ACTIONS API

A low-level interface for providing virtualized device input actions to the web browser. In addition to the high-level element interactions, the Actions API provides granular control over exactly what designated input devices can do. Selenium provides an interface for 3 kinds of input sources: a key input for keyboard devices, a pointer input for a mouse, pen or touch devices, and wheel inputs for scroll wheel devices (introduced in Selenium 4.2). Selenium allows you to construct individual action commands assigned to specific inputs and chain them together and call the associated perform method to execute them all at once.

Action Builder:

In the move from the legacy JSON Wire Protocol to the new W3C WebDriver Protocol, the low level building blocks of actions became especially detailed. It is extremely powerful, but each input device has a number of ways to use it and if you need to manage more than one device, you are responsible for ensuring proper synchronization between them.

Pause:

Pointer movements and Wheel scrolling allow the user to set a duration for the action, but sometimes you just need to wait a beat between actions for things to work correctly.

```
WebElement clickable = driver.findElement(By.id("clickable"));
        new Actions(driver)
                .moveToElement(clickable)
                .pause(Duration.ofSeconds(1))
                .clickAndHold()
                .pause(Duration.ofSeconds(1))
                .sendKeys("abc")
                .perform();
```

Release All Actions:

An important thing to note is that the driver remembers the state of all the input items throughout a session. Even if you create a new instance of an actions class, the depressed keys and the location of the pointer will be in whatever state a previously performed action left them.

There is a special method to release all currently depressed keys and pointer buttons. This method is implemented differently in each of the languages because it does not get executed with the perform method.

```
((RemoteWebDriver) driver).resetInputState();
```

Key Board Actions

A representation of any key input device for interacting with a web page. There are only 2 actions that can be accomplished with a keyboard: pressing down on a key, and releasing a pressed key. In addition to supporting ASCII characters, each keyboard key has a representation that can be pressed or released in designated sequences.

Keys:

In addition to the keys represented by regular Unicode, Unicode values have been assigned to other keyboard keys for use with Selenium. Each language has its own way to reference these keys.

Use the Java Keys Enum

```
   NULL        ('\uE000’)
   CANCEL      ('\uE001')
// Number pad keys
   NUMPAD0     ('\uE01A')
   NUMPAD1     ('\uE01B')
// Function keys
   F1          ('\uE031')
   F2          ('\uE032')
```

Key down:

```
new Actions(driver)
        .keyDown(Keys.SHIFT)
        .sendKeys("a")
        .perform();
```

Key up:

```
 new Actions(driver)
        .keyDown(Keys.SHIFT)
        .sendKeys("a")
        .keyUp(Keys.SHIFT)
        .sendKeys("b")
        .perform();
```

Send keys:

This is a convenience method in the Actions API that combines keyDown and keyUp commands in one action. Executing this command differs slightly from using the element method, but primarily this gets used when needing to type multiple characters in the middle of other actions.

Active Element:

```
new Actions(driver)
        .sendKeys("abc")
        .perform();
```

Designated Element:

```
new Actions(driver)
        .sendKeys(textField, "Selenium!")
        .perform();
```

Copy and Paste:

Here's an example of using all of the above methods to conduct a copy / paste action. Note that the key to use for this operation will be different depending on if it is a Mac OS or not. This code will end up with the text:SeleniumSelenium!

```
 Keys cmdCtrl = Platform.getCurrent().is(Platform.MAC) ? Keys.COMMAND : Keys.CONTROL;
```

```
WebElement textField = driver.findElement(By.id("textInput"));
        new Actions(driver)
                .sendKeys(textField, "Selenium!")
                .sendKeys(Keys.ARROW_LEFT)
                .keyDown(Keys.SHIFT)
                .sendKeys(Keys.ARROW_UP)
                .keyUp(Keys.SHIFT)
                .keyDown(cmdCtrl)
                .sendKeys("xvv")
                .keyUp(cmdCtrl)
                .perform();

        Assertions.assertEquals("SeleniumSelenium!", textField.getAttribute("value"));
```

Alerts

An Alert in Selenium is a small message box which appears on screento give the user some information or notification. It notifies the user with some specific information or error, asks for permission to perform certain tasks and it also provides warning messages as well.

