Table of Contents

[Overview 2](#_Toc4395870)

[1. Introduction to TestNG 3](#_Toc4395871)

[2. Installing TestNG 4](#_Toc4395872)

[Other Methods 5](#_Toc4395873)

[Resources 5](#_Toc4395874)

[3. TestNG Annotations 6](#_Toc4395875)

[What Are TestNG Annotations 6](#_Toc4395876)

[How to Add TestNG Annotations 6](#_Toc4395877)

[Configuration Annotations 7](#_Toc4395878)

[TestNG Annotations Execution Flow 11](#_Toc4395879)

[Which Annotation(s) Do We Choose for Testing 14](#_Toc4395880)

[Resources 18](#_Toc4395881)

[4. TestNG XML Files 19](#_Toc4395882)

[The TestNG XML file 19](#_Toc4395883)

[Resources 21](#_Toc4395884)

[5. Priority Attribute 22](#_Toc4395885)

[Default Execution Order for Test Methods 22](#_Toc4395886)

[Priority Attributes for Test Methods 25](#_Toc4395887)

[Resources 27](#_Toc4395888)

[6.1 Intro to TestNG Assertions 28](#_Toc4395889)

[Resources 29](#_Toc4395890)

[6.2 Hard Asserts 30](#_Toc4395891)

[Resources 31](#_Toc4395892)

[Source Code 31](#_Toc4395893)

[6.3 Soft Asserts 32](#_Toc4395894)

[Difference Between Hard Asserts and Soft Asserts 32](#_Toc4395895)

[Hard Asserts vs Soft Asserts 33](#_Toc4395896)

[Resources 34](#_Toc4395897)

[7.1 Intro to Dependency Testing 35](#_Toc4395898)

[Resources 36](#_Toc4395899)

[7.2 dependsOnMethods Attribute 37](#_Toc4395900)

[Executing without dependsOnMethods Attribute 37](#_Toc4395901)

[Execute with dependsOnMethods Attribute 38](#_Toc4395902)

[Exclude Test Method via XML File 40](#_Toc4395903)

[Resources 41](#_Toc4395904)

[7.3 groups and dependsOnGroups Attributes 42](#_Toc4395905)

[groups Attribute 42](#_Toc4395906)

[dependsOnGroups Attribute 42](#_Toc4395907)

[Execute Groups via XML File 44](#_Toc4395908)

[Run Test Methods According to Group Name 47](#_Toc4395909)

[Resources 48](#_Toc4395910)

[8. Data Driven Testing 49](#_Toc4395911)

[DataProvider annotation 50](#_Toc4395912)

[dataProvider Attribute 50](#_Toc4395913)

[dataProviderClass Attribute 52](#_Toc4395914)

[Resources 53](#_Toc4395915)

[9. Cross-Browser Testing 54](#_Toc4395916)

[Parameter Tag via XML File 54](#_Toc4395917)

[Parameters Annotation 55](#_Toc4395918)

[Different Ways to Supply test data 56](#_Toc4395919)

[Additional Concepts 57](#_Toc4395920)

[Resources 57](#_Toc4395921)

**Introduction to TestNG**

# Overview

xUnit is a family of Unit Test Frameworks that consists of JUnit for Java, PyUnit for Python, and NUnit for the .Net programming languages such as C Sharp.

JUnit is the most popular test framework but TestNG is the most powerful test framework. TestNG was influenced by JUnit so it adopted the same concepts then added more testing features.

Our lesson plan will include:

* Installing TestNG
* Annotations
* Assertions
* Attributes (priority, dependsOnMethods, and dependsOnGroups)
* Data-driven testing
* Cross-browser testing

# Introduction to TestNG

In this chapter, we will discuss what is a test framework and what separates TestNG from other frameworks

A test framework is a pattern for writing and running test scripts. The purpose is to monitor testing and development of an Application Under Test (AUT).

A test framework is different from an automation design framework. Some of the most popular automation design frameworks are data-driven, keyword-driven, and hybrid-driven frameworks. With Selenium, we utilize a test framework to create our automation design framework.

Test frameworks simplify the process of testing by giving power to developers and automation engineers to write a quick test. We can add an annotation and assert statement then start our test. We also have the ability to run our tests as a collection in a test suite.

Some of the core functions within a test framework include:

* creating and executing test scripts
* generating test reports
* generating logs
* reading and writing test data

xUnit is a name for a compilation of Unit Testing Frameworks. Some of those frameworks are JUnit, PyUnit, and NUnit. JUnit is for Java, PyUnit is for Python, and NUnit is for the .Net programming languages such as C#. The xUnit family became popular after releasing JUnit. Later, many programming languages embraced the common architecture and joined the xUnit family.

However, as their family name implies, xUnit is a unit testing framework. These framewoorks can test units of an application at a time, where units are the smallest part of an application.

So, what separates TestNG from other frameworks? TestNG is able to test more than a unit. Just like the xUnit family, TestNG can perform the same functions because it was influenced by JUnit, however, TestNG has more testing features which support parallel testing, integration testing, dependency testing, data-driven testin, cross-browser testing, and end-to-end testing.

While the xUnit family is good, TestNG has been a step ahead when it comes to testing. However, TestNG has also influenced the lastest versions of JUnit and has included a lot of the same concepts.

# Installing TestNG

In this chapter, we will discuss the various ways to install TestNG and then demonstrate how to install TestNG using Eclipse.

Ways to Install TestNG

Let’s start with the compatible IDEs. IDE stands for Integrated Development Environment. The compatible IDEs are Eclipse, NetBeans, and IntelliJ. Next, we can setup TestNG using build tools Ant or Maven.

In addition to the IDEs and build tools, we can use the command line or download the TestNG jars. The focus in this chapter will be installing TestNG using Eclipse but I will provide a reference link for NetBeans, IntelliJ, Ant, and Maven.

How to Install TestNG Using Eclipse Marketplace

First, I’ll show you to how install TestNG using Eclipse Marketplace.

1. Go to Help
2. Select Eclipse Marketplace
3. Type TestNG in the Find text box
4. Click the Go button
5. Click the Install button
6. Click the Confirm button after making sure all of the checkboxes are checked
7. Select "I accept the terms of the license agreement"
8. Click Finish

Note

A security warning shows up to let you know you are installing software that contains unsigned content. This shows up because TestNG is 3rd Party plugin.

1. Click Install anyway
2. Click Restart Now to restart Eclipse

How to Install TestNG Using Install New Software

Another way to install TestNG is through "Install New Software".

1. Go to Help
2. Select Install New Software
3. Type TestNG
4. Click Enter
   * If TestNG Eclipse does not come up for you as an auto-suggestion then click the Add button
   * In the Add Repository pop-up, type TestNG for Name and http://beust.com/eclipse for Location
   * Click OK
   * Select the checkbox for TestNG
   * Click Next
   * Accept the License Agreement
   * Click Finish button.

After installing TestNG, we must add the TestNG Library.

Add TestNG Library

We have the option of adding our TestNG Library now or waiting until we import an annotation. Let’s go ahead and add the TestNG Library now. There’s more than one way to add a TestNG Library but we are going to do so by configuring the build path.

1. Right click on the src directory within the Package Explorer
2. Right click JRE System Library (or Referenced Libraries)
3. Select Build Path
4. Select Configure Build Path
5. Select Classpath
6. Click the Add Library button
7. Select TestNG
8. Click Next
9. Click Finish (now we see TestNG()
10. Click the Apply and Close button.

We are finished installing TestNG and adding TestNG as a library.

### Other Methods

[TestNG IntelliJ](https://testng.org/doc/idea.html)

[TestNG NetBeans](http://wiki.netbeans.org/TestNG#Installation)[TestNG Maven](https://testng.org/doc/maven.html)[TestNG Ant Task](https://testng.org/doc/ant.html)

## Resources

[TestNG Eclipse](https://tinyurl.com/Steps-To-Install-TestNG-Doc)[TestNG IntelliJ](https://testng.org/doc/idea.html)

[TestNG NetBeans](http://wiki.netbeans.org/TestNG#Installation)[TestNG Maven](https://testng.org/doc/maven.html)[TestNG Ant Task](https://testng.org/doc/ant.html)

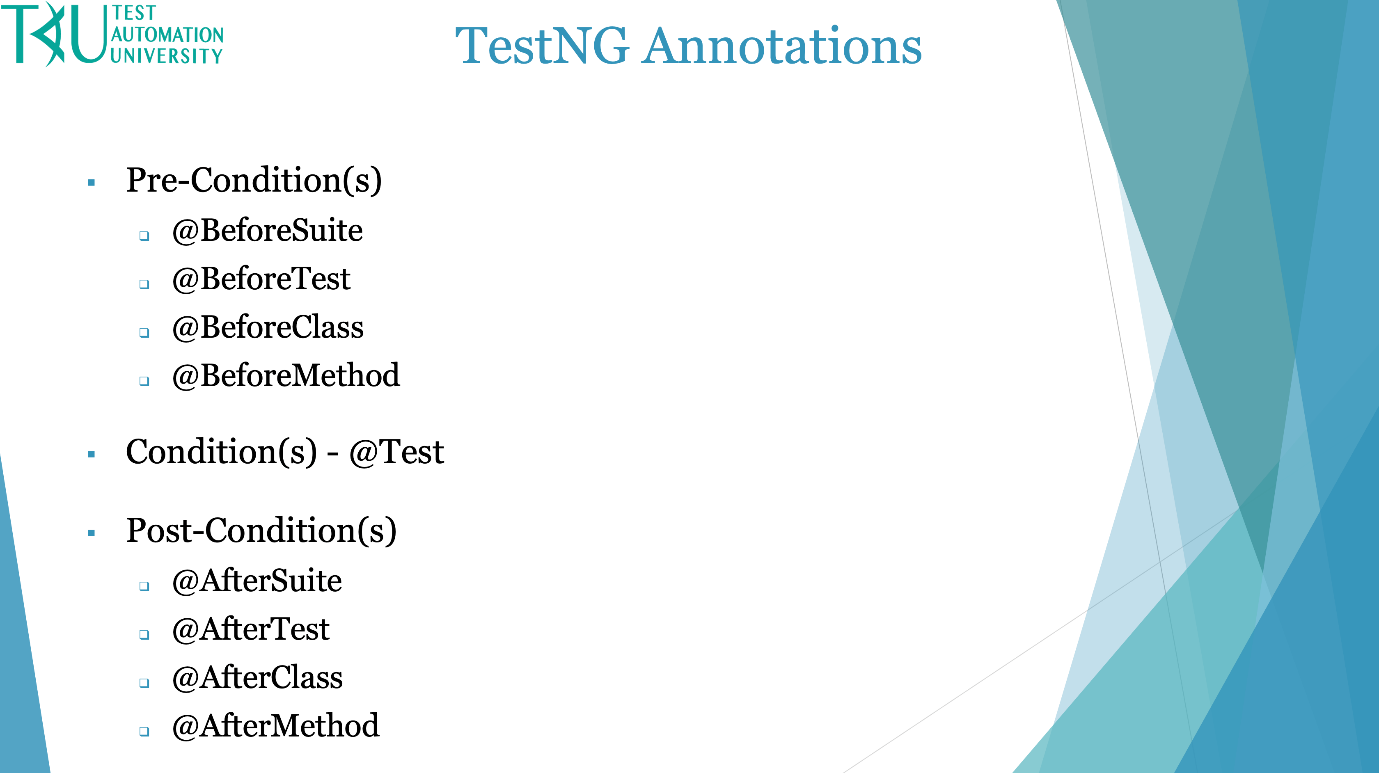
# TestNG Annotations

### What Are TestNG Annotations

A TestNG annotation is data that has a special meaning for a Java method. It provides information about how the annotation will control the execution order.

