SELENIUM TUTORIAL

Selenium is one of the most widely used open source Web UI automation testing tools. It supports automation of websites across different browsers, platforms and programming languages. Our tutorials are designed for beginners with basic knowledge of programming language - Java. We will start with the basics of Selenium and then as the tutorial progresses we will move to the more advanced stuff.

RIGHT CLICK IN SELENIUM WEBDRIVER

Hello friends, quite often during automation we need to right click or context click an element. Later, this action is followed up by pressing the UP/DOWN arrow keys and ENTER key to select the desired context menu element (check our tutorial on pressing the non-text keys in selenium - Pressing ARROW KEYS, FUNCTION KEYS and other non-text keys in Selenium). For right clicking an element in Selenium, we make use of the Actions class. The Actions class provided by Selenium Webdriver is used to generate complex user gestures including right click, double click, drag and drop etc.

CODE SNIPPET TO RIGHT CLICK AN ELEMENT

```
Actions action = new Actions(driver);
WebElement element = driver.findElement(By.id("elementId"));
action.contextClick(element).perform();
```

Here, we are instantiating an object of Actions class. After that, we pass the WebElement to be right clicked as parameter to the contestClick() method present in the Actions class. Then, we call the perform() method to perform the generated action.

SAMPLE CODE TO RIGHT CLICK AN ELEMENT

For the demonstration of the right click action, we will be launching Sample Site for Selenium Learning. Then we will right-click a textbox after which it's context menu will get displayed, asserting that right click action is successfully performed.

```
package seleniumTutorials;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.interactions.Actions;

public class RightClick {

	public static void main(String[] args) throws InterruptedException{
		WebDriver driver = new FirefoxDriver();

		//Launching Sample site
		driver.get("http://artoftesting.com/sampleSiteForSelenium.html");

		//Right click in the TextBox
		Actions action = new Actions(driver);
		WebElement searchBox = driver.findElement(By.id("fname"));
		action.contextClick(searchBox).perform();

		//Thread.sleep just for user to notice the event
		Thread.sleep(3000);

		//Closing the driver instance
		driver.quit();
	}


}
```

DOUBLE CLICK IN SELENIUM WEBDRIVER

Hello friends! in this post, we will learn to double click an element using Selenium Webdriver with Java. For double clicking an element in Selenium we make use of the Actions class. The Actions class provided by Selenium Webdriver is used to generate complex user gestures including right click, double click, drag and drop etc.

CODE SNIPPET TO DOUBLE CLICK AN ELEMENT

```
Actions action = new Actions(driver);
WebElement element = driver.findElement(By.id("elementId"));
action.doubleClick(element).perform();
```

Here, we are instantiating an object of Actions class. After that, we pass the WebElement to be double clicked as parameter to the doubleClick() method present in the Actions class. Then, we call the perform() method to perform the generated action.

SAMPLE CODE TO DOUBLE CLICK AN ELEMENT

For the demonstration of the double click action, we will be launching Sample Site for Selenium Learning. Then we will double click the button on which the

text "Double-click to generate alert box" is written. After that an alet box will appear, asserting that double-click action is successfully performed.

```
package seleniumTutorials;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.interactions.Actions;

public class DoubleClick {

	public static void main(String[] args) throws InterruptedException{
		WebDriver driver = new FirefoxDriver();

		//Launching sample website
		driver.get("http://artoftesting.com/sampleSiteForSelenium.html");
		driver.manage().window().maximize();

		//Double click the button to launch an alertbox
		Actions action = new Actions(driver);
		WebElement btn = driver.findElement(By.id("dblClkBtn"));
		action.doubleClick(btn).perform();

		//Thread.sleep just for user to notice the event
		Thread.sleep(3000);

		//Closing the driver instance
		driver.quit();
	}

}
```

MOUSEOVER IN SELENIUM WEBDRIVER

Hello friends! in this post, we will learn to automate the mouseover over an element using Selenium Webdriver with Java. For performing the mouse hover over an element in Selenium, we make use of the Actions class. The Actions class provided by Selenium Webdriver is used to generate complex user gestures including mouseover, right click, double click, drag and drop etc.

CODE SNIPPET TO MOUSEOVER

```
Actions action = new Actions(driver);
WebElement element = driver.findElement(By.id("elementId"));
action.moveToElement(element).perform();
```

Here, we are instantiating an object of Actions class. After that, we pass the WebElement to be mouse hovered as parameter to the moveToElement() method present in the Actions class. Then, we will call the perform() method to perform the generated action.

SAMPLE CODE TO MOUSE HOVER OVER AN ELEMENT

For the demonstration of the mouseover action, we will be launching Sample Site for Selenium Learning. Then we will mouse hover over the 'Submit' button. This will resut in a tooltip generation, asserting that the mouseover action is successfully performed.

```
package seleniumTutorials;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.interactions.Actions;

public class MouseOver {

	public static void main(String[] args) throws InterruptedException{
		WebDriver driver = new FirefoxDriver();

		//Launching sample website
		driver.get("http://artoftesting.com/sampleSiteForSelenium.html");
		driver.manage().window().maximize();

		//Mouseover on submit button
		Actions action = new Actions(driver);
		WebElement btn = driver.findElement(By.id("idOfButton"));
		action.moveToElement(btn).perform();

		//Thread.sleep just for user to notice the event
		Thread.sleep(3000);

		//Closing the driver instance
		driver.quit();
	}

}
```

DRAG AND DROP IN SELENIUM WEBDRIVER

Drag and Drop is one of the common scenarios in automation. In this tutorial, we are going to study the handling of drag and drop event in Selenium WebDriver using **Actions** class. For practicing, you can check the dummy page having a draggable web element.

ACTIONS IN SELENIUM WEBDRIVER

For performing a complex user gestures like drag and drop, we have a **Actions** class in Selenium WebDriver. Using the Actions class, we first build a sequence of composite events and then perform it using **Action** (an interface which represents a single user-interaction). The different methods of Actions class we will be using here are-

- **clickAndHold(WebElement element)** - Clicks a web element at the middle(without releasing).
- **moveToElement(WebElement element)** - Moves the mouse pointer to the middle of the web element without clicking.
- **release(WebElement element)** - Releases the left click (which is in pressed state).
- **build()** - Generates a composite action

CODE SNIPPET TO PERFORM DRAG AND DROP

Below is the code snippet for performing drag and drop operation-

```
//WebElement on which drag and drop operation needs to be performed
WebElement fromWebElement = driver.findElement(By Locator of fromWebElement);

//WebElement to which the above object is dropped
WebElement toWebElement = driver.findElement(By Locator of toWebElement);

//Creating object of Actions class to build composite actions
Actions builder = new Actions(driver);

//Building a drag and drop action
```

```
Action dragAndDrop = builder.clickAndHold(fromWebElement)
                      .moveToElement(toWebElement)
                      .release(toWebElement)
              .build();

//Performing the drag and drop action
dragAndDrop.perform();
```

SCREENSHOTS IN SELENIUM WEBDRIVER

While doing manual testing, we always have the machine in front of us to check what happened when the test case failed. In automation, we rely on the assertion messages that we print in case of failure. In addition to that, we can also have screenshot of the browser in case of failure due to assertion or unavailability of any web element.

