Selenium with Java
Lecture 9

What are Wait commands in Selenium?

In Selenium, wait commands are used to synchronize the execution of test scripts with the state of the web application. Wait Commands help ensure that the script waits for certain conditions to be met before proceeding, which is crucial for dynamic web pages where elements may take time to load.

This helps in avoiding exceptions that occur when the elements to be tested are not loaded. Wait commands are essential when it comes to executing Selenium tests. They help to observe and troubleshoot issues that may occur due to variation in time lag.

While running Selenium tests, it is common for testers to get the message “Element Not Visible Exception“. This appears when a particular web element with which WebDriver has to interact, is delayed in its loading. To prevent this Exception, Selenium Wait Commands must be used.

Selenium WebDriver provides three commands to implement waits in tests.

1. Implicit Wait
2. Explicit Wait
3. Fluent Wait

Implicit Wait in Selenium

Implicit wait makes WebDriver wait for a specified amount of time when trying to locate an element before throwing a NoSuchElementException. When implicit wait is set, the WebDriver will wait for a defined period, allowing for elements to load dynamically.

Implicit Wait setting is a Global setting and applies to all elements in the script, and it remains in effect for the duration of the WebDriver instance.

Once the command is run, Implicit Wait remains for the entire duration for which the browser is open. It's default setting is 0, and the specific wait time needs to be set by the following protocol.

Implicit Wait Syntax

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

However, implicit wait increases test script execution time. It makes each command wait for the defined time before resuming test execution. If the application responds normally, the implicit wait can slow down the execution of test scripts.

```
package MYPackage;
import java.time.Duration;
import java.util.List;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
public class ImplicitWait {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		WebDriver driver = new ChromeDriver();
	    driver.get("http://www.google.com");
	    driver.manage().window().maximize();
	    driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(2));
	    WebElement element = driver.findElement(By.xpath("//textarea[@id='APjFqb']"));
	    element.sendKeys("Selenium WebDriver Interview questions");
	    element.sendKeys(Keys.ENTER);
	    List<WebElement> list = driver.findElements(By.className("_Rm"));
	    System.out.println(list.size());
	}
}
```

Explicit Wait in Selenium:

Explicit wait in Selenium is a synchronization mechanism that allows the WebDriver to wait for a specific condition to occur before proceeding with the next step in the code. Unlike Implicit waits, which apply globally, explicit waits are applied only to specific elements or conditions, making them more flexible and precise.

Explicit wait is more intelligent, but can only be applied for specified elements. However, it is an improvement on implicit wait since it allows the program to pause for dynamically loaded Ajax elements.

The following Expected Conditions can be used in Explicit Wait.

1. alertIsPresent()
2. elementSelectionStateToBe()
3. elementToBeClickable()
4. elementToBeSelected()
5. frameToBeAvaliableAndSwitchToIt()
6. invisibilityOfTheElementLocated()
7. invisibilityOfElementWithText()
8. presenceOfAllElementsLocatedBy()
9. presenceOfElementLocated()
10. textToBePresentInElement()
11. textToBePresentInElementLocated()
12. textToBePresentInElementValue()
13. titleIs()
14. titleContains()
15. visibilityOf()
16. visibilityOfAllElements()
17. visibilityOfAllElementsLocatedBy()
18. visibilityOfElementLocated()

Explicit Wait Syntax:

```
WebDriverWait wait = new WebDriverWait(driver,30);
wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("//div[contains(text()));
```

Selenium WebDriver to wait for 30 seconds before throwing a TimeoutException. If it finds the element before 30 seconds, then it will return immediately.

Fluent Wait in Selenium

Fluent Wait in Selenium marks the maximum amount of time for a Selenium WebDriver to wait for a certain condition (web element) becomes visible. It also defines how frequently WebDriver will check if the condition appears before throwing the “ElementNotVisibleException”.

To put it simply, Fluent Wait looks for a web element repeatedly at regular intervals until timeout happens or until the object is found.

Fluent Wait commands are most useful when interacting with web elements that can take longer durations to load. This is something that often occurs in Ajax applications.

Fluent waits are also sometimes called smart waits because they don’t wait out the entire duration defined in the code. Instead, the test continues to execute as soon as the element is detected – as soon as the condition specified in .until(YourCondition) method becomes true.

Fluent Wait Syntax

```
Wait wait = new FluentWait(WebDriver reference)
.withTimeout(timeout, SECONDS)
.pollingEvery(timeout, SECONDS)
.ignoring(Exception.class);

WebElement foo=wait.until(new Function<WebDriver, WebElement>() {
public WebElement apply(WebDriver driver) {
return driver.findElement(By.id("foo"));
}
});
```

```
//Declare and initialise a fluent wait
FluentWait wait = new FluentWait(driver);
//Specify the timout of the wait
wait.withTimeout(5000, TimeUnit.MILLISECONDS);
//Sepcify polling time
wait.pollingEvery(250, TimeUnit.MILLISECONDS);
//Specify what exceptions to ignore
wait.ignoring(NoSuchElementException.class)
wait.until(ExpectedConditions.alertIsPresent());
```

How to Handle Web Table in Selenium:

Web Tables are like normal tables where the data is presented in a structured form using rows and columns. The only difference is that they are displayed on the web with the help of HTML code.

