**Selenium Introduction**

**What is Selenium :**

Selenium is a web automated testing tool that supports cross-browser testing across various operating systems.

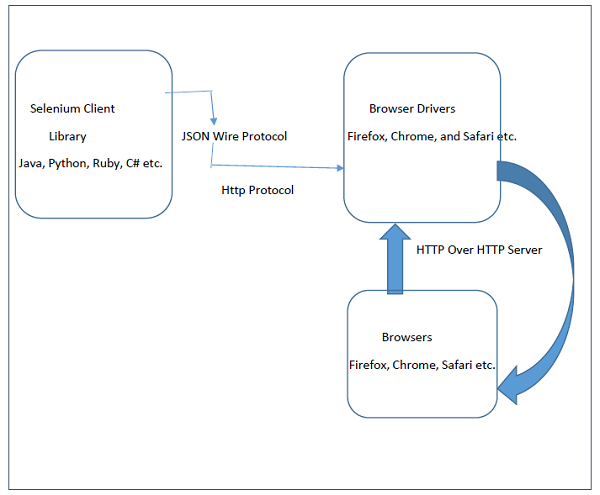
Selenium supports JAVA, Python, C#, Ruby, and JavaScript.

**What is Selenium WebDriver :**

SeleniumWebDriver is a popular open-source testing framework that allows you to automate browser actions. It supports multiple operating systems (OS) and programming languages. Here are some of the supported operating systems and languages for Selenium WebDriver:

* **Operating Systems Supported by Selenium:**
* Windows
* macOS
* Linux
* Unix
* **Programming Languages Supported by Selenium:**
* Java
* C#
* Python
* Ruby
* JavaScript (Node.js)
* **Web Browsers Supported by Selenium:**
* Google Chrome
* Mozilla Firefox
* Microsoft Edge
* Safari (limited support on macOS)
* Opera
* Internet Explorer (limited support)

**Selenium WebDriver Architecture :**



Selenium WebDriver API enables interaction between browsers and browser drivers. This architecture consists of four layers namely the Selenium Client Library, JSON Wire Protocol, Browser Drivers and Browsers.

Selenium Client Library consists of languages like Java, Ruby, Python, C# and so on. After the test cases are triggered, entire Selenium code will be converted to Json format.

JSON stands for Javascript Object Notation. It takes up the task of transferring information from the server to the client. JSON Wire Protocol is primarily responsible for transfer of data between HTTP servers. Generated Json is made available to browser drivers through http Protocol.

Each browser has a specific browser driver. Browser drivers interact with its respective browsers and execute the commands by interpreting Json which they received from the browser. As soon as the browser driver gets any instructions, they run them on the browser. Then the response is given back in the form of HTTP response.

**Let’s consider the following block of code −**

WebDriver driver = new ChromeDriver();

driver.get (“https://www.tutorialspoint.com/index.htm“);

Once we run this block of code, the entire code will be converted with the help of JSON Wire Protocol over HTTP as a URL. The converted URL will be fed to the ChromeDriver.

The browser driver utilizes HTTP server to get the request from HTTP. As the browser driver gets the URL, it passes the request to its browser via HTTP. It will trigger the event of executing the Selenium instructions on the browser.

Now if the request is that of POST, it will trigger an action on the browser. If it’s a GET request, then the response will be produced at the browser end. Finally it will be passed over HTTP to the browser driver. The browser driver will in turn send it to the UI via JSON Wire Protocol.

This sums up the overall explanation of the Selenium WebDriver Architecture.

**Installation of Selenium with Java :**

**Step 1: Install Java Development Kit (JDK):**

Java Development Kit (JDK) Installation:

Ensure that you have the Java Development Kit (JDK) installed on your system. Selenium requires Java to work. You can download the JDK from the official Oracle website (https://www.oracle.com/java/technologies/javase-jdk11-downloads.html) and follow the installation instructions for your operating system.

Integrated Development Environment (IDE):

**Step 2: Choose an Integrated Development Environment (IDE):**

Step 2: Choose an Integrated Development Environment (IDE):

Select an IDE for Java development. Some popular options include

Eclipse (<https://www.eclipse.org/downloads/>),

IntelliJ IDEA (<https://www.jetbrains.com/idea/download/>),

**Step 3: Set up a Java project:**

Create a new Java project in your chosen IDE. Set up the project structure and configure it appropriately.

