SK Institute

**TestNG**

*by*

Srikanth Pittala Lead Automation Engineer

**What is TestNG**?

It is a testing framework developed by Cedric Beust where NG, stands for **Next Generation.**

it is an advance version of JUnit (in Java) and NUnit (in C#). we can use it for unit testing, functional testing, integration testing, and end-to-end testing.

TestNG is written in Java, so mainly contains Java annotations to configure and write test methods.

Features that TestNG has over JUnit4 are:

* Extra Before and After annotations such as Before/After Suite and Before/After Group
* Dependency test
* Grouping of test methods
* Multithreaded execution
* In-built reporting framework

In TestNG, suites and tests are configured through XML files. By default, the name of the file is testng.xml, and it is not mandatory to put the same name, we can change the name if we want to.

Using testng.xml file we can include or exclude respective packages, classes, and methods in test suite. We can also group test methods with particular name, and we can include or exclude these groups from the test execution.

Parameterization of test methods is very easy using TestNG and it also provides an easy method of creating data-driven tests.

**Advantages and features of TestNG:**

**Multiple Before and After annotation options:**

TestNG provides multiple kinds of Before/After annotations for support of different setup and clean up options.

**XML-based test configuration and test suite definition:**

Test suites in TestNG are configured mainly using XML files. An XML file can be used to create suites using classes, test methods, and packages, as well as by using TestNG groups. This file is also used to pass parameters to test methods or classes.

**Dependent methods:**

This is one of the major features of TestNG where you can tell TestNG to execute a dependent test method to run after a given test method. You can also configure whether the dependent test method has to be executed or not in case the earlier test method fails.

**Groups/group of groups:**

Using this feature, you can assign certain test methods into particular named groups (ex: Regression tests, Smoke tests and Sanity tests etc...) and tell TestNG to include or exclude a particular group in a test. Means, you can execute only Regression tests if you want to...

**Dependent groups:**

Like dependent methods, this feature allows test methods belonging to one group being dependent upon another group.

**Parameterization of test methods:**

This feature helps users to pass parameter values through an XML configuration file to the test methods, which can then be used inside the tests.

**Data-driven testing**:

TestNG allows users to do data-driven testing of test methods using this feature. The same test method gets executed multiple times based on the data.

**Multithreaded execution:**

This allows execution of test cases in a multithreaded environment. This feature can be used for parallel test execution to reduce execution time or to test a multithreaded test scenario.

**Better reporting:**

TestNG internally generates an XML and HTML report by default for its test execution. You can also add custom reports to the framework if required.

**Open API**:

TestNG provides easy extension of API, this helps in adding custom extensions or plugins to the framework depending upon the requirement.

**Downloading TestNG**

1. Download & add testng.jar file to your project.
2. Add testng plug-in to eclipse IDE (Install new software or marketplace in eclipse)

**Note : For testng installation we required minimum JDK5**

**Understanding testng.xml file**

testng.xml file is a configuration file in TestNG. It is used to define test suites and tests. It is also used to pass Parameters to the test methods. It provides different options to include packages, classes, and independent test methods in our test suite. It also allows us to configure multiple tests in a single test suite and run them in multi-threaded environment.

**TestNG allows you to:**

* Create tests with packages
* Create tests using classes
* Create tests using test methods
* Include/exclude a particular package, class, or test method
* Use of regular expression while using the include/exclude feature
* Store parameter values for passing to test methods at runtime
* Configure multi-threaded execution options

**Creating testng.xml With Multiple Tests**

**Example:**

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

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

<suite parallel=*"classes"* name=*"Suite"*>

<test thread-count=*"5"* parallel=*"classes"* name=*"Test1"*>

<classes>

<class name=*"PackageOne.GoogleDemo"*/>

<class name=*"PackageOne.GoogleDemo2"*/>

</classes>

</test> <!-- Test -->

<test thread-count=*"5"* parallel=*"classes"* name=*"Test2"*>

<classes>

<class name=*"PackageOne.GoogleDemo3"*/>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

**Adding Packages to testng.xml file**

**Example:**

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

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

<suite name=*"Suite"*>

<test thread-count=*"5"* name=*"Test"*>

<packages>

<package name=*"PackageOne1"*/>

<package name=*"PackageOne2"*/>

<package name=*"PackageOne3"*/>

</packages>

</test> <!-- Test -->

</suite> <!-- Suite -->

**Adding Classes to testng.xml file**

**Example:**

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

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

<suite parallel=*"classes"* name=*"Suite"*>

<test thread-count=*"5"* parallel=*"classes"* name=*"Test1"*>

<classes>

<class name=*"PackageOne.GoogleDemo1"*/>

<class name=*"PackageOne.GoogleDemo2"*/>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

**Adding Methods to testng.xml file**

**Example:**

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

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

<suite name=*"Suite"*>

<test thread-count=*"5"* name=*"Test"*>

<classes>

<class name=*"PackageOne.GoogleDemo2"*>

<methods>

<include name=*"firstmethod"*></include>

<include name=*"secondmethod"*></include>

<include name=*"thirdmethod"*></include>

<exclude name=*"fourthmethod"*></exclude>

</methods>

</class>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

**Or**

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

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

<suite name=*"Suite"*>

<test thread-count=*"5"* name=*"Test"*>

<classes>

<class name=*"PackageOne.GoogleDemo2"*>

<methods>

<exclude name=*"fourthmethod"*></exclude>

</methods>

</class>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

**TestNG Annotations:**

TestNg Annotations will direct the execution flow of the program.

1. **@Before Suite:**

The annotated method will be executed Before any tests declared inside the TestNG suite

1. **@AfterSuite:**

The annotated method will be executed After any tests declared inside the TestNG suite

1. **@Before Test:**

The annotated method will be executed Before test section declared inside the TestNG Suite

1. **@AfterTest:**

The annotated method will be executed After test section declared inside the TestNG suite

1. **@BeforeGroups:**

The annotated method will be executed Before any of the test method belongs to specified group.

1. **@AfterGroups:**

The annotated method will be executed After any of the test method belongs to specified group.

1. **@BeforeClass:**

The annotated method will be executed Before any of the test method of a test class.

1. **@AfterClass:**

The annotated method will be executed After any of the test method of a test class.

1. **@BeforeMethod:**

The annotated method will be executed Before each test method.

1. **@AfterMethod:**

The annotated method will be executed After each test method.

1. **@DataProvider:**

Marks a method as a data providing method for a test method. The said method has to return an Object double array (Object [] []) as data.

1. **@Factory:**

Marks an annotated method as a factory that returns an array of class objects

(Object []). These class objects will then be used as test classes by TestNG. This is used to run a set of test cases with different values.

