Selenium WebDriver

Last updated: 17 March 2022

Contents

Assert and Verify

There are two mechanisms for validating elements available in the application under test. The first is "`assert`"; this allows the test to check if the element is on the page. If it is not available, then the test will stop at the step that failed. The second is "`verify`"; this also allows the test to check if the element is on the page, but if it isn't, then the test will carry on executing.

Some of the "`verify`" and "`assert`" methods are:

```
verifyElementPresent
assertElementPresent
verifyElementNotPresent
assertElementNotPresent
verifyText
assertText
verifyAttribute
assertAttribute
verifyChecked
assertChecked
verifyAlert
assertAlert
verifyTitle
assertTitle
```

Locating Elements

Locating elements is done by using the "`findElement()`" and "`findElements()`" methods.

The "`findElement()`" method returns the first WebElement object based on a specified search criteria, or throws an exception, if it does not find any element matching the search criteria.

The "`findElements()`" method returns a list of WebElements matching the search criteria. If no elements are found, it returns an empty list.

The "`findElement()`" and "`FindElements()`" methods throw a "`NoSuchElementFoundException`" exception when they fail to find the desired element using the specified locator strategy

Strategy	Description
By id	Locates an element using the ID attribute
By name	Locates an element using the Name attribute
By class name	Locates an element using the (CSS style) Class attribute
By tag name	Locates an element using the HTML tag (for example "Select", "Input", "List", "Button" etc.). The "Input" tag has multiple different types (such as "RADIO", "CHECKBOX", "TEXTBOX", "PASSWORD") that need to be further filtered by using the "type" attribute.

By link text	Locates link using its text
By partial link text	Locates link using its partial text

By CSS	Locates element using the CSS selector.
	
	The By CSS locator is similar to the By XPath locator in its usage, but the difference is that it is slightly faster.
	
	The following example uses two attributes for the “<input>” element:
	
	`WebElement previousButton = driver.findElement(By.cssSelector("input[type='submit'][value='Login']"));`
	There is also a partial match syntax (a kind of “regular expression”), featuring “`^=`” (starting with), “`$=`” (ending with) and “`*=`” (containing).
	
	Checking for text inside of an element:
	
	`WebElement cell = driver.findElement(By.cssSelector("td:contains('MySearchText')"));`
	
	More examples:
	
	• To identify an element using the “`div`” element with “`id #flrs`”, use the “`#flrs`” syntax.
	• To identify the child “`anchor`” element, use the “`#flrs > a`” syntax, which returns the “`link`” element.
	• To identify the “`anchor`” element with its attribute, use the “`#flrs > a[a[href="/intl/en/about.html"]]`” syntax.

By XPath	Locates element using XPath query. However, this is the least preferable locator strategy due its slow performance, as XPath provides the ability to search elements bi-directionally (downwards in the hierarchy, as well as upwards). In web browsers, the built-in developer tools can be used to identify the XPath of elements. XPath examples: • The root element is identified as “`//`”. • To identify all the “`div`” elements, the syntax is “`//div`”. • To identify the “`link`” tags that are within the “`div`” element, the syntax is “`//div/a`”. • To identify all the elements with a tag, use “`*`”. The syntax is “`//div/*`”. • To identify all the “`div`” elements that are at three levels down from the root, use “`//*/*/div`”. • To identify specific elements, use attribute values of those elements, such as “`//*/div/a[@id='attrValue']`”, which returns the “`anchor`” element. This element is at third level from root within a “`div`” element, and has an “`id`” value “`attrValue`”. • The following example searches for the “`<DIV>`” node (in the DOM) that contains an attribute named “`class`” with the value "*myclassname*"): `//div[contains(@class,"myclassname")]`

Example for finding a child element:

```
WebElement topLink = driver.findElement(By.id("div1")).findElement(By.linkText("top"));
```

Example for extracting all the links and print their targets by using the "`findElements()`" method:

```
@Test
public void testFindElements()
{
	//Get all the links displayed on Page
	List<WebElement> links = driver.findElements(By.tagName("a"));
	//Verify that there are four links displayed on the page
	assertEquals(4, links.size());
	//Iterate though the list of links and print target for each link
	for(WebElement link : links)
		System.out.println(link.getAttribute("href"));
}
```

Relative Locators

Relative Locators find web elements based on their GUI location.

