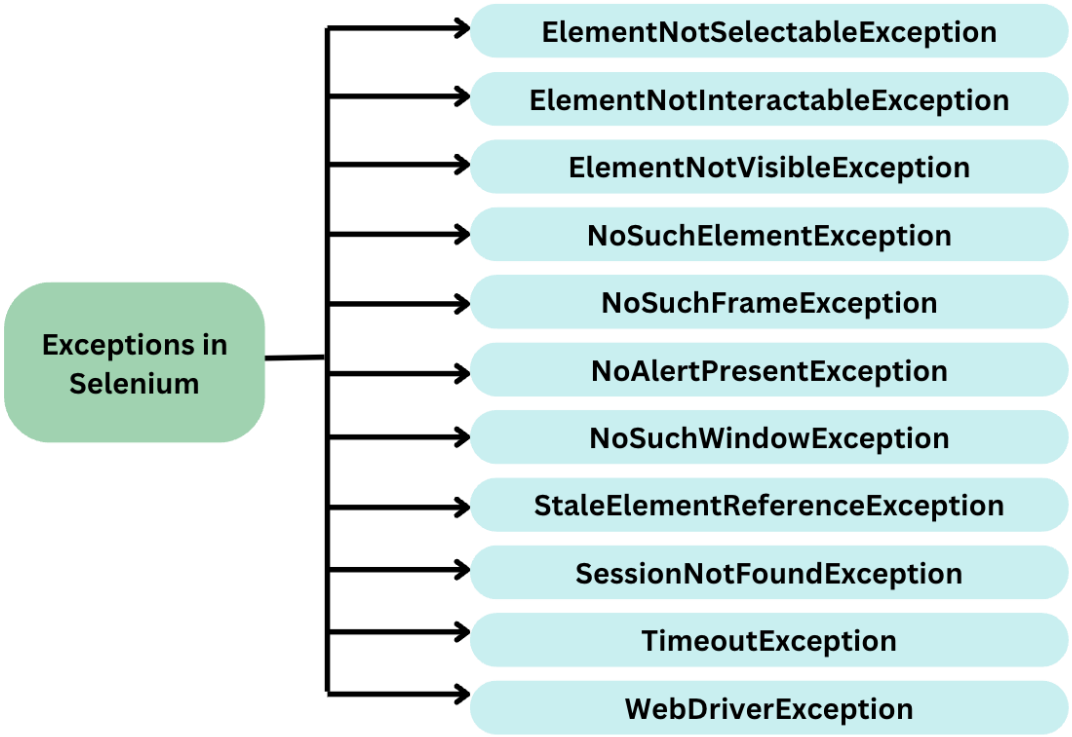
11 Common Exceptions in Selenium WebDriver



1. **ElementNotSelectableException:** This exception belongs to InvalidElementStateException class. This exception occurs when the web element is present on the web page but cannot be selected.**Solution:**
    - Try for other interfaces to select the element (**selectByIndex**, **selectByVisibleText**).
    - A wait command can also be added to wait until the element becomes clickable.

2. **ElementNotInteractableException:** This exception is thrown when an element is present in DOM, but it is not interactable, like unable to perform any action on it such as clicking or sending keys. This happens probably when the element to be interacted with is either hidden or disabled.**Solution:**

- Wait until the element is visible or clickable using Explicit wait.
- Scroll until the element gets in display using the Actions class.
- Use JS Executor to interact directly with the DOM.

3. **ElementNotVisibleException:** Selenium throws ElementNotVisibleException when an element is present in the DOM, but it is not visible to the user.**Solution:**

- Try to write unique locators that match with a single element only.
- Wait until the element is visible or clickable using explicit wait.

4. **NoSuchElementException:** This is the most common Selenium exception which is a subclass of NotFoundException class and thrown by findElement() method of Selenium WebDriver. This exception occurs when the locators defined in the Selenium program is unable to locate the element in the DOM.**Solution:**

- Reverify the locator by inspecting the browser and checking whether the element is present on DOM or not. Try switching to more reliable locators.
- Wait for the element to load using explicit wait.

5. **NoSuchFrameException:** This exception occurs when Selenium is unable to locate the desired frame or iframe using the specified iframe identifier (By iframe id, name or index). Iframe is a web page inside a web page and to work with the iframe elements, we need to first switch to the desired iframe. This exception triggers when Selenium is unable to find the iframe or if it is switching to an invalid iframe.**Solution:**

- Use wait after the action which triggers to open the iframe to ensure the iframe is loaded properly.

- Ensure the iframe locator is correct. (Switch between iframe name, id or index and recheck)

6. **NoAlertPresentException:** This exception occurs when Selenium tries to switch to an alert box which is not available on the web page which means the driver is switching to an invalid or non-existing Alert pop up. Like iframe, the driver first needs to switch to the desired Alert box to interact with it and then perform actions on it such as clicking on OK/ Submit/ Cancel button or fetching Alert message.**Solutions:**

- Use wait after the action which triggers to open the Alert to ensure it is loaded properly before interacting.
- Ensure alert locator is correct and visible on DOM by inspected browser. Try with different alert locators.

