Selenium Interview Questions

1. Explain Selenium architecture

Selenium is made up of several components that work together to automate web browsers. The main components are:

- ***Selenium WebDriver:*** This is the core component of Selenium and is responsible for interacting with web browsers. It communicates with the browser by sending HTTP requests and receiving HTTP responses.
- ***Selenium Grid:*** This component allows you to run tests on multiple machines and browsers in parallel, which can greatly reduce the time it takes to run a large test suite.
- ***Selenium IDE:*** This is a Firefox plugin that allows you to record and play back user interactions with a web page. It can also be used to export scripts in various programming languages.

The architecture of Selenium is client-server based, where the client libraries communicate with the browser through a driver. The driver is responsible for translating the Selenium commands into a format that the browser can understand and execute. The driver is specific to a particular browser and operating system, such as the ChromeDriver for Google Chrome on Windows or the SafariDriver for Safari on macOS.

In summary, Selenium architecture consists of several components that work together to automate web browsers, which are Selenium WebDriver, Selenium Grid, Selenium IDE, and Selenium RC. The client-server based architecture communicates with the browser through a specific driver, which is responsible for translating the Selenium commands into a format that the browser can understand and execute.

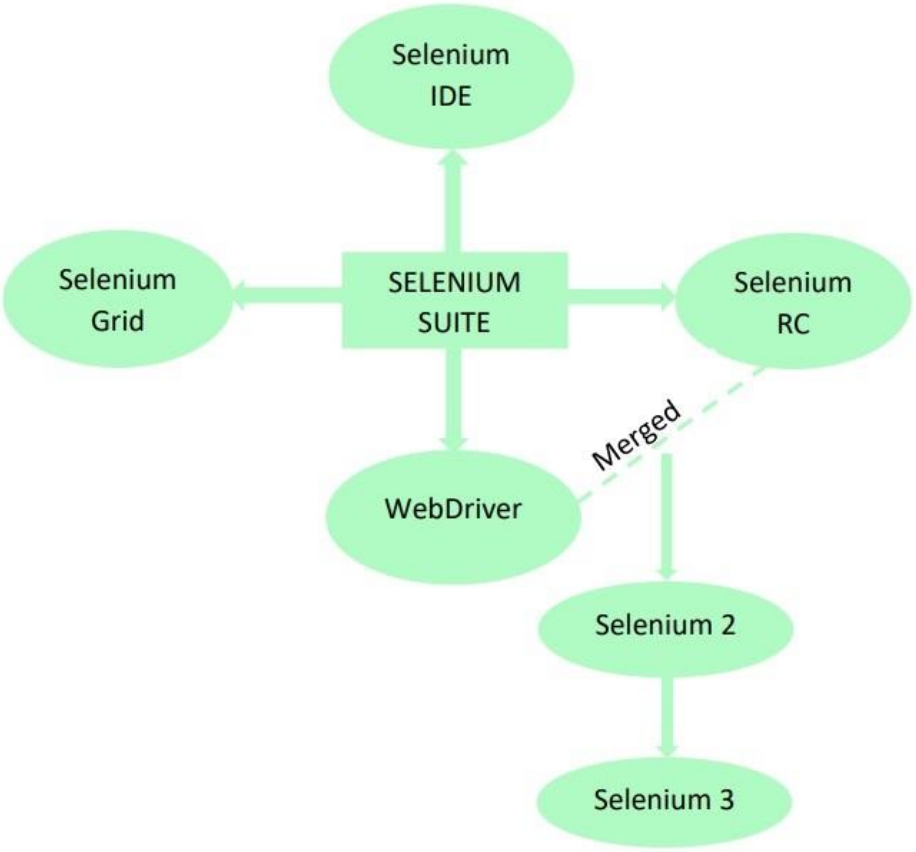

Fig. Selenium Architecture

2. How Selenium works internally?

When using Selenium with Java, the interactions with the browser are done through the Selenium WebDriver Java bindings. The WebDriver Java bindings provide a convenient API for interacting with the browser and executing Selenium commands. Here's an example of how Selenium might be used in a Java script:

- First, you need to import the Selenium WebDriver and other necessary libraries:

  ***import** org.openqa.selenium.WebDriver;*

  ***import** org.openqa.selenium.chrome.ChromeDriver;*
- Next, you will create a WebDriver object and specify the driver to use, in this case, Chrome:

```
driver = new ChromeDriver();
```

- Once the driver is created and configured, you can use it to navigate to a website, and interact with the page like a user would:

```
driver.get("http://leaftaps.com/opentaps");
driver.findElement(By.id("username")).sendKeys("DemoCSR");
driver.findElement(By.id("password")).sendKeys("crmsfa");
```

When the script is executed, the WebDriver Java bindings will send the commands to the browser driver (in this case, the ChromeDriver). The ChromeDriver will then translate these commands into a format that the Chrome browser can understand, such as the Chrome DevTools Protocol. The Chrome browser will then perform the requested actions, such as navigating to a website or clicking a button. The WebDriver Java bindings will listen for any responses from the browser and use them to update the state of the WebDriver object, such as the current URL or the page source.

In summary, Selenium WebDriver in Java interacts with the browser through the browser driver, which is responsible for translating Selenium commands into a format that the browser can understand and execute. The WebDriver Java bindings provide a convenient API for interacting with the browser and executing Selenium commands.

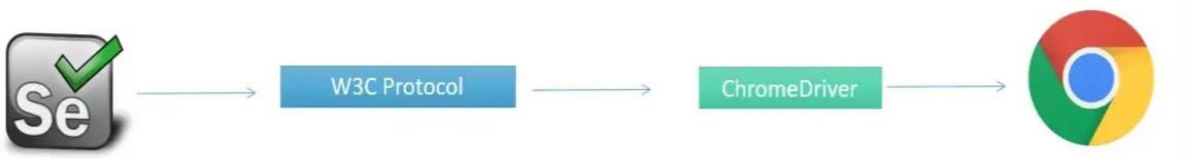

Fig. Way of Working - Selenium WebDriver

3. What are the common exceptions that you faced in selenium?

Common exceptions that may be encountered while using Selenium include:

- ***NoSuchElementException:*** This occurs when the element you are trying to interact with cannot be found on the page.
- ***StaleElementReferenceException:*** This occurs when the element you are trying to interact with is no longer attached to the DOM.
- ***ElementNotVisibleException:*** This occurs when the element you are trying to interact with is present on the page, but is not visible.
- ***TimeoutException:*** This occurs when a command does not complete in the time specified by the timeout setting.
- ***InvalidElementStateException:*** This occurs when the element is in an invalid state for the desired action, such as trying to send keys to a disabled element.
- ***WebDriverException:*** This is a general exception that can be thrown by Selenium when there is an error with the web driver.
- ***InvalidSelectorException:*** This occurs when the selector used to find an element is invalid.

4. Tell about StaleElementReferenceException and how did you handle?

StaleElementReferenceException is an exception that occurs in Selenium when the element you are trying to interact with is no longer attached to the DOM (Document Object Model). This can happen when the page is refreshed, an element is replaced with another element, or the element is deleted from the page. One common way to handle this exception is to use a try-catch block and then re-find the element in the catch block. For example:

```
try {
   element.click ();
}
catch (StaleElementReferenceException e) {
   element = driver.findElement (By.id ("element_id"));
   element.click();
```

Another approach is to use the ExpectedConditions class in Selenium to wait for the element to be present and interactable before interacting with it. For example:

```
new WebDriverWait(driver,Duration.ofSeconds(10)).
until(ExpectedConditions.refreshed(ExpectedConditions.
presenceOfElementLocated(By.xpath("element locator"))));
driver.findElement(By.xpath("element locator")).click();
```

This can help to ensure that the element is present and interactable before interacting with it, reducing the chances of a StaleElementReferenceException occurring.
Additionally, you can also wrap your code block with a while loop and check for stale element exception and retry for a few times before giving up.
It is also important to note that this exception can occur when you are working with a dynamic page where elements are constantly changing, in this case it's better to refactor your code to handle dynamic page or use explicit wait or wait for condition to make sure the element is stable before interacting with it.