There are five Relative Locators:

Relative Locator	Description
`above()`	Web element appears above the specified web element
`below()`	Web element appears below the specified web element
`toLeftOf()`	Web element appears to the left of the specified web element
`toRightOf()`	Web element appears to the right of the specified web element
`near()`	Web element is at most 50 pixels away from the specified web element. There is also an overloaded method to specify the distance.

Example for "`toLeftof()`" and "`below()`" locators:

```
WebElement book;

book =
driver.Findelement(RelativeLocators.withTagName("li").toLeftOf(By.id("pid1")).be
low(By.id("pid2")));

String strId = book.getAttribute ("id");
```

Example for "`toRightOf()`" and "`above()`" locators:

```
WebElement book;

book =
driver.Findelement(RelativeLocators.withTagName("li").toRightOf(By.id("pid1")).a
bove(By.id("pid2")));

String strId = book.getAttribute ("id");
```

If they are multiple results for the query, then the first matching item will be returned. It is also important to consider that the results might vary depending on the viewport. The view on a mobile phone might look very different than the view on a desktop browser.

Determine if an element exists without throwing an exception

WebDriver is really good at letting you know when an element does not exist. If it throws a "`NoSuchElementException`" exception, then you know it's not there. You must handle these errors.

To get around this, you can use "`findElements()`", and check that the size of the list returned is 0. For example:

```
List<WebElement> elements = driver.findElements(By.Id("myElement"));
elements.size(); //This should be zero and can be checked accordingly
```

Get information from elements

Method	Result

`getText();`	Returns the value of the "`innerText`" attribute of the element.
	
	You can also perform a partial match using Java String API methods such as "`contains()`", "`startsWith()`" , and "`endsWith()`". You can use these methods in the following way:
	
	`assertTrue(messageText.contains("color"));`
	`assertTrue(messageText.startsWith("Click on"));`
	`assertTrue(messageText.endsWith("will change"));`

`getAttribute()`	Retrieve an element's attribute. Example: `assertEquals("justify",message.getAttribute("align"));`

`getCSSValue()`	Returns the value of a specified style attribute. Example:
	
	`String width = message.getCssValue("width");`
	`assertEquals("150px",width);`

`getLocation()`	The "`getLocation()`" method can be used with all WebElements. It is used to get the relative position of an element where it is rendered on the web page. This position is

	calculated relative to the top-left corner of the web page of which the (x, y) coordinates are assumed as (0, 0). This method can be of use if a test script tries to validate the layout of a web page.
`getSize()`	The "`getSize()`" method can also be used with all visible HTML elements. It returns the width and height of the rendered WebElement.
`getTagName()`	The "`getTagName()`" method can be used on all WebElements. It returns the tag name of the WebElement. For example, in the following HTML code, "`button`" is the tag name of the HTML element: `<button id="gbqfba" class="gbqfba" name="btnK" aria-label="Google Search">`

Get the content of a whole page

The "`driver.getPageSource()`" method gets the text of a whole page, for example:

```
if(driver.getPageSource().contains("Whatever you look for"))
{
      ...
}
```

Enter data in Editboxes

Data for "`textbox`" or "`textarea`" HTML elements can be entered with the "`sendKeys()`" method, for example:

```
driver.findElement(By.xpath("//input[@name='q']")).sendKeys("hello");
```

Special keys, such as Backspace, Enter, Tab, or Shift, require using a special "enum" class of WebDriver, named "`Keys`". This example uses the Shift key to type the text in uppercase:

```
driver.findElement(By.xpath("//input[@name='q']")).sendKeys(Keys.chord(Keys.SHIF
T,"phones"));
```

Clear data in Editboxes

The "`clear()`" method action is similar to the "`sendKeys()`" method, which is applicable for "`textbox`" and "`textarea`" HTML elements. It is used to erase the text that is entered, for example:

```
driver.findElement(By.xpath("//input[@name='q']")) .clear();
```

The "submit()" method

The "`submit()`" method can be used on a form, or on an element within a form. It is used to submit a form of a web page to the server hosting the web application, for example:

```
driver.findElement(By.xpath("//input[@name='q']")) .submit();
```

Set information on Dropdown and Lists

WebDriver supports testing Dropdown and List controls using a special "`Select`" class instead of the "`WebElement`" class.