I view TestNG annotations as pre-conditions, conditions, and post-conditions. Our test automation scripts are very similar to manual test cases. Sometimes with manual test cases, there are pre-conditions that must be set up before we start our test. Our test is the condition, and after our test is the post- condition.

It’s the same with automation, we have pre-conditions that must be set up before we test; then we have our test which is the condition; next is the post-condition that is performed after we complete our testing.



Notice the @ symbol (or @ sign). This is short for at a rate which is an accounting term. This symbol is placed in front of every TestNG annotation.

* All of the pre-conditions begin with @Before.
* @Test is a key annotation because it performs our test.
* Last, we have the post-conditions that all start with @After.

### How to Add TestNG Annotations

Let’s go to Eclipse.

There is more than one way to add an annotation. The most straightforward way to add an annotation is just to type it on the line above the method you'd like to associate it with.

@BeforeMethod

public void setUp ()

{

}

After typing it, you'll need to import the annotation if it's the first time you're using it in this class. A short cut for importing and organizing our imports is CTRL + SHIFT + O. (If you want to see a list of Eclipse shortcuts select CTRL + SHIFT + L and we see a lot of shortcuts including Organize Imports.)

Another way to add TestNG annotations is to:

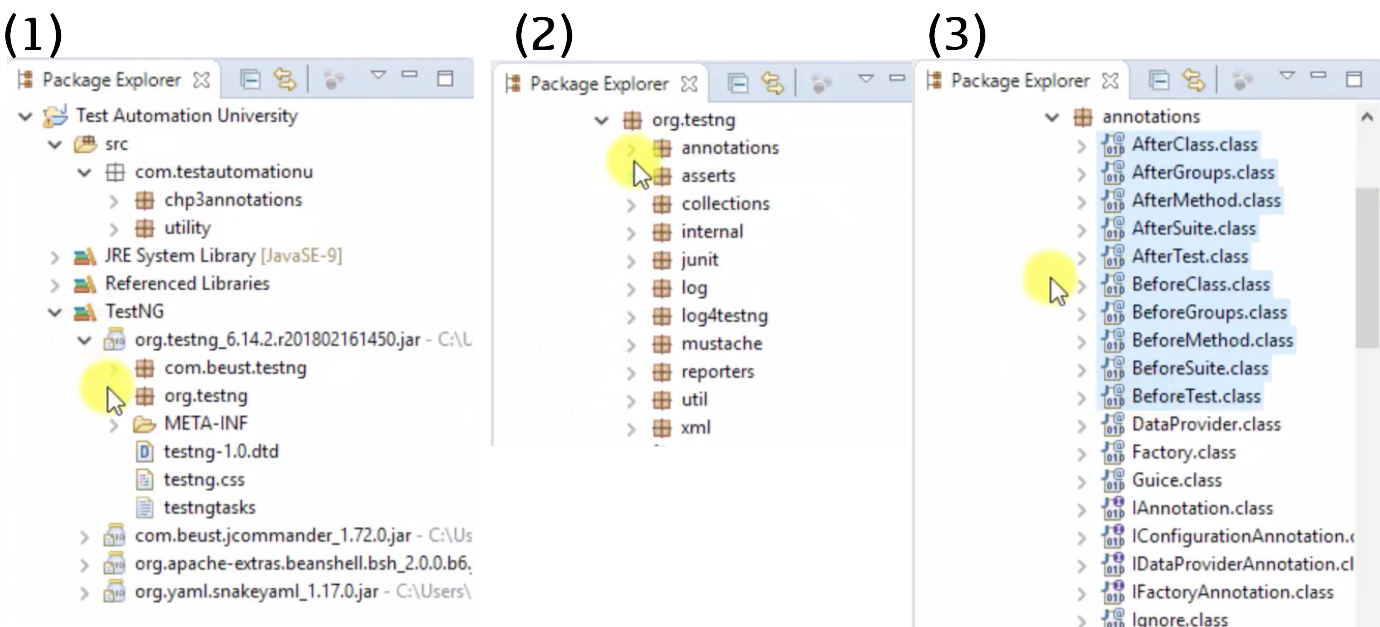
1. Right click on the package in which the class resides
2. Select New
3. Select Other
4. Type TestNG
5. Select TestNG class
6. Click Next
7. Select all of the annotations except for DataProvider (we are going to cover DataProvider in a subsequent chapter called Data Driven Testing)
8. Change the Class Name to Configuration Annotations
9. Click Finish



### Configuration Annotations

A configuration annotation is an annotation that begins with Before or After. They are called configuration annotations because the Before annotations help us set up variables and configurations before starting test execution, while the After annotations help us clean up everything after executing our test.

If we take a look at the TestNG annotations package, we will see all of the annotations. The configuration annotations start at AfterClass and stop at BeforeTest.



In our editor, the annotations are displayed in order from lowest to highest. Let’s start with the highest level and add some print statements to demonstrate how the annotations can be used.

package com.testautomationu.chp3annotations;

import org.testng.annotations.Test;

import org.testng.annotations.BeforeMethod;

import org.testng.annotations.AfterMethod;

import org.testng.annotations.BeforeClass;

import org.testng.annotations.AfterClass;

import org.testng.annotations.BeforeTest;

import org.testng.annotations.AfterTest;

import org.testng.annotations.BeforeSuite;

import org.testng.annotations.AfterSuite;

public class Configuration\_Annotations {

@BeforeSuite

public void beforeSuite() {}

@BeforeTest

public void beforeTest() {}

@BeforeClass

public void beforeClass() {}

@BeforeMethod

public void beforeMethod() {}

@Test

public void searchCustomer() {}

@Test

public void searchProduct() {}

@AfterMethod

public void afterMethod() {}

@AfterClass

public void afterClass() {}

@AfterTest

public void afterTest() {}

@AfterSuite

public void afterSuite() {}

}

@BeforeSuite runs before a suite starts. So, let’s use the @BeforeSuite annotated method to set up our system property for the browser.

@BeforeSuite

public void beforeSuite()

{

System.out.println("Chrome - Set Up System Property");

}

@AfterSuite runs after all @Test methods within the suite have completed running. So, we'll use the @AfterSuite annotated method to clean up cookies.

@AfterSuite

public void afterSuite()

{

System.out.println("Chrome - Clean Up All Cookies");

}

@BeforeTest runs before all tests. We can demonstrate this by using @BeforeTest to open the browser before any tests begin.

@BeforeTest

public void beforeTest()

{

System.out.println("Open Chrome");

}

@AfterTest runs after all @Test methods. We can use our @AfterTest method to close the browser after all @Test methods have completed.

@AfterTest

public void afterTest()

{

System.out.println("Close Chrome");

}

@BeforeClass runs before a test class starts, so let's use that method to open the test application.

@BeforeClass

public void beforeClass()

{

System.out.println("Open Test Application");

}

@AfterClass runs after all @Test methods, so we will use that method to close the test application.

@AfterClass

public void afterClass()

{

System.out.println("Close Test Application");

}

Next, we have @BeforeMethod which runs before each @Test method. Let's demonstrate this by having our @BeforeMethod sign into the application before each test runs.

@BeforeMethod

public void beforeMethod()

{

System.out.println("Sign In");

}

@AfterMethod runs after each @Test method. We can use this to sign out of our application after each test finishes.

@AfterMethod

public void afterMethod()

{

System.out.println("Sign Out");

}

Next is the @Test annotation. After all Before methods have executed, the @Test methods will run. We can have as many @Test annotations as we want. Let's add a few here.

@Test

public void searchCustomer()

{

System.out.println("Search For Customer");

}

@Test

public void searchProduct()

{

System.out.println("Search For Product");

}

The annotations can be placed in any order in the editor because TestNG identifies the methods by looking up the annotation. For example, we can place the @BeforeSuite method anywhere in this editor and it will always execute first.

### TestNG Annotations Execution Flow

The execution flow depends on our annotations. Therefore, the methods execute according to the rank of each annotation. Let’s run!

package com.testautomationu.chp3annotations;

import org.testng.annotations.Test;

import org.testng.annotations.BeforeMethod;

import org.testng.annotations.AfterMethod;

import org.testng.annotations.BeforeClass;

import org.testng.annotations.AfterClass;

import org.testng.annotations.BeforeTest;

import org.testng.annotations.AfterTest;

import org.testng.annotations.BeforeSuite;

import org.testng.annotations.AfterSuite;

public class Configuration\_Annotations {

@BeforeSuite

public void beforeSuite()

{

System.out.println("Chrome - Set Up System Property");

}

@BeforeTest

public void beforeTest()

{

System.out.println("Open Chrome");

}

@BeforeClass

public void beforeClass()

{

System.out.println("Open Test Application");

}

@BeforeMethod

public void beforeMethod()

{

System.out.println("Sign In");

}

@Test

public void searchCustomer()

{

System.out.println("Search For Customer");

}

@Test

public void searchProduct()

{

System.out.println("Search For Product");

}

@AfterMethod

public void afterMethod()

{

System.out.println("Sign Out");

}

@AfterClass

public void afterClass()

{

System.out.println("Close Test Application");

}

@AfterTest

public void afterTest()

{

System.out.println("Close Chrome");

}

@AfterSuite

public void afterSuite()

{

System.out.println("Chrome - Clean Up All Cookies");

}

}

On the console, we see the first few print statements in the following order:

Chrome – Set Up System Property

Open Chrome

Open Test Application

Sign In

Search For Customer

Sign Out

This shows that the annotations have been executed in the following order:

1. @BeforeSuite
2. @BeforeTest
3. @BeforeClass
4. @BeforeMethod
5. @Test
6. @AfterMethod

Here’s the interesting part, the @BeforeMethod always runs before the @Test method. In this case, the first @Test method prints "Search For Customer" which has a @Test annotation. And the @AfterMethod always runs after the @Test method. In this demo, the @AfterMethod prints "Sign Out".

At this point, we have finished searching for a customer, however, Chrome is still set up, Chrome is still open, and the application remains open but we are not signed into the application. Now we start over with the @BeforeMethod, which is a pre-condition to allow us to sign in before moving on to the next @Test to search for a product.

Here's an update on the print statements:

Chrome – Set Up System Property

Open Chrome

Open Test Application

Sign In

Search For Customer

Sign Out

Sign In

Search For Product

Sign Out

This shows that the annotations have been executed in the following order:

1. @BeforeSuite
2. @BeforeTest
3. @BeforeClass
4. @BeforeMethod
5. @Test searchCustomer
6. @AfterMethod
7. @BeforeMethod
8. @Test searchProduct
9. @AfterMethod

This shows us how TestNG executes the pre-condition, the condition, and the post-condition based on our annotations.

Next we see that the @AfterClass, @AfterTest, and @AfterSuite annotations are executed.

Chrome – Set Up System Property

Open Chrome

Open Test Application

Sign In

Search For Customer

Sign Out

Sign In

Search For Product

Sign Out

Close Test Application

Close Chrome

Clean Up All Cookies

1. @BeforeSuite
2. @BeforeTest
3. @BeforeClass
4. @BeforeMethod
5. @Test searchCustomer
6. @AfterMethod
7. @BeforeMethod
8. @Test searchProduct
9. @AfterMethod
10. @AfterClass
11. @AfterTest
12. @AfterSuite

### Which Annotation(s) Do We Choose for Testing

The annotations we choose for testing depends on our test requirements. Do we need our test to set up a pre-condition before every test method and clean up with a post-condition after every test method? Or do we need one pre-condition before all of the test methods and one post-condition after all test methods?