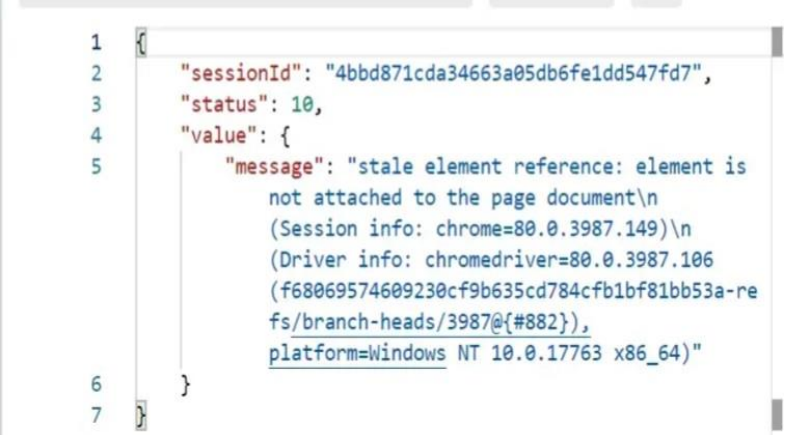

Fig. Stale Element Exception

5. Difference between get(String Url) and Navigate().to(String Url)

In Selenium, both get(String Url) and Navigate().to(String Url) are used to navigate to a specific URL. However, there is a slight difference between the two methods.

- ***get(String Url):*** This method is used to navigate to a specific URL and is a shorthand for Navigate().to(String Url). It is a simple method that loads a new web page in the current browser window.

  driver.get("http://leaftaps.com/opentaps");
- ***navigate().to(String Url):*** This method is used to navigate to a specific URL and provides more functionality than get(String Url). It allows you to navigate forward and backward in your browser history using the navigate().back() and navigate().forward() methods.

  driver.navigate().to("http://leaftaps.com/opentaps ");

In summary, both get() and Navigate().to() are used to navigate to a specific URL, but Navigate().to() provides more functionality, such as the ability to navigate forward and backward in your browser history, while get() is a shorthand for Navigate().to() and is used to load a new web page in the current browser window.

6. What is your automation failure rates and how do you manage them?

Automation failure rates can be caused by a number of factors such as:

- ***Flaky tests:*** These are tests that sometimes pass and sometimes fail for no apparent reason. This can be caused by tests that are not properly isolated from the environment or tests that rely on timing in a non-deterministic way.
- ***Incorrect test design:*** This can occur when tests are not designed to properly handle failure scenarios or when tests are not designed to be robust enough to handle changes in the application under test.
- ***Environment issues:*** This can occur when the test environment is not properly configured or when the test environment is not stable enough to run tests.

To manage automation failure rates, you should:

- ***Invest in test design and maintenance:*** This includes developing a robust test architecture, creating and maintaining test documentation, and continuously improving test quality.
- ***Use a test management tool:*** These tools can help you track and manage test failures, identify flaky tests, and provide reporting and analytics on test results.
- ***Use a continuous integration and continuous delivery pipeline:*** This can help you catch test failures early in the development process, and make it easier to fix them before they make it to production.
- Use techniques like flaky test detection and re-run, parallel execution, and smart test selection to minimize the impact of flaky tests.
- Regularly analyze test results and metrics, and use this data to identify and fix problem areas in your testing process.

- Continuously improve the test environment and test infrastructure.
- Improve test data management, test environment management and test maintenance.
- Have a continuous process improvement and monitoring mechanism in place to track and improve the test automation process.

Overall, Automation failure rates can be managed by having good test design, test maintenance, test environment, and test infrastructure and also by having a continuous process improvement mechanism in place to track and improve the automation process.

7. Explain different waits in selenium?

In Selenium, there are several types of waits that can be used to control the execution of tests:

- ***Implicit Wait:*** This type of wait is set at the WebDriver level and applies to all elements that are accessed through that driver instance. It tells the driver to poll the DOM for a certain amount of time when trying to find an element. If the element is not found within the specified time, a NoSuchElementException is thrown.