A sample test for a Dropdown control:

```
@Test

public void testDropdown()

{

      //Get the Dropdown as a Select using its name attribute
      Select make = new Select(driver.findElement(By.name("make")));

      //Verify Dropdown does not support multiple selection
      assertFalse(make.isMultiple());

      //Verify Dropdown has four options for selection
      assertEquals(4, make.getOptions().size());

      //You can select an option in Dropdown using Visible Text
      make.selectByVisibleText("Honda");
      assertEquals("Honda", make.getFirstSelectedOption().getText());

      //or you can select an option in Dropdown using value attribute
      make.selectByValue("Audi");
      assertEquals("Audi", make.getFirstSelectedOption().getText());

      //or you can select an option in Dropdown using index
      make.selectByIndex(0);
      assertEquals("BMW", make.getFirstSelectedOption().getText());

}
```

Sample code to check the options in the Dropdown control:

```
//Verify that the Dropdown has expected values as listed in a array
List<String> exp_options = Arrays.asList(new String[]{"BMW", "Mercedes",
"Audi","Honda"});
List<String> act_options = new ArrayList<String>();

//Retrieve the option values from Dropdown using getOptions() method
for(WebElement option : make.getOptions())
act_options.add(option.getText());

//Verify expected options array and actual options array match
assertArrayEquals(exp_options.toArray(),act_options.toArray());
```

For checking whether a specific option is available for selection, you can simply perform a check on the "`act_options`" array list in the following way:

```
assertTrue(act_options.contains("BMW"));
```

A sample test for a List control, which has multi-selection enabled.

```
@Test

public void testMultipleSelectList()

{

	//Get the List as a Select using its name attribute
	Select color = new Select(driver.findElement(By.name("color")));

	//Verify List support multiple selection
	assertTrue(color.isMultiple());

	//Verify List has five options for selection
	assertEquals(5, color.getOptions().size());

	//Select multiple options in the list using visible text
	color.selectByVisibleText("Black");
	color.selectByVisibleText("Red");
	color.selectByVisibleText("Silver");

	//Deselect an option using visible text
	color.deselectByVisibleText("Silver");

	//Deselect an option using value attribute of the option
	color.deselectByValue("rd");

	//Deselect an option using index of the option
	color.deselectByIndex(0);

}
```

Clicking on Buttons and Links (and other web elements)

The "`click()`" method is used to click on Buttons and Links (and other web elements).

Set information on Radio Buttons and Radio Groups

WebDriver supports Radio Button and Radio Group controls using the "`WebElement`" class. You can select and deselect the radio buttons using the "`click()`" method of the "`WebElement`" class and check whether a radio button is selected or deselected using the "`isSelected()`" method

You can also get all the radio buttons from a Radio Group in a list using the "`findElements()`" method along with the Radio Group identifier.

Set information on Checkboxes

WebDriver supports Checkbox control using the "`WebElement`" class. You can select or deselect a checkbox using the "`click()`" method of the WebElement class and check whether a checkbox is selected or deselected using the "`isSelected()`" method.

Working with Tables

A table can be identified by using name, id or xpath and then the rows can be accessed one by one through the "`findElements()`" method.

The first example finds all row elements from a given table. Please note that "t" in this example stands for the table object.

```
List<WebElement> rows = t.findElements(By.tagName("tr"));
```

The second example illustrates how to find the column number for the given column name in a table.

```
int getColumnNumber(WebElement r, String columnName)
{
	List<WebElement> cells = r.findElements(By.tagName("th"));
	int c = 0;

	for (WebElement cell : cells)
	{
		c=c+1;
		System.out.println(c + " --> "+ cell.getText() );
		if (columnName.equals(cell.getText())) break;
	}
	return c;
}
```

The third example illustrates how to check, if the value in a given cell matches the desired value. The function takes 3 parameters. The first parameter is the row element, the second parameter is the column number, and third parameter is the expected value.

```
boolean verifyValue (WebElement r, int a, String expValue)
{
	List<WebElement> mcells = r.findElements(By.tagName("td"));
	int c = 0;

	for (WebElement cell : mcells)
	{
		c=c+1;
		if (c==a)
		{
			// get the value inside cell with the getText() method
			if (expValue.equals(cell.getText())) return true;
		}
```

```
        }
        return false;
}
```

Check the status of an element

The WebElement class provides the following methods to check the state of an element:

Method	Purpose
`isEnabled()`	This method checks if an element is enabled. Returns “true” if enabled, else “false” for disabled.
`isSelected()`	This method checks if element is selected (radio button, checkbox, and so on). It returns “true” if selected, or “false” for deselected. It can be executed only on a radio button, options in select, and checkbox WebElements. When executed on other elements, it will return “false”.
`isDisplayed()`	This method checks if element is displayed.