1. **@Listeners:**

Applied on a test class. Defines an array of test listeners classes extending org testng.ITestNGListener. Helps in tracking the execution status and logging purpose.

1. **@Parameters:**

This annotation is used to pass parameters to a test method. These parameter values are provided using the testng xml configuration file at run time.

1. **@Test:**

Marks a class or a method as a test method. If used at class level, all the **public** methods of a class will be considered as a test method.

**Before and After Annotations:**

Before and After Annotations are very useful while writing the selenium programs. These will control the execution flow of your program. What should execute what time and which statement, or block of code should execute first, and which should be last.

One of the important things is the order of writing the annotations is not predefined. It is your wish where you want to write those in the program. TestNG will take care of the execution flow according to the annotations which we used in the program.

**TestNG Test Annotation:**

The most important annotation of TestNG is Test annotation. This annotation marks a method or class as TestNG test. It you apply at class level then it will mark all the public methods present inside the class as test methods. It had lot of attributes which you can use along with the Test annotation, which will enable you to use the different features provided by the TestNG unit test framework.

**TestNG Test Attribute:**

1. **alwaysRun**

Takes a true or false value. If set to true, this method will always run even if it's **depending on method** or **depending on group** fails.

1. **dataProvider**

The name of the data provider, which will provide data for data-driven testing to this method.

1. **dataProviderClass**

The class where TestNG should look for the data-provider method mentioned in the dataProvider attribute. By default, it's the

Note: it we extend classes, no need to mention this attribute in test annotation.

1. **dependsOnGroups**

Specifies the list of groups this method depends on.

1. **dependsOnMethods**

Specifies the list of methods this method depends on.

1. **Description**

Add description to the test method and display in testng html report.

1. **Enabled**

Sets whether the said method or the methods inside the said class should be enabled for execution or not. By default, its value is true.

1. **expectedExceptions**

This attribute is used for exception testing. This attribute specifies the list of exceptions this method is expected to throw.

1. **Groups**

List of groups the said method or class belongs to.

1. **timeOut**

This attribute is used for a time out test and specifies the time (in millisecs) this method should take to execute.

1. **InvocationCount**

TestNG supports multi-invocation of a test method, i.e., a @Test method can be invoked multiple times sequentially or in parallel. If we want to run single @Test 10 times at a single thread, then invocationCount can be used.

**Running testng.xml Using Command Prompt:**

**There are 2 ways to run the testng.xml file:**

* Using Command Prompt
* Using IDE (i.e., Eclipse)

**Required Jar files**

* testng.jar
* jcommander.jar

Can do in 2 steps:

1. First, we need to set the classpath from the command prompt. For this we need to mention the bin (all compiled files will be residing in this) folder path and required jar files folder.

set classpath=[Project Folder]\bin;[Project Folder]\libs\\*

Example:

set classpath=E:\YouTube\bin;E:\YouTube\libs\\*

1. Second, execute the testng.xml file from the command prompt. java org.testng.TestNG testng.xml.

Instead giving 2 different commands, we can combine and run as a single command to execute the testng.xml file.

Find the command below:

Here, we are in the project folder. So, no need to mention the project folder path explicitly.

java -cp ".\bin;.\libs\\*" org.testng.TestNG testng.xml

In the above command we are adding the TestNG JAR and the project compiled code to the Java classpath by using the -cp option of Java.

In case you would like to execute multiple testng.xml files, we can use the previous command by passing the other XML files as added arguments to the command line.

The following is a sample command:

**java -cp ".\bin;.\libs\\*" org.testng.TestNG testng.xml testng1.xml**

TestNG also allows executing a particular test from the testng.xml file. To execute a particular test from the testng.xml, use the option -testnames at the end of command line with comma-separated names of test that need to be executed.

The following is a sample command:

**java -cp ".\bin;.\libs\\*" org.testng.TestNG-testnames "SampleTest" testng.xml**

**Enable/Disable TestNG Test :**

In some scenarios you may need NOT to Execute/Run some of the tests or set of tests from getting executed in a particular class. For example, consider a scenario where a serious bug exists in application due to certain tests belonging to certain scenarios that cannot be executed. As the issue has already been identified we may need to disable the said test scenarios from being executed.

This test attribute used to enable or disable a test at class level or method level.

Note : This test attribute override xml include node

**Disabling a test can be achieved in 2 ways:**

1. From testng.xml configuration file

Example:

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

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

<suite name=*"Suite"*>

<test thread-count=*"5"* name=*"Test"*>

<classes>

<class name=*"PackageOne.GoogleDemo2"*>

<methods>

<include name=*"firstmethod"*></include>

<include name=*"firstmethod"*></include>

<include name=*"firstmethod"*></include>

<exclude name=*"fourthmethod"*></exclude>

</methods>

</class>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

1. From the Test annotation property/attribute (i.e., enabled).

* By default, this attribute is **true.**
* If this attribute is set to false for the Test annotation at class level, all the public methods inside the class will be disabled.
* If this attribute is set to false for the Test annotation at method level, then only that method will be disabled.

**TestNG Exception Test:**

This test attribute used to specify the type of exceptions that are expected to be thrown by a test method during execution. It supports multiple values being provided for verification. If the exception thrown by the test is not part of the user entered list, the test method will be marked as failed.

**Example**:

@Test

**public** **class** ClassThree

{

@Test(expectedExceptions = {NoSuchElementException.**class**,ElementNotFoundException.**class**})

**public** **void** MethodOne()

{

System.***out***.println("Class Three - Method One");

}

}

**TestNG Time-Out Test:**

While running tests there is a chance of where certain tests get stuck or may take much more time than expected. In such a case you may need to mark the said test case as fail and then continue. TestNG allows user to configure a time period to wait for a test to completely execute. For this we will use Time-out Test annotation property.

**This can be configured in three ways:**

1. **At Suite Level:** This will be applicable for all the tests in the TestNG test suite

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

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

<suite name=*"Suite"* time-out=*"3000"* >

<test thread-count=*"5"* name=*"Test"*>

<classes>

<class name=*"TestNGPractice.ClassOne"*></class>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

2. **At test Level:** This will be applicable for the test in the TestNG test suite

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

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

<suite name=*"Suite"*>

<test thread-count=*"5"* name=*"Test"* time-out=*"3000"*>

<classes>

<class name=*"TestNGPractice.ClassOne"*></class>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

**3.At Method Level:** This will be applicable for the said test method.

**public** **class** ClassOne

{

@Test(timeOut = 2000)

**public** **void** MethodOne() **throws** InterruptedException

{

System.***out***.println("Class One - Method One");

Thread.*sleep*(5000);

}

}

**4.Method Overrides Suite Level**: Method level override the time period if configured at the suite level.

**4.Method Overrides test Level**: Method level override the time period if configured at the test level.

**Parameterization using testng.xml file:**

There are two ways to pass the parameters in the TestNG. For this, need to use Parameters and DataProvider annotations of the TestNG. Parameterization using testng.xml file will allow users to pass parameters to the test methods as an argument using xml file.