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

- ***Explicit Wait:*** This type of wait is used to tell the driver to wait for a specific condition to be met before proceeding with the next step. It uses the WebDriverWait class and the ExpectedConditions class to wait for elements to be present, visible, or interactable.

  ```
  WebElement firstResult = new
  WebDriverWait(driver,Duration.ofSeconds(10)).until(ExpectedConditions.
  elementToBeClickable(By.xpath("//a/h3")));
  ```

- ***Fluent Wait:*** This type of wait is a variation of the explicit wait. It allows you to specify the polling interval and the maximum amount of time to wait for the element to be present.

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

- ***Sleep:*** This type of wait is used to pause the execution of the script for a specific amount of time. It is not recommended to use this method, as it can cause tests to be slow and unreliable. It is only recommended for debugging purposes.

  ```
  Thread.sleep(1000);
  ```

It is also important to note that the best practice is to use explicit waits instead of implicit waits, as explicit waits give you more control over the waiting process and make your tests more predictable and robust.

8. What are the ways to click an element using WebDriver?

- ***click() method:*** This method clicks on the element and can be used on any element that is interactable.

  ```
  element = driver.findElement(By.id("element_id"));
  element.click();
  ```

- ***sendKeys(Keys.RETURN) method:*** This method simulates the pressing of the "Enter" key on the element and can be used on any element that is interactable.

  ```
  element = driver.findElement(By.id("element_id"));
  element.sendKeys(Keys.RETURN);
  ```

- ***submit() method:*** This method submits the form that the element is associated with. It can be used on any element that is a part of a form.

  ```
  element = driver.findElement(By.id("element_id"));
  element.submit();
  ```

- ***javascript method :*** This method uses JavaScript to click on the element. It can be used when an element is not interactable by normal means.

  ```
  element = driver.findElement(By.id("element_id"));
  driver.execute_script("arguments[0].click();", element);
  ```

It is important to note that when interacting with elements, it is a best practice to first use explicit wait or expected conditions to make sure the element is present, visible and interactable before interacting with it. It is also important to check if the element is still present before interacting with it.

```
if (element.is_displayed())
        element.click();
else{
      element = driver.findElement(By.id("element_id"));
       element.click();
}
```

This will make sure that the element is present before interacting with it and reduces the chances of StaleElementReferenceException.

9. How to switch window using selenium?

In Selenium, you can switch between windows using the switchTo().window() method. The switchTo().window() method takes the window handle as its argument. A window handle is a unique identifier that is assigned to each window by the browser. Here is an example of how to switch to a new window:

Get the current window handle

```
String currentWindowHandle  = driver.getWindowHandle();
system.out.println(currentWindowHandle);
```

Get all window handles

```
Set<String> windowHandles = driver.getWindowHandles();
List<String> windows = new ArrayList<String>(windowHandles);
```

Switch the control to the child window

```
driver.switchTo().window(windows.get(1));
```

Switch the control back to the parent window

```
driver.switchTo().window(windows.get(0));
```

In the above example, the script first gets the current window handle using the getWindowHandle() property. Then, it clicks on an element that opens a new window and gets all window handles using the getWindowHandles() property. Then it iterates through all the window handles and switch to the window that is not the current window. You can also switch back to the parent window using the switch_to.default_content() method.

```
driver.switch_to.default_content();
```

It is also important to note that when switching windows, it is a best practice to use explicit wait or expected conditions to make sure the new window is loaded before interacting with it.