Mouse and Keyboard events

The “`Actions`” class implements the builder pattern to create a composite action containing a group of other actions, such as dragging-and-dropping, or holding a key and then performing mouse operations.

There are three different actions that are available in the “`Actions`” class that are specific to the keyboard. They are the “`keyUp()`”, “`keyDown()`”, and “`sendKeys()`” actions, each having two overloaded methods. One method is to execute the action directly on the WebElement, and the other is to just execute the method irrespective of its context.

Example: Let's create a test to select multiple rows from different positions in a table using the Ctrl key. You can select multiple rows by selecting the first row, then holding the Ctrl key, and then selecting another row and releasing the Ctrl key. This will select the desired rows from the table.

```
@Test
public void testRowSelectionUsingControlKey() {
        List<WebElement> tableRows =
        driver.findElements(By.xpath("//table[@class='iceDatTbl']/tbody/tr"));
        //Select second and fourth row from table using Control Key.
        //Row Index start at 0
        Actions builder = new Actions(driver);
        builder.click(tableRows.get(1))
        .keyDown(Keys.CONTROL)
        .click(tableRows.get(3))
        .keyUp(Keys.CONTROL)
        .build().perform();
        //Verify Selected Row table shows two rows selected
        List<WebElement> rows =
        driver.findElements(By.xpath("//div[@class='icePnlGrp
```

```
        exampleBox']/table[@class='iceDatTbl']/tbody/tr"));
        assertEquals(2,rows.size());
}
```

You can create a composite action that is ready to be performed by calling the "`build()`" method. Finally the test will perform this composite action by calling the "`perform()`" method of the "`Actions`" class.

The "`click()`" method is used to simulate the left-click of your mouse at its current point of location. It is also possible to specify a WebElement to click on as input parameter: "`click(`*WebElementName*`)`"

Double clicks can be implemented in the same way by clicking on the spot with the "`doubleClick()`" method, or on a specific WebElement: "`doubleClick(`*WebElementName*`)`"

The "`contextClick()`" method is also known as right-click. The context is nothing but a menu; a list of items is associated to a WebElement based on the current state of the web page. It is also possible to right-click on a specific WebElement: "`contextClick(`*WebElementName*`)`"

The "`moveByOffset()`" method is used to move the mouse from its current position to another point on the web page.

Drag and Drop operations can be implemented in the same way with the "`dragAndDrop(`*SourceWebElementName*`, `*TargetWebElementName*`)`" (to move to a target element) and the "`dragAndDropBy(`*WebElementName*`, `*intXoffset*`, `*intYoffset*`)`" (move by an offset) methods.

The "`clickAndHold()`" method is another method of the "`Actions`" class that left-clicks on an element and holds it without releasing the left button of the mouse. This method is useful when executing operations such as drag-and-drop. It is also possible to specify a WebElement to click on as input parameter: "`clickAndHold(`*WebElementName*`)`"

The "`release()`" method is the one that can release the left mouse button. By using "`release(`*WebElementName*`)`", you can actually release the currently held WebElement in the middle of another WebElement, so that you don't have to calculate the offset of the target WebElement from the held WebElement.

The "`moveToElement()`" method is another method that helps to move the mouse cursor to a WebElement on the web page.

WebDriver also provides a "`JavascriptExecutor`" interface that can be used to execute arbitrary JavaScript code within the context of the browser.

The "`JavascriptExecutor`" can inject and then use entire JavaScript libraries (such as JQuery). The "`JavascriptExecutor`" also has a method called "`executeAsyncScript()`", that allows execution of JavaScript that does not respond instantly.

Synchronisation

"`WaitFor`" Commands are very useful for AJAX. The "`waitFor`" commands will timeout after 30 seconds by default.