**Step 4: Add Selenium Java libraries as dependencies:**

To use Selenium in your Java project, you need to add the Selenium Java libraries as dependencies. There are two common ways to do this: using a build management tool like Maven or Gradle, or manually adding the JAR files to your project.

**Option 1:** Maven Project (Recommended):

If you're using Maven as your build management tool, follow these steps:

1. Open the project's pom.xml file.
2. Inside the <dependencies> section, add the following dependency:

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>4.10.0</version>

</dependency>

Save the pom.xml file. Maven will automatically download the Selenium Java libraries for your project.

**Option 2:** Manual JAR file addition:

If you're not using a build management tool or prefer to add the JAR files manually, follow these steps:

1. Download the Selenium Java libraries (JAR files) from the official Selenium website (<https://www.selenium.dev/downloads/>). Make sure to download the latest stable version.
2. In your IDE, locate the project's library or dependencies settings.
3. Add the Selenium JAR files to the project's dependencies or classpath. The exact steps depend on the IDE you're using. Refer to your IDE's documentation for instructions on how to add external JAR files.

**Step 5: Configure WebDriver:**

Selenium requires WebDriver, which is a browser-specific driver, to interact with browsers. You need to download the appropriate WebDriver for the browser you intend to automate.

Identify the browser you want to automate (e.g., Chrome, Firefox, Safari).

Download the WebDriver executable for the chosen browser. The WebDriver downloads page can be found at the following URLs:

Chrome: ChromeDriver [(https://sites.google.com/a/chromium.org/chromedriver/]((https:/sites.google.com/a/chromium.org/chromedriver/))

Firefox: GeckoDriver (<https://github.com/mozilla/geckodriver/>)

Safari: SafariDriver (<https://webkit.org/blog/6900/webdriver-support-in-safari-10/>)

Place the WebDriver executable in a location accessible from your system's PATH environment variable.

**Step 6: Start using Selenium:**

You're now ready to write Selenium code in your Java project and execute browser automation tasks. Import the necessary Selenium classes in your Java classes, create WebDriver instances, and interact with the browsers using Selenium's API.

Here's a simple example:

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class SeleniumExample {

public static void main(String[] args) {

// Set the path to the chromedriver executable

System.setProperty("webdriver.chrome.driver", "path\_to\_chromedriver");

// Create a new instance of the ChromeDriver

WebDriver driver = new ChromeDriver();

// Perform browser automation tasks

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

// ...

// Close the browser

driver.quit();

}

}

Replace "path\_to\_chromedriver" with the actual path to the chromedriver executable on your system.

That's it! You have successfully installed Selenium for Java. You can now start automating browser interactions using Selenium in your Java project.

* **Configuration in Eclipse:**

1. **Open Eclipse and create a new Java project:**

* Go to "File" > "New" > "Java Project".
* Give your project a name and click "Finish".

1. **Set up the project build path:**

* Right-click on the project and select "Build Path" > "Configure Build Path".
* In the "Libraries" tab, click "Add External JARs".
* Browse to the location where you have saved the Selenium JAR files, select them, and click "Open".
* Click "Apply and Close" to save the changes.

1. **Create a new Java class:**

* Right-click on the project and select "New" > "Class".
* Give your class a name and click "Finish".

1. **Start writing Selenium code:**

Inside the Java class, you can import the necessary Selenium classes and start writing your Selenium code.

**WebDriver Methods :**

When working with Selenium WebDriver in Java, there are several methods and operations available to interact with web elements and perform various actions. Here are some common WebDriver methods you can use:

1. **get(String url):** Loads a new web page in the current browser window/tab.

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

1. **findElement(By locator):** Finds and returns the first web element that matches the given locator.

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

1. **findElements(By locator):** Finds and returns a list of all web elements that match the given locator.

: List<WebElement> elements = driver.findElements(By.className("myClass"));

1. **sendKeys(CharSequence... keysToSend):** Sends a sequence of keystrokes to an input field or element.

: WebElement inputField = driver.findElement(By.name("myInput"));

inputField.sendKeys("Hello, world!");

1. **click():** Performs a click action on a web element.