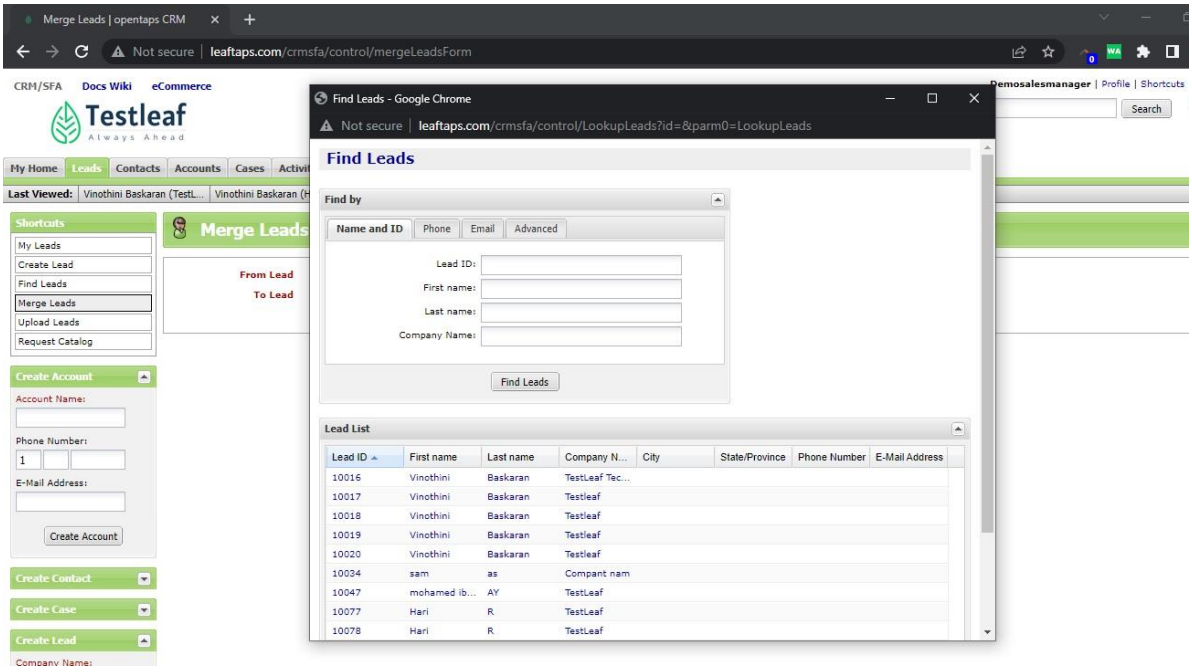

Fig. Windows Handling

10. How to work on lazyloading with selenium WebDriver with java?

To work with lazy loading using Selenium WebDriver with Java, you can use the ExpectedConditions class to wait for elements to become visible or interactable before interacting with them. For example, you can use the ExpectedConditions.visibilityOfElementLocated() method to wait for an element to be visible before performing an action on it. Here's an example:

```
WebDriver driver;
// initialize webdriver
WebDriverWait wait = new WebDriverWait(driver, 10);
WebElement element = wait.until(ExpectedConditions
.visibilityOfElementLocated(By.id("element_id")));
element.click();
```

This code snippet uses the WebDriverWait class to wait for up to 10 seconds for an element with the id "element_id" to be visible, then clicks on it.
Another way to handle lazy loading is by using javascript to scroll down the page until the element is in the viewport. You can use the Selenium's JavascriptExecutor to execute this script.

```
JavascriptExecutor js = (JavascriptExecutor) driver;
WebElement element = driver.findElement(By.id("element_id"));
js.executeScript("arguments[0].scrollIntoView();", element);
```

This script will scroll the page until the element is visible. Once the element is in view, you can interact with it as usual.

11. How to handle notifications in browser using WebDriver?

Handling browser notifications in Selenium WebDriver can be a bit tricky, as the methods for interacting with them are not built into the WebDriver API. However, there are a few ways to handle them:

- ***Using the Alert class:*** WebDriver provides an Alert class that can be used to interact with JavaScript alerts, confirmations, and prompts. To handle a notification, you can switch the focus of the WebDriver to the alert using the switchTo.alert method. Once the focus is on the alert, you can use the accept(), dismiss() or sendKeys() method to interact with it.

  ```
  Alert alert = driver.switchTo().alert();
  String text = driver.switchTo().alert().getText();
  System.out.println(text);
  Thread.sleep(5000);
  alert.accept();
  ```

- ***Using browser's web API:*** Some browsers like Chrome and Firefox, provide web API to handle notifications. For example, in Chrome, you can use chrome.notifications API to interact with notifications.

  ```
  ChromeOptions options = new ChromeOptions();
  options.addArguments("--disable-notifications");
  ChromeDriver driver =new ChromeDriver(options);
  ```

- ***Using a third-party library:*** There are also some third-party libraries such as pytest-selenium and selenium-webdriver-manager that provide additional functionality for handling notifications.