HOW TO TAKE SCREESHOT

The code to take the screenshot make use of **getScreenshotAs** method of **TakesScreenshot** interface. Following code will take screenshot of the web page opened by webDriver instance.

```
File scrFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
FileUtils.copyFile(scrFile, new File("D:\\testScreenShot.jpg"));
```

TAKE SCREENSHOT ON FAILURE

Now in order to take screenshot in case of test failure we will use @AfterMethod annotation of TestNG. In the @AfterMethod annotation we will use **ITestResult**interface's **getStatus()** method that returns the test result and in case of failure we can use the above commands to take screenshot. One more thing to mention here is in order to uniquely identify the screenshot file, we are naming it as the name of the test method appended with the test parameters (passed through a data-provider). For getting the test name and parameters we are using the getName() and getParameters() methods of ITestResult interface. In case you are not using any data-provider(like in case of this demo) then you can just have the getName() method to print the test method name.

```
@AfterMethod
public void takeScreenShotOnFailure(ITestResult testResult) throws IOException {
        if (testResult.getStatus() == ITestResult.FAILURE) {
```

```
                File scrFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.
FILE);
                FileUtils.copyFile(scrFile, new File("errorScreenshots\\" + testResu
lt.getName() + "-"
                                        + Arrays.toString(testResult.getParameters()) +  ".j
pg"));
        }
}
```

DEMO WITH GOOGLE CALCULATOR

Following is the complete sample script for google calculator test that is intentionally made to fail by asserting result for 2+2 as 5. Just change the path of the screenshot file to the desired location and run the test script.

```
public class ScreenshotDemo{

        String driverExecutablePath = "lib\\chromedriver.exe";
        WebDriver driver;

        @BeforeTest
        public void setup(){
                System.setProperty("webdriver.chrome.driver", driverExecutablePath);
                driver = new ChromeDriver();

                //Set implicit wait of 3 seconds
                driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);
        }


        @Test
        //Tests google calculator
        public void googleCalculator() throws IOException{

                //Launch google
                driver.get("http://www.google.co.in");

                //Write 2+2 in google textbox
                WebElement googleTextBox = driver.findElement(By.id("lst-ib"));
                googleTextBox.sendKeys("2+2");

                //Hit enter
                googleTextBox.sendKeys(Keys.ENTER);
```

```
                //Get result from calculator
                WebElement calculatorTextBox = driver.findElement(By.id("cwtltblr"))
;
                String result = calculatorTextBox.getText();

                //Intentionaly checking for wrong calculation of 2+2=5 in order to t
ake screenshot for failing test
                Assert.assertEquals(result, "5");
        }

        @AfterMethod
        public void takeScreenShotOnFailure(ITestResult testResult) throws IOExceptio
n {
                if (testResult.getStatus() == ITestResult.FAILURE) {
                        System.out.println(testResult.getStatus());
                        File scrFile = ((TakesScreenshot)driver).getScreenshotAs(Out
putType.FILE);
                        FileUtils.copyFile(scrFile, new File("errorScreenshots\\" +
testResult.getName() + "-"
                                        + Arrays.toString(testResult.getParameters(
)) +  ".jpg"));
                }
        }

}
```

PRESS KEYBOARD KEYS IN SELENIUM

During automation, we are often required to press enter, control, tab, arrow keys, function keys and other non-text keys as well from keyboard. In this post, we will find how to simulate pressing of these non-text keys using selenium webdriver in java. Here, we will be using **Keys** enum provided by Selenium webdriver for all the non-text keys.

PRESS ENTER/RETURN KEY IN SELENIUM

For pressing Enter key over a textbox we can pass Keys.ENTER or Keys.RETURN to the sendKeys method for that textbox.

```
WebElement textbox = driver.findElement(By.id("idOfElement"));
textbox.sendKeys(Keys.ENTER);
```

or

```
WebElement textbox = driver.findElement(By.id("idOfElement"));
textbox.sendKeys(Keys.RETURN);
```

Similarly, we can use Keys enum for different non-text keys and pass them to the sendKeys method. The following table has an entry for each of the non-text key present in a keyboard.

Keyboard's Key	Keys enum's value
Arrow Key - Down	Keys.ARROW_DOWN
Arrow Key - Up	Keys.ARROW_LEFT

Arrow Key - Left	Keys.ARROW_RIGHT
Arrow Key - Right	Keys.ARROW_UP
Backspace	Keys.BACK_SPACE
Ctrl Key	Keys.CONTROL
Alt key	Keys.ALT
DELETE	Keys.DELETE
Enter Key	Keys.ENTER
Shift Key	Keys.SHIFT
Spacebar	Keys.SPACE
Tab Key	Keys.TAB
Equals Key	Keys.EQUALS

Esc Key	Keys.ESCAPE
Home Key	Keys.HOME
Insert Key	Keys.INSERT
PgUp Key	Keys.PAGE_UP
PgDn Key	Keys.PAGE_DOWN
Function Key F1	Keys.F1
Function Key F2	Keys.F2
Function Key F3	Keys.F3
Function Key F4	Keys.F4
Function Key F5	Keys.F5
Function Key F6	Keys.F6

Function Key F7	Keys.F7
Function Key F8	Keys.F8
Function Key F9	Keys.F9
Function Key F10	Keys.F10
Function Key F11	Keys.F11
Function Key F12	Keys.F12

KEYBOARD ACTIONS IN SELENIUM WEBDRIVER

In our beginner's tutorials, we have seen the sendKeys() method which simulates the keyboard typing action on a textbox or input type element. But this method is not sufficient for handling complex keyboard actions. For this, Selenium has an **Actions**class which provides different methods for Keyboard interactions. In this tutorial, we wil be studying the three actions for handling keyboard action - keyDown(), keyUp() and sendKeys() along with their overloaded method implementations.