Command	Condition
`waitForPageToLoad`	Delays execution until the page is fully loaded in the browser

`waitForElementPresent`	Delays execution until the specified element is present on the page
`waitForElementNotPresent`	Delays execution until the specified element is removed from the page
`waitForTextPresent`	Delays execution until the specified text is present on the page
`waitForFrameToLoad`	Delays execution until the contents of the frame are fully loaded in the browser
`waitForAlertPresent`	Delays execution until an alert window is displayed on the page

A number of these commands are run implicitly when other commands are being executed. For example, with respect to the "`clickAndWait`" command, when you click on an element, the "`waitForPageToLoad`" command is also executed.

Set the Implicit Wait timeout to 10 Seconds

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

Implicit timeouts are common to all WebElements and have a global timeout period associated with them.

However, an implicit wait condition may slow down your tests when an application responds normally, as it will wait for each element to appear in the DOM and increase the overall execution time. It is recommended to avoid or minimize the use of an implicit wait condition. Try to handle synchronization issues with an explicit wait condition instead, which provides more control as compared to an implicit wait condition.

Explicit wait

WebDriver provides the "`WebDriverWait`" and "`ExpectedCondition`" classes for implementing an explicit wait condition. The following example sets the explicit wait time for a new WebElement called "searchBox" that is identified by its name ("q") to 20 seconds. The implicit timeout doesn't get applied for this WebElement. This effectively allows you to override the implicit wait time exclusively for a WebElement, if you think that it might require more time.

```
WebElement searchBox = (new WebDriverWait(driver,
Duration.ofSeconds(20))).until((ExpectedCondition<WebElement>) d ->
d.findElement(By.name("q")));
```

The "`ExpectedCondition`" class provides a set of predefined conditions to wait before proceeding further. The following table shows some common conditions that you frequently come across when automating web browsers supported by the "`ExpectedCondition`" class:

Predefined condition	WebDriver (Java)
An element is visible and enabled	`elementToBeClickable(By locator)`
An element is selected	`elementToBeSelected(WebElement element)`
Presence of an element	`presenceOfElementLocated(By locator)`
Specific text present in an element	`textToBePresentInElement(By locator, java.lang.String text)`
Element value	`textToBePresentInElementValue(By locator, java.lang.String text)`
Title	`titleContains(java.lang.String title)`

Set the default browser navigation timeout

The following statement sets the navigation timeout to 50. This means that WebDriver will wait for a maximum of 50 seconds for a page to load. If it doesn't, then it will throw an exception.

```
driver.manage().timeouts().pageLoadTimeout(Duration.ofSeconds(50));
```

Controlling browsers using "navigate"

WebDriver talks to individual browsers natively. This way it has better control, not just on the web page, but on the browser itself. Navigate is one such feature of WebDriver that allows the test script developer to work with the browser's "Back", "Forward", and "Refresh" controls of the browser.

The method that is used for this purpose is "`navigate()`", for example:

```
driver.navigate().back();
driver.navigate().forward();
driver.navigate().refresh();
```

Cookies

WebDriver provides methods for handling cookies, for example "`driver.manage().getCookies()`" to fetch all cookies that are loaded for a web page, or "`driver.manage().addCookie(CookieObjectName)`" for adding cookie information to the driver.

Open new windows and tabs

Open a new window:

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

// Opens a new window and switches to new window

driver.switchTo().newWindow(WindowType.WINDOW);

// Opens Microsoft homepage in the newly opened window

driver.navigate().to("https://www.microsoft.com/");
```

Open a new tab within the same window:

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

// Opens a new tab in existing window

driver.switchTo().newWindow(WindowType.TAB);

// Opens Microsoft homepage in the newly opened tab

driver.navigate().to("https://www.microsoft.com/");
```

Handling pop-up windows

Testing pop-up windows involves identifying a pop-up window, switching the driver context to the pop-up window, then executing steps in the pop-up window, and finally switching back to the parent window.

Every time you open a web page using WebDriver in a browser window, WebDriver assigns a window handle to that window. WebDriver uses this window handle to identify the window.

WebDriver allows you to identify a pop-up window by its window handle or its "`name`" attribute and switching between the pop-up window and the browser window is done using the "`Webdriver.switchTo().window()`"method.