Below are the two ways to pass the arguments:

1. Using **testng.xml** configuration file.

2. Using DataProvider.

Using testng.xml configuration file:

Can pass some simple values such as Strings to the test methods at runtime. You can use this approach for sending parameter values through TestNG xml configuration files. For this, you have to use Parameter’s annotation in the code and Parameter tag in the XML file to pass the parameter values to test methods.

1. Suite Level

2. Test Level

3. Class Level

4. Method Level

5. Method Level Overrides Class Level

4. Class Level Overrides Test Level

5. Test Level Overrides Suite Level

The parameter is declared using parameter tag and name attribute defines the name of the attribute and it should match with the @Parameters annotation attribute in the code. And value attribute defines the value of the parameter. The parameter tag can be used in suite level as well as test level.

If you have a parameter at both suite level and test level then the test level parameter will override the suite level parameter, that means the test level will take high priority.

If you forget to give @Parameters annotation in the program and executed the code using xml file, then will get TestNGException. In the above program we will miss the Parameters for the second test and will try to execute the program.

**Optional Parameters in TestNG:**

TestNG optional parameters option will provide optional values to a parameter, this value will be used if parameter value is not found or not provided in the xml configuration file. To achieve this we will use @Optional annotation in the test method as an argument.

**Points to remember:**

1. We can pass the parameters to the test method Two ways.

2. Can pass the parameter using xml file using parameter tag.

3. Can pass the parameter using @Optional annotation in the test method.

4. If NOT provided parameter from the xml file then it will take the optional parameter which is provided in the test method using @Optional annotation.

5. It provided the parameter value from the xml then it will consider this value.

6. The parameter value which is provided in the xml file take the high priority than the optional value and it will override the optional value in the execution.

Example:

**public** **class** ClassOne

{

@Parameters({"sample1","sample2"})

@Test

**public** **void** MethodOne(@Optional("optional text")String text1,String text2) **throws** InterruptedException

{

System.***out***.println("Class One - Method One");

System.***out***.println(text1);

System.***out***.println(text2);

}

}

Xml file:

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

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

<suite name=*"Suite"* >

<parameter name=*"sample1"* value=*"s -Hello siddhi-1"*></parameter>

<parameter name=*"sample2"* value=*"s -Hello siddhi-2"*></parameter>

<test thread-count=*"5"* name=*"Test"* >

<parameter name=*"sample1"* value=*"t- Hello siddhi-1"*></parameter>

<parameter name=*"sample2"* value=*"t -Hello siddhi-2"*></parameter>

<classes>

<class name=*"TestNGPractice.ClassOne"* >

<parameter name=*"sample1"* value=*"c -Hello siddhi-1"*></parameter>

<parameter name=*"sample2"* value=*"c -Hello siddhi-2"*></parameter>

<methods>

<parameter name=*"sample1"* value=*"m- Hello siddhi-1"*></parameter>

<parameter name=*"sample2"* value=*"m- Hello siddhi-2"*></parameter>

</methods>

</class>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

**DataProvider In TestNG:**

TestNG DataProvider is one of the most important features provided by TestNG. By using this feature user can write data-driven tests, that means a test method can run multiple times with different sets of data. In Parameterization using testng.xml file Video we have seen how to parameterize tests using testng.xml file. This feature is also one of the ways to pass the parameters to the test methods. It helps in providing the complex parameters to the test methods as this way is not possible with the testng.xml file.

To use the DataProvider feature in your tests you have to declare a method annotated by DataProvider (i.e. @DataProvider(name = "<data-provider name >")) and then use the said method in the test method using the dataProvider attribute (i.e. @Test(dataProvider = "< data-provider name >")) in the Test annotation.

**Different ways:**

1. DataProvider method and its using method are in Same class.

Example:

public class SameClass

{

@Test(dataProvider = "testdata")

public void MethodOne(String text1) throws InterruptedException

{

System.out.println("Class One - Method One");

System.out.println(text1);

}

@DataProvider(name = "testdata")

public Object[][] data()

{

Object[][] obj = new Object[][]{ {"data1"},{"data2"},{"data1"},{"data2"}};

return obj;

}

}

1. DataProvider method and its using method are in Different classes.

**TestClass**:

**public** **class** DifferentClass

{

@Test(dataProvider = "testdata",dataProviderClass = DifferentData.**class**)

**public** **void** MethodOne(String text1,String text2) **throws** InterruptedException

{

System.***out***.println("Class One - Method One");

System.***out***.println(text1);

System.***out***.println(text2);

}

}

DataProviderClass:

**public** **class** DifferentData

{

@DataProvider(name = "testdata")

**Public static** Object[][] data()

{

Object[][] obj = **new** Object[][]

{ {"data1","data2"},{"data3","data4"}};

**return** obj;

}

}

1. Multiple Parameters.

**TestClass**:

**public** **class** DifferentClass

{

@Test(dataProvider = "testdata",dataProviderClass = DifferentData.**class**)

**public** **void** MethodOne(String text1,String text2,String text3) **throws** InterruptedException

{

System.***out***.println("Class One - Method One");

System.***out***.println(text1);

System.***out***.println(text2);

System.***out***.println(text3);

}

}

**DataClass:**

**public** **class** DifferentData

{

@DataProvider(name = "testdata")

**public** Object[][] data()

{

Object[][] obj = **new** Object[2][3];

obj[0][0] = "Text1";

obj[0][1] = "Text2";

obj[0][2] = "Text3";

obj[1][0] = "Text4";

obj[1][1] = "Text5";

obj[1][2] = "Text6";

**return** obj;

}

}

**Point to Remember:**

* Can pass the complex data to test methods as parameters.
* Data Reading Class and Data Provider Class can be in the same class.
* Data Reading Class and Data Provider Class can be in different classes.

**Grouping Tests in TestNG:**

In TestNG Grouping Tests, we can group multiple test methods into a named group. You can execute a particular set of test methods belonging to a group or multiple groups. This feature allows the test methods to be segregated into different sections or modules. You can have a set of test that belong to sanity test whereas others may belong to regression tests. Apart from this you can also segregate the tests based on the functionalities or features that the test method verifies. This helps in executing only a particular set of tests as and when required.