ACTIONS CLASS METHOD FOR KEYBOARD INTERACTION

1. **keyDown(Keys modifierKey)-**
   The keyDown(Keys modifierKey) method takes the modifier Keys as parameter (Shift, Alt and Control Keys - that modifies the purpose of other keys, hence the name). It is used to simulate the action of pressing a modifier key, without releasing. The expected values for the keyDown() method are - Keys.SHIFT, Keys.ALT and Keys.CONTROL only, passing key other than these results in IllegalArgumentException.

2. **keyDown(WebElement element, Keys modifierKey)-**
   This another implementation of keyDown() method in which the modifier key press action is performed on a WebElement.

3. **keyUp(Keys modifierKey)-**
   The keyUp() method is used to simulate the modifier key-up or key-release action. This method follows a preceeding key press action.

4. **keyUp(WebElement element, Keys modifierKey)-**
   This implementation of keyUp() method performs the key-release action on a web element.

5. **sendKeys(CharSequence KeysToSend)-**
   The sendKeys(CharSequence KeysToSend) method is used to send a sequence of keys to a currently focussed web element. Here, we need to note that it is different from the webElement.sendKeys() method. The Actions sendKeys(CharSequence KeysToSend) is particularly helpful when dealing with modifier keys as it doesn't release those keys when passed(resulting in correct behaviour) unlike the

webElement.sendKeys() method.

6. **sendKeys(WebElement element, CharSequence KeysToSend)-** This implementation of sendKeys() method is used to send a sequence of keys to a web element.

CODE SNIPPET FOR KEYBOARD ACTIONS

```
//WebElement to which the keyboard actions are performed
WebElement textBoxElement = driver.findElement(By Locator of textBoxElement);

//Creating object of Actions class
Actions builder = new Actions(driver);

//Generating an action to type a text in CAPS
Action typeInCAPS = builder.keyDown(textBoxElement, Keys.SHIFT)
                    .sendKeys(textBoxElement, "artOfTesting")
                    .keyUp(textBoxElement, Keys.SHIFT)
             .build();

//Performing the typeInCAPS action
typeInCAPS.perform();
```

MOUSE ACTIONS IN SELENIUM WEBDRIVER

In this tutorial, we will be studying the advanced mouse interactions using **Actions**class. Using these methods,we can perform mouse operations like right click, double click, mouse hover, click and hold etc.

ACTIONS CLASS METHOD FOR MOUSE INTERACTIONS

1. **click()-**
   This method is used to click at the current mouse pointer position. It is particularly useful when used with other mouse and keyboard events, generating composite actions.

2. **click(WebElement webElement)-**
   This method is used to click at the middle of a web element passed as parameter to the click() method.

3. **clickAndHold()-**
   The clickAndHold() method is used to perform the click method without releasing the mouse button.

4. **clickAndHold(WebElement onElement)-**
   This method performs the click method without releasing the mouse button over a web element.

5. **contextClick()-**
   This method is used to perform the right click operation(context-click) at the current mouse position.

6. **contextClick(WebElement onElement)-**
   This method performs the right click operation at a particular web element.

7. **doubleClick()-**
   As the name suggest, this method performs double click operation at a current mouse position.

8. **doubleClick(WebElement onElement)-**
   Performs the double click operation at a particular web element.

9. **dragAndDrop(WebElement fromElement, WebElement toElement)-**
   This is a utility method to perform the dragAndDrop operation directly wherein, we can pass the source element and the target element as parameter.

10. **dragAndDropBy(WebElement fromElement, int xOffset, int yOffset)-**
    This method is a variation of dragAndDrop(fromElement, toElement) in which instead of passing the target element as parameter, we pass the x and y offsets. The method clicks the source web element and then releases at the x and y offsets.

11. **moveByOffset(int xOffset, int yOffset)-**
    This method is used to move the mouse pointer to a particular position based on the x and y offsets passed as parameter.

12. **moveToElement(WebElement toElement)-**
    This method is used to move the mouse pointer to a web element passed as parameter.

13. **moveToElement(WebElement toElement, int xOffset, int yOffset)-**
    This method moves the mouse pointer by the given x and y offsets from the top-left corner of the specified web element.

14. **release()-**
    This method releases the pressed left mouse button at the current mouse pointer position.

15. **release(WebElement onElement)-**
    This method release the pressed left mouse button at a particular web element.

CODE SNIPPET FOR A MOUSE ACTION

```
//WebElement which needs to be right clicked
WebElement rtClickElement = driver.findElement(By Locator of rtClickElement);

//Generating a Action to perform context click or right click
Actions rightClickAction = new Actions(driver).contextClick(rtClickElement);

//Performing the right click Action generated
rightClickAction.build().perform();
```

PAGE OBJECT MODEL - POM

In this tutorial, we will study about "Page Object Model", a design pattern in UI automation testing. Before starting with Page Object Model, let's first know what are design patterns.

DESIGN PATTERN

A Design pattern is a generic solution to a common software design/architecture problem. Implementation of these design patterns leads to inclusion of best practices and best solution, evolved over the time by others while working with similar problems.

PAGE OBJECT MODEL IN SELENIUM

A Page Object Model is a design pattern that can be implemented using selenium webdriver. It essentially models the pages/screen of the application as objects called Page Objects, all the functions that can be performed in the specific page are encapsulated in the page object of that screen. In this way any change made in the UI will only affect that screens page object class thus abstracting the changes from the test classes.

ADVANTAGES OF USING PAGE OBJECT MODEL

- Increases code reusability - code to work with events of a page is written only once and used in different test cases
- Improves code maintainability - any UI change leads to updating the code in page object classes only leaving the test classes unaffected
- Makes code more readable and less brittle

CREATING A PAGE OBJECT MODEL IN JAVA

Here, we'll create an automation framework implementing Page Object Model using Selenium with Java.
Suppose we have to test a dummy application with only a login page and a home page. To start with we first need to create page objects for all available pages in our application - LoginPage.java and HomePage.java. Then we will create a test class that will create instance of these page objects and invoke there methods to create tests.

Let's take the scenario where in the login page user enters valid credentials and on clicking submit button, user is redirected to home page.