: WebElement button = driver.findElement(By.id("myButton"));

button.click();

1. **getText()**: Retrieves the visible text of a web element.

: WebElement element = driver.findElement(By.className("myClass"));

String text = element.getText();

1. **getAttribute(String attributeName):** Retrieves the value of the specified attribute of a web element.

: WebElement link = driver.findElement(By.linkText("Click me"));

String href = link.getAttribute("href");

1. **clear():** Clears the text or value of an input field.

: WebElement inputField = driver.findElement(By.name("myInput"));

inputField.clear();

1. **isDisplayed():** Checks if a web element is displayed on the page.

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

boolean isDisplayed = element.isDisplayed();

1. **isEnabled()**: Checks if a web element is enabled or interactive.

: WebElement button = driver.findElement(By.id("myButton"));

boolean isEnabled = button.isEnabled();

1. **isSelected()**: Checks if a web element is selected, typically used for checkboxes or radio buttons.

: WebElement checkbox = driver.findElement(By.id("myCheckbox"));

boolean isSelected = checkbox.isSelected();

These are just a few examples of commonly used methods in Selenium WebDriver. There are many more methods available to perform various operations, handle alerts, switch frames/windows, execute JavaScript, and more. Refer to the Selenium WebDriver documentation for a comprehensive list of methods and their usage: <https://www.selenium.dev/documentation/en/webdriver/web_element/>.

**How to invoke different browsers using web driver :**

To invoke different browsers using Selenium WebDriver in Java, you need to download the appropriate WebDriver executable for the desired browser and use the corresponding WebDriver implementation. Here's an example of how to invoke different browsers:

1. **Chrome:**

To use Chrome with Selenium WebDriver, you need to download ChromeDriver executable and set the system property "webdriver.chrome.driver" to the path of the ChromeDriver executable.

: import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class ChromeExample {

public static void main(String[] args) {

// Set the path to the chromedriver executable

System.setProperty("webdriver.chrome.driver", "path\_to\_chromedriver");

// Create a new instance of the ChromeDriver

WebDriver driver = new ChromeDriver();

// Perform browser automation tasks

// ...

// Close the browser

driver.quit();

}

}

1. **Firefox:**

For Firefox, you need to download GeckoDriver executable and set the system property "webdriver.gecko.driver" to the path of the GeckoDriver executable.

: import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

public class FirefoxExample {

public static void main(String[] args) {

// Set the path to the geckodriver executable

System.setProperty("webdriver.gecko.driver", "path\_to\_geckodriver");

// Create a new instance of the FirefoxDriver

WebDriver driver = new FirefoxDriver();

// Perform browser automation tasks

// ...

// Close the browser

driver.quit();

}

}

1. **Other Browsers:**

Selenium WebDriver also supports other browsers like Edge, Opera, and Internet Explorer. For each browser, you need to download the respective WebDriver executable and set the appropriate system property.

: import org.openqa.selenium.WebDriver;

import org.openqa.selenium.edge.EdgeDriver;

import org.openqa.selenium.opera.OperaDriver;

import org.openqa.selenium.ie.InternetExplorerDriver;

public class OtherBrowsersExample {

public static void main(String[] args) {

// Set the path to the MicrosoftWebDriver executable for Edge

System.setProperty("webdriver.edge.driver", "path\_to\_edgedriver");

WebDriver edgeDriver = new EdgeDriver();

// Set the path to the operadriver executable for Opera

System.setProperty("webdriver.opera.driver", "path\_to\_operadriver");

WebDriver operaDriver = new OperaDriver();

// Set the path to the IEDriverServer executable for Internet Explorer

System.setProperty("webdriver.ie.driver", "path\_to\_iedriver");

WebDriver ieDriver = new InternetExplorerDriver();

// Perform browser automation tasks

// ...

// Close the browsers

edgeDriver.quit();

operaDriver.quit();

ieDriver.quit();

}

}

**First Web driver programme :**

: import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.By;

public class WebDriverExample {

public static void main(String[] args) {

// Set the path to the chromedriver executable

System.setProperty("webdriver.chrome.driver", "path\_to\_chromedriver");

// Create a new instance of the ChromeDriver

WebDriver driver = new ChromeDriver();

// Navigate to a webpage

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

// Find an element and interact with it

WebElement searchBox = driver.findElement(By.name("q"));

searchBox.sendKeys("Selenium WebDriver");

searchBox.submit();

// Print the page title

String pageTitle = driver.getTitle();

System.out.println("Page Title: " + pageTitle);

// Close the browser

driver.quit();

}

}

**Note** that you may need to adjust the wait time or use more advanced techniques for waiting until elements are visible or interactions are complete.

