear, for loading spinners to disappear, or for network calls to complete using Selenium 4's Chrome DevTools Protocol (CDP) support.

Advanced Page Object Model

- **Q42: What are the differences between Page Object Model and Page Factory?**
  - **A: POM** is a **design pattern**, while **Page Factory** is a **class** that helps implement the POM pattern more easily. With POM, you manually find elements using driver.findElement(). With Page Factory, you use the @FindBy annotation, and elements are initialized automatically with lazy loading, which can simplify code.

- **Q43: How would you implement lazy loading in Page Objects?**
    - **A:** A common way is to use **Page Factory's** lazy initialization with AjaxElementLocatorFactory. Alternatively, in modern Java, you can use the Supplier interface to defer the element location until it's actually needed.

Performance & Optimization

- **Q44: How do you measure and optimize test execution performance?**
    - **A:** You can optimize performance by:
        - **Parallelizing** tests to run simultaneously.
        - Using **robust locators** (e.g., ID over complex XPath).
        - Adding **strategic waits** to avoid unnecessary delays.
        - Using **performance logs** from the browser via Selenium 4's CDP.
        - Creating **modular test suites** for targeted execution (e.g., running only smoke tests).
- **Q45: What strategies do you use for handling large test suites?**
    - **A:** Use **TestNG** to:
        - **Categorize** tests into groups (smoke, regression).
        - Run tests **in parallel** to reduce total execution time.
        - Use testng.xml to select and run specific tests or groups.
        - Integrate with **CI/CD pipelines** to run tests automatically.
        - Use **cloud services** like Selenium Grid to distribute test execution.

Advanced TestNG Concepts

- **Q46: How do you implement retry logic for flaky tests?**
    - **A:** You implement retry logic by creating a custom class that implements the **IRetryAnalyzer interface**. In this class, you define the logic for retrying a failed test a certain number of times. You then apply this class to your @Test annotation.
- **Q47: Explain TestNG's factory annotation and its use cases.**
    - **A:** The @Factory annotation is used to create **multiple instances of a test class** at runtime. This is particularly useful for complex **data-driven testing** where you want to run the same set of test methods with different parameters or configurations, effectively creating a new test for each data set.

Selenium 4 Advanced Features

- **Q48: How do you use Chrome DevTools Protocol (CDP) in Selenium 4?**
    - **A:** The CDP allows you to interact with the browser's developer tools directly. You can use it to capture **performance metrics**, listen to **console logs**, intercept **network traffic**, or manipulate the browser's geolocation without using a separate tool.
- **Q49: How do you handle relative locators in complex scenarios?**
    - **A:** For complex scenarios, you can **chain multiple relative locators** together to pinpoint an element. For example, you can locate a button that is both below() a form and toRightOf() another button. This combination provides a powerful way to locate elements in a dynamic UI.

Advanced Exception Handling

- **Q50: How do you create custom exceptions for an automation framework?**
    - **A:** You create a custom exception class by extending RuntimeException or a more specific exception like ElementNotClickableException. This allows you to throw more meaningful exceptions that provide specific information about why a test failed, which can improve debugging and reporting.

CI/CD Integration

- **Q51: How do you integrate Selenium tests with CI/CD pipelines?**
    - **A:** You integrate by:
        1. Using a build tool like **Maven** or **Gradle** to manage dependencies and trigger tests.
        2. Configuring the test execution command in your CI tool's script (e.g., Jenkins, GitLab CI).
        3. Using **TestNG** to run tests in parallel.
        4. Generating and publishing test reports (like Allure or ExtentReports) within the pipeline.
        5. Using **Docker** to ensure a consistent test environment.