In this tutorial, we will learn how to handle popup in Selenium and different types of alerts found in web application Testing. We will also see how to handle Alert in Selenium WebDriver and learn how do we accept and reject the alert depending upon the alert types.

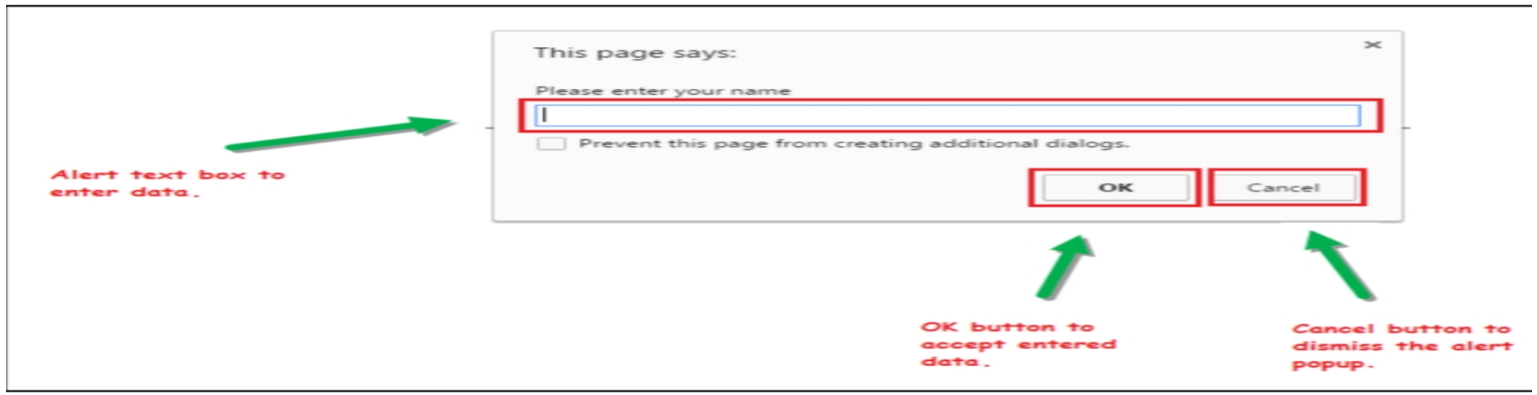
Types of Alerts in Selenium

1) Simple Alert

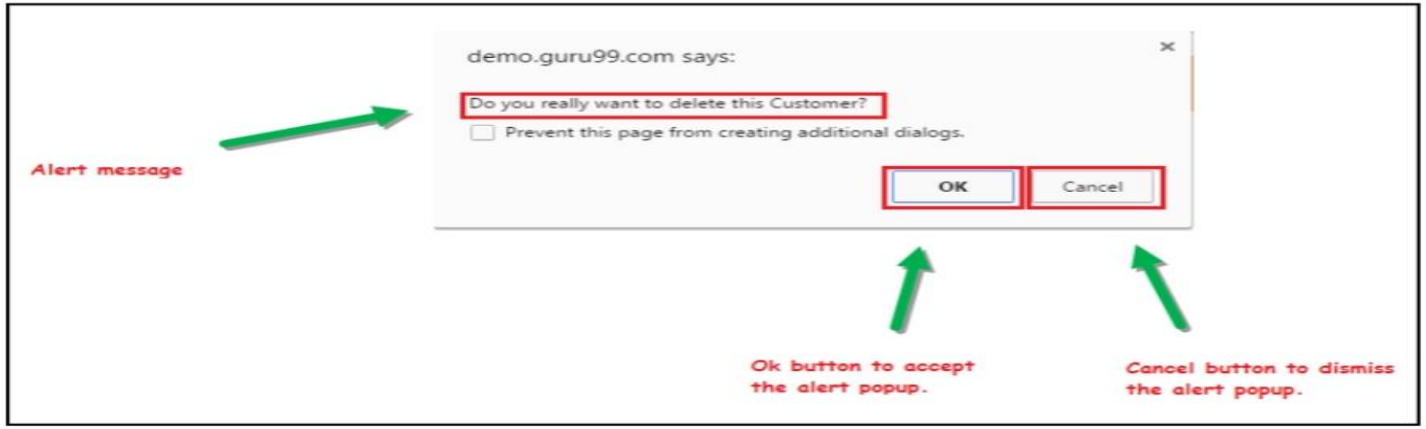
The simple alert class in Selenium displays some information or warning on the screen.