* **Locators**

In Selenium, locators are used to identify and locate web elements on a web page. Locators provide a way to interact with specific elements for actions such as clicking, inputting text, or retrieving information. Selenium supports various types of locators to uniquely identify elements. Here are the commonly used locators in Selenium:

Types of Locators :

1. **ID:**

Locates elements by their unique ID attribute.

**Example:** driver.findElement(By.id("elementId"));

1. **Name:**

Locates elements by their name attribute.

**Example:** driver.findElement(By.name("elementName"));

1. **Class Name:**

Locates elements by their CSS class name.

**Example:** driver.findElement(By.className("className"));

1. **Tag Name:**

Locates elements by their HTML tag name.

**Example:** driver.findElement(By.tagName("tagName"));

1. **Link Text:**

Locates anchor elements ( <a> ) by the exact text displayed within them.

**Example:** driver.findElement(By.linkText("linkText"));

1. **Partial Link Text:**

Locates anchor elements ( <a> ) by a partial match of the text displayed within them.

**Example:** driver.findElement(By.partialLinkText("partialLinkText"));

1. **CSS Selector:**

Locates elements using CSS selectors.

**Example:** driver.findElement(By.cssSelector("cssSelector"));

1. **XPath:**

Locates elements using XPath expressions.

**Example:** driver.findElement(By.xpath("xpathExpression"));

Note: You can use findElements() instead of findElement() to locate multiple elements that match a locator.

**Syntaxes of locators:**

// Using ID locator

WebElement elementById = driver.findElement(By.id("elementId"));

// Using Name locator

WebElement elementByName = driver.findElement(By.name("elementName"));

// Using Class Name locator

WebElement elementById = driver.findElement(By.className("class name));

// Using Tag Name locator

WebElement elementById = driver.findElement(By.tagName("tag name"))

// Using Link Text locator

WebElement elementById = driver.findElement(By.LinkText("link text"));

// Using Partial Link Text locator

WebElement elementById = driver.findElement(By.partialLinkText("partiallink"))

// Using CSS Selector locator

WebElement elementById = driver.findElement(By.cssSelector("css"));

// Using Xpath locator

WebElement elementById = driver.findElement(By.xpath("xpath"));

**Different plugins for locators**

There are various plugins available for different Integrated Development Environments (IDEs) that can assist in locating web elements using different locators in Selenium. Here are a few popular plugins:

1. **Selenium IDE (for Chrome and Firefox):**

* Selenium IDE is a browser plugin that provides a record-and-playback functionality for Selenium tests.
* It allows you to record interactions with a web page and generates Selenium code using locators.
* Website: <https://www.selenium.dev/selenium-ide/>

1. **Selenium WebDriver (for Eclipse):**

* Selenium WebDriver is a plugin for Eclipse IDE that provides support for Selenium testing.
* It offers features like code completion, syntax highlighting, and easy integration with WebDriver.
* Installation: You can install it through the Eclipse Marketplace by searching for "Selenium WebDriver" or by downloading it from the Selenium website.
* Website: <https://www.selenium.dev/downloads/>

1. **Firebug and Firepath (for Firefox):**

* Firebug and Firepath are Firefox extensions that assist in inspecting and locating elements on a web page.
* Firebug helps inspect the HTML structure and CSS properties of elements.
* Firepath is an add-on to Firebug that aids in generating XPath or CSS selectors for elements.
* Firebug: https://getfirebug.com/
* Firepath: <https://addons.mozilla.org/en-US/firefox/addon/firepath/>

1. **ChroPath (for Chrome):**

* ChroPath is a Chrome extension that helps generate XPath and CSS selectors for web elements.
* It provides an intuitive interface to inspect elements and automatically generates locators.
* Website: <https://chrome.google.com/webstore/detail/chropath/ljngjbnaijcbncmcnjfhigebomdlkcjo>