Let’s walk through our [test application](https://opensource-demo.orangehrmlive.com/)

then I will show you the difference using two pairs of configuration annotations and three test methods.  


The test requirement is to sign in with username:Admin and password:admin123 then click the login button. After signing in, we click the Admin tab then search for a user. Finally, we sign out.

Our code uses a @BeforeMethod annotation and @AfterMethod annotation. First, we setup the test, then sign into the application, after signing into the application, we search for a user and sign out. After signing out we teardown our test which is close the browser.

package com.testautomationu.chp3annotations;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

import org.testng.annotations.AfterMethod;

import org.testng.annotations.BeforeMethod;

import org.testng.annotations.Test;

import com.testautomationu.utility.Highlighter;

public class BM\_AM\_OrangeHRM

{

WebDriver driver;

@BeforeMethod

public void setUp ()

{

System.setProperty("webdriver.chrome.driver", "C:\\Users\\Rex Allen Jones II\\Downloads\\Drivers\\chromedriver.exe");

driver = new ChromeDriver ();

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

driver.get("https://opensource-demo.orangehrmlive.com/");

System.out.println("1. Open Chrome & Application");

}

@Test

public void signIn ()

{

WebElement textUsername = driver.findElement(By.id("txtUsername"));

Highlighter.highlightElement(driver, textUsername);

textUsername.sendKeys("Admin");

WebElement textPassword = driver.findElement(By.id("txtPassword"));

Highlighter.highlightElement(driver, textPassword);

textPassword.sendKeys("admin123");

WebElement buttonLogin = driver.findElement(By.id("btnLogin"));

Highlighter.highlightElement(driver, buttonLogin);

buttonLogin.click();

System.out.println("2. Sign In");

}

@Test

public void userSearch ()

{

WebElement menuAdmin = driver.findElement(By.id("menu\_admin\_viewAdminModule"));

Highlighter.highlightElement(driver, menuAdmin);

menuAdmin.click();

WebElement textUserName = driver.findElement(By.id("searchSystemUser\_userName"));

Highlighter.highlightElement(driver, textUserName);

textUserName.sendKeys("Admin");

WebElement buttonSearch = driver.findElement(By.id("searchBtn"));

Highlighter.highlightElement(driver, buttonSearch);

buttonSearch.click();

System.out.println("3. Search For User");

}

@Test

public void userSignOut ()

{

WebElement linkWelcome = driver.findElement(By.id("welcome"));

Highlighter.highlightElement(driver, linkWelcome);

linkWelcome.click();

WebElement linkLogout = driver.findElement(By.xpath("//div[@id='welcome-menu']/descendant::a[contains(@href,'logout')]"));

Highlighter.highlightElement(driver, linkLogout);

linkLogout.click();

System.out.println("4. Sign Out");

}

@AfterMethod

public void tearDown ()

{

System.out.println("5. Close Chrome & Application");

driver.quit();

}

}

When we run this, we see two failures: userSearch and userSignOut.

If we look at the output of the first test, signIn, we have:

1. Open Chrome & Application

2. Sign In

5. Close Chrome & Application

* Step 1 shows we opened Chrome and the application
* Step 2 shows we signed into the application
* Step 3 is missing
* Step 4 is missing
* Step 5 shows we close Chrome and the application

Let’s look at the output for the next test userSearch.

1. Open Chrome & Application

5. Close Chrome & Application

* Step 1 shows we opened Chrome and the application
* Step 2 is missing
* Step 3 is missing
* Step 4 is missing
* Step 5 shows we close Chrome and the application

Why were steps 2 through 4 skipped? The test script skipped those steps because we cannot open Chrome and the application then search for a user because we have not signed into the application yet. The console shows:

Failed: userSearch. Unable to locate element. It could not locate the Admin tab to begin searching for a user.

Same with the userSignOut. Our test script could not sign out of the application because it never signed into the application. Steps 2, 3, and 4 are missing. The console shows:

FAILED: userSignOut

With this test requirement, we must use @BeforeClass/@AfterClass or @BeforeTest/@AfterTest so that our code does not open the application before every test method and close the application after every test method.

Now, watch what happens when we use the same code but change the configuration annotations to @BeforeClass and @AfterClass.

@BeforeClass

public void setUp ()

{

System.setProperty("webdriver.chrome.driver", "C:\\Users\\Rex Allen Jones II\\Downloads\\Drivers\\chromedriver.exe");

driver = new ChromeDriver ();

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

driver.get("https://opensource-demo.orangehrmlive.com/");

System.out.println("1. Open Chrome & Application");

}

@AfterClass

public void tearDown ()

{

System.out.println("5. Close Chrome & Application");

driver.quit();

}

Now if we run this, we see that all tests pass and the console shows all 5 steps.

TestNG provides a lot of annotations for our test requirements. This is a list of the annotations.



## Resources

[Source Code](https://github.com/RexJonesII/Test-Automation-University/tree/master/chp3annotations)[Test Application](https://opensource-demo.orangehrmlive.com/)

[Code for Highlighter utility](https://github.com/RexJonesII/Test-Automation-University/tree/master/utility)

# TestNG XML Files

### The TestNG XML file

The purpose of a TestNG XML file is to store data and to carry data for all of our testing. Here's an XML file for our test suite.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Test Automation University Suite">

<test name="configuration annotation Test">

<classes>

<class name="com.testautomationu.chp3annotations.Configuration\_Annotations">

<methods>

<include name = "searchCustomer"/>

<include name = "searchProduct"/>

</methods>

</class>

</classes>

</test> <!-- configuration annotation Test -->

</suite> <!-- Test Automation University Suite -->

In this example, we see the following tags:

* <suite>
* <test>
* <class>
* <methods>

A <suite> can have one or more tests.

A <test> can have one or more classes.

A <class> can have one or more methods.

The XML file gives us a picture on why the annotations execute in a particular order.

If we run our test suite, we see the same output as before. It’s the same output because this execution has one <test> tag.

Chrome – Set Up System Property

Open Chrome

Open Test Application

Sign In

Search For Customer

Sign Out

Sign In

Search For Product

Sign Out

Close Test Application

Close Chrome

Clean Up All Cookies

Let’s look at an XML file that has two <test> tags:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Test Automation University Suite">

<test name="Search For A Customer">

<classes>

<class name="com.testautomationu.chp3annotations.Configuration\_Annotations">

<methods>

<include name = "searchCustomer"/>

</methods>

</class>

</classes>

</test> <!-- configuration annotation Test -->

<test name="Search For A Product">

<classes>

<class name="com.testautomationu.chp3annotations.Configuration\_Annotations">

<methods>

<include name = "searchProduct"/>

</methods>

</class>

</classes>

</test> <!-- configuration annotation Test -->

</suite> <!-- Test Automation University Suite -->

This XML file does not include the test methods together. We see the first test name is Search For A Customer and the second test name is Search For A Product.

After running this, we can see that the output is different than before.

Chrome – Set Up System Property

Open Chrome

Open Test Application

Sign In

Search For Customer

Sign Out

Close Test Application

Close Chrome

Open Chrome

Open Test Application

Sign In

Search For Product

Sign Out

Close Test Application

Close Chrome

Clean Up All Cookies

This time our execution does not immediately sign back into the application after signing out of the application. Now, the execution processes @AfterClass (Close Test Application) and @AfterTest (Close Chrome) after signing out. Why? Because in the XML file, the test methods are not included the same <class> tag. Therefore, execution will not sign back in to start our next test.

Notice, Chrome is still set up after executing @BeforeSuite. That means the @AfterSuite annotation is the only annotation that has not been executed after the first test. It is only executed after both tests have completed.

In reality, we probably would not use four Before configuration annotations and four After configuration annotations. However, I wanted you to see how the XML file is connected to the annotations and how the XML file allows us to execute the same methods for one test compared to more than one test.

## Resources

[Source Code](https://github.com/RexJonesII/Test-Automation-University/tree/master/chp4xml)[Getting Started With TestNG: A Java Test Framework](https://tinyurl.com/TestNG-Getting-Started-Book)

[Next Generation Java Testing: TestNG and Advanced Concepts](https://testng.org/doc/book.html)

# Priority Attribute

In this chapter, we will discuss the default execution order for @Test methods and the priority attribute for the test methods.

### Default Execution Order for Test Methods

As a reminder, a test method is marked by the @Test annotation. In addition to setting our methods as tests, we can also set our entire class as a test.

In this example, I marked the class as part of the TestNG tests by adding a @Test annotation at the class level.

@Test

public class DefaultExecutionOrder\_Class

{

}

Let's add setUp and tearDown methods with configuration annotations.

@Test

public class DefaultExecutionOrder\_Class

{

@BeforeClass

public void setUp() {}

@AfterClass

public void tearDown() {}

}

We'll add three more methods with no annotations: signIn, searchTShirt and signOut.

@Test

public class DefaultExecutionOrder\_Class

{

@BeforeClass

public void setUp() {}

public void signIn() {}

public void searchTShirt() {}

public void signOut() {}

@AfterClass

public void tearDown() {}

}

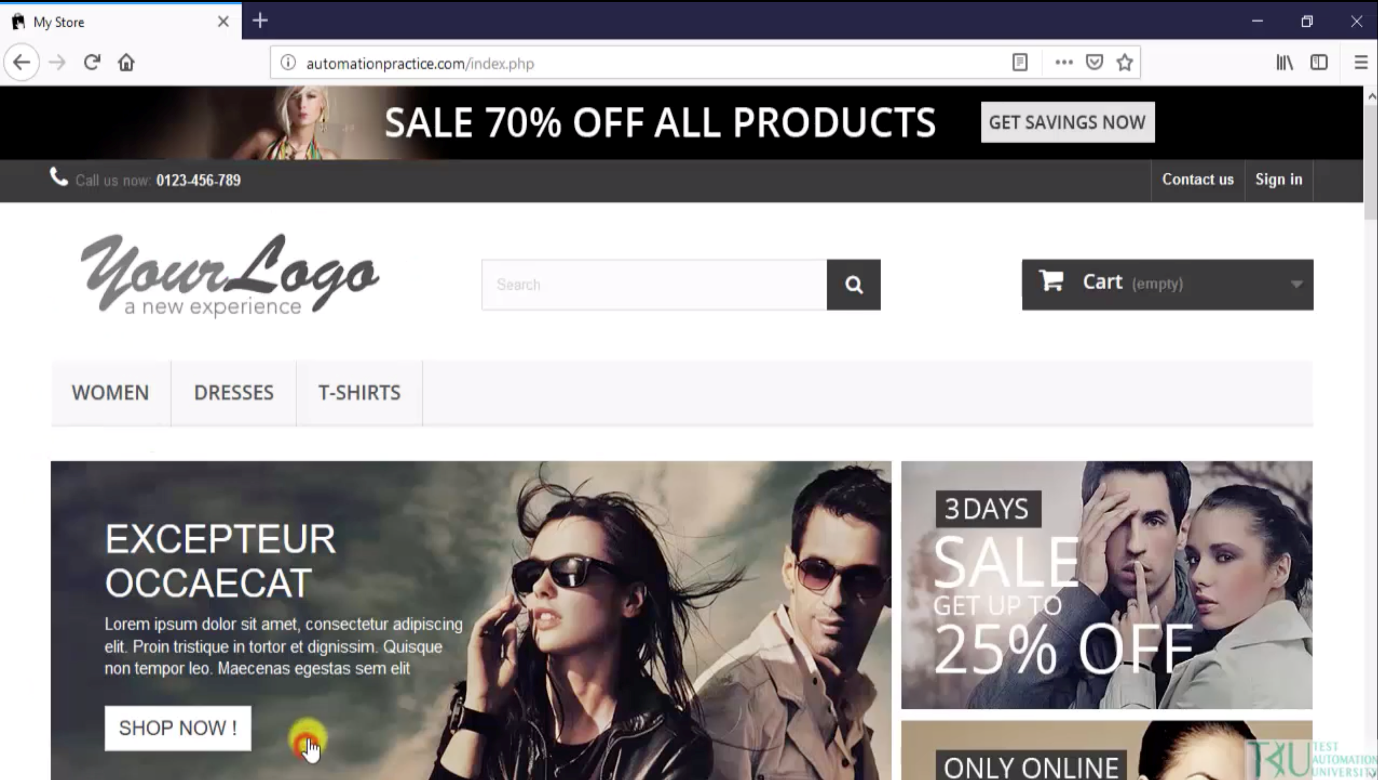
Looking at this layout, it seems the execution order should be:

1. setUp
2. signIn
3. searchTShirt
4. signOut
5. tearDown

However, that’s not the case. Execution is going to run searchTShirt before signIn and signOut. This is because the default execution order is ascending alphabetical order from A – Z. Therefore, in ascending order, se in searchTShirt comes before si in signIn and signOut.