**Different Ways:**

1. Test belongs to Single group.
2. Test belongs to Multiple groups.
3. Including and Excluding groups.

Example:

**public** **class** ClassTwo

{

@Test(groups = "smoke")

**public** **void** MethodOne()

{

System.***out***.println("Class Two - Method One");

}

@Test(groups = "regression")

**public** **void** MethodTwo()

{

System.***out***.println("Class Two - Method Two");

}

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

**public** **void** MethodThree()

{

System.***out***.println("Class Two - Method Three");

}

}

Xml-file:

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

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

<suite name=*"Suite"*>

<test thread-count=*"5"* name=*"Test"*>

<groups>

<run>

<include name=*"smoke"*></include>

<exclude name=*"regression"*></exclude>

</run>

</groups>

<classes>

<class name=*"TestNGPractice.ClassOne"*></class>

<class name=*"TestNGPractice.ClassTwo"*></class>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

**Dependency Tests in TestNG:**

Dependency test in TestNG will discuss about how a test can dependent on other tests or group of tests. This feature help in executing a set of tests to be executed before a test method. Method dependency only works if the depend-on-method is part of the same class or any of the inherited base class.

**Different Ways:**

* Test depends on Single Method.
* Test depends on Multiple Methods.
* Test that depends on its Parent Class Methods.

Example:

Child class:

**public** **class** DependentClass **extends** DepParentClass

{

@Test(dependsOnMethods = "methodThree")

**public** **void** methodOne()

{

System.***out***.println("Class Two - Method One");

}

@Test(dependsOnMethods = {"methodFour","methodFive"})

**public** **void** methodTwo()

{

System.***out***.println("Class Two - Method Two");

}

@Test(dependsOnMethods = "methodsix")

**public** **void** methodThree()

{

System.***out***.println("Class Two - Method Three");

}

**public** **void** methodFour()

{

System.***out***.println("Class Two - Method Four");

}

**public** **void** methodFive()

{

System.***out***.println("Class Two - Method Five");

}

}

Parent class:

**public** **class** DepParentClass

{

@Test()

**public** **void** methodsix()

{

System.***out***.println("Class DepParentClass - Method six");

}

}

**Tests Depends on in TestNG GROUPS**

Tests Depends on Groups in TestNG will discuss about how a test can dependent on groups or multiple groups. This feature help in executing a set of groups to be executed before a test method. Group dependency only works if the depend-on-group is part of the same class or any other classes which are in the same package.

**Different Ways:**

1. Test depends on Single Group.

2. Test depends on Multiple Groups.

3. Test that depends on other Classes from the same package.

Example:

**public** **class** DependentClass

{

@Test(dependsOnGroups = "smoke")

**public** **void** methodOne()

{

System.***out***.println("Class Two - Method One");

}

@Test(dependsOnGroups = {"smoke",”regression”})

**public** **void** methodTwo()

{

System.***out***.println("Class Two - Method Two");

}

@Test(dependsOnGroups = {"otherclassgroup"})

**public** **void** methodOther()

{

System.***out***.println("Class Two -Other Class Method");

}

@Test(groups = {"smoke"})

**public** **void** methodThree()

{

System.***out***.println("Class Two - Method Three");

}

@Test(groups = {"regression"})

**public** **void** methodFour()

{

System.***out***.println("Class Two - Method Four");

}

}

Class Two:

**public** **class** ClassTwo

{

@Test(groups={"otherclassgroup"})

**public** **void** methodsix()

{

System.***out***.println("Class Two – Other Method");

}

}

**Parallel Execution of Methods in TestNG:**

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

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

<suite name=*"Suite"* parallel=*"methods"* thread-count=*"5"*>

<test name=*"Test"*>

<classes>

<class name=*"TestNGPractice.ClassOne"*></class>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

**Parallel Execution of classes in TestNG:**

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

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

<suite name=*"Suite"* parallel=*"classes"* thread-count=*"2"*>

<test name=*"Test"*>

<classes>

<class name=*"TestNGPractice.ClassOne"*></class>

<class name=*"TestNGPractice.ClassTwo"*></class>

<class name=*"TestNGPractice.ClassThree"*></class>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

**Assertion in TestNG:**

**Difference between Soft Assert/Verify and hard Assert/Assert in selenium**?

A soft assert will run the test and not throw an exception if the assert failed, while a hard assert will throw the exception immediately, and then continue with the testing process.

**@Test**

public void softAssertion()

{

SoftAssert softAssertion= new SoftAssert();

System.out.println("softAssert");

softAssertion.assertTrue(false);

softAssertion.assertAll();

}

**@Test**

public void hardAssertion()

{

System.out.println("hardAssert");

Assert.assertTrue(false);

System.out.println("hardAssert");

}

Soft Assertions are the type of assertions that do not throw an exception when an assertion fails and continue with the next step after the assert statement. This is used when the test requires multiple assertions to be executed and we want all of the assertions/codes to be executed before failing/skipping the tests.

Hard Assert is a type of TestNG assertion which throws Exception Immediately if the assert statement fails and move to the next testcase and if there is any code in the current test case after assert statement it will not execute that statement.

**TestNG ASSERTIONS:**

TestNG Assertions will discuss about the Assertions available in the testNG in detail. When you write a test script there should be a verification point in order to mark that test script is passed or failed or any other state. To achieve this in testNG we will use Assertions. Assertion helps you to check or verify the success of conditions in your test. If the conditions don't satisfy, it will stop the test execution of the said test and mark it as failing.

TestNG supports assertion of a test using the Assert class which is part of the testNG library.

**Assertions:**

1. Assert.assert True(condition, message):

It will check the condition is true or not. If the condition is false, then it fails and prints the message. Here the message is optional.

1. Assert.assertFalse (condition, message)

It will check the condition is false or not. If the condition is true, then it fails and prints the message. Here the message is optional.

1. Assert.assertNotNull(object, message)

It will check the object is NOT Null or not. If the object is null, then it fails and prints the message. Here the message is optional.

1. Assert.assertNull(object, message)

It will check the object is Null or not. If the object is NOT Null, then it fails and prints the message. Here the message is optional.

1. Assert.assertEquals(actual, expected, message)

It will check the actual and expected or Equal or not. If both are NOT Equal, then it fails and prints the message. Here the message is optional.

1. Assert.assertNotEquals(actual, expected, message)

It will check the actual and expected or NOT Equal or not. If both are Equal, then it fails and prints the message. Here the message is optional.

**Custom Logger in TestNG:**

Custom logger in the TestNG will discuss about writing our own logger in TestNG. Logging is one of the most important things to remember in automation testing. Logger will keep on eye of the execution flow that where went wrong in the script or it very useful to debug the script.