**Practical Execution :**

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

public class SimpleExample {

public static void main(String[] args) {

// Set the path to the chromedriver executable

System.setProperty("webdriver.chrome.driver", "path\_to\_chromedriver");

// Create a new instance of the ChromeDriver

WebDriver driver = new ChromeDriver();

// Open a webpage

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

// Locate the page heading by tag name and retrieve its text

WebElement heading = driver.findElement(By.tagName("h1"));

String headingText = heading.getText();

System.out.println("Page Heading: " + headingText);

// Locate the search input field by name and enter text

WebElement searchInput = driver.findElement(By.name("q"));

searchInput.sendKeys("Selenium WebDriver");

// Locate the search button by CSS selector and click it

WebElementSearchButton=driver.findElement(By.cssSelector("input[type='submit']"));

searchButton.click();

// Get the current URL and print it

String currentURL = driver.getCurrentUrl();

System.out.println("Current URL: " + currentURL);

// Close the browser

driver.quit();

}

}

**Synchronization :**

Synchronization is an important concept in web UI automation that ensures the test script waits for certain conditions to be met before proceeding further. It helps in handling timing issues and ensuring the web application is in a stable state before interacting with elements.

**Different Types Of Waits :**

**Implicit Wait:**

* Implicit wait instructs the WebDriver to wait for a certain amount of time before throwing an exception if an element is not immediately found.
* Set the implicit wait once at the beginning of your script using the manage().timeouts().implicitlyWait() method.
* **Example**: driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

**Explicit Wait:**

* Explicit wait allows you to wait for a specific condition to occur before proceeding.
* You can specify the expected condition using the ExpectedConditions class and wait for it using the WebDriverWait class.
* **Example:** Wait for an element to be visible before performing an action.

WebDriverWait wait = new WebDriverWait(driver, 10);

WebElementelement=wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("elementId")));

**Fluent Wait:**

* Fluent wait is a more flexible form of explicit wait that allows you to define custom polling intervals and ignore specific exceptions during the wait.
* It is useful when waiting for elements with dynamic loading or when handling intermittent synchronization issues.
* **Example:** Wait for an element to be clickable with a custom polling interval of 2 seconds.

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

.withTimeout(Duration.ofSeconds(10))

.pollingEvery(Duration.ofSeconds(2))

.ignoring(NoSuchElementException.class);

WebElementelement=wait.until(ExpectedConditions.elementToBeClickable(By.id("elementId")));

Practical Execution :

**Implicit Wait:**

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

import java.util.concurrent.TimeUnit;

public class ImplicitWaitExample {

public static void main(String[] args) {

// Set system property for ChromeDriver

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

// Initialize ChromeDriver instance

WebDriver driver = new ChromeDriver();

// Set implicit wait

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

// Launch the web application

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

// Find the element using implicit wait

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

// Perform actions on the element

element.click();

// Close the browser

driver.quit();

}

}

**Explicit Wait:**

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

import org.openqa.selenium.support.ui.ExpectedConditions;

import org.openqa.selenium.support.ui.WebDriverWait;

public class ExplicitWaitExample {

public static void main(String[] args) {

// Set system property for ChromeDriver

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

// Initialize ChromeDriver instance

WebDriver driver = new ChromeDriver();

// Launch the web application

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

// Explicit wait: Wait for the element to be clickable

WebDriverWait wait = new WebDriverWait(driver, 10);

WebElementelement=wait.until(ExpectedConditions.elementToBeClickable(By.id("elementId")));

// Perform actions on the element

element.click();

// Close the browser

driver.quit();

}

}

**Fluent Wait:**

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

import org.openqa.selenium.support.ui.FluentWait;

import org.openqa.selenium.support.ui.Wait;

import org.openqa.selenium.support.ui.WebDriverWait;

import java.time.Duration;

public class FluentWaitExample {

public static void main(String[] args) {

// Set system property for ChromeDriver

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

// Initialize ChromeDriver instance

WebDriver driver = new ChromeDriver();

// Launch the web application

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

// Fluent wait: Wait for the element to be clickable

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

.withTimeout(Duration.ofSeconds(10))

.pollingEvery(Duration.ofSeconds(2))

.ignoring(NoSuchElementException.class);

WebElementelement=wait.until(driver>driver.findElement(By.id("elementId")));

// Perform actions on the element

element.click();

// Close the browser

driver.quit();

}

}

**Handling of Drop down**

Handling dropdowns or select elements in Selenium WebDriver involves selecting an option from the dropdown menu. Here's an example of how to handle dropdowns using Selenium WebDriver with Java:

1. Locate the Dropdown Element:

* Identify the dropdown element using appropriate locators such as ID, name, class, or XPath.