CONTENT OF LOGINPAGE.JAVA

```
public class LoginPage {

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    //Using FindBy for locating elements
    @FindBy(id = "userName")
    private WebElement userName;

    @FindBy(id = "password")
    private WebElement password;

    @FindBy(id = "submitButton")
    private WebElement submit;

    /*Defining all the user actions that can be performed in the loginPage
    in the form of methods*/

    public void typeUserName(String text) {
        userName.sendKeys(text);
    }

    public void typePassword(String text) {
        password.sendKeys(text);
    }

    /*Take note of return type of this method, clicking submit will navigate
    user to Home page, so return type of this method is marked as HomePage.*/
    public HomePage clickSubmit() {
        submit.click();
        return new HomePage(driver);
    }

    public HomePage loginWithValidCredentials(String userName, String pwd) {
        typeUserName(userName);
        typePassword(pwd);
        return clickSubmit();
    }
}
```

CONTENT OF HOMEPAGEPAGE.JAVA

```
public class HomePage {

    public HomePage(WebDriver driver) {
        this.driver = driver;
    }

    @FindBy(id = "userInfo")
    private WebElement userInfo;

    public void clickUserInfo(){
        userInfo.click();
    }

    public String showUserInfo(){
        String userData = clickUserInfo();
        return userData;
    }
}
```

CONTENT OF TEST CLASS - POMTEST.JAVA

```
public class POMTest{

    //Create firefox driver's instance
    WebDriver driver = new FirefoxDriver();

    @Test
    public void verifyUserInfo() {

    //Creating instance of loginPage
    LoginPage loginPage = new LoginPage(driver);

    //Login to application
    HomePage homePage = loginPage.loginWithValidCredentials("user1","pwd1");
```

```
    //Fetch user info
    String userInfo = homePage.showUserInfo();

    //Asserting user info
    Assert.assertTrue(userInfo.equalsIgnoreCase("XYZ"),"Incorrect userInfo");
    }
}
```

This was just a demo for understanding of Page Object Model, an actual project would require several updations like creating abstract classes for page objects, creating base classes for test classes, creating helper and utility classes for database connectivity, for passing test data through config files etc.

PAGEFACTORY IN SELENIUM WEBDRIVER

Hello friends! in our previous tutorial on Page Object Model, we studied about POM design pattern, its advantages and implementation in Selenium WebDriver. In this tutorial, we will study about PageFactory, an enhanced implementation of Page Object Model.

WHAT IS PAGEFACTORY?

PageFactory is class provided by Selenium WebDriver to support the Page Object design pattern. It makes handling "Page Objects" easier and optimized by providing the following-

- @FindBy annotation
- initElements method

@FINDBY-

@FindBy annotation is used in PageFactory to locate and declare web elements using different locators. Here, we pass the attribute used for locating the web element along with its value to the @FindBy annotation as parameter and then declare the element. Example-

```
@FindBy(id="elementId") WebElement element;
```

In the above example, we have used 'id' attribute to locate the web element 'element'. Similarly, we can use the following locators with @FindBy annotations.

- className
- css
- name
- xpath
- tagName
- linkText
- partialLinkText

INITELEMENTS()-

The initElements is a static method of PageFactory class which is used in conjunction with @FindBy annotation. Using the initElements method we can initialize all the web elements located by @FindBy annotation. Thus, instantiating the Page classes easily.

```
initElements(WebDriver driver, java.lang.Class pageObjectClass)
```

CODE SNIPPET FOR PAGEFACTORY

Below is a code snippet with a demo Page class. Here, the web elements are located by @FindBy annotation and the initElements method is invoked in the constructor of the PageFactoryDemoClass.
PS: The implementation of test classes will remain same (as stated in Page Object Model tutorial).

```
public class PageFactoryDemoClass {

    WebDriver driver;

    @FindBy(id="search")
    WebElement searchTextBox;

    @FindBy(name="searchBtn")
    WebElement searchButton;

    //Constructor invoking initElements method
    public PageFactoryDemoClass(WebDriver driver){
        this.driver = driver;

        //initializing all the web elements located by @FindBy
        PageFactory.initElements(driver, this);
    }

    //Sample method of the page class
    public void search(String searchTerm){
        searchTextBox.sendKeys(searchTerm);
        searchButton.click();
    }
}
```

DIFFERENCE BETWEEN ASSERT AND VERIFY

Both Assert and Verify statements are used in the test suites for adding validations to the test methods. Testing frameworks like TestNG and JUnit are used with Selenium to provide assertions.
The major difference between "Assert" and "Verify" commands is- In case of "Assert" command, as soon as the validation fails the execution of that particular test method is stopped and the test method is marked as failed.
Whereas, in case of "Verify", the test method continues execution even after the failure of an assertion statement. Although the test method will still be marked as failed but the remaining statements of the test method will be executed normally. In TestNG, the "Verify" functionality is provided by means of "Soft Assertions" or "SoftAssert" class.

Now, let's get deeper into Assert and Verify/Soft Assert.

ASSERT

We use Assert when we have to validate critical functionality, failing of which makes the execution of further statements irrelevant. Hence, the test method is aborted as soon as failure occurs. Example-

```
@Test
public void assertionTest(){

   //Assertion Passing
   Assert.assertTrue(1+2 == 3);
   System.out.println("Passing 1");

   //Assertion failing
   Assert.fail("Failing second assertion");
   System.out.println("Failing 2");
}
```

Output-

```
Passing 1
```

```
FAILED: assertionTest
java.lang.AssertionError: Failing second assertion
```

Here, we can observe that only the text "Passing 1" gets printed. The second assertion aborts the test method as it fails preventing further statement from getting executed.

VERIFY

At times, we might require the test method to continue execution even after the failure of the assertion statements. In TestNG, **Verify** is implemented using **SoftAssert** class.

In case of SoftAssert, all the statements in the test method are executed (including multiple assertions). Once, all the statements are executed, the test result is collated based on the assertion results and test is marked as passed or fail.

Example-

```
@Test
public void softAssertionTest(){

   //Creating softAssert object
   SoftAssert softAssert = new SoftAssert();

   //Assertion failing
   softAssert.fail("Failing first assertion");
   System.out.println("Failing 1");

   //Assertion failing
   softAssert.fail("Failing second assertion");
   System.out.println("Failing 2");

   //Collates the assertion results and marks test as pass or fail
   softAssert.assertAll();
}
```

Output-

```
Failing 1
Failing 2
FAILED: softAssertionTest
java.lang.AssertionError: The following asserts failed:
        Failing first assertion,
        Failing second assertion
```

Here, we can see that even though both the test methods are bound to fail, still the test continues to execute.

TESTNG INTRODUCTION

TestNG is a testing framework inspired from JUnit and NUnit but introducing some new functionality that make it more powerful and easier to use.