To generate custom loggers in TestNG we need to use Listeners.

**What are Listeners in TestNG?**

Listener is defined as interface that modifies the default TestNG’s behaviour. As the name suggests Listeners “listen” to the event defined in the selenium script and behave accordingly. It is used in selenium by implementing Listeners Interface. It allows customizing TestNG reports or logs. There are many types of TestNG listeners available.

**Types of Listeners in TestNG**

There are many types of listeners which allows you to change the TestNG’s behaviour.

Below are the few TestNG listeners:

1. IAnnotationTransformer ,
2. IExecutionListener,
3. IHookable ,
4. IInvokedMethodListener ,
5. IMethodInterceptor ,
6. IReporter,
7. ISuiteListener,
8. ITestListener

Above Interface are called TestNG Listeners. These interfaces are used in selenium to generate logs or customize the TestNG reports.

**IAnnotationTransformer**:

The best part about TestNG is the naming convention it uses for its keywords, like the ‘listener’ which listens to the code. Similarly, IAnnotationTransformer transforms the TestNG annotations at run time. A scenario may appear in which the user seeks to override the content of the annotation based on a condition. In such a case it is not necessary to make changes in the source code. Simply use IAnnotationTransformer to override the content of the annotations. IAnnotationTransformer has only one method named transform() that accepts four parameters:

1. ITestAnnotation annotation
2. Class testClass
3. Constructor testConstructor
4. Method testMethod

Let’s implement this listener in code, to understand its usage. This scenario will change invocation count at run time for the required method of TestNG class.

import org.testng.annotations.Test;

public class IAnnotationTransformerWithExample

{

MyListener obj=new MyListener();

@Test(invocationCount=5)

public void changeInvocationCountOfMethod()

{

System.out.println("This method have invocation count set to 5 but at run time it shall become "+ obj.counter);

}

}

The class implementing the interface IAnnotationTransformer that shall change this invocation count:

import java.lang.reflect.Constructor;

import java.lang.reflect.Method;

import org.testng.IAnnotationTransformer;

import org.testng.annotations.ITestAnnotation;

public class MyListener implements IAnnotationTransformer

{

int counter=3;

@Override

public void transform(ITestAnnotation testAnnotation, Class testClass, Constructor testConstrutor, Method testMethod)

{

if(testMethod.getName().equals("ChangeInvocationCountOfMethod"))

{

System.out.println("Changing invocation for the following method: " + testMethod.getName());

testAnnotation.setInvocationCount(counter);

}

}

}

**TestNG.xml file:**

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

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

<suite name="Parent\_Suite">

<listeners>

<listener class-name="MyListener"/>

</listeners>

<test name="ItestReporter">

<classes>

<class name="IAnnotationTransformerWithExample" />

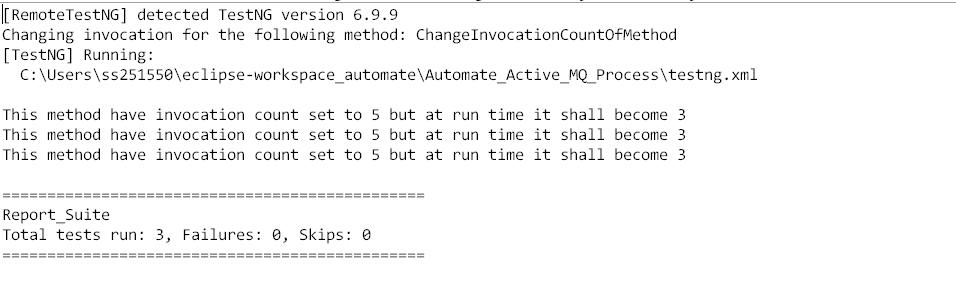
</classes>

</test>

</suite>

<!-- Suite -->

**Console Output**:



**IExecutionListener**:

As the name suggests, it monitors the beginning and end of TestNG execution. This listener is mostly used to start/stop the server while starting or ending code execution. It may also be used to inform respective stakeholders via email that execution shall start or when it ends.

It has two methods:

1. onExecutionStart() – invoked before TestNG starts executing the suites
2. onExecutionFinish() – invoked after all TestNG suites have finished execution

Let’s look at an example. This example has a class with 5 methods which shall be executed after the onExecutionStart method of the IExecutionListener interface. After these methods are completed the onExecutionFinish method shall be executed. The two methods in this example highlight the start and end time of the test.

import org.testng.annotations.Test;

public class IExecutionListenerWithExample

{

@Test

public void method1()

{

System.out.println("this method is method 1");

}

@Test

public void method2()

{

System.out.println("this method is method 2");

}

@Test

public void method3()

{

System.out.println("this method is method 3");

}

@Test

public void method4()

{

System.out.println("this method is method 4");

}

@Test

public void method5()

{

System.out.println("this method is method 5");

}

}

**Class implementing IExecutionListener interface:**

import java.sql.Time;

import org.testng.IExecutionListener;

public class MyListener implements IExecutionListener

{

@Override

public void onExecutionFinish()

{

long endTime= System.currentTimeMillis();

System.out.println("Inform all the suite have finished execution at"+ endTime);

}

@Override

public void onExecutionStart()

{

long startTime= System.currentTimeMillis();

System.out.println("Inform all the suite have started execution at"+ startTime);

}

}

**TestNG.xml file:**

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

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

<suite name="Parent\_Suite">

<listeners>

<listener class-name="MyListener"/>

</listeners>

<test name="ItestReporter">

<classes>

<class name="IExecutionListenerWithExample" />

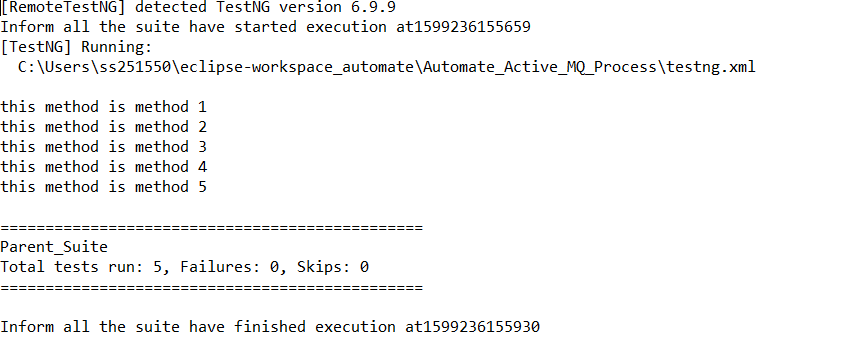
</classes>

</test>

</suite>

<!-- Suite -->

**Console Output:**



**IHookable**:

If a class implements this interface, its run method will be invoked instead of each test method. Using the callback method of the IHookCallBack parameter, the invocation of the test method can be performed.