Example:

WebElement dropdown = driver.findElement(By.id("dropdownId"));

1. Create an Instance of the Select Class:

Create an instance of the Select class by passing the dropdown element as a parameter.

Example:

Select select = new Select(dropdown);

1. Select an Option by Visible Text:

* Use the selectByVisibleText() method to select an option from the dropdown by its visible text.

Example:

select.selectByVisibleText("Option 1");

1. Select an Option by Value:

Use the selectByValue() method to select an option from the dropdown by its value attribute.

Example:

select.selectByValue("option1Value");

1. Select an Option by Index:

Use the selectByIndex() method to select an option from the dropdown by its index (starting from 0).

Example:

select.selectByIndex(0); // Selects the first option

**Handling of radio button**

Handling radio buttons in Selenium WebDriver involves selecting a radio button option from a group of options. Here's an example of how to handle radio buttons using Selenium WebDriver with Java:

1. Locate the Radio Button Elements:

* Identify the radio button elements using appropriate locators such as ID, name, class, or XPath.
* It's important to note that radio buttons are typically grouped together using the same name attribute value.

Example:

WebElement radioButton1 = driver.findElement(By.id("radioButton1Id"));

WebElement radioButton2 = driver.findElement(By.id("radioButton2Id"));

1. Select a Radio Button:

* Use the click() method to select the desired radio button option.

Example:

radioButton1.click(); // Selects the first radio button

1. Check if a Radio Button is Selected:

* Use the isSelected() method to check if a radio button is selected.

Example:

boolean radioButton1Selected = radioButton1.isSelected();

1. Select a Radio Button from a Group:

* If there are multiple radio buttons in a group and you want to select a specific one, you need to locate each radio button individually and click on the desired option.

Example:

WebElement radioButton1 = driver.findElement(By.id("radioButton1Id"));

WebElement radioButton2 = driver.findElement(By.id("radioButton2Id"));

if (!radioButton1.isSelected()) {

radioButton1.click();

}

These are the basic steps to handle radio buttons in Selenium WebDriver.

**Handling of check box**

Handling checkboxes in Selenium WebDriver involves selecting or deselecting checkboxes based on your test requirements. Here's an example of how to handle checkboxes using Selenium WebDriver with Java:

1. Locate the Checkbox Element:

* Identify the checkbox element using appropriate locators such as ID, name, class, or XPath.

Example:

WebElement checkbox = driver.findElement(By.id("checkboxId"));

1. Select or Deselect a Checkbox:

* Use the click() method to select or deselect the checkbox based on its current state.
* If the checkbox is not selected, clicking it will select it. If it is already selected, clicking it will deselect it.

Example:

checkbox.click(); // Selects or deselects the checkbox

1. Check if a Checkbox is Selected:

* Use the isSelected() method to check if a checkbox is selected.

Example:

boolean checkboxSelected = checkbox.isSelected();

1. Select or Deselect Multiple Checkboxes:

* If there are multiple checkboxes and you want to select or deselect multiple options, locate each checkbox element individually and perform the desired action.

Example:

WebElement checkbox1 = driver.findElement(By.id("checkbox1Id"));

WebElement checkbox2 = driver.findElement(By.id("checkbox2Id"));

checkbox1.click(); // Selects checkbox1

checkbox2.click(); // Selects checkbox2

These are the basic steps to handle checkboxes in Selenium WebDriver.

**Handling of pop up**

Handling pop-ups or alert boxes in Selenium WebDriver involves interacting with dialog boxes that appear during the execution of a web application. Here's an example of how to handle pop-ups using Selenium WebDriver with Java:

1. Switch to the Alert:

Use the switchTo().alert() method to switch the driver's focus to the alert dialog.