We can place the test methods anywhere in the editor and it will run in the same order every time because the Test annotations identify the test methods.

Before running this program, let’s walk through the [test application](http://automationpractice.com/index.php).



The plan is to first sign in by entering an email of TestNG@Framework.com and password of TestNG1234 then click the Sign In button. After clicking the Sign In button, we click T-Shirts, search for a Blue T-shirt, then click the Search button. Finally, we sign out.

Let’s code this and run it.

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

import org.testng.annotations.AfterClass;

import org.testng.annotations.BeforeClass;

import org.testng.annotations.Test;

import com.testautomationu.utility.Highlighter;

@Test

public class DefaultExecutionOrder\_Class

{

WebDriver driver;

@BeforeClass

public void setUp () throws Exception

{

System.setProperty("webdriver.chrome.driver", "C:\\Users\\Rex Allen Jones II\\Downloads\\Drivers\\chromedriver.exe");

driver = new ChromeDriver ();

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

driver.get("http://automationpractice.com/index.php");

}

public void signIn ()

{

driver.findElement(By.xpath("//div[@class='header\_user\_info']/a")).click();

WebElement emailAddress = driver.findElement(By.id("email"));

Highlighter.highlightElement(driver, emailAddress);

emailAddress.sendKeys("TestNG@Framework.com");

WebElement password = driver.findElement(By.id("passwd"));

Highlighter.highlightElement(driver, password);

password.sendKeys("TestNG1234");

WebElement buttonSignIn = driver.findElement(By.id("SubmitLogin"));

Highlighter.highlightElement(driver, buttonSignIn);

buttonSignIn.click();

System.out.println("1. Sign In");

}

public void searchTShirt ()

{

WebElement menu = driver.findElement(By.xpath("//div[@id='block\_top\_menu']/ul/li/a[text()='T-shirts']"));

Highlighter.highlightElement(driver, menu);

menu.click();

WebElement searchBox = driver.findElement(By.id("search\_query\_top"));

Highlighter.highlightElement(driver, searchBox);

searchBox.sendKeys("Blue");

WebElement buttonSearch = driver.findElement(By.xpath("//\*[@id='searchbox']/button"));

Highlighter.highlightElement(driver, buttonSearch);

buttonSearch.click();

System.out.println("2. Search For T-Shirt");

}

public void signOut ()

{

WebElement linkSignOut = driver.findElement(By.className("logout"));

Highlighter.highlightElement(driver, linkSignOut);

linkSignOut.click();

System.out.println("3. Sign Out");

}

@AfterClass

public void tearDown () throws Exception

{

driver.quit();

}

}

**Output:**

2. Search For T-Shirt

1. Sign In

3. Sign Out

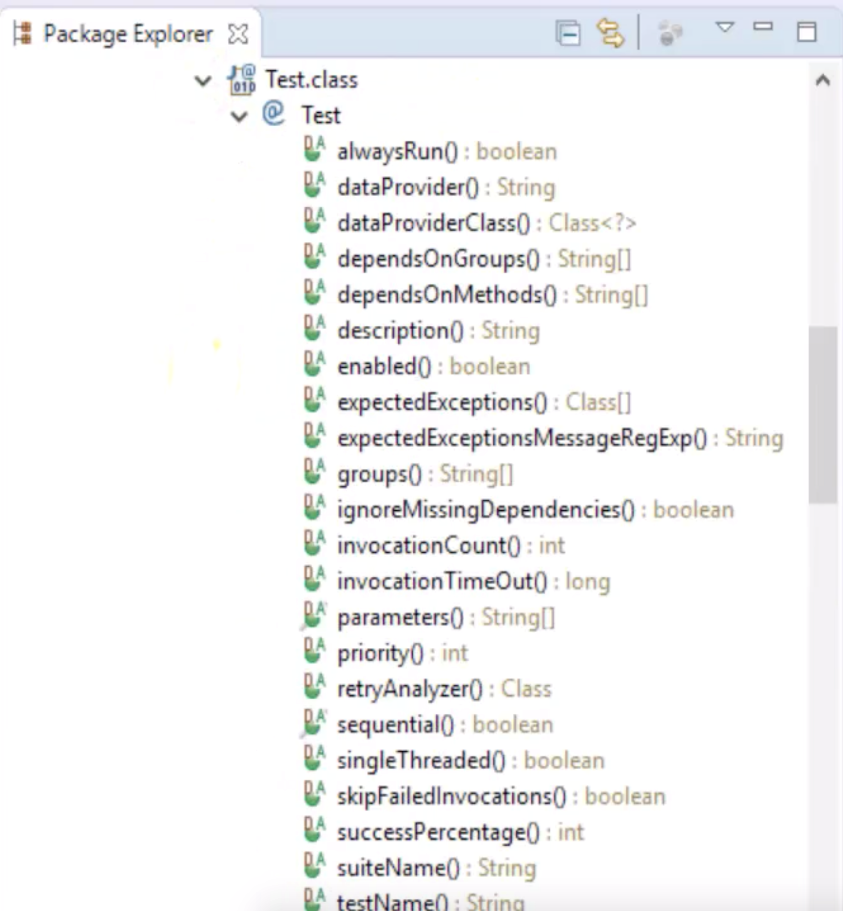
This works the same as if we had a @Test annotation at the method levels. If we removed the class @Test annotation and added @Test above the three test methods, the default execution order would be the same.

Note

As a side note, only the methods with a public access modifier are considered as test methods when marking the class with a @Test annotation. Any other access modifier will not set the method as a test method. If I change one of these methods to private then that method will not be executed.

### Priority Attributes for Test Methods

The @Test annotation has a lot of attributes. We can see those attributes by going to the TestNG Library -> TestNG jar file -> org.testng -> annotations, then scroll down to the Test.class and expand.



Let's focus on the priority attribute. The purpose of a priority attribute is to determine the execution order for our test method.

We use a priority attribute by writing priority within a parenthesis after the @Test annotation. The priority attribute uses an Integer data type. The lowest number gets executed first. Some people start at zero but I prefer to use 1 since it’s the first test method.

Let's update our tests to use priority.

public class PriorityExecutionOrder

{

@BeforeClass

public void setUp() {}

@Test (priority = 1)

public void signIn() {}

@Test (priority = 2)

public void searchTShirt() {}

@Test (priority = 3)

public void signOut() {}

@AfterClass

public void tearDown() {}

}

If we now execute this, we see it run in the following order:

1. Sign In

2. Search for T-Shirt

3. Sign Out

## Resources

[Source Code](https://github.com/RexJonesII/Test-Automation-University/tree/master/chp5priority)[Test Application](http://automationpractice.com/index.php)

[Code for highlighter utility](https://github.com/RexJonesII/Test-Automation-University/tree/master/utility)

# 6.1 Intro to TestNG Assertions

I have a question for you. Up to this point, what have we tested? We have automated opening Chrome and the application, signing into the application, searching for a user, signing out, closing chrome, and closing the application. We have not verified our test script and are not sure if it truly passed or failed. While the console shows that the tests have passed, that’s only because there was no error in our automation code. If I were to remove all code in the searchUser test method, then run it, this would still show as passing. That’s not right.

**What Are TestNG Assertions**

TestNG assertions verify if our test truly passed or failed. It’s a line of code that is placed in our test method to verify a condition.

**TestNG Assertion Methods**

There are many assertions for TestNG but most of them are overloaded versions of the following methods:

* assertTrue - verifies a condition is true
* assertFalse - verifies a condition is false
* assertSame - verifies that two objects refer to the same object
* assertNotSame - verifies that two objects do not refer to the same object
* assertNotNull - verifies that an object is not null
* assertEquals - verifies that two objects are equal

Generally, all of these TestNG assertions have the same three parameters: actual result, expected result, and a String. It’s the same with JUnit assertions which have an assertion class located in TestNG’s distribution.

**JUnit & TestNG Assertions**

JUnit has a class, junit.framework.Assert, that has similar overloaded methods. TestNG turned around and added the same JUnit class, org.testng.AssertJUnit, to its distribution. TestNG added the same class as JUnit to guarantee all assertions keep working if we migrate our test from JUnit to TestNG. TestNG also added another class called org.testng.Assert.

The main difference between JUnit’s class and TestNG’s class is the syntax. Their parameters are available in reverse order. For example, the assertEquals method for JUnit has a String as the first parameter followed by an expected result then an actual result. The same method for TestNG has actual result as the first parameter, expected result as the second parameter then String as the last parameter.

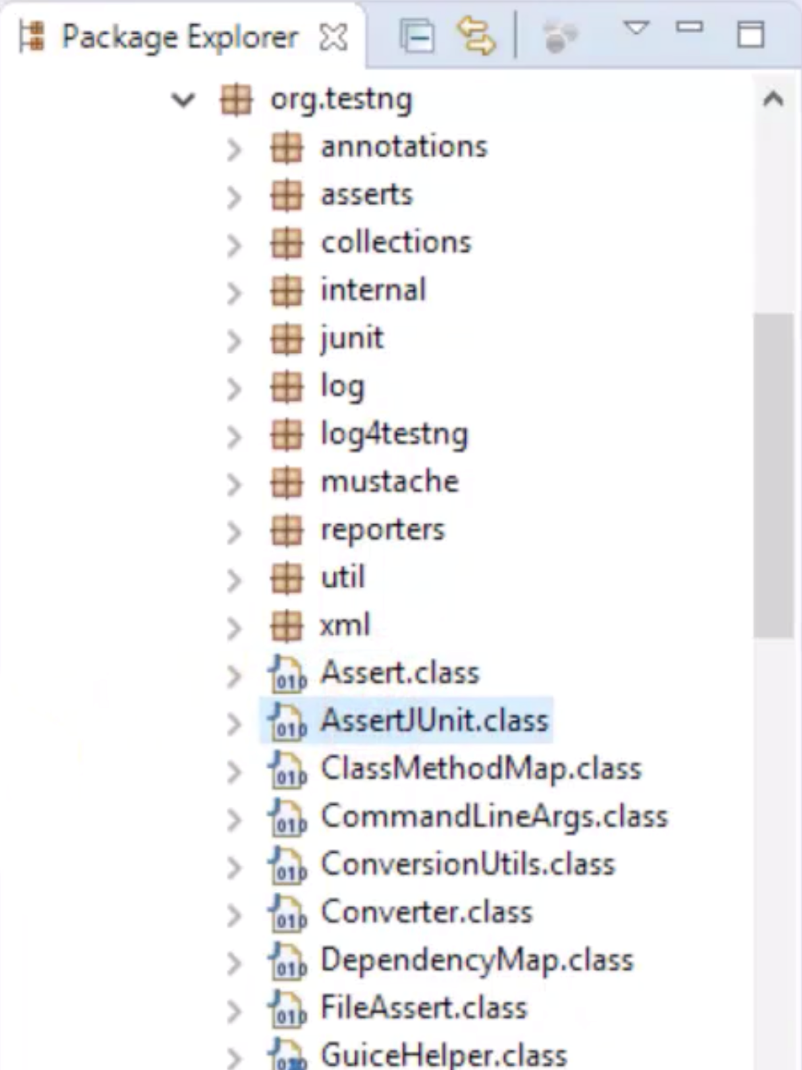
//JUnit Syntax

assertEquals("", expected, actual);

//TestNG Syntax

assertEquals(actual, expected, "");

Let’s go to Eclipse and I’ll show you the assertions. Go to the TestNG Library, maximize the TestNG jar file, select org.testng package, and we see both classes: Assert and AssertJUnit class.



