Part 3- The Complete Selenium WebDriver & SearchContext Method Cheat Sheet!

✅ 25. Window Handles & Switching Between Multiple Windows 🪟

- **Declared in:** `WebDriver` interface
- **Implemented by:** `RemoteWebDriver` class
- **How to use:** Use `getWindowHandles()` to retrieve handles for all open windows and `switchTo().window()` to switch between them.
- **Arguments:** Takes the window handle (a `String`) as an argument to switch between windows.
- **Returns:** `void`.
- **Purpose:** This allows you to handle and switch between multiple browser windows or tabs in Selenium.

i. Switching Between Windows

```
// Store the current window handle
String mainWindowHandle = driver.getWindowHandle();

// Iterate through all open windows
for (String windowHandle : driver.getWindowHandles()) {
    driver.switchTo().window(windowHandle);  // Switch to the new window
    if (!windowHandle.equals(mainWindowHandle)) {
        // Close the new window if needed
        driver.close();
    }
}

// Switch back to the main window
driver.switchTo().window(mainWindowHandle);
```

ii. Handle New Window with `driver.switchTo().newWindow()`

```
// Open a new tab
driver.switchTo().newWindow(WindowType.TAB);

// Open a new window
driver.switchTo().newWindow(WindowType.WINDOW);
```

✅ 26. Action Class - Performing Complex User Interactions

- **Declared in:** `Actions` class
- **Implemented by:** `Actions` class
- **How to use:** Use `Actions` to simulate complex user interactions like mouse movements, key presses, drag and drop, etc.
- **Arguments:** Takes `WebDriver` object and the actions you want to perform.
- **Returns:** `Actions` object that can be chained with other actions.
- **Purpose:** Enables the automation of complex user interactions such as hover, double-click, drag-and-drop, etc.

i. Performing a Hover Action

```
Actions actions = new Actions(driver);
WebElement element = driver.findElement(By.id("hoverElement"));
actions.moveToElement(element).perform();  // Hover over the element
```

ii. Performing a Double-Click Action

```
Actions actions = new Actions(driver);
WebElement element = driver.findElement(By.id("doubleClickElement"));
actions.doubleClick(element).perform();  // Double-click on the element
```

iii. Drag and Drop Action

```
Actions actions = new Actions(driver);
WebElement source = driver.findElement(By.id("source"));
```

```
WebElement target = driver.findElement(By.id("target"));
actions.dragAndDrop(source, target).perform();  // Drag source to target
```

✅ 27. Fluent Wait ⏳

- **Declared in:** `FluentWait` class
- **Implemented by:** `FluentWait` class
- **How to use:** FluentWait is a more flexible waiting mechanism that allows you to define the frequency of checking for a condition.
- **Arguments:** Takes `WebDriver` and `Duration` for timeouts and polling frequency.
- **Returns:** `FluentWait` object.
- **Purpose:** FluentWait can be used to define waiting conditions with the ability to configure how often the condition should be checked.

```
FluentWait<WebDriver> wait = new FluentWait<>(driver)
    .withTimeout(Duration.ofSeconds(30))  // Max time to wait
    .pollingEvery(Duration.ofSeconds(5))  // Poll every 5 seconds
    .ignoring(NoSuchElementException.class);  // Ignore specific exceptions

// Wait until a specific condition is true
WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("someElement")));
```

✅ 28. JavaScriptExecutor - Executing JavaScript in WebDriver 🖥️

- **Declared in:** `JavascriptExecutor` interface
- **Implemented by:** `RemoteWebDriver` class
- **How to use:** Use `JavascriptExecutor` to execute custom JavaScript code in the context of the current page.
- **Arguments:** Takes a `String` containing JavaScript code, and optionally an array of arguments.
- **Returns:** Object – result of the JavaScript execution.

- **Purpose:** Useful for scenarios where WebDriver's standard actions might not be enough, and JavaScript can be used to execute more complex actions.

i. Executing JavaScript to Click an Element

```
JavascriptExecutor js = (JavascriptExecutor) driver;
WebElement element = driver.findElement(By.id("submitButton"));
js.executeScript("arguments[0].click();", element);  // Click using JavaScrip
t
```

ii. Executing JavaScript to Scroll the Page

```
JavascriptExecutor js = (JavascriptExecutor) driver;
js.executeScript("window.scrollBy(0, 250);");  // Scroll down 250px
```

✅ 29. Explicit Wait ⏳

- **Declared in:** `WebDriverWait` class
- **Implemented by:** `WebDriverWait` class
- **How to use:** Use `WebDriverWait` with `ExpectedConditions` to wait for a specific condition to occur before proceeding.
- **Arguments:** Takes `WebDriver` and `Duration` for maximum wait time.
- **Returns:** The object that corresponds to the expected condition.
- **Purpose:** Waits for an element or condition to be true (e.g., visibility of an element, clickability, etc.).

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
WebElement element = wait.until(ExpectedConditions.visibilityOfElementLo
cated(By.id("submitButton")));
```

✅ 30. TakesScreenshot - Capturing Screenshots 📸

- **Declared in:** `TakesScreenshot` interface
- **Implemented by:** `RemoteWebDriver` class

- **How to use:** Use `TakesScreenshot` to capture a screenshot of the current browser state.
- **Arguments:** None.
- **Returns:** `File` – the screenshot file.
- **Purpose:** Captures a screenshot of the current browser page, typically used for debugging and reports.

```
File screenshot = ((TakesScreenshot) driver).getScreenshotAs(OutputType.
FILE);
File destinationFile = new File("path/to/screenshot.png");
FileUtils.copyFile(screenshot, destinationFile);  // Save the screenshot
```

Let me know if you'd like more Selenium concepts or examples!