Example:

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

1. Accept or Dismiss the Alert:

* Use the accept() method to accept or click the "OK" button on the alert.
* Use the dismiss() method to dismiss or click the "Cancel" button on the alert.

Example:

alert.accept(); // Clicks the "OK" button

// OR

alert.dismiss(); // Clicks the "Cancel" button

1. Get the Text of the Alert:

* Use the getText() method to retrieve the text displayed on the alert.

Example:

String alertText = alert.getText();

1. Enter Text into an Alert (if applicable):

* If the alert has an input field and requires user input, use the sendKeys() method to enter text into the input field.

Example:

alert.sendKeys("Text to enter");

1. Switch Back to the Main Window or Default Content:

* If the pop-up opens in a new window or frame, use the appropriate methods to switch back to the main window or default content.

Example:

driver.switchTo().defaultContent(); // Switch back to the default content

These are the basic steps to handle pop-ups in Selenium WebDriver. By using the switchTo().alert() method, you can interact with alert dialogs, accept or dismiss them, retrieve the text, and perform actions if an input field is available.

**Importance of assertions:Assertion commands**

Assertions play a crucial role in automated testing as they help verify expected behavior and ensure the correctness of the application under test. Here are the key reasons why assertions are important in test automation:

* **Verification of Expected Results:**

Assertions allow you to compare the actual results of an action or operation with the expected results. They provide a mechanism to validate that the application is functioning as intended. If the actual result matches the expected result, the assertion passes, indicating that the test is successful. If the results don't match, the assertion fails, indicating an issue or a bug in the application.

* **Early Detection of Issues:**

Assertions enable early detection of issues and failures in the application. By incorporating assertions throughout your test scripts, you can identify problems as soon as they occur during test execution. This helps in identifying and debugging issues at an early stage, reducing the time and effort required for troubleshooting.

* **Improved Test Coverage:**

Assertions help in achieving better test coverage by ensuring that the critical functionality of the application is tested thoroughly. By asserting specific conditions, you can verify that the application behaves correctly under different scenarios and inputs.

* **Documentation of Expected Behavior:**

Assertions act as documentation for the expected behavior of the application. By including assertions in your test scripts, you are explicitly stating what behavior or outcome you expect from the application. This makes your test scripts more understandable and maintainable, both for yourself and other team members.

* **Facilitates Debugging:**

When an assertion fails, it provides valuable information about the failure, such as the expected and actual values. This information assists in debugging and narrowing down the root cause of the issue, making it easier to fix the problem.

**Common Assertion Commands in Java:**

assertEquals(): Compares two values for equality.

assertNotEquals(): Verifies that two values are not equal.

assertTrue(): Asserts that a condition or expression is true.

assertFalse(): Asserts that a condition or expression is false.

assertNull(): Verifies that a value is null.

assertNotNull(): Verifies that a value is not null.

Example:

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculatorTest {

@Test

public void testAddition() {

Calculator calculator = new Calculator();

int result = calculator.add(2, 3);

assertEquals(5, result); // Verifies that the actual result is equal to the expected result (5)

}

}

**Handling of interactions:**

Handling mouse actions and keyboard actions in Selenium WebDriver involves using the Actions class to simulate user interactions with the web application. Here are examples of mouse and keyboard actions using Selenium WebDriver with Java:

**Mouse Actions:**

1. **Moving the Mouse to an Element:**

* Locate the target element using appropriate locators.
* Create an instance of the Actions class.
* Use the moveToElement() method to move the mouse to the element.
* Perform the action using the perform() method.

Example :

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

Actions actions = new Actions(driver);

actions.moveToElement(element).perform();

1. **Clicking and Holding on an Element:**

* Locate the target element using appropriate locators.
* Create an instance of the Actions class.
* Use the clickAndHold() method to click and hold the mouse button on the element.
* Perform the action using the perform() method.

Example :

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

Actions actions = new Actions(driver);

actions.clickAndHold(element).perform();

1. **Releasing the Mouse Button:**

* Create an instance of the Actions class.
* Use the release() method to release the mouse button.
* Perform the action using the perform() method.

Example:

Actions actions = new Actions(driver);

actions.release().perform();

**Keyboard Actions:**

1. **Sending Keys to an Element:**

* Locate the target element using appropriate locators.
* Use the sendKeys() method to send keys to the element.