7. **NoSuchWindowException:** Selenium throws this exception when the WebDriver cannot find the desired browser window or tab using the specified window handle or name. This may occur when the window browser or tab we are attempting to work with is either not present, has been closed during the execution or is not completely loaded.**Solutions:**

- Ensure that the window handle or name being used is accurate.
- Wait for the browser window or tab to completely load and then switch the WebDriver to the desired window instance.

8. **StaleElementReferenceException:** This exception pops up when the element that was referenced by the locator is no longer present in the DOM or has become stale. It is a runtime exception that occurs when the page gets dynamically loaded, deleting the referenced element completely from the DOM, so that it becomes stale.**Solutions:**

- Refresh the page before accessing the element that causes StaleElementException.
- Use try-catch block to handle the exception and attempt to locate the element again in the catch block.

9. **SessionNotFoundException:** SessionNotFoundException will occur when the Webdriver is trying to perform actions on the web application after the browser is closed or when the browser session is not available.**Solution:** To handle such exceptions, we need to revisit our code and check if the code is not accidentally closing the browser. We need to make sure that the browser remains active throughout the execution and should be closed only at the end of the execution.

10. **TimeoutException:** In Selenium, TimeOutException occurs when a command takes longer than the wait time to avoid the ElementNotVisible Exception. Due to certain environment conditions such as low internet speed or web application taking more time than usual to completely load, the element to be interacted with does not load. And in such conditions when the WebDriver tries to find the element on the webpage, **TimeoutException** occurs.**Solutions:**

- Check the load time of the web element manually and add wait accordingly.
- Add explicit wait using JavaScript executor until the page is loaded.
- Try using some other property to locate the element such as CSS Selector or Xpath.

11. **WebDriverException:** This exception occurs when the WebDriver is acting immediately after closing the driver session.**Solution:** Use **driver.close()** after the completion of all tests at the suite level and not at the test level. If using TestNG, use **driver.close()** in **@AfterSuite** and not in **@AfterTest**.

Handling Exceptions In Selenium WebDriver

Following are a few standard ways using which one can handle Exceptions in Selenium WebDriver:

Try-catch

This method can catch Exceptions by using a combination of the try and catch keywords. ***Try*** indicates the start of the block, and ***Catch*** is placed at the end of the try block to handle or resolve the Exception. The code that is written within the Try/Catch block is referred to as “protected code.” The following code represents the syntax of Try/Catch block –

```
try

   {

     // Some code

   }

catch(Exception e)

  {

     // Code for Handling the exception

  }
```

Multiple catch blocks

There are various types of Exceptions, and one can expect more than one exception from a single block of code. Multiple catch blocks are used to handle every kind of Exception separately with a separate block of code. One can use more than two catch blocks, and there is no limitation on the number of catch blocks. The code below represents the syntax of multiple catch blocks –

```
try

   {

      //Some code

   }

catch(ExceptionType1 e1)

   {

     //Code for Handling the Exception 1

   }

catch(ExceptionType2 e2)

   {

     //Code for Handling the Exception 2

   }
```

Throw / Throws

When a programmer wants to generate an Exception explicitly, the Throw keyword is used to throw Exception to runtime to handle it. When a programmer is throwing an Exception without handling it, then he/she needs to use Throws keyword in the method signature to enable the caller program to understand the exceptions that might be thrown by the method. The syntax for Throws is as follows:

```
public static void anyFunction() throws Exception

{

try

   {

     // write your code here

   }

catch (Exception e)

   {

      // Do whatever you wish to do here



      // Now throw the exception back to the system

      throw(e);

   }

}
```

Multiple Exceptions

One can mention various Exceptions in the throws clause. Refer to the example below:

```
public static void anyFunction() throws ExceptionType1, ExceptionType2



{

 try

   {

     // write your code here

   }

 catch (ExceptionType1 e1)

   {

    // Code to handle exception 1

   }

 catch (ExceptionType1 e2)

   {

    // Code to handle exception 2

   }

}
```

Finally

The Finally keyword is used to create a block of code under the try block.
This **finally** code block always executes irrespective of the occurrence of an exception

```
try

  {

    //Protected code

  }

catch(ExceptionType1 e1)

  {

    //Catch block

  }

catch(ExceptionType2 e2)

  {

    //Catch block

  }

catch(ExceptionType3 e3)

  {

    //Catch block

  }



finally

   {

     //The finally block always executes.

   }
```

One can also use the following methods to display Exception Information:

- **printStackTrace()**: It prints the stack trace, name of the exception, and other useful description
- **toString()**: It returns a text message describing the exception name and description
- **getMessage()**: It displays the description of the exception

Conclusion

- Exception handling is a very crucial part of every Selenium Script
- Exceptions in Selenium can not be ignored as they disrupt the normal execution of programs
- One can write optimal and robust scripts by handling the Exceptions in unique ways
- Programmers can handle standard exceptions using various techniques like Try/catch, Multiple catch blocks, Finally, and others depending upon the requirement of scripts.
- For analyzing the exceptions in detail, one can also use methods like **printStackTrace()**, **toString()**, and **getMessage()**