It's important to note that handling browser notifications can be different for different browsers, and the above-mentioned methods might not work for all browsers. So, you should check the documentation for the browser you're using and the version of WebDriver you're using to ensure that the method you're using is compatible.

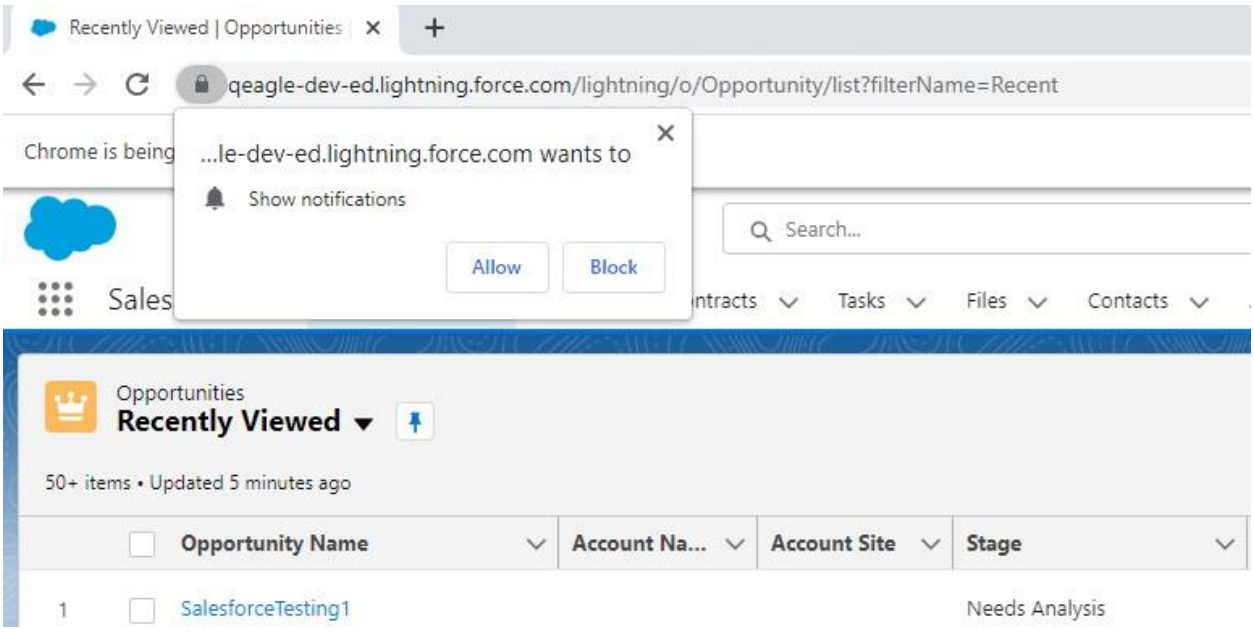

Fig. Browser Notification

12. How to find the broken link using WebDriver?

To find broken links using WebDriver, you can use the following steps:

- First, you will need to import the necessary modules, such as Selenium WebDriver and requests.
- Then, you will need to initialize the WebDriver and navigate to the website you want to check for broken links.
- Next, you will need to find all of the links on the page using the find_elements_by_tag_name method. You can store these links in a list.
- Then, you will need to iterate through the list of links and check each one for a status code of 200 (OK) using the requests module.
- If a link returns a status code other than 200, you can print the link and the status code to the console or store it in a separate list for later analysis.
- Finally, you can close the WebDriver and exit the script.

```
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Iterator;
import java.util.List;
import org.openqa.selenium.By;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class BrokenLink {
```

```
public static void main(String[] args) throws InterruptedException,
IOException {
        String homePage = "https://leafground.com/link.xhtml";
        String url = "";
        HttpURLConnection huc = null;
        int respCode = 200;
 ChromeDriver driver = new ChromeDriver();
driver.manage().window().maximize();
driver.get(homePage);
List<WebElement> links = driver.findElements(By.tagName("a"));
Iterator<WebElement> it = links.iterator();
        while(it.hasNext()){
          url = it.next().getAttribute("href");
               System.out.println(url);
        if(url == null || url.isEmpty()){
System.out.println("URL is either not configured for anchor tag or it is
empty");
               continue;
            }
     if(!url.startsWith(homePage)){
               System.out.println("URL belongs to another domain, skipping
it.");
               continue;
            }
    try {
               huc = (HttpURLConnection)(new URL(url).openConnection());
```

```
                huc.setRequestMethod("HEAD");
                huc.connect();
               respCode = huc.getResponseCode();
             if(respCode >= 400){
                    System.out.println(url+" is a broken link");
                }
             else{
                    System.out.println(url+" is a valid link");
                }
            } catch (MalformedURLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
              } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
                    }
              }
             driver.quit();
                    }
              }
        }
}
```