Example:

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

element.sendKeys("Hello, World!");

1. **Performing Keyboard Actions:**

* Create an instance of the Actions class.
* Use methods such as keyDown(), keyUp(), and sendKeys() to perform keyboard actions.

Example:

Actions actions = new Actions(driver);

actions.keyDown(Keys.CONTROL).sendKeys("a").keyUp(Keys.CONTROL).perform();

These are some of the common mouse and keyboard actions that can be performed using Selenium WebDriver. By utilizing the Actions class and its methods, you can simulate various user interactions with the web application during your automation tests.

**Handling of multiple window**

Handling multiple windows in Selenium WebDriver involves switching between different browser windows or tabs during the execution of your tests. Here's an example of how to handle multiple windows using Selenium WebDriver with Java:

1. **Get the Window Handles:**

* Use the getWindowHandles() method to get a set of window handles currently open by the WebDriver.
* Store the window handles in a set or list for later use.

Example:

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

1. **Switch to a Specific Window:**

* Use the switchTo().window() method to switch the WebDriver's focus to a specific window using its handle.
* Provide the handle of the desired window as the argument to the switchTo().window() method.

Example:

String windowHandle = "windowHandleValue";

driver.switchTo().window(windowHandle);

1. **Perform Actions in the Current Window:**

* Once you have switched to a specific window, you can perform actions in that window using the WebDriver instance.
* Find elements and interact with them using appropriate locators and methods.

Example:

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

element.click();

1. **Close a Window:**

Use the close() method to close the current window that the WebDriver is focused on.

Example:

driver.close();

1. **Switch Back to the Main Window:**

* If you need to switch back to the main window or the original window from where you started, use the switchTo().window() method again.
* Provide the handle of the main window as the argument to the switchTo().window() method.

Example:

String mainWindowHandle = "mainWindowHandleValue";

driver.switchTo().window(mainWindowHandle);

These are the basic steps to handle multiple windows in Selenium WebDriver. You can use the getWindowHandles() method to get all the window handles, switch between windows using switchTo().window(), and perform actions in each window. Remember to store the handles of the main window or any other windows you want to switch back to.

**Handling of IFrame**

Handling iframes (inline frames) in Selenium WebDriver involves switching the driver's focus to the iframe element so that you can interact with the elements inside it. Here's an example of how to handle iframes using Selenium WebDriver with Java:

1. Switch to the Iframe:

* Identify the iframe element using appropriate locators such as ID, name, or index.
* Use the switchTo().frame() method to switch the WebDriver's focus to the iframe.
* Provide the iframe element as the argument to the switchTo().frame() method.

Example:

// Switch by ID or name

WebElement iframe = driver.findElement(By.id("iframeId"));

driver.switchTo().frame(iframe);

// Switch by index

driver.switchTo().frame(0); // Switch to the first iframe on the page

1. Perform Actions Inside the Iframe:

* Once the WebDriver is switched to the iframe, you can perform actions on the elements inside it.
* Find elements and interact with them using appropriate locators and methods.

Example:

WebElementelementInsideIframe=driver.findElement(By.id("elementInsideIframeId"));

elementInsideIframe.click();

1. Switch Back to the Main Frame:

* If you need to switch back to the main frame or the default content of the page, use the switchTo().defaultContent() method.

Example:

driver.switchTo().defaultContent();

1. Nested Iframes:

* If you have nested iframes, you need to switch to each iframe in a hierarchical manner.
* Switch to the parent iframe, perform actions, then switch to the child iframe, and so on.

Example:

// Switch to parent iframe

driver.switchTo().frame("parentIframe");

// Switch to child iframe

WebElement childIframe = driver.findElement(By.id("childIframeId"));

driver.switchTo().frame(childIframe);

// Perform actions inside the child iframe

WebElementelementInsideChildIframe=driver.findElement(By.id("elementInsideChildIframeId"));

elementInsideChildIframe.click();

// Switch back to the parent iframe

driver.switchTo().parentFrame();

Remember to switch back to the default content (switchTo().defaultContent()) or parent frame (switchTo().parentFrame()) when you finish interacting with iframes to ensure subsequent actions are performed on the main content of the page.