It has a single method name **run**, which accepts two parameters. run (IHookCallBack callBack, ITestResult testResult) Now let’s look into its real-time example. In this example, based on a certain parameter value, the test shall be skipped using the IHookable listener interface. These values will be provided by a data provider in a separate TestNG class.

import org.testng.annotations.DataProvider;

import org.testng.annotations.Test;

public class IHookableListenerWithExample

{

@Test(dataProvider="parametersToBeSent")

public void t(String parameter)

{

System.out.println("test method to be called with the following parameter is " + parameter);

}

@DataProvider

public Object[][] parametersToBeSent()

{

return new Object[][]{{"parameter 1"}, {"parameter 2"}, {"parameter 3"}};

}

}

**The class implementing the IHookable listener interface:**

import org.testng.IHookCallBack;

import org.testng.IHookable;

import org.testng.ITestResult;

public class MyListener implements IHookable

{

@Override

public void run(IHookCallBack callBack, ITestResult testResult)

{

Object[] parameterValues = callBack.getParameters();

if (parameterValues[0].equals("parameter 3"))

{

System.out.println("Skip the required parameter");

}

else

{

callBack.runTestMethod(testResult);

}

}

}

**The testNG.xml file:**

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

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

<suite name="Parent\_Suite">

<listeners>

<listener class-name="MyListener"/>

</listeners>

<test name="ItestReporter">

<classes>

<class name="IHookableListenerWithExample" />

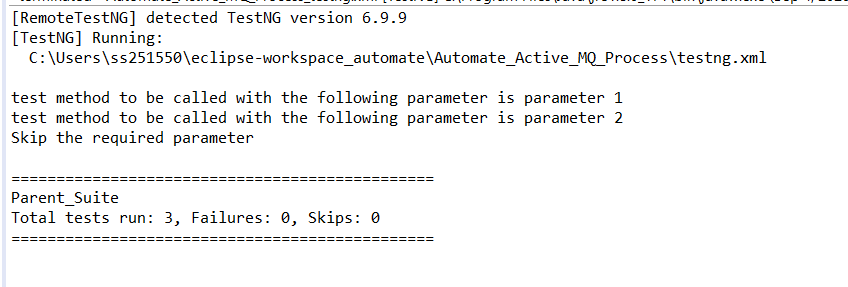
</classes>

</test>

</suite>

<!-- Suite -->

**Console output:**



**IInvokedMethodListener**:

This listener gets invoked before and after a method in TestNG. These methods constitute both test and other configuration methods.

These listeners are useful for setting up configuration or other cleanup activities.

It contains two methods:

1. beforeInvocation(): this method gets invoked before every method
2. afterInvocation(): this method gets invoked after every method

Let’s look at an example. The TestNG class contains different configuration methods. The other class implements the InvokedMethodInterceptor which implements the beforeInvocation and afterInvocation methods. These defined methods execute before and after every config method of the TestNG class.

import org.testng.annotations.AfterMethod;

import org.testng.annotations.AfterSuite;

import org.testng.annotations.BeforeMethod;

import org.testng.annotations.BeforeSuite;

import org.testng.annotations.Test;

public class IInvokedMethodListenerWithExample

{

@BeforeSuite

public void method1()

{

System.out.println("before suite");

}

@BeforeMethod

public void method2()

{

System.out.println("before method");

}

@Test

public void method3()

{

System.out.println("test method 1 ");

}

@Test

public void method4()

{

System.out.println("test method 2 ");

}

@AfterMethod

public void method5()

{

System.out.println("after method");

}

@AfterSuite

public void afterSuite()

{

System.out.println("after suite");

}

}

**The class implementing the InvokedMethodListener interface:**

import org.testng.IInvokedMethod;

import org.testng.IInvokedMethodListener;

import org.testng.ITestResult;

public class MyListener implements IInvokedMethodListener

{

@Override

public void afterInvocation(IInvokedMethod method, ITestResult result)

{

System.out.println("This method is invoked after every config method - " + method.getTestMethod().getMethodName());

}

@Override

public void beforeInvocation(IInvokedMethod method, ITestResult result)

{

System.out.println("This method is invoked before every config method - " + method.getTestMethod().getMethodName());

}

}

**TestNG.xml file:**

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

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

<suite name="Parent\_Suite">

<listeners>

<listener class-name="MyListener"/>

</listeners>

<test name="ItestReporter">

<classes>

<class name="IInvokedMethodListenerWithExample" />

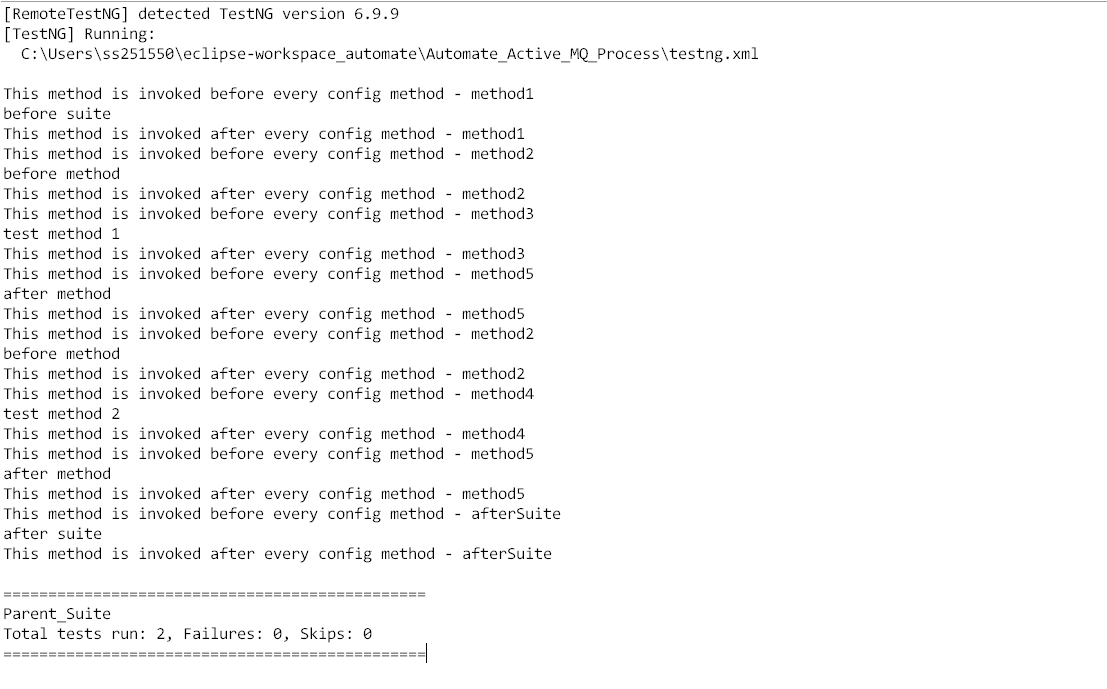
</classes>

</test>

</suite>

<!-- Suite -->

**Console Output:**



**IMethodInterceptor**: This listener helps to alter the methods that TestNG is supposed to run. It gets invoked just before TestNG invokes the methods. It just has one method name **intercept** that returns an altered list of methods.