Maximize Assert and there’s a lot of methods. Maximize AssertJUnit and we some of the same overloaded methods (you can see both of these classes in the **Resources** section below). The methods within our Assert class are considered **hard asserts**. In the next section, we will cover hard asserts.

## Resources

[TestNG Assert class](https://static.javadoc.io/org.testng/testng/6.8.17/org/testng/Assert.html)[TestNG AssertJUnit class](https://static.javadoc.io/org.testng/testng/6.8.17/org/testng/AssertJUnit.html)

# 6.2 Hard Asserts

In the previous chapter, we ran our test script and it showed that the test passed even without a verification step.

Let's create new tests with assertions. Our test requirement is to verify the home page for [OrangeHRM](https://opensource-demo.orangehrmlive.com/)

The code will sign in which will take us to the home page. From there, we'll verify the Welcome hyperlink, Admin tab, and Dashboard.

public class VerifySign\_HA\_PASS

{

@BeforeClass

public void setUp() {}

@Test

public void signIn() {}

@Test

public void testHomePageVerification() {}

@Test

public void userSearch() {}

@Test

public void userSignOut() {}

@AfterClass

public void tearDown() {}

}

Let's add assertions to the testHomePageVerification method. Our first assertion will be assertEquals.

@Test

public void testHomePageVerification()

{

Assert.assertEquals(true, true, "The Welcome Link Is Not Correct On The Home Page");

}

The first parameter represents the **actual** value. The second parameter represents the **expected** value. Let's set those both to true for now, just to see how it works. The third parameter is a String and it represents the message that will show up only in the event of an assertion failure.

Let's also add assertions assertFalse and assertTrue. Both of these methods accept a boolean condition, and an optional String for the message. assertTrue asserts that the condition is true, and assertFalse asserts that the condition is false.

@Test

public void testHomePageVerification()

{

Assert.assertEquals(true, true, "The Welcome Link Is Not Correct On The Home Page");

Assert.assertFalse(false, "The Admin Tab Is Not Displayed On The Home Page");

Assert.assertTrue(true, "The Dashboard Is Not Correct On The Home Page");

}

After running this, we see all tests passed. Now, let’s see what happens when this test fails. Let's change the condition in the assertEquals and assertTrue methods. I'll also add print statements after each to demonstrate the flow.

@Test

public void testHomePageVerification()

{

Assert.assertEquals(true, false, "The Welcome Link Is Not Correct On The Home Page");

System.out.println("3. Verify Welcome Link");

Assert.assertFalse(false, "The Admin Tab Is Not Displayed On The Home Page");

System.out.println("4. Verify Admin Tab");

Assert.assertTrue(false, "The Dashboard Is Not Correct On The Home Page");

System.out.println("5. Verify Dashboard");

}

As expected, after running this, testHomePageVerification failed. Let’s look at the output. Steps 3, 4, and 5 are missing. The AssertionError shows our message "The Welcome Link Is Not Correct On The Home Page" because our test failed. However, this is the only AssertionError shown, even though we know that assertTrue should also fail. There is only one AssertionError because we used a **hard assert**.

A hard assert stops immediately after a failure then continues on to the next annotation. In this case, assertEquals failed so the subsequent assertions within this method were skipped and the script went on to the next @Test method, userSearch.

To have all assertions within a test evalauted, you can use **soft asserts**. We'll discuss those in the next section.

## Resources

## [Source Code](https://github.com/RexJonesII/Test-Automation-University/tree/master/chp6assertions)

# 6.3 Soft Asserts

In this chapter, we will discuss the difference between hard asserts and soft asserts.

### Difference Between Hard Asserts and Soft Asserts

A hard assert stops execution after a fail and moves on to the next annotation. It does not matter if the next annotation is a @Test annotation or a configuration annotation.

We saw in the previous chapter that steps 3, 4, and 5 did not execute because they were inside the same annotation that failed and used hard assertion. Therefore, the steps after the first fail were skipped and execution picked up at the next test method.

A soft assert continues execution after a fail and moves on to the next statement line. It was designed to keep executing even when a verification step fails.

To use a soft assert, first we declare SoftAssert which is a class in TestNG.

SoftAssert softassert = new SoftAssert();

Next, we replace all of the Assert references with our object reference softassert. The assertion methods assertEquals, assertFalse, and assertTrue will stay the same.

@Test

public void testHomePageVerification()

{

softassert.assertEquals(true, false, "The Welcome Link Is Not Correct On The Home Page");

System.out.println("3. Verify Welcome Link");

softassert.assertFalse(false, "The Admin Tab Is Not Displayed On The Home Page");

System.out.println("4. Verify Admin Tab");

softassert.assertTrue(false, "The Dashboard Is Not Correct On The Home Page");

System.out.println("5. Verify Dashboard");

}

The SoftAssert class actually only has two methods: assertAll and doAssert. We need to use assertAll every time for soft assert to work. If assertAll is not used then our test will pass. I’m going to run without assertAll then run with assertAll to show you the difference.

Without assertAll, we see that testHomePageVerification passes even though we know that the conditions are not met for steps 3 and 5. SoftAssert kept executing although the verification step failed.

Now, let’s add assertAll to the end of the test method

softassert.assertAll();

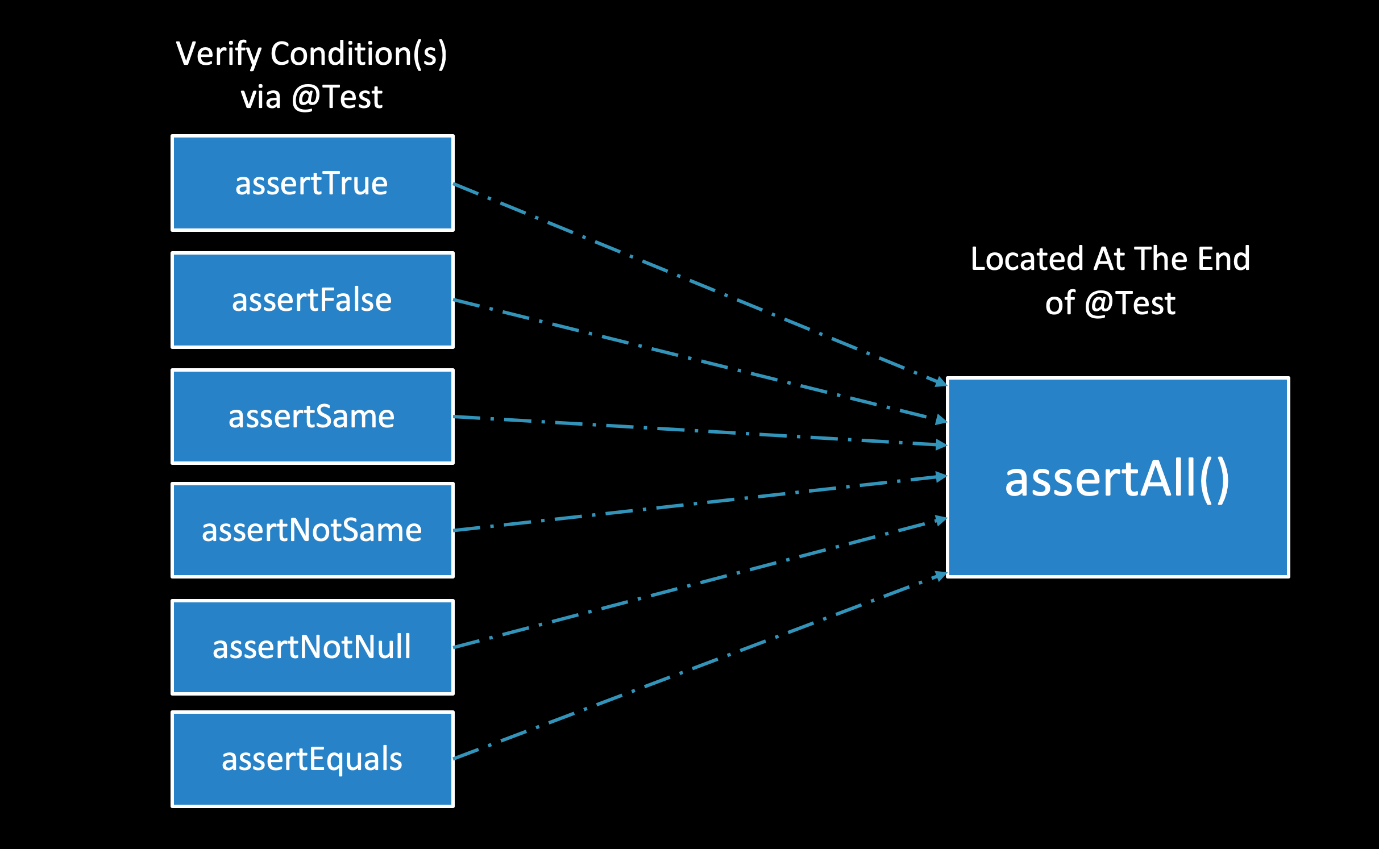
When we run this we see that testHomePageVerification now fails and the assertions which failed have been printed to the console:

java.lang.AssertionError: The following asserts failed:

The Welcome Link Is Not Correct On The Home Page expected [false] but found [true],

The Dashboard Is Not Correct On The Home Page expected [true] but found [false]

Here’s a diagram that shows assertAll. On the left, we see a few assertion methods and assertAll on the right. The arrows are pointing to assertAll because they indicate an AssertionError gets stored into assertAll if there is a failure. That’s why we place the assertAll method at the end of our test method.



### Hard Asserts vs Soft Asserts

Which one should we use for automation? It depends on our test. Sometimes it’s good to have a hard assertion and other times it’s not good to have a hard assertion. The same with soft assertions. Let’s consider these print statements.

import org.testng.annotations.Test;

public class HardAssert\_SoftAssert

{

@Test

public void exampleAssertTest ()

{

System.out.println("1. Open Browser");

System.out.println("2. Open Application");

System.out.println("3. Sign Into The Application");

System.out.println("4. Go To Home Page & Verify Home Page");

System.out.println("5. Go To Search Page & Verify Search Page");

System.out.println("6. Search For User");

System.out.println("7. Sign Out Of The Application");

}

}

We know hard asserts stop if a verification step fails and will not execute the next statement but soft asserts will execute the next statement.

If we had a failure after opening the browser then we would not want to continue executing the next step. There is no reason. With that scenario, it’s best to use a hard assert.