<table> is the HTML tag that is used to define a web table. While <th> is used for defining the header of the table, <tr> and <td> tags are used for defining rows and columns respectively for the web table.

1. Static Web Tables

These tables have fixed data that remains unchanged throughout. Due to the static nature of their content, they are called Static web tables.

2. Dynamic Web Tables

These tables have data that changes over time, and hence the number of rows and columns might also change depending upon the data shifts. Due to the dynamic nature of their content, they are called Dynamic web tables.

```
package MYPackage;
import java.time.Duration;
import java.util.List;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
public class WebTableDemo {
	public static void main(String[] args) {
		// TODO Auto-generated method stub

		WebDriver driver = new ChromeDriver();
		driver.get("https://www.nyse.com/ipo-center/recent-ipo");
		driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(5));
		driver.manage().window().maximize();

		//Finding number of Rows
		List<WebElement> rowsNumber = driver.findElements(By.xpath("//table[1]/tbody/tr"));
		int rowCount = rowsNumber.size();
		System.out.println("No of rows in this table : " + rowCount);
		//Finding number of Columns
		List<WebElement> columnsNumber =
driver.findElements(By.xpath("((//table[1]/thead)[1])//th"));
		int columnCount = columnsNumber.size();
		System.out.println("No of columns in this table : " + columnCount);
	       //Finding cell value at 2nd row and 3rd column
		WebElement cellAddress = driver.findElement(By.xpath("(//table[1]//tbody//tr[1])//td[3]"));
				String value = cellAddress.getText();
				System.out.println("The Cell Value is :" +value);
				driver.quit();

	}
}
```

How to use JavascriptExecutor in Selenium

JavascriptExecutor is an interface that is used to execute JavaScript with Selenium. To simplify the usage of JavascriptExecutor in Selenium, think of it as a medium that enables the WebDriver to interact with HTML elements within the browser. JavaScript is a programming language that interacts with HTML in a browser, and to use this function in Selenium, JavascriptExecutor is required.

Sometimes, Selenium WebDriver alone will not be able to perform certain operations or interact with web elements. In that case, JavaScript is needed to make sure those actions are being performed accurately.

Let’s suppose a tester has written an automation script to click a few buttons, but there seems to be an issue due to which the script fails every time. To resolve this, the tester uses JavascriptExecutor.

JavascriptExecutor consists of two methods that handle all essential interactions using JavaScript in Selenium.

1. **executeScript method** – This method executes the test script in the context of the currently selected window or frame. The script in the method runs as an anonymous function. If the script has a return statement, the following values are returned:

```
package MYPackage;
import org.openqa.selenium.JavascriptExecutor;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
public class DemoJavaScriptExecutor {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		WebDriver driver = new ChromeDriver();
	   driver.get("https://www.browserstack.com/users/sign_in");
	   driver.manage().window().maximize();
	   JavascriptExecutor js = (JavascriptExecutor)driver;
	   js.executeScript("document.getElementById('user_email_login').value='rbc@xyz.com';");
	   js.executeScript("document.getElementById('user_password').value='password';");
	   js.executeScript("document.getElementById('user_submit').click();");
	   js.executeScript("alert('enter correct login credentials to continue');");
	}
}
```

2. **executeAsyncScript method** – This method executes the asynchronous piece of JavaScript on the current window or frame. An asynchronous script will be executed while the rest of the page continues parsing, which enhances responsiveness and application performance.

```
package MYPackage;

import org.openqa.selenium.JavascriptExecutor;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

public class DemoJavaScriptExecutor {

	public static void main(String[] args) {

		// TODO Auto-generated method stub

		WebDriver driver = new ChromeDriver();

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

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

	   JavascriptExecutor js = (JavascriptExecutor)driver;

	   js.executeAsyncScript("window.scrollBy(0,document.body.scrollHeight)");
```

```
        }
}
```

How to take Screenshot in Selenium WebDriver

```
File file = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
```

```
String screenshotBase64 =
((TakesScreenshot)driver).getScreenshotAs(OutputType.BASE64);
```

To capture screenshots in Selenium, one has to utilize the method **TakesScreenshot**. This notifies WebDriver that it should take a screenshot in Selenium and store it.

OutputType defines the output type for the required screenshot in the above snippet.

If the user intends to take a screenshot in Selenium and store it in a designated location, the method is **getScreenshotAs**.

```
package MYPackage;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import java.io.File;
import java.io.IOException;
import org.apache.commons.io.FileUtils;
import org.openqa.selenium.OutputType;
import org.openqa.selenium.TakesScreenshot;
public class DemoTakeScreenShot {
        public static void main(String[] args) throws IOException {
                WebDriver driver = new ChromeDriver();
           driver.get("https://www.browserstack.com");
           driver.manage().window().maximize();
           //Convert web driver object to TakeScreenshot
           TakesScreenshot scrShot =((TakesScreenshot)driver);
           //Call getScreenshotAs method to create image file
           File SrcFile=scrShot.getScreenshotAs(OutputType.FILE);
           //Move image file to new destination
           File DestFile=new File("C:\\Users\\piyush.keshari_bitsi\\Desktop\\Selenium with
Java\\test.png");
           //Copy file at destination
           FileUtils.copyFile(SrcFile, DestFile);
           }
           }
```