Let’s look at an example.

The code here will run only methods with priority 1 in the test class. Other methods with different priority shall not be executed. This will be done after implementing the IMethodInterceptor listener.

import org.testng.annotations.Test;

public class IMethodInterceptorWithExample

{

@Test(priority=2)

public void method1()

{

System.out.println("Method 1 will not be executed");

}

@Test(priority=2)

public void method2()

{

System.out.println("Method 2 will not be executed");

}

@Test(priority=1)

public void method3()

{

System.out.println("Method 3 will be executed");

}

@Test(priority=1)

public void method4()

{

System.out.println("Method 4 will be executed");

}

}

**The IMethodInterceptor class implementing the interface:**

import java.util.ArrayList;

import java.util.List;

import org.testng.IMethodInstance;

import org.testng.IMethodInterceptor;

import org.testng.ITestContext;

import org.testng.annotations.Test;

public class MyListener implements IMethodInterceptor

{

@Override

public List<IMethodInstance> intercept(List<IMethodInstance> methodsInstance, ITestContext testContext)

{

List<IMethodInstance> result = new ArrayList<IMethodInstance>();

for (IMethodInstance method : methodsInstance)

{

Test testMethod = method.getMethod().getConstructorOrMethod().getMethod().getAnnotation(Test.class);

if (testMethod.priority() == 1)

{

result.add(method);

}

}

return result;

}

}

**TestNG.xml file:**

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

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

<suite name="Parent\_Suite">

<listeners>

<listener class-name="MyListener"/>

</listeners>

<test name="ItestReporter">

<classes>

<class name="IMethodInterceptorWithExample" />

</classes>

</test>

</suite>

<!-- Suite -->

**Console Output:**

Text

Description automatically generated

**IReporter**:

This listener helps to generate custom reports in TestNG, based on desired conditions. It contains a method called **generateReport**() which is invoked when all suites of TestNG are executed.

The method uses three arguments:

1. xmlSuite: It contains a list of suites for execution in the xml file
2. suites: It contains all information about test execution and suites like class name, package name, method name and test execution results
3. outputDirectory: It contains the path where the report shall be saved.

Let’s look at an example of how to customize reports through IReporter listeners. In this example, through the IReporter listener, the code shall run only methods belonging to a particular group.

In this class, the group has been defined as ‘Sanity’, which shall be executed. The methods which are not part of the group shall not be executed.

In the other class that implements IReporter, the generateReport() method has been used to customize the results accordingly. The customized results shall be visible in console with a corresponding report generated under the suite folder name specified in the testNG.xml file

import org.testng.Assert;

import org.testng.annotations.Test;

public class IReporterWithExample

{

@Test(groups="smoke")

public void testcase1()

{

System.out.println("This test case will pass");

}

@Test(groups="smoke")

public void testcase2()

{

System.out.println("This test case will fail");

Assert.assertTrue(false);

}

@Test

public void testcase3()

{

System.out.println("this tet case does not belong to the group smoke");

}

}

**The class implementing the IReporter listener interface:**

import java.util.Collection;

import java.util.List;

import java.util.Map;

import java.util.Set;

import org.testng.IReporter;

import org.testng.IResultMap;

import org.testng.ISuite;

import org.testng.ISuiteListener;

import org.testng.ISuiteResult;

import org.testng.ITestContext;

import org.testng.ITestListener;

import org.testng.ITestNGMethod;

import org.testng.ITestResult;

import org.testng.xml.XmlSuite;

public class MyListener implements IReporter

{

@Override

public void generateReport(List<XmlSuite> xmlSuites, List<ISuite> suites, String outputDirectory)

{

// TODO Auto-generated method stub

ISuite suite = suites.get(0);

Map<String, Collection<ITestNGMethod>> methodsByGroup = suite.getMethodsByGroups();

Map<String, ISuiteResult> tests = suite.getResults();

for (String key : tests.keySet())

{

System.out.println("Key: " + key + ", Value: " + tests.get(key));

}

Collection<ISuiteResult> suiteResults = tests.values();

ISuiteResult suiteResult = suiteResults.iterator().next();

ITestContext testContext = suiteResult.getTestContext();

Collection<ITestNGMethod> perfMethods = methodsByGroup.get("smoke");

IResultMap failedTests = testContext.getFailedTests();

for (ITestNGMethod perfMethod : perfMethods)

{

Set<ITestResult> testResultSet = failedTests.getResults(perfMethod);

for (ITestResult testResult : testResultSet)

{

System.out.println("Test " + testResult.getName() + " failed, error " + testResult.getThrowable());

}

}

IResultMap passedTests = testContext.getPassedTests();

for (ITestNGMethod perfMethod : perfMethods)

{

Set<ITestResult> testResultSet = passedTests.getResults(perfMethod);

for (ITestResult testResult : testResultSet)

{

System.out.println("Test " + testResult.getName() + " passed, time took " +

(testResult.getEndMillis() - testResult.getStartMillis()));

}

}

}

}

**Below is the tesNG.xml for the classes to be run:**

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

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

<suite name="Report\_Suite">

<listeners>

<listener class-name="MyListener"/>

</listeners>

<test name="ItestReporter">

<classes>

<class name="IReporterWithExample" />

</classes>

</test>

</suite>

<!-- Suite -->

**Console Output:**

Graphical user interface, text, application, email

Description automatically generated

A corresponding customized report shall also be created under the folder name which is the suite name defined in the xml file. In this example, it is **Report\_Suite**.

Text

Description automatically generated

**Sample Report:**

Table

Description automatically generated

**ISuiteListener**: As the name suggests, this listener works at the suite level. It listens and runs before the start and end of suite execution. It contains two methods:

1. onStart: invoked before test suite execution starts
2. onFinish: invoked after test suite execution finishes.