The same with opening an application. There is no reason to keep executing our test script if it’s a failure opening the application. The next step which is to sign into the application would return a failure.

However, we would not use a hard assert after step 4. If there’s a failure on the home page, we still want to verify step 5.

To sum up assertions, we want to use hard asserts where it does not make sense to keep executing our test after a verification failure. We use soft asserts when we want to continuing executing our test even in the event of a failure.

## Resources

[Source Code](https://github.com/RexJonesII/Test-Automation-University/tree/master/chp6assertions)

# 7.1 Intro to Dependency Testing

Dependency testing is one of the main features that separates TestNG from all other test frameworks. After reading [Next Generation Java Testing](https://www.amazon.com/Next-Generation-Java-Testing-Advanced/dp/0321503104) , I came across a section called "Dependent" Code and it mentioned some developers in the testing community are strongly opposed to any hint of dependencies in their tests.

One of their arguments stated that as soon as test methods depend on other test methods, it becomes hard to run those methods in isolation. That argument was true until TestNG.

Now, we are allowed to run test methods in isolation and part of a dependency. TestNG provides a way to clearly indicate those dependencies. Therefore, it has no problem considering the requirements for any test method we want to run in isolation.

Attributes are used to assist annotations with our dependency testing. All three of these attributes can be placed in the configuration and test annotations:

* dependsOn
* groups
* dependsOnGroups

So what exactly is dependency testing? Dependency testing is when two or more actions, two or more tasks, or two or more functions depend on the order of their methods. For example, we must sign into the application before we can sign out of the application. Therefore, signing out depends on us first signing in. If we, reverse that order and execute, then the sign out method would fail.

We have seen a form of dependency testing in Chapter 3 with annotations. The test methods depend on the configuration annotations. If a configuration annotation fails then the test methods would not pass. Here’s an example of a failed configuration. Let’s run.

public class FAIL\_ConfigurationAnnotation

{

@BeforeClass

public void setUp() {}

@Test

public void signIn() {}

@Test

public void userSearch() {}

@Test

public void userSignOut() {}

@AfterClass

public void tearDown() {}

}

We see one configuration failure and one skip. The @BeforeClass configuration failed and the @AfterClass configuration was skipped.

Then we see the test methods. There are three runs, three skips, and no failures. The test methods were skipped because the @BeforeClass configuration failed. At the core, that is dependency testing - the test methods were skipped and did not fail. You can get more information about Dependencies on [TestNG website](https://testng.org/doc/documentation-main.html).

## Resources

[Source Code](https://github.com/RexJonesII/Test-Automation-University/tree/master/chp7dependencytesting)

[Dependencies via TestNG Table of Contents](https://testng.org/doc/documentation-main.html)

# 7.2 dependsOnMethods Attribute

### Executing without dependsOnMethods Attribute

In this example, we have seven test methods.

public class DependsOnMethods\_No

{

@Test

public void test1\_SetUpChrome() {}

@Test

public void test2\_OpenOrangeHRM() {}

@Test

public void test3\_SignIn() {}

@Test

public void test4\_SearchUser() {}

@Test

public void test5\_SearchEmployee() {}

@Test

public void test6\_SearchCandidate() {}

@Test

public void test7\_SignOut() {}

}

Let's add code to set up Chrome and then also code to open the application. However, we'll use an invalid URL for the application so that test2\_OpenOrangeHRM fails.

public class DependsOnMethods\_No

{

WebDriver driver;

@Test

public void test1\_SetUpChrome()

{

System.setProperty("webdriver.chrome.driver", "C:\\Users\\Rex Allen Jones II\\Downloads\\Drivers\\chromedriver.exe");

driver = new ChromeDriver ();

System.out.println("1. Set Up Chrome");

}

@Test

public void test2\_OpenOrangeHRM()

{

//Invalid URL

driver.get("https://opensource-demo.orangehrmlive1234.com/");

Assert.assertEquals(false, true, "Could Not Access OrangeHRM");

System.out.println("2. Open OrangeHRM");

}

@Test

public void test3\_SignIn() {}

@Test

public void test4\_SearchUser() {}

@Test

public void test5\_SearchEmployee() {}

@Test

public void test6\_SearchCandidate() {}

@Test

public void test7\_SignOut() {}

}

Let’s run our test script and see what happens. As expected, the site cannot be reached.

**Console Output**

Tests run: 7, Failures: 6, Skips: 0

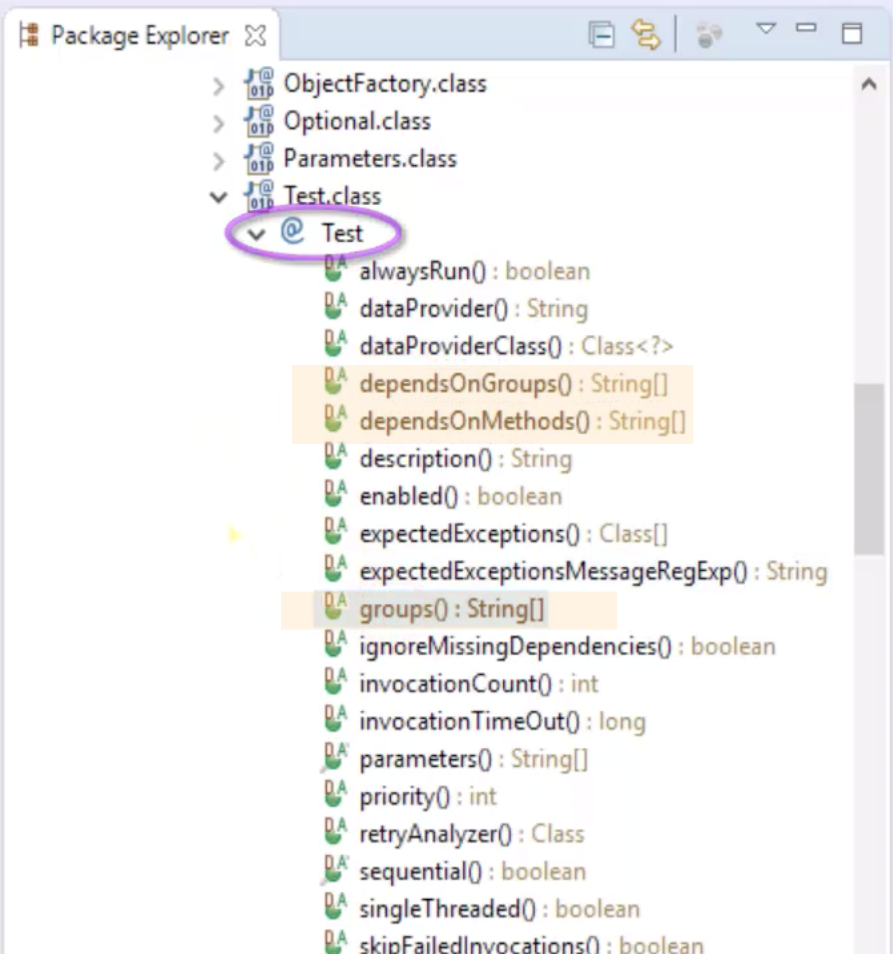
On the console, we see there's a cascade of failures - that's when one failure forces the remaining tests in the suite to fail. We see 7 runs and 6 of those runs are failures.

This is not correct. Ideally, we should have 1 pass (test1\_SetUpChrome), 1 failure (test2\_OpenOrangeHRM), and the remaining 5 tests should be skipped.

To solve this problem, we can use the dependsOnMethods attribute.

### Execute with dependsOnMethods Attribute

If we go to the annotations package, we can find the dependsOnMethods attribute in the @Test annotation along with the groups and dependsOnGroups attributes. There are several others, but we'll only focus on these three within this course.



To add the dependsOnMethod attribute to a test, we add a parentheses after the @Test annotation, write in "dependsOnMethod =" and then paste the name of the method that this test depends on.

@Test

public void test1\_SetUpChrome ()

{

System.setProperty("webdriver.chrome.driver", "C:\\Users\\Rex Allen Jones II\\Downloads\\Drivers\\chromedriver.exe");

driver = new ChromeDriver ();

System.out.println("1. Set Up Chrome");

}

//Invalid URL

@Test (dependsOnMethods = "test1\_SetUpChrome")

public void test2\_OpenOrangeHRM ()

{

driver.get("https://opensource-demo.orangehrmlive1234.com/");

Assert.assertEquals(false, true, "Could Not Open OrangeHRM");

System.out.println("2. Open OrangeHRM");

}

Here we have specified that test2\_OpenOrangeHRM depends on test 1\_SetUpChrome.

Let's add additional dependencies for the other methods.

@Test

public void test1\_SetUpChrome() {}

@Test (dependsOnMethods = "test1\_SetUpChrome")

public void test2\_OpenOrangeHRM() {}

@Test (dependsOnMethods = "test2\_OpenOrangeHRM")

public void test3\_SignIn() {}

@Test (dependsOnMethods = "test3\_SignIn")

public void test4\_SearchUser() {}

@Test (dependsOnMethods = { "test2\_OpenOrangeHRM", "test3\_SignIn" } )

public void test5\_SearchEmployee() {}

@Test (dependsOnMethods = { "test2\_OpenOrangeHRM", "test3\_SignIn" } )

public void test6\_SearchCandidate() {}

@Test (dependsOnMethods = { "test2\_OpenOrangeHRM", "test3\_SignIn" } )

public void test7\_SignOut() {}

}

There are some cases where a test needs to depend on more than one test method. This is the case for test5\_SearchEmployee, test6\_SearchCandidate, and test7\_SignOut. In those cases, noticed we have used curly braces ({}) to enclose the methods that the test depends on.

If we run this now, we'll see 1 pass (test1\_SetUpChrome), 1 fail(test2\_OpenOrangeHRM) and the remaining 5 tests as skipped.

### Exclude Test Method via XML File

I've converted our class into this XML file:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Test Automation University Suite">

<test thread-count="5" name="Depends On Methods Attribute Test">

<classes>

<class name="com.testautomationu.chp6dependencytesting.DependsOnMethods\_PASS">

<methods>

<exclude name = "test4\_SearchUser"/>

</methods>

</class>

</classes>

</test> <!-- Depends On Methods Attribute Test -->

</suite> <!-- Test Automation University Suite -->

By default, a TestNG XML file does not include a <methods> tag because it’s optional. For our example, I added an opening and closing <methods> tag then used it to exclude a test method. That test method is test4\_SearchUser.

Let’s imagine, everyone in the organization knows that searching for a user is not working. The defect has been reported and we will add it back to our test suite once the defect has been resolved. In the meantime, we don’t have to keep executing that test. We can exclude it for now.

If we run this, we'll see there are no failures or skips, and also that test4\_SearchUser was not executed.

## Resources

[Source Code](https://github.com/RexJonesII/Test-Automation-University/tree/master/chp7dependencytesting)

[Dependencies via TestNG Table of Contents](https://testng.org/doc/documentation-main.html)

# 7.3 groups and dependsOnGroups Attributes

In this chapter, we will discuss the groups and dependsOnGroups attributes, and execute groups using an XML file.

### groups Attribute

The groups attribute is a way to add any number of test methods to a named list. This feature is beneficial because it’s an opportunity to run only those lists of test methods according to their group name. The group names can consist of a module of our application, a certain test type, or whatever else we decide. Plus, one test method can be a part of more than one group. For example, you can add the sign in test method to a group for smoke tests, and to another group for regression tests. A class can also be added to a group.

In this example, test1\_SetUpChrome is added to a group called initialize.