In the following example, a user can open a pop-up window by clicking on the Help button. In this example, the developer has provided "HelpWindow" as its name:

```
@Test

public void testWindowPopup()

{

//Save the WindowHandle of Parent Browser Window
String parentWindowId = driver.getWindowHandle();

//Clicking Help Button will open Help Page in a new Popup Browser Window
WebElement helpButton = driver.findElement(By.id("helpbutton"));
helpButton.click();

try {

     //Switch to the Help Popup Browser Window
     driver.switchTo().window("HelpWindow");
```

```
} catch (NoSuchWindowException e) {
     e.printStackTrace();
}
     //Verify the driver context is in Help Popup Browser Window
     assertTrue(driver.getTitle().equals("Help"));
     //Close the Help Popup Window
     driver.close();
     //Move back to the Parent Browser Window
     driver.switchTo().window(parentWindowId);
     //Verify the driver context is in Parent Browser Window
     assertTrue(driver.getTitle().equals("Build my Car - Configuration"));
}
```

Handling JavaScript alerts (pop ups)

WebDriver provides an "`Alert`" class for handling alerts. To access the alert box displayed on the screen as an instance of the "`Alert`" class, the "`driver.switchTo().alert()`" method is used as follows:

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

If there is no dialog currently present, and you invoke this API, it throws a "`NoAlertPresentException`" exception.

A test might need to verify what message is displayed in an alert box. You can get the text from an alert box by calling the "`getText()`" method of the "`Alert`" class as follows:

```
String textOnAlert = alert.getText();
```

An alert box is closed by clicking on the "OK" button. This can be done by calling the "`accept()`" method as follows:

```
alert.accept();
```

Alternatively, an alert box can also be accessed without creating an instance of the "`Alert`" class by directly calling the desired methods as follows:

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

An alert box can also be cancelled (instead of "OK") by:

```
alert.dismiss();
```

Handling OK/Cancel Message Boxes (=Confirm Box Alert)

When a confirm box pops up, the user has to click either on the "OK" or the "Cancel" button to proceed. If the user clicks on the "OK" button, the box returns true. If the user clicks on the "Cancel" button, the box returns false.

The handling is essentially the same as with an Alert Box. The "`accept()`" method clicks the "OK" button, "`dismiss()`" clicks on the "Cancel" button.

Handling an Input Box (Prompt Box Alert)

An Input Box allows the input of a value and displays OK/Cancel boxes to proceed with the input or cancel it. If the user clicks on the "OK" button, the box returns the input value. If the user clicks on the "Cancel" button, the box returns null.

The handling is essentially the same as with an Alert Box. Input is sent via the "`sendKeys()`" method.

Frames (including IFRAME)

Frames can be identified by an "`ID`" or "`name`" attribute, or by their index. The focus can be switched to a frame by the "`driver.switchTo().frame()`" method. The "`defaultContent()`" method can then be used to switch the focus back to the page.

Example of switching to a frame by the "`ID`" or "`name`" attribute:

```
driver.switchTo().frame("left");
```

Example of switching to a frame by the index of the frame. The index is zero-based, meaning that the first frame has the number 0:

```
driver.switchTo().frame(1);
```

Switch back to the default page. This is very important. If you don't do this and try to switch to another frame (for example the frame with the index 2) while you are still in the first frame, your WebDriver will complain, saying that it can't find a frame with index 2. This is because the WebDriver searches for the second frame in the context of the first frame. So, you have to first come back to the top-level container, before you switch to another frame:

```
driver.switchTo().defaultContent();
```

Another alternative for switching frames is by passing the frame WebElement:

`driver.switchTo().frame(`*frameElement*`);`

Event handling

WebDriver provides a very good framework for tracking the various events that happen while running test scripts using WebDriver. Many navigation events that get fired before and after an event occurs (such as before and after navigating to a URL, before and after browser back-navigation, and so on) can be tracked and captured. To fire an event, WebDriver uses a class named "`EventFiringWebDriver`", and to catch that event, it provides an interface named "`WebDriverEventListener`". Test script developers should provide their own implementations for the overridden methods from the interface.

The "`EventFiringWebDriver`" class **is a wrapper** around the WebDriver that gives the driver the capability to fire events. That means that it effectively replaces the original WebDriver inside the script. For example, instead of "*webDriver*`.get("`*https://www.google.com*`");`", something like "*eventFiringDriver*`.get("`*https://www.google.com*`");`" will be used.