**Note**: In case of a child suite to a parent suite, the child suite shall run before the parent suite. This is done to ensure the results reflect the parent suite which automatically contains the results of the child suite.

The following example incorporates the ISuiteListener. The code will use two child classes containing before and after suites. These shall be run through a TestNG xml file containing a reference to the ISuiteListener class which shall run its onStart and onFinish methods first and last respectively.

import org.testng.ISuite;

import org.testng.ISuiteListener;

import org.testng.ITestContext;

import org.testng.ITestListener;

import org.testng.ITestResult;

public class MyListener implements ISuiteListener

{

@Override

public void onFinish(ISuite suite1)

{

System.out.println("onFinish function started of ISuiteListener " );

}

@Override

public void onStart(ISuite suite2)

{

System.out.println("onStart function started of ISuiteListener " );

}

}

**Below are the two respective child classes:**

import org.testng.annotations.AfterSuite;

import org.testng.annotations.BeforeSuite;

import org.testng.annotations.Test;

public class ISuiteListenerWithExample

{

@BeforeSuite

public void bsuite()

{

System.out.println("BeforeSuite method started for the first IsuiteListener example class");

}

@Test

public void test()

{

System.out.println("Test method started for the first IsuiteListener example class");

}

@AfterSuite

public void asuite()

{

System.out.println("AfterSuite method started for the first IsuiteListener example class");

}

}

**And the second class:**

import org.testng.annotations.AfterSuite;

import org.testng.annotations.BeforeSuite;

import org.testng.annotations.Test;

public class ISuiteListenerExample2

{

@BeforeSuite

public void bsuite()

{

System.out.println("BeforeSuite method started for the first IsuiteListener example 2 class");

}

@Test

public void test()

{

System.out.println("Test method started for the first IsuiteListener example 2 class");

}

@AfterSuite

public void asuite()

{

System.out.println("AfterSuite method started for the first IsuiteListener example 2 class");

}

}

**Their respective xml files:**

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

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

<suite name="SuiteOne">

<test thread-count="5" name="Test">

<classes>

<class name="ISuiteListenerExample2"/>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

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

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

<suite name="Suite2">

<test thread-count="5" name="Test">

<classes>

<class name="ISuiteListenerWithExample"/>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

**The final testNG xml file:**

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

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

<suite name="Parent\_Suite">

<listeners>

<listener class-name="MyListener"/>

</listeners>

<suite-files>

<suite-file path="ISuiteListenerWithExample.xml"> </suite-file>

<suite-file path="ISuiteListenerExample2.xml"> </suite-file>

</suite-files>

</suite>

<!-- Suite -->

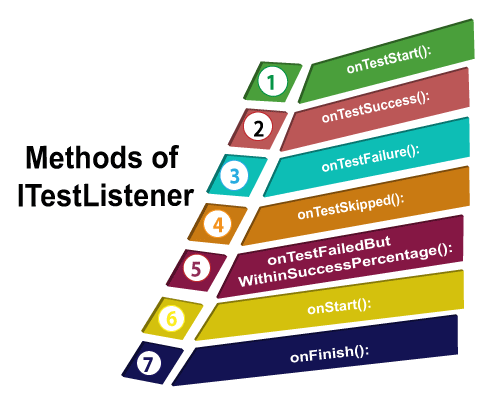
**Console Output:**

Graphical user interface, text, application, email

Description automatically generated

**ITestListener:**

To implement listeners, need to implement the ITestListener interface in your class. These classes are notified at runtime by TestNG when the test case executes.



For this we need to have at least 2 classes. One is for actual test cases and other is for to implement the ITestListener.

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

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

<suite name=*"Suite"* parallel=*"classes"* thread-count=*"3"*>

<listeners>

<listener class-name=*"TestNGPrac.ListnerTest"*></listener>

</listeners>

<test thread-count=*"5"* name=*"Test"* >

<classes>

<class name=*"TestNGPrac.ClassOne3"*/>

<!-- <class name="TestNGPrac.ClassTwo"/>

<class name="TestNGPrac.ClassThree"/>

<class name="TestNGPrac.ClassFour"/>

<class name="TestNGPrac.ClassFive"> -->

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

**Execute Multiple XML files:**

Execute Multiple Xml files in TestNG will discuss about how to execute multiple xml files in TestNG at a time. For ex. You have more than one xml suite files to execute different types of test scripts in your project. Now you can execute the individual xml files simply that right click on xml file and run as TestNG suite. But if you have multiple files then you need to put those xml files in a separate xml file and need to execute the same way as right click on xml and run as TestNG suite. This way you can manage the test cases execution how we want.

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

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

<suite name=*"Suite"* parallel=*"classes"* thread-count=*"3"*>

<suite-files>

<suite-file path=*".\testng3.xml"*></suite-file>

<suite-file path=*".\testng.xml"*></suite-file>

</suite-files>

</suite> <!-- Suite -->

**Prioritizing Tests:**

Prioritizing tests in TestNG will discuss about the order of execution tests in test suite. We will write test methods our own way using @Test annotation. After writing the test cases we will execute the test cases either normally or from the testng.xml file. After execution of the tests if we observe the order of execution then we can find that tests executed by taking the alphabetical order. And all the tests will have the equal priority as we did not set any priority to the tests.

In order to set the priority to the tests then we can use one of the @Test attributes called "Priority". By default, all the tests will have the same priority called Zero(i.e. If you not set any priority then it will take the priority as Zero).

If you give the priority equal to all the test cases, then all the tests will execute in alphabetical order. Here we will see how the priority will work while executing the tests by taking few examples:

Examples will discuss in this:

* Without Priority
* Partially Use Priority
* With Priority
* With negative (-) priority

**TestNG topics :**

1. Downloading TestNG
2. Understanding testng.xml file
3. Creating testng.xml With Multiple Tests
4. Adding Packages to testng.xml file
5. Adding Classes to testng.xml file
6. Adding Methods to testng.xml file
7. TestNG Annotations:
8. Before and After Annotations:
9. TestNG Test Annotation:
10. TestNG Test Attribute:
11. Parallel Execution in TestNG:
    1. Parallel Execution of Methods in TestNG:
    2. Parallel Execution of classes in TestNG:
    3. Parallel Execution of tests in TestNG:
12. TestNG ASSERTIONS:
13. Custom Logger in TestNG:
14. Custom Reporter in TestNG:
15. Execute Multiple XML files:
16. Prioritizing Tests:
17. Capture Screenshot for Failed Tests:
18. Executing only Failed Tests in TestNG:
19. Running TestNG Tests Programmatically:
20. Running testng.xml Using Command Prompt:
21. Advantages and features of TestNG
22. TestNG Reports