TestNG is an open source automated testing framework; where NG of TestNG means Next Generation. TestNG is similar to JUnit but it is much more powerful than JUnit but still it's inspired by JUnit. It is designed to be better than JUnit, especially when testing integrated classes. Pay special thanks to Cedric Beust who is the creator of TestNG.

TestNG eliminates most of the limitations of the older framework and gives the developer the ability to write more flexible and powerful tests with help of easy annotations, grouping, sequencing & parametrizing.

TestNG is a testing framework inspired from ***JUnit*** and ***NUnit*** but introducing some new functionality that make it more powerful and easier to use.
It is an open source automated testing framework; where **NG** OF TEST**NG** MEANS **N**EXT **G**ENERATION. TestNG is similar to JUnit but it is much more powerful than JUnit but still it's inspired by JUnit. It is designed to be better than JUnit, especially when testing integrated classes. Pay special thanks to *Cedric Beust who is the creator of TestNG*.
TestNG eliminates most of the limitations of the older framework and gives the developer the ability to write more flexible and powerful tests with help of easy annotations, grouping, sequencing & parametrizing.

BENEFITS OF TESTNG

There are number of benefits but from Selenium perspective, major advantages of TestNG are :

1. It gives the ability to produce ***HTML Reports*** of execution
2. ***Annotations*** made testers life easy
3. Test cases can be ***Grouped & Prioritized*** more easily
4. ***Parallel*** testing is possible
5. Generates ***Logs***
6. Data ***Parameterization*** is possible

TEST CASE WRITING

Writing a test in TestNG is quite simple and basically involves following steps:

Step 1 – Write the business logic of the test
Step 2 – Insert TestNG annotations in the code
Step 3 – Add the information about your test (e.g. the class names, methods names, groups names etc…) in a testng.xml file
Step 4 – Run TestNG

ANNOTATIONS IN TESTNG

@BeforeSuite: The annotated method will be run before all tests in this suite have run.
@AfterSuite: The annotated method will be run after all tests in this suite have run.
@BeforeTest: The annotated method will be run before any test method belonging to the classes inside the tag is run.
@AfterTest: The annotated method will be run after all the test methods belonging to the classes inside the tag have run.
@BeforeGroups: The list of groups that this configuration method will run before. This method is guaranteed to run shortly before the first test method that belongs to any of these groups is invoked.
@AfterGroups: The list of groups that this configuration method will run after. This method is guaranteed to run shortly after the last test method that belongs to any of these groups is invoked.
@BeforeClass: The annotated method will be run before the first test method in the current class is invoked.
@AfterClass: The annotated method will be run after all the test methods in the current class have been run.
@BeforeMethod: The annotated method will be run before each test method.
@AfterMethod: The annotated method will be run after each test method.
@Test: The annotated method is a part of a test case.

BENEFITS OF USING ANNOTATIONS

1. It identifies the methods it is interested in by looking up annotations. Hence method names are not restricted to any pattern or format.
2. We can pass additional parameters to annotations.
3. Annotations are strongly typed, so the compiler will flag any mistakes right away.
4. Test classes no longer need to extend anything (such as Test Case, for JUnit 3).

SELENIUM WEBDRIVER WITH TESTNG SAMPLE SCRIPT

Selenium WebDriver with TestNG in Java - in this example, we will test the Google Calculator feature using Selenium for UI automation and TestNG as testing framework. You can download this java file here calculatorTest.java (right click on 'calculatorTest.java' and click on 'save link as...' to save the sample selenium test script).

```
public class calculatorTest {

	@Test
	//Tests google calculator
	public void googleCalculator(){

		//Create firfox driver's instance
		WebDriver driver = new FirefoxDriver();

		//Set implicit wait of 10 seconds
		driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

		//Launch google
		driver.get("http://www.google.co.in");

		//Write 2+2 in google textbox
		WebElement googleTextBox = driver.findElement(By.id("gbqfq"));
		googleTextBox.sendKeys("2+2");
		//Click on searchButton
		WebElement searchButton = driver.findElement(By.id("gbqfb"));
		searchButton.click();
		//Get result from calculator
		WebElement calculatorTextBox = driver.findElement(By.id("cwos"));
		String result = calculatorTextBox.getText();

		//Verify that result of 2+2 is 4
		Assert.assertEquals(result, "4");
	}

}
```

TESTNG ANNOTATIONS

In this tutorial, we will be studying all the annotations of TestNG along with the different attributes supported. **An annotation is a tag or metadata that provides additional information about a class, interface or method.** TestNG make use of these annotations to provide several features that aid in creation of robust testing framework. Here, we will refer to each TestNG annotation in detail and study their syntax and usage.

@TEST

The @Test is the most important and commonly used annotation of TestNG. It is used to mark a method as Test. So, any method over which we see @Test annotation, is considered as a TestNG test.

```
@Test
public void sampleTest() {
   //Any test logic
   System.out.println("Hi! ArtOfTesting here!");
}
```

Now, let's see some important attributes of @Test annotations-

1. description - The 'description' attribute is used to provide a description to the test method. It generally contains a one-liner test summary.

```
@Test(description = "Test summary")
```

2. **dataProvider** - This attribute helps in creating a data driven tests. It is used to specify the name of the data provider for the test.

```
@Test(dataProvider = "name of dataProvider")
```

3. **priority** - This attribute helps in prioritizing the test methods. The default priority starts with 0 and tests execute in ascending order.

```
@Test(priority = 2)
```

4. **enabled** - This attribute is used to specify whether the given test method will run with the suite or class or not.

```
@Test(enabled = false)
```

5. **groups** - Used to specify the groups, the test method belongs to.

```
@Test(groups = { "sanity", "regression" })
```

7. **dependsOnMethods** - Used to specify the methods on which the test method depends. The test method only runs after successful execution of the dependent tests.

```
@Test(dependsOnMethods = { "dependentTestMethodName" })
```

8. **dependsOnGroups** - Used to specify the groups on which the test method depends.

```
@Test(dependsOnGroups = { "dependentGroup" })
```

9. **alwaysRun** - When set as True, the test method runs even if the dependent methods fail.

```
@Test(alwaysRun=True)
```

10. **timeOut** - This is used to specify a timeout value for the test(in milli seconds). If test takes more than the timeout value specified, the test terminates and is marked as failure.

```
@Test (timeOut = 500)
```

@BEFORESUITE

The annotated method will run only once before all tests in this suite have run.

@AFTERSUITE

The annotated method will run only once after all tests in this suite have run.