@Test (groups = "initialize")

public void test1\_SetUpChrome() {}

### dependsOnGroups Attribute

The dependsOnGroups attribute is similar to the dependsOnMethods attribute in that it allows us to define a group name that our test method depends on. Every method member of a group is guaranteed to be invoked before a method that depends on this group.

Let's make the test method test2\_OpenOrangeHRM depend on the initialize group.

@Test (groups = "initialize")

public void test1\_SetUpChrome() {}

@Test (dependsOnGroups = "initialize")

public void test2\_OpenOrangeHRM() {}

We can also make test2\_OpenOrangeHRM a part of a new group which we'll call env\_ application.

@Test (groups = "initialize")

public void test1\_SetUpChrome() {}

@Test (dependsOnGroups = "initialize", groups = "env\_application")

public void test2\_OpenOrangeHRM() {}

Let’s pretend there is a different URL for each environment such as a dev environment, QA environment, and production environment. We can have a group for each of these environments.

We are not required to add our test methods to a group. The remaining test methods do not belong to a group but we'll have them depend on a group.

@Test (groups = "initialize")

public void test1\_SetUpChrome() {}

@Test (dependsOnGroups = "initialize", groups = "env\_application")

public void test2\_OpenOrangeHRM() {}

@Test (dependsOnGroups = "env\_application")

public void test3\_SignIn() {}

@Test (dependsOnGroups = "env\_application")

public void test4\_SearchUser() {}

@Test (dependsOnGroups = "env\_application")

public void test5\_SearchEmployee() {}

@Test (dependsOnGroups = "env\_application")

public void test6\_SearchCandidate() {}

@Test (dependsOnGroups = "env\_application")

public void test7\_SignOut() {}

When we run this, we see that all tests pass. But let's see what happens when there's a failure. We'll change our application's URL to an invalid one then run again.

@Test (groups = "initialize")

public void test1\_SetUpChrome() {}

@Test (dependsOnGroups = "initialize", groups = "env\_application")

public void test2\_OpenOrangeHRM()

{

//Invalid URL

driver.get("https://opensource-demo.orangehrmlive1234.com/");

Assert.assertEquals(false, true, "Could Not Open OrangeHRM");

System.out.println("2. Open OrangeHRM");

}

@Test (dependsOnGroups = "env\_application")

public void test3\_SignIn() {}

@Test (dependsOnGroups = "env\_application")

public void test4\_SearchUser() {}

@Test (dependsOnGroups = "env\_application")

public void test5\_SearchEmployee() {}

@Test (dependsOnGroups = "env\_application")

public void test6\_SearchCandidate() {}

@Test (dependsOnGroups = "env\_application")

public void test7\_SignOut() {}

**Console Output**

Total test runs: 7, Failures: 1, Skips: 5

There's one failure and five skips, just like with the dependsOnMethods attribute.

### Execute Groups via XML File

TestNG provides a few ways to run groups at runtime. The main way is to use an XML file but we can also use the command line or Ant.

Let's look at a new class with six test methods.

import org.testng.annotations.Test;

public class Amazon\_FakeSite

{

@Test (groups = "smoke")

public void test1\_LogIn ()

{

System.out.println("1. Log In / Group = smoke" + "\n");

}

@Test (groups = { "smoke", "regression" } )

public void test2\_SearchProducts ()

{

System.out.println("2. Search Products / Group = smoke, regression" + "\n");

}

@Test (groups = { "regression", "integration" , "defect.fix"} )

public void test3\_PlaceOrder ()

{

System.out.println("3. Place Order / Group = regression, integration, defect.fix" + "\n");

}

@Test (groups = { "system", "defect.backlog" } )

public void test4\_SendConfirmation ()

{

System.out.println("4. Send Confirmation / Group = system, defect.backlog" + "\n");

}

@Test (groups = { "regression", "defect.progress" } )

public void test5\_ShipOrder ()

{

System.out.println("5. Ship Order / Group = regression, defect.progress" + "\n");

}

@Test

public void test6\_LogOut ()

{

System.out.println("6. Log Out / Group = None" + "\n");

}

}

We've added test1 and test2 to a `smoke` group.

test2 is also a part of a regression group along with test3 and test 5.

In addition to the regression group, test3 also belongs to the integration and defect.fix groups. We have the option of being very descriptive with our group name so noted I added .fix as part of the group name indicating that this test method has a defect that should be fixed. When it is fixed, we can remove this test from the defect.fix group.

I've added test4 to the system and defect.backlog groups.  
And test5 has been added to the regression and defect.progress groups.

test6 is not a part of a group.

Let’s execute some scenarios for groups at runtime using this XML file:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Test Automation Suite">

<test name="All Groups Test">

<groups>

<run></run>

</groups>

<classes>

<class name="com.testautomationu.chp6dependencytesting.Amazon\_FakeSite"/>

</classes>

</test> <!-- All Groups Test -->

</suite> <!-- Test Automation Suite -->

Notice, the <groups> tag and <run> tag. The <groups> tag allows two tags: <run> and <define>.

The <run> tag lists the groups that can be included or excluded. This file does not include or exclude any groups. Therefore, all groups will show up.

The <define> tag helps us create a new group based on existing groups.

Running this XML file will execute all groups within the Amazon\_FakeSite class.

The next xml file includes a regression group.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Test Automation Suite">

<test name="Include Regression Defect Groups Test">

<groups>

<run>

<include name = "regression"/>

</run>

</groups>

<classes>

<class name="com.testautomationu.chp6dependencytesting.Amazon\_FakeSite"/>

</classes>

</test> <!-- All Groups Test -->

</suite> <!-- Test Automation Suite -->

When we run this, we see the following:

**Console Output**

2. Search Products / Group = smoke, regression

3. Place Order / Group = regression, integration, defect.fix

5. Ship Order / Group = regression, defect.progress

Notice only the tests that were apart of the regression group were executed, because this is what we specified in our XML file. These methods happen to be a part of other groups as well, but they were only executed because they belong to the regression group.

In the XML file, we can include more than one group. This example includes the smoke and system groups, which includes three methods from our class.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Test Automation Suite">

<test name="Include Smoke and System Groups Test">

<groups>

<run>

<include name = "smoke"/>

<include name = "system"/>

</run>

</groups>

<classes>

<class name="com.testautomationu.chp6dependencytesting.Amazon\_FakeSite"/>

</classes>

</test> <!-- All Groups Test -->

</suite> <!-- Test Automation Suite -->

**Console Output**

1. Log In / Group = smoke

2. Search Products / Group = smoke, regression

4. Send Confirmation / Group = system, defect.backlog

Look what happens when we exclude a group, in this case `regression`.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Test Automation Suite">

<test name="Exclude Regression Groups Test">

<groups>

<run>

<exclude name = "regression"/>

</run>

</groups>

<classes>

<class name="com.testautomationu.chp6dependencytesting.Amazon\_FakeSite"/>

</classes>

</test> <!-- All Groups Test -->

</suite> <!-- Test Automation Suite -->

**Console Output**

1. Log In / Group = smoke

4. Send Confirmation / Group = system, defect.backlog

6. Log Out / Group = None

As expected, no test methods which are assigned to the regression group have been executed.

We also have the ability to use regular expressions. In this XML file, defects has a regular expression using a dot and asterisk.

<exclude name = "defect.\*"/>

Let's use that example where all groups that begin with defect. are excluded, and let's also indicate to include the regression group:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Test Automation Suite">

<test name="Include Regression - Exclude Defect Groups Test">

<groups>

<run>

<include name = "regression"/>

<exclude name = "defect.\*"/>

</run>

</groups>

<classes>

<class name="com.testautomationu.chp6dependencytesting.Amazon\_FakeSite"/>

</classes>

</test> <!-- All Groups Test -->

</suite> <!-- Test Automation Suite -->

Even though we have three tests as part of the regression group, only one of them are executed because it has filtered out the ones that are a part of group that begins with defect.

**Console Output**

2. Search Products / Group = smoke, regression

### Run Test Methods According to Group Name

Let’s recap the rules we covered when executing groups at runtime:

* a <run> that has no <include> and no <exclude> will run all test methods
* if we include a group then only test methods assigned to that group will run
* if we include more than one group then members of all included groups will be run
* excluding a group runs all test methods except the test methods assigned to that group
* we can use regular expressions to specify group names that should be included or excluded
* if a test method belongs to a group that is included but also belongs to another group that is excluded, then the test will not run because excluded wins

Up to this point, all of the test methods that we executed were from the same class. However, TestNG is not limited to executing a group using one class. We can execute groups using more than one class or even a package.

This XML file has two <class> tags.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Test Automation Suite">

<test name="Include 2 Classes Groups Test">

<groups>

<run>

<include name = "regression"/>

<include name = "env\_application"/>

</run>

</groups>

<classes>

<class name="com.testautomationu.chp6dependencytesting.Amazon\_FakeSite"/>

<class name="com.testautomationu.chp6dependencytesting.Groups\_DependsOnGroups\_PASS"/>

</classes>

</test> <!-- All Groups Test -->

</suite> <!-- Test Automation Suite -->

We have included the regression and env\_application groups from both the Amazon\_FakeSite and Groups\_DependsOnGroups\_PASS.

When we run this, all test methods that belong to these two groups and are within these two classes are executed.

## Resources

[Source Code](https://github.com/RexJonesII/Test-Automation-University/tree/master/chp7dependencytesting)

# Data Driven Testing

In this chapter, we will discuss the @DataProvider annotation, and the dataProvider and dataProviderClass attributes.

Let’s take a look at the code before discussing the annotation and both attributes.

public class LogIn1

{

public void logIn (String email, String password, boolean success)

{

System.out.println("Log In Credentials: " + "\n" +

" Email = " + email + "\n" +

" Password = " + password + "\n" +

" Successful Log In = " + success + "\n" );

}

public Object [] [] logInData ()

{

Object [][] data = new Object [3][3];

data [0][0] = "TestNG@Framework.com"; data [0][1] = "TestNG1234"; data [0][2] = true;

data [1][0] = "Joe@Doe.com"; data [1][1] = "DoeDoe34"; data [1][2] = false;

data [2][0] = "Test@AutomationU.com"; data [2][1] = "TAU1234"; data [2][2] = true;

return data;

}

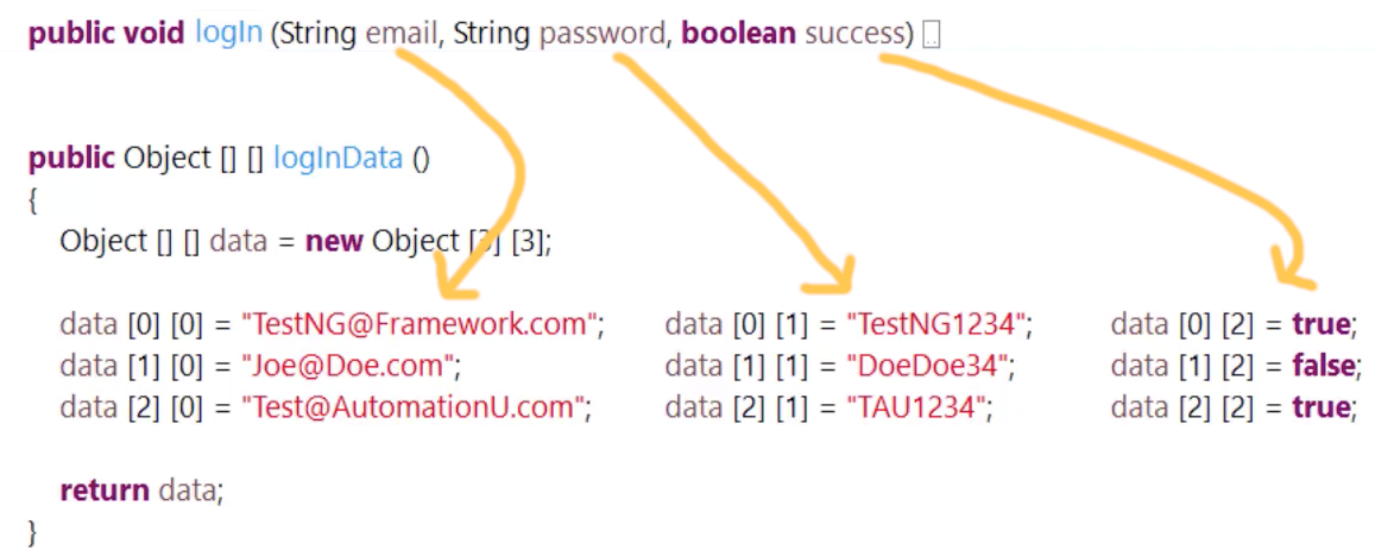
}

Let's look at the logIn method.

public void logIn (String email, String password, boolean success)

This method has 3 parameters: email, password, and success. They will receive their argument values from the logInData method.