These are the methods of "`WebDriverEventListener`":

`beforeAlertAccept(WebDriver webDriver)`
`afterAlertAccept(WebDriver webDriver)`
`afterAlertDismiss(WebDriver webDriver)`
`beforeAlertDismiss(WebDriver webDriver)`
`beforeNavigateTo(String url, WebDriver webDriver)`
`afterNavigateTo(String s, WebDriver webDriver)`
`beforeNavigateBack(WebDriver webDriver)`
`afterNavigateBack(WebDriver webDriver)`
`beforeNavigateForward(WebDriver webDriver)`
`afterNavigateForward(WebDriver webDriver)`
`beforeNavigateRefresh(WebDriver webDriver)`
`afterNavigateRefresh(WebDriver webDriver)`
`beforeFindBy(By by, WebElement webElement, WebDriver webDriver)`
`afterFindBy(By by, WebElement webElement, WebDriver webDriver)`
`beforeClickOn(WebElement webElement, WebDriver webDriver)`
`afterClickOn(WebElement webElement, WebDriver webDriver)`
`beforeChangeValueOf(WebElement webElement, WebDriver webDriver)`
`afterChangeValueOf(WebElement webElement, WebDriver webDriver)`
`beforeScript(String s, WebDriver webDriver)`
`afterScript(String s, WebDriver webDriver)`
`onException(Throwable throwable, WebDriver webDriver)`

Taking Screenshots with WebDriver

You can save the file object returned by the "`getScreenshotAs()`" method using the "`copyFile()`" method of the Java "`FileUtils`" class from the "`org.apache.commons.io.FileUtils`" class. It will create a "`*.png`" file.

Example:

```
@Test
public void testTakesScreenshot()
{
	try {
		File scrFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
		FileUtils.copyFile(scrFile, new File("c:\\tmp\\main_page.png"));
	} catch (Exception e) {
		e.printStackTrace();
	}
}
```

The "`TakesScreenshot`" interface captures the screenshot of the entire page, current window, visible portion of the page, or of the complete desktop window in their respective order, as supported by the browser.

Taking Screenshots with RemoteWebDriver

"`RemoteWebDriver`" doesn't implement the "`TakesScreenshot`" interface. There are multiple ways to work around this.

One way is to create an own "`WebDriver`" class that extends the "`RemoteWebDriver`" class and implements the "`TakesScreenshot`" interface by providing the implementation for the "`getScreenshotAs()`" method. Another ways is to use the "`Augmenter`" class.

Add the following code to the test using RemoteWebDriver:

```
driver = new Augmenter().augment(driver);
File scrFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
FileUtils.copyFile(scrFile, new File("c:\\tmp\\screenshot.png"));
```

The Page Object pattern

The Page Object pattern brings the following advantages:

- It helps in building a layer of abstraction separating automation code, which knows about locating application elements and the one which interacts with these elements for actual testing.
- It provides a central repository of pages from the application for tests.
- It provides high maintainability and reduction in code duplication.

A Page Object is an object in a Java class.

A Factory method is just a fancy name for a method that instantiates objects. Like a factory, the job of the factory method is to create -- or manufacture – objects.

Page Object Classes	You split the test logic out into separate PageObject classes. This allows you to create a Java class for each page and/or business process that you can use in an application.
	
	Elements in PageObject classes should be declared by using the "`@FindBy`" annotation. There are two different ways to use "`@FindBy`". The following one works for ID, NAME, or CLASS_NAME):
	`@FindBy(id="`*user_login*`")`
	`WebElement` *email;*
	
	The second way to use "`@FindBy`" works with all the different locator mechanisms (CLASS_NAME, CSS, ID, ID_OR_NAME, LINK_TEXT, NAME, PARTIAL_LINK_TEXT, TAG_NAME, XPATH):
	`@FindBy(how=How.ID, using="`*user_login*`")`
	`WebElement` *email;*
	
	A constructor method is used to call the web page of the page objects, so that they can be initialised:
	`public` *NameOfThePageObjectClass*`(WebDriver` *driver*`){`
	`this.`*driver* `=` *driver;*
	driver`.get("`*http://url.html*`");`
	`}`

Page Factory	Once the PageObject class declares elements using the "`@FindBy`" annotation, you can instantiate that PageObject class and its elements using the "`PageFactory`" class. The elements of the PageObject class get initialised when

	you call "`PageFactory.initElements();`" in your tests.
	