@BEFORECLASS

The annotated method will run only once before the first test method in the current class is invoked.

@AFTERCLASS

The annotated method will run only once after all the test methods in the current class have been run.

@BEFORETEST

The annotated method will run before any test method belonging to the classes inside the <test> tag is run.

@AFTERTEST

The annotated method will run after all the test methods belonging to the classes inside the <test> tag have run.

@DATAPROVIDER

Using @DataProvider we can create a data driven framework in which data is passed to the associated test method and multiple iteration of the test runs for the different test data values passed from the @DataProvider method. The method annotated with @DataProvider annotation return a 2D array or object.

```
//Data provider returning 2D array of 3*2 matrix
 @DataProvider(name = "dataProvider1")
   public Object[][] dataProviderMethod1() {
      return new Object[][] {{"kuldeep","rana"}, {"k1","r1"},{"k2","r2"}};
   }

   //This method is bound to the above data provider
   //The test case will run 3 times with different set of values
   @Test(dataProvider = "dataProvider1")
   public void sampleTest(String str1, String str2) {
      System.out.println(str1 + " " + str2);
   }
```

@PARAMETER

The @Parameter tag is used to pass parameters to the test scripts. The value of the @Parameter tag can be passed through testng.xml file. Sample testng.xml with parameter tag-

```
@Test (timeOut = 500)<suite name="sampleTestSuite">
   <test name="sampleTest">
      <parameter name="sampleParamName" value="sampleParamValue"/>
      <classes>
         <class name="TestFile" />
      </classes>
   </test>
</suite>
```

Sample Test Script with @Parameter annotation-

```
public class TestFile {
   @Test
   @Parameters("sampleParamName")
   public void parameterTest(String paramValue) {
      System.out.println("sampleParamName = " + sampleParamName);
```

```
    }
```

@LISTENER

TestNG provides us different kind of listners using which we can perform some action in case an event has triggered. Usually testNG listeners are used for configuring reports and logging.

```
@Listeners(PackageName.CustomizedListenerClassName.class)

public class TestClass {
    WebDriver driver= new FirefoxDriver();@Test
    public void testMethod(){
    //test logic
    }
}
```

@FACTORY

The @Factory annotation helps in dynamic execution of test cases. Using @Factory annotation, we can pass parameters to the whole test class at run time. The parameters passed can be used by one or more test methods of that class.
In the below example, the test class TestFactory will run the test method in TestClass with different set of parameters - 'k1' and 'k2'.

```
public class TestClass{
    private String str;

    //Constructor
    public TestClass(String str) {
        this.str = str;
    }

    @Test
    public void TestMethod() {
        System.out.println(str);
    }
}
```

```
public class TestFactory{
    //Because of @Factory, the test method in class TestClass
    //will run twice with data "k1" and "k2"
    @Factory
    public Object[] factoryMethod() {
        return new Object[] { new TestClass("K1"), new TestClass("k2") };
    }
}
```

DATA DRIVEN TESTING USING TESTNG

In this tutorial, we will be studying about data driven testing. We will refer to the @DataProvider annotation of TestNG using which we can pass data to the test methods and create a data driven testing framework.

WHAT IS DATA DRIVEN TESTING?

Data driven testing is a test automation technique in which the test data and the test logic are kept separated. The test data drives the testing by getting iteratively loaded to the test script. Hence, instead of having hard-coded input, we have new data each time the script loads the data from the test data source.

DATA DRIVEN TESTING USING @DATAPROVIDER

Data driven testing can be carried out through TestNG using its @DataProvider annotation. A method with @DataProvider annotation over it returns a 2D array of object where the rows determines the number of iterations and columns determine the number of input parameters passed to the Test method with each iteration.
This annotation takes only the name of the data provider as its parameter which is used to bind the data provider to the Test method. If no name is provided, the method name of the data provider is taken as the data provider's name.

```
@DataProvider(name = "nameOfDataProvider")
public Object[][] dataProviderMethodName() {
        //Data generation or fetching logic from any external source
        //returning 2d array of object

        return new Object[][] {{"k1","r1",1},{"k2","r2",2}};
}
```

After the creation of data provider method, we can associate a Test method with data provider using 'dataProvider' attribute of @Test annotation. For

successful binding of data provider with Test method, the number and data type of parameters of the test method must match the ones returned by the data provider method.

```
@Test(dataProvider = "nameOfDataProvider")
public void sampleTest(String testData1, String testData2, int testData3) {
        System.out.println(testData1 + " " + testData2 + " " + testData3);
}
```

CODE SNIPPET FOR DATA DRIVEN TESTING IN TESTNG

```
@DataProvider(name = "dataProvider1")
public Object[][] dataProviderMethod1() {
        return new Object[][] {{"k1","r1"},{"k2","r2"},{"k3","r3"}};
}

//The test case will run 3 times with different set of values
@Test(dataProvider = "dataProvider1")
public void sampleTest(String str1, String str2) {
        System.out.println(str1 + " " + str2);
}
```

The above test "sampleTest" will run 3 times with different set of test data - {"k1","r1"},{"k2","r2"},{"k3","r3"} received from 'dataProvider1' dataProvider method.

RERUN FAILED TESTS IN TESTNG

In this post, we will learn to rerun failed test cases using TestNG. We will explore the two approaches to achieve this, namely - using the testng-failed.xml file and by implementing the testNG IRetryAnalyzer.

RERUN FAILED TESTS USING TESTNG-FAILED.XML

WHEN TO USE?

Sometimes after some bug fixes, as a test automation engineer we are required to run only the failed tests reported by test automation suite. Running only the failed tests validate the bug fixes quickly.

HOW TO ACHIEVE?

Running only the failed tests is fairly simple as TestNG provides inherent support for this. Whenever a test suite is run using the testng.xml file then after the test execution, a **testng-failed.xml file gets created in the test-output folder**. Later on, we can run this file just like we run the testng.xml file. As this file only keeps track of the failed tests, so running this file, runs the failed tests only.

RETRY FAILED TESTS AUTOMATICALLY USING IRETRYANALYZER

WHEN TO USE?

At times, the test execution report comes up with some failures that are not because of the issues in the application. The underlying cause of these issues might be related to the test environment setup or some occasional server issue. In order to make sure that the failure reported in the test report are

genuine and not just one-off cases, we can retry running the failed test cases to eliminate false negative tests results in our test reports.

HOW TO ACHIEVE?