The values are placed in a 2-dimensional array. There are three rows of test data.



The first column of data will pass email values to String email.

The second column of data will pass password values to String password.

The third column of data will pass success values to boolean success.

The statement return data helps us return values to the logIn method. After the logIn method receives the argument values then the parameters use those values. That’s a brief overview of this Java code for parameters, arguments, and the 2-dimensional array.

### DataProvider annotation

Now, let’s discuss the @DataProvider annotation. The @DataProvider annotation returns Java objects which are values to the test method. We implement this annotation by writing @DataProvider above the method that has the test data.

In this case, logInData has the test data. The logInData method will supply data to the logIn test method.

The @DataProvider annotation presents two purposes at the same time. The first purpose is to pass an unlimited number of values to a test method. There is no restriction. The values can be any Java data type. In our example, we use a String and boolean data type.

The second purpose is to allow the test method to be invoked with different data sets. Each set will run and have its own test results. In our example, we have three sets of data.

### dataProvider Attribute

The dataProvider attribute connects the @DataProvider annotation to the test method. This is how both methods converse with each other. Since it’s an attribute of the @Test annotation, we write dataProvider with a lowercase d and set it equal to the @DataProvider's method name surrounded by double quotes.

public class LogIn1

{

@Test (dataProvider = "logInData")

public void logIn (String email, String password, boolean success)

{

System.out.println("Log In Credentials: " + "\n" +

" Email = " + email + "\n" +

" Password = " + password + "\n" +

" Successful Log In = " + success + "\n" );

}

@DataProvider

public Object [] [] logInData ()

{

Object [][] data = new Object [3][3];

data [0][0] = "TestNG@Framework.com"; data [0][1] = "TestNG1234"; data [0][2] = true;

data [1][0] = "Joe@Doe.com"; data [1][1] = "DoeDoe34"; data [1][2] = false;

data [2][0] = "Test@AutomationU.com"; data [2][1] = "TAU1234"; data [2][2] = true;

return data;

}

}

Let’s run.

**Console Output**

Log In Credentials:

Email = TestNG@Framework.com

Password = TestNG1234

Successful Log In = true

Log In Credentials:

Email = Joe@Doe.com

Password = DoeDoe34

Successful Log In = false

Log In Credentials:

Email = Test@AutomationU.com

Password = TAU1234

Successful Log In = true

The output shows three separate tests, each with a row of the data provider.

We can also give the @DataProvider annotation a name. In this example, the name is login-provider. If a data provider has a name, we connect the @DataProvider annotation and the test method by using the data provider's name and not the method name. But as you can see, the data provider's name is optional as the test was executed successfully the last time without a name. Either way is okay.

### dataProviderClass Attribute

The dataProviderClass attribute allows us to separate the test method and data provider into different classes. Up to this point, our test method and data provider were located in the same class. The dataProviderClass is an attribute of the @Test annotation just like the dataProvider.

Here we have one class which contains the data provider:

import org.testng.annotations.DataProvider;

public class SignInDP

{

@DataProvider (name = "signin-provider")

public static Object [] [] signInData ()

{

Object [] [] data = new Object [4] [3];

data [0] [0] = "Invalid"; data [0] [1] = "Invalid123"; data [0] [2] = false;

data [1] [0] = "Admin"; data [1] [1] = "admin123"; data [1] [2] = true;

data [2] [0] = "admin"; data [2] [1] = "admin123"; data [2] [2] = true;

data [3] [0] = "NotValid"; data [3] [1] = "NotValid34"; data [3] [2] = false;

return data;

}

}

and another class which contains the test method:

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

import org.testng.Assert;

import org.testng.annotations.Test;

public class OrangeHRM

{

WebDriver driver;

@Test (dataProviderClass = SignInDP.class, dataProvider = "signin-provider")

public void signIn (String usename, String password, boolean success)

{

System.setProperty("webdriver.chrome.driver", "C:\\Users\\Rex Allen Jones II\\Downloads\\Drivers\\chromedriver.exe");

driver = new ChromeDriver ();

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

driver.get("https://opensource-demo.orangehrmlive.com");

driver.findElement(By.id("txtUsername")).sendKeys(usename);

driver.findElement(By.id("txtPassword")).sendKeys(password);

driver.findElement(By.id(btnLogin)).click();

System.out.println("Sign In Credentials: " + "\n" +

" Username = " + usename + "\n" +

" Password = " + password + "\n" +

" Successful Sign In = " + success + "\n" );

String actualResult = driver.findElement(By.id("welcome)).getText();

String expectedResult = "Welcome Admin";

Assert.assertEquals(actualResult, expectedResult, "The Actual & Expected Results Do Not Match");

driver.quit();

}

}

## Resources

[Source Code](https://github.com/RexJonesII/Test-Automation-University/tree/master/chp8datadriventesting)

# 9. Cross-Browser Testing

In this chapter, we will discuss the <parameter> tag for the TestNG XML file, the @Parameters annotation, and the different ways to supply test data.

Cross-browser testing is a form of data-driven testing because we can drive different data sets using the <parameter> tag and @Parameters` annotation.

I’m going to send different browser names from the XML file to the @Parameters annotation.

### Parameter Tag via XML File

The <parameter> tag specifies the name and value of a parameter. In this XML file, we have the same class name “Test Automation U” located in three different tests: one for Internet Explorer, one for Firefox, and one for Chrome.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Cross Browser Testing">

<parameter name = "URL" value = "https://testautomationu.applitools.com/"/>

<test name = "Test On IE">

<parameter name = "BrowserType" value = "Internet Explorer"/>

<classes>

<class name = "com.testautomationu.chp8crossbrowsertesting.TestAutomationU"/>

</classes>

</test> <!-- Test On IE -->

<test name = "Test On Firefox">

<parameter name = "BrowserType" value = "Firefox"/>

<classes>

<class name = "com.testautomationu.chp8crossbrowsertesting.TestAutomationU"/>

</classes>

</test> <!-- Test On Firefox -->

<test name = "Test On Chrome">

<parameter name = "BrowserType" value = "Chrome"/>

<classes>

<class name = "com.testautomationu.chp8crossbrowsertesting.TestAutomationU"/>

</classes>

</test> <!-- Test On Chrome -->

</suite> <!-- Cross Browser Testing -->

We have the option of placing the <parameter> tag within the <suite> level or <test> level. The <parameter> tag will get overridden at the <test> level if we add the tag in both places with the same parameter name. This is because the <test> level is the closest to the <class> level which uses the parameter name. I decide the level to place the parameter name and value by determining if all classes need the same value. If all classes need the same value then I add the <parameter> tag at the suite level. If the classes require a unique value then I add the <parameter> tag at the test level.

This XML file has different parameter names. We see URL and BrowserType. All three classes need to access the same website. Therefore, at the <suite> level, we have parameter name as “URL” in parenthesis then the value of the URL is https://testautomationu.applitools.com.

Next, is the <test> level. The parameter name will be the same for each test but the values are going to be different. Let’s start with a test on IE. The parameter name is BrowserType and the value is InternetExplorer. This test will use Internet Explorer as a browser.

The next test uses Firefox as the browser, and the last test is uses Chrome.

### Parameters Annotation

The purpose of a @Parameters annotation is to point out how to pass parameters and which parameters to pass to the test method.

We are going to pass the URL and BrowserType from the XML to the @Parameters annotation. The value we write in this annotation must match the parameter name from the XML file. The parameter names are URL and BrowserType.

@Test

@Parameters ( {"URL", "BrowserType"} )

public void verifyTAU (String url, String browserType) {}

Parameters in the annotation are different from parameters in the test method. Parameters in the annotation are a list of parameter names to be looked up in the XML file. Parameters in the test method receive the values from the XML file. Therefore, one gets the parameter name and the other one gets the parameter value.

Let's add code to this method to instantiate the proper driver based on the browser specified.

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

import org.openqa.selenium.ie.InternetExplorerDriver;

import org.testng.annotations.Parameters;

import org.testng.annotations.Test;

public class TestAutomationU

{

WebDriver driver;

@Test

@Parameters ( {"URL", "BrowserType"} )

public void verifyTAU (String url, String browserType)

{

if (browserType.equalsIgnoreCase("Internet Explorer"))

{

System.setProperty("webdriver.ie.driver", "C:\\Users\\Rex Allen Jones II\\Downloads\\Drivers\\IEDriverServer\_Win32\_2.53.1\\IEDriverServer.exe");

driver = new InternetExplorerDriver ();

}

else if (browserType.equalsIgnoreCase("Firefox"))

{

driver = new FirefoxDriver ();

}

else if (browserType.equalsIgnoreCase("Chrome"))

{

System.setProperty("webdriver.chrome.driver", "C:\\Users\\Rex Allen Jones II\\Downloads\\Drivers\\chromedriver.exe");

driver = new ChromeDriver ();

}

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

driver.get(url);

System.out.println("\n" + "Open " + browserType);

System.out.println(" " + driver.getTitle());

System.out.println("Close " + browserType + "\n");

driver.quit();

}

}

To execute this, we must run the XML file. A failure will occur if we execute from the class.

### Different Ways to Supply test data

In this chapter, we hard-coded our test data sets. However, we can also use other ways to supply our test data such as CSV file, database, properties file, or Microsoft Excel. All of the ways have their pros and cons.

### Additional Concepts

Here’s some additional TestNG concepts that were not covered in this course:

* The ability to disable a test method by setting enable to false
* Execute a package at runtime
* Executing TestNG from the command prompt
* Provide optional values using the @Optional annotation
* Add Listeners which is a registration for test results
* Add Logs
* And the ability to view default reports
* Create custom reports
* Multithreading which executes multiple components of a program at the same time and there’s more TestNG concepts available to us

I want to end by saying thanks to Angie Jones. She’s good, I like her and thanks to Applitools. They gave me an opportunity to be your instructor for this course.

You can reach me at [Rex.Jones@Test4Success.org](mailto://Rex.Jones@Test4Success.org). I have a social network that provides videos on Selenium, Java, and TestNG. The videos are available on YouTube, LinkedIn, and Facebook. I am also the author of 6 books that covers programming and automation. Some of the books are getting updated to include videos. Thank you for watching this course and I wish you much success.

## Resources

[Source Code](https://github.com/RexJonesII/Test-Automation-University/tree/master/chp9crossbrowsertesting)