13. How to take screenshot for element?

TakeScreenshot is the interface that takes the snapshot for the element or a visible portion of the browser using getScreenshotAs method.

- Define the target file

```
WebElement webElement = driver.findElement(By.id("userame"));
File source = webElement.driver.getScreenshotAs(OutputType.FILE);
```

- Define the target file

```
File target = new File("./snaps/Phone.jpg");
```

- Save the source to target

```
FileUtils.copyFile(source1, target1);
```

14. How do you handle SessionNotCreatedException in WebDriver?

A "SessionNotCreatedException" can be handled in Selenium WebDriver by wrapping the instantiation of the WebDriver in a try-catch block and catching the exception. You can then print out the error message to help debug the issue.

Here is an example of how to handle "SessionNotCreatedException" in Selenium WebDriver:

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.SessionNotCreatedException;
public class Main {
  public static void main(String[] args) {
    try {
      FirefoxDriver driver = new FirefoxDriver();
      driver.get("http://www.example.com");
    } catch (SessionNotCreatedException e) {
      System.out.println(e.getMessage());
    }
  }}
```

In the above example, the FirefoxDriver is instantiated inside a try-catch block. If the browser is not able to start a new session, an exception of type "SessionNotCreatedException" will be thrown and caught. The error message is then printed out to the console. You can also check the error message in the exception and cross verify it with the error codes available in the Selenium WebDriver documentation to find out what exactly went wrong. As a best practice, you should also check if the browser executable is located in the expected location and that the browser version is compatible with the version of the WebDriver.

You can also try adding the DesiredCapabilities argument when instantiating the WebDriver. This allows you to set the browser version, platform, and other settings.

```
DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setBrowserName("firefox");
capabilities.setVersion("latest");
capabilities.setPlatform(Platform.WINDOWS);
driver = new FirefoxDriver(capabilities);
```

You can also try closing the browser if it is already open and in use before instantiating the WebDriver.

15. How to handle alerts in Selenium?

In Selenium, you can handle alerts by using the switchTo().alert() method. This method allows you to switch the focus of the webdriver to the alert dialog box. Once you have switched the focus to the alert, you can interact with it using the accept(), dismiss(), or sendKeys() methods. For example:

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

This will accept the alert dialog box.

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

This will dismiss the alert dialog box.

```
Alert alert = driver.switchTo().alert();
alert.sendKeys("Test");
alert.accept();
```

This will send keys to the alert dialog box and accept the alert.

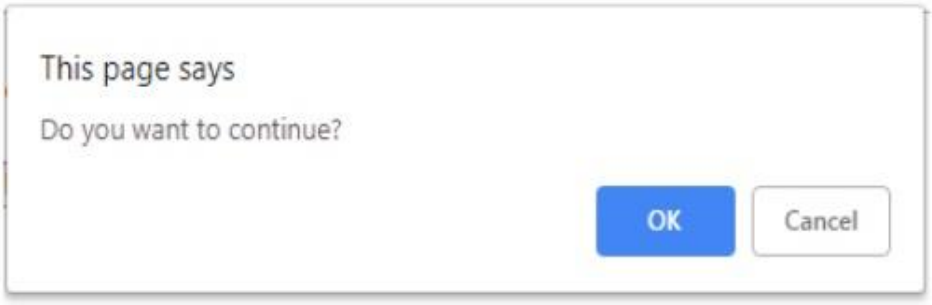

Fig. Alert Handling