	The following example creates an object called "*loginPage*" from the PageObject class "*AdminLoginPage*". It then runs a method called "*login*":
	
	`AdminLoginPage loginPage= PageFactory.initElements(driver, AdminLoginPage.class);`
	
	`loginPage.login();`
	
	Because it is best practice to nest PageObjects when it makes business sense to build a business function (such as "create a new customer") in a method of a PageObject class, the same example program code just mentioned is also often used within a PageObject class method to include another (sub) PageObject as part of a method that reflects a business function.

LoadableComponent
 	
(This is an optional recommended extension. When used, it replaces/changes some of the code just mentioned, as described under "Note")
	The loadable component is an extension to the PageObject pattern. The "`LoadableComponent`" class in the WebDriver library helps test developers to ensure that the page or a component of the page is loaded successfully. It tremendously reduces the efforts to debug your test cases. The PageObject class should extend this "`LoadableComponent`" abstract class and, as a result, it is bound to provide implementation for the following two methods:
	
	`protected abstract void load()`
	
	`protected abstract void isLoaded() throws java.lang.Error`
	
	The page or component that has to be loaded in the "`load()`" and "`isLoaded()`" methods determines whether or not the page or component is fully loaded. If it is not fully loaded, it throws an exception. The URL that has to be loaded is specified in the "`load()`" method and the "`isLoaded()`" method validates whether or not the correct page is loaded.
	
	Examples:
	
	`@Override`
	`protected void load() {`
	`  driver.get("`*`http://url.html`*`");`
	`  PageFactory.initElements(driver, this);`
	`}`
	
	`@Override`
	`protected void isLoaded() throws Error {`
	`  Assert.assertTrue(`*`anyPageObjectWebElement`*` != null);`
	`}`
	
	The "`get()`" method ensures that the component is loaded by invoking the "`isLoaded()`" method (which calls the "`load()`" method if the assertion fails), for example:
	
	`AdminLoginPage loginPage= new AdminLoginPage(driver).get();`

	Note: If `LoadableComponent`" is used, then this code replaces the previously recommended way of creating the "loginPage" (when "`LoadableComponent`" is not used), because "`PageFactory.initElements()`" is now a part of the

	"`load()`"method. Please also note that "`driver.get("`*http://url.html*"`);`" has been moved from the constructor of the PageObject class to the "`load()`" method.

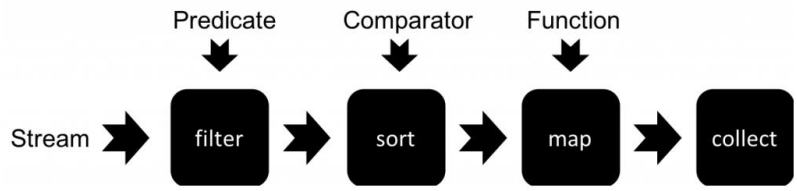
Dealing with I/O

WebDriver provides three important classes to handle the filesystem.

Class	Used for
`FileHandler`	Copy files, create directories, delete files or directories, check if a file is a zipped file, make a file executable, make a file writeable, read a file as a string, check if a file is executable etc.
`TemporaryFilesystem`	As the name suggests, the files that are created under temporary filesystem are temporary; that is, the files are deleted as soon as the test script is executed.
`zip`	Allows zipping and unzipping of files.

Using Java 8 Features with Selenium

The Stream API is a new addition to the Collections API in Java 8. The Stream API brings new ways to process collections of objects. A stream represents a sequence of elements and supports different kinds of operations (filter, sort, map, and collect) from a collection. You can chain these operations together to form a pipeline to query the data, as shown in this diagram:



Example: Print all members of a collection:
MyCollection`.stream().forEach(System.out::println);`

Example for using a filter on "`isDisplayed()`", showing first the number of all links, and then just the number of displayed links:

```
List<WebElement> links = driver.findElements(By.tagName("a"));
System.out.println("Total Links : " + links.size());
long count = links.stream().filter(item -> item.isDisplayed()).count();
System.out.println("Total Link visible " + count);
```

This example collects all HTML links (anchor tags) into a collection of WebElements. It then maps the WebElements collection to a new collection of Strings and prints them to the console:

```
List<WebElement> linkElements = driver.findElements(By.tagName("a"));
List<String> linkTexts =
       linkElements.stream().map(WebElement::getText).collect(Collectors.toList());
linkTexts.stream().forEach(System.out::println);
```