2) Prompt Alert:

This Prompt Alert asks some input from the user and Selenium webdriver can enter the text using sendkeys(" input.... ").



3) **Confirmation Alert:** This confirmation alert asks permission to do some type of operation.



Finders

Finding web elements

Locating the elements based on the provided locator values.

One of the most fundamental aspects of using Selenium is obtaining element references to work with. Selenium offers a number of built-in locator strategies to uniquely identify an element. There are many ways to use the locators in very advanced scenarios.

For the purposes of this documentation, let’s consider this HTML snippet:

```
<ol id="vegetables">
 <li class="potatoes">…
 <li class="onions">…
 <li class="tomatoes"><span>Tomato is a Vegetable</span>…
</ol>
<ul id="fruits">
  <li class="bananas">…
  <li class="apples">…
  <li class="tomatoes"><span>Tomato is a Fruit</span>…
</ul>
```

First matching element:

Many locators will match multiple elements on the page. The singular find element method will return a reference to the first element found within a given context.

Evaluating entire DOM:

When the find element method is called on the driver instance, it returns a reference to the first element in the DOM that matches with the provided locator. This value can be stored and used for future element actions. In our example HTML above, there are two elements that have a class name of “tomatoes” so this method will return the element in the “vegetables” list.

```
WebElement vegetable = driver.findElement(By.className("tomatoes"));
```

Evaluating a subset of the DOM:

Rather than finding a unique locator in the entire DOM, it is often useful to narrow the search to the scope of another located element. In the above example there are two elements with a class name of “tomatoes” and it is a little more challenging to get the reference for the second one.

One solution is to locate an element with a unique attribute that is an ancestor of the desired element and not an ancestor of the undesired element, then call find element on that object:

```
WebElement fruits = driver.findElement(By.id("fruits"));
WebElement fruit = fruits.findElement(By.className("tomatoes"));
```

Optimized locator:

A nested lookup might not be the most effective location strategy since it requires two separate commands to be issued to the browser. To improve the performance slightly, we can use either CSS or XPath to find this element in a single command. See the Locator strategy suggestions in our Encouraged test practices section.

For this example, we’ll use a CSS Selector:

```
WebElement fruit = driver.findElement(By.cssSelector("#fruits .tomatoes"));
```

All matching elements:

There are several use cases for needing to get references to all elements that match a locator, rather than just the first one. The plural find elements methods return a collection of element references. If there are no matches, an empty list is returned. In this case, references to all fruits and vegetable list items will be returned in a collection.

```
List<WebElement> plants = driver.findElements(By.tagName("li"));
```

Get element:

Often you get a collection of elements but want to work with a specific element, which means you need to iterate over the collection and identify the one you want.

```
List<WebElement> elements = driver.findElements(By.tagName("li"));
```

Find Elements From Element:

It is used to find the list of matching child WebElements within the context of parent element. To achieve this, the parent WebElement is chained with ‘findElements’ to access child elements.

// Get element with tag name 'div'

```
WebElement element = driver.findElement(By.tagName("div"));
```

// Get all the elements available with tag name 'p'

```
List<WebElement> elements = element.findElements(By.tagName("p"));
```

Get Active Element

It is used to track (or) find DOM element which has the focus in the current browsing context.

// Get attribute of current active element

```
String attr = driver.switchTo().activeElement().getAttribute("title");
```