For retrying the failure test runs automatically during the test run itself, we need to implement the **IRetryAnalyzer** interface provided by TestNG. The IRetryAnalyzer interface provide methods to control retrying the test runs. Here, we will override the retry() method of IRetryAnalyzer to make sure the test runs in case of failure with specififed number of retry limit. The comments in the code snippet make it self-explainatory.

CODE SNIPPET

```
package com.artoftesting.test;
import org.testng.IRetryAnalyzer;
import org.testng.ITestResult;

public class RetryAnalyzer implements IRetryAnalyzer {

	//Counter to keep track of retry attempts
	int retryAttemptsCounter = 0;

	//The max limit to retry running of failed test cases
	//Set the value to the number of times we want to retry
	int maxRetryLimit = 1;

	//Method to attempt retries for failure tests
	public boolean retry(ITestResult result) {
		if (!result.isSuccess()) {
			if(retryAttemptsCounter < maxRetryLimit){
				retryAttemptsCounter++;
				return true;
			}
		}
		return false;
	}
}
```


For the demo, I am creating a dummy test method below and intentionally failing it using the assert.fail() method. Here, we will set the @Test

annotation's **retryAnalyzer attribute with RetryAnalyzer.class** that we created above.

```
@Test(retryAnalyzer = RetryAnalyzer.class)
    public void intentionallyFailingTest(){
        System.out.println("Executing Test");
        Assert.fail("Failing Test");
    }
```

TEST OUTPUT

```
===============================================
Test suite
Total tests run: 2, Failures: 1, Skips: 1
===============================================
```

Here, we can observe that the number of runs for the method are displayed as 2 with one failure and one skipped. The first run of the test method when failed will be marked as Skipped and then the test will run again due to the logic specified in the RetryAnalyzer.

Now, there is just one problem, we need to set the **retryAnalyzer** attribute in each of the @Test annotation. To deal with this we can implement **IAnnotationTransformer**interface. This interface provides the capability to alter the testNG annotation at runtime. So, we will create a class implementing the IAnnotationTransformer and make it set the RetryAnalyzer for @Test annotations.

```
package com.artoftesting.test;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

import org.testng.IAnnotationTransformer;
import org.testng.IRetryAnalyzer;
import org.testng.annotations.ITestAnnotation;

public class FailureRetryListener implements IAnnotationTransformer {

        //Overriding the transform method to set the RetryAnalyzer
        public void transform(ITestAnnotation testAnnotation, Class testClass,
                        Constructor testConstructor, Method testMethod)   {
                IRetryAnalyzer retry = testAnnotation.getRetryAnalyzer();
```

```
                if (retry == null)
                        testAnnotation.setRetryAnalyzer(RetryAnalyzer.class);
        }
}
```

Having created a listener, we can specify it in the testNG.xml file like this-

```
<listeners>
        <listener class-name="com.artoftesting.test.FailureRetryListener"/>
</listeners>
```

Now, we don't have to set the **retryAnalyzer** in each @Test annotation. Having the listener specified in the testng.xml file will make it work for all the tests.

PS: If you want to add retry functionality to only limited set of test method than just set the @Test annotations with **retryAnalyzer attribute with RetryAnalyzer.class**, there is no need to implement IAnnotationTransformer and adding the listener in the testng.xml file.

PRIORITY IN TESTNG

In automation, many times we are required to configure our test suite to run test methods in a specific order or we have to give precedence to certain test methods over others. TestNG allows us to handle scenarios like these by providing a **priority**attribute within @Test annotation. By seeting the value of this priority attribute we can order the test methods as our need.

PRIORITY PARAMETER

We can assign a priority value to a test method like this-

```
@Test(priority=1)
```

The tests with lower priority value will get executed first. Example-

DEFAULT PRIORITY

The default priority of test when not specified is integer value 0. So, if we have one test case with priority 1 and one without any priority value then the test without any priority value will get executed first (as default value will be 0 and tests with lower priority are executed first).

CODE SNIPPET

```
@Test(priority = 1)
public void testMethodA() {
   System.out.println("Executing - testMethodA");
}

@Test
public void testMethodB() {
   System.out.println("Executing - testMethodB");
}

@Test(priority = 2)
public void testMethodC() {
   System.out.println("Executing - testMethodC");
}
```

OUTPUT

```
Executing - testMethodB
Executing - testMethodA
Executing - testMethodC
```

Here, we can see that testMethodB got executed first as it had default priority of 0. Since the other tests had priority value as 1 and 2 hence, the execution order was testMethodB then testMethodA and then testMethodC.

PS: If we want to give a test method, priority higher than the default priority then we can simply assign a negative value to the priority attribute of that test method.

```
@Test(priority = -1)
public void testMethod() {
   System.out.println("Priority higher than default");
}
```


DEPENDENCY IN TESTNG

As a rule of thumb each test method must be independent from other i.e. result or execution of one test method must not affect the other test methods. But at times in automation, we may require one test method to be dependent on other. We may want to maintain dependecy between the test methods in order to save time or for scenarios where the call to run a particular test depends on the test result of a some other tests. TestNG as a testing framework provides us two attributes **dependsOnMethod** and **dependsOnGroup** within @Test annotation to achieve dependency between the tests. Now let's explore these attributes.

DEPENDSONMETHOD

Using 'dependsOnMethod' attribute within @Test annotation, we can specify the name of parent test method on which the test should be dependent. Some points to remember about dependsOnMethod-

- On execution of the whole test class or test suite, the parentTest method will run before the dependentTest.
- If we run the dependentTest alone even then the parentTest method will also run that too before the dependentTest.
- In case, the parentTest method gets failed then the dependentTest method will not run and will be marked as skipped.

```
@Test(dependsOnMethods={"parentTest"})
```

CODE SNIPPET

In the below code snippet, we have two test methods - parentTest() and dependentTest(). The dependentTest method is made dependent on the parentTest method using **dependsOnMethods** attribute.

```
public class TutorialExample {

    @Test
    public void parentTest() {
      System.out.println("Running parent test.");
    }

    @Test(dependsOnMethods={"parentTest"})
    public void dependentTest() {
```

```
        System.out.println("Running dependent test.");
    }
}
```

OUTPUT

On running the dependentTest() alone, the parentTest() method will also run and that too before the dependentTest(). This is done to maintain the dependency.

```
Running parent test.
Running dependent test.
PASSED: parentTest
PASSED: dependentTest

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
===============================================
```

DEPENDSONGROUP

The **dependsOnGroup** attribute is used to make a test method dependent on a collection of tests belonging to a particular groups. Thus ensuring that the dependent test will run after all the tests of the parent group are executed. Some points to remember about dependsOnMethod-

- On execution of the whole test class or test suite, the tests belonging to the parent group will run before the dependentTest.
- If we run the dependentTest alone even then all the tests belonging to the parent group will also run that too before the dependentTest.
- In case, anyone or all of the tests of the parent group method gets failed then the dependentTest method will not run and will be marked as skipped.

```
@Test(dependsOnGroups = "groupA")
```

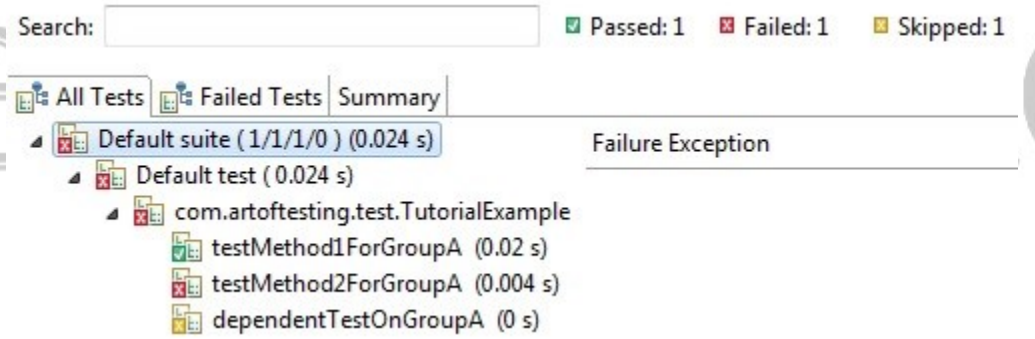
CODE SNIPPET

In the below code snippet, we have two test methods - testMethod1ForGroupA() and testMethod2ForGroupA() belonging to group **groupA**. Then we have a dependentTest method - **dependentTestOnGroupA** that is made dependent on the methods of **groupA** using **dependsOnGroups** attribute. For the sake of demo, we are intentionally failing the **testMethod2ForGroupA**.

```
public class TutorialExample {

    @Test(groups = "groupA")
    public void testMethod1ForGroupA() {
        System.out.println("Running test method1 of groupA");
    }

    @Test(groups = "groupA")
    public void testMethod2ForGroupA() {
        System.out.println("Running test method2 of groupA");
        Assert.fail();
    }

    @Test(dependsOnGroups = "groupA")
    public void dependentTestOnGroupA() {
        System.out.println("Running the dependent test");
    }
}
```

TESTNG RESULT

On running the dependentTestOnGroupA() alone or running the whole test class, we can observe the following output.



Here, we can observe that TestNG attemps to run all the 3 tests and because of the dependecy, runs **testMethod1ForGroupA** and **testMethod1ForGroupB** first. Now, since **testMethod1ForGroupB** gets failed, so, the dependentTest i.e. **dependentTestOnGroupA** is not run and marked as skipped.

TIMEOUT IN TESTNG

The automation test suites have the tendency to take too much time in case the elements are not readily available for interaction. Also, in certain tests we might have to wait for some asynchronous event to occur in order to proceed with the test execution. In these cases, we may want to limit the test execution time by specifying an upper limit of timeout exceeding which the test method is marked as failure. TestNG provides us **timeOut** attribute for handling these requirements.

TIMEOUT

Time **timeOut** attribute within the @Test annotation method is assigned a value specifying the number of milliseconds. In case the test method exceeds the timeout value, the test method is marked as failure with **ThreadTimeoutException**.

```
@Test(timeOut = 1000)
```

CODE SNIPPET

In the below code snippet we have specified a timeout of 1000ms. Inside the test methods we can see that a Thread.sleep() of 3seconds is introduced. On test execution, we can notice in the output that the test fails with ThreadTimeoutException as the timeout is 1 second and the test takes liitle over 3 seconds to execute.

```
@Test(timeOut = 1000)
public void timeOutTest() throws InterruptedException {
   Thread.sleep(3000);
   //Test logic
}
```

OUTPUT

```
FAILED: timeOutTest
org.testng.internal.thread.ThreadTimeoutException:
Method org.testng.internal.TestNGMethod.timeOutTest()
didn't finish within the time-out 1000
```

MYSQL AUTOMATION IN JAVA

WHY DO WE NEED DATABASE AUTOMATION?

- To get test data - If we automate the database, we can directly fetch the test data from database and then work on them in test automation script
- To verify result - In automation we can verify the front end result with backend entry in the database

- To delete test data created - In automation it is good practice to delete the test data created, using database automation, we directly fire the delete query to delete the test data created
- To update certain data - As per the need of test script, the test data can be updated using update query

CONNECTING TO MYSQL DATABASE IN JAVA

Database automation in mySQL database involves the following steps-

- Loading the required JDBC Driver class, which in our case is com.mysql.jdbc.Driver. You can download the jar from here and add it to your classpath or add it as maven dependency in you pom.xml file in case you are using maven project.

```
Class.forName("com.mysql.jdbc.Driver");
```

- Creating a connection to the database-

```
Connection conn = DriverManager.getConnection("DatabaseURL","UserName", "Password");
```

- Executing SQL queries-

```
Statement st = conn.createStatement();
String Sql = "select * from [tableName] where <condition>";
ResultSet rs = st.executeQuery(Sql);
```

- Fetching data from result set-

```
while (rs.next()) {
  System.out.println(rs.getString(<requiredField>));
}
```

SAMPLE CODE TO CONNECT TO A MYSQL DATABASE

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class connectingToMySQLDBExample {

    private static String userName;
    private static String password;
    private static String dbURL;
    private static Connection connection;

    public static void main(String[] args) {
```

```
        try {
                userName = "username";
                password = "password";
                dbURL = "jdbc:mysql://artoftesting.com/testDB";
                try {
                        Class.forName("com.mysql.jdbc.Driver");
                }
               catch (ClassNotFoundException e) {
                        System.out.println("MySQL JDBC driver not found.");
                        e.printStackTrace();
                }
                try {
                        connection = DriverManager.getConnection(dbURL, userName, pa
ssword);
                        Statement st = connection.createStatement();
                        String sqlStr = "select * from testTable";
                        ResultSet rs = st.executeQuery(sqlStr);
                        while (rs.next()) {
                                System.out.println(rs.getString("name"));
                        }

                        } catch (SQLException e) {
                                System.out.println("Connection to MySQL db failed");
                         e.printStackTrace();
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```