# TestNG @Test Annotation, Supported Attributes

**@Test annotation** is one of the basic annotations in [TestNG](https://www.scientecheasy.com/2019/04/testng-tutorial.html/). This annotation marks a method as a test method.

The test method annotated with @Test annotation tells TestNG that this method is a “test” and it should be executed when the user runs the class.

If we apply this annotation at a class level, it will mark all public methods declared inside the class as test methods for TestNG test.

Let’s create a simple example program where we will declare [test annotation](https://www.scientecheasy.com/2020/07/testng-annotations.html/) on class.

**Program source code 1:**

package testAnnotation;

import org.testng.annotations.Test;

// Declaration of @Test annotation on class.

@Test

public class TestClass

{

public void m1()

{

System.out.println("Test method one");

}

public void m2()

{

System.out.println("Test method two");

}

private void m3()

{

System.out.println("Test method three");

}

protected void m4()

{

System.out.println("Test method four");

}

}

Output:

Test method one

Test method two

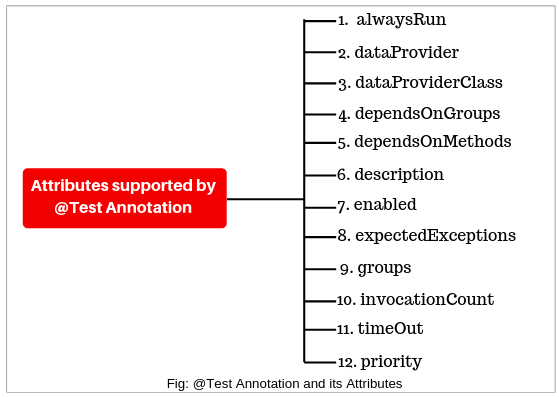
In the preceding example program, a class has been annotated by Test annotation of TestNG which contains four methods two public, one private, and protected.

As you can see above output, only two methods out of four methods of class have been executed by TestNG.

This is because TestNG considers only methods with public access modifiers as test methods and executes them. All the methods with other access modifiers will be neglected by TestNG.

## Supported Attributes by @Test Annotation In TestNG

@Test annotation in TestNG supports lots of attributes that we can use with this annotation. Some of the important attributes supported by Test annotation is as follows:

[](https://www.scientecheasy.com/2020/07/testng-test-annotation.html/)

**1. alwaysRun:** This attribute takes a value as true or false. If we set true, this method will always execute even its depending method fails. It has the following general form:

Syntax:

    @Test(alwaysRun = true)

Let’s take an example program based on the alwaysRun attribute.

**Program source code 2:**

package testAnnotation;

import org.testng.annotations.Test;

public class AlwaysRunEx

{

@Test

public void m1()

{

System.out.println("m1 method");

}

@Test(alwaysRun = true)

public void m2()

{

System.out.println("m2 method");

}

@Test(alwaysRun = false)

public void m3()

{

System.out.println("m3 method");

}

}

Output:

m1 method

m2 method

m3 method

**2. dataProvider:** This attribute is used to provide data to the test method directly in data-driven testing.

Syntax:

    @Test(dataProvider = "getData")

**3. dataProviderClass:** This attribute is used to call the DataProvider method from another class.

Syntax:

    @Test(dataProvider = "getData", dataProviderClass = Hello.class)

where, Hello is the name of class where DataProvider method has been declared.

**4. dependsOnGroups:** This attribute is used to make test methods depend on a particular group. We can also specify a list of groups this method depends on.

All of the methods of these groups are executed first before this method. If any test method belonging to a particular group is failed, the dependent test method will not be run and will be flagged as a SKIP. It has the following general form.

Syntax:

    @Test(dependsOnGroups = {"GroupA", "GroupB", . . .})

Let’s understand dependsOnGroups attribute with the help of an example program. In this example program, a test method depends on a particular group. Look at the following source code.  
  
**Program source code 3:**

package testAnnotation;

import org.testng.annotations.Test;

public class dependsOnGroupsEx

{

@Test(groups = "GroupA")

public void m1()

{

System.out.println("m1-GroupA");

}

@Test(groups = "GroupA")

public void m2()

{

System.out.println("m2-GroupA");

}

@Test(groups = "GroupB")

public void m3()

{

System.out.println("m3-GroupB");

}

@Test(dependsOnGroups = "GroupA") // This test method depends on GroupA.

public void m4()

{

System.out.println("m4 is dependent on GroupA");

}

}

Output:

m1-GroupA

m2-GroupA

m3-GroupB

m4 is dependent on GroupA

As you can observe output that after the execution of all methods of particular groups, m4 has been executed.

Let’s take one more example program based on this attribute.

**Program source code 4:**

package testAnnotation;

import org.junit.Assert;

import org.testng.annotations.Test;

public class dependsOnGroupsEx

{

@Test(groups = "GroupA")

public void m1()

{

Assert.fail();

System.out.println("m1-GroupA");

}

@Test(groups = "GroupA")

public void m2()

{

System.out.println("m2-GroupA");

}

@Test(groups = "GroupB")

public void m3()

{

System.out.println("m3-GroupB");

}

@Test(dependsOnGroups = "GroupA") // This test method depends on GroupA.

public void m4()

{

System.out.println("m4 is dependent on GroupA");

}

}

Output:

m2-GroupA

m3-GroupB

FAILED: m1

SKIPPED: m4

As you can observe in the above output that only two methods are successfully executed. One method m1() of GroupA is failed, the dependent method m4() on GroupA is also skipped.

**5. dependsOnMethods:** The attribute dependsOnMethods is used to make the test method depend on a particular method. We can also specify a list of methods this method depends on.

The test method annotated with @Test and attribute dependsOnMethods will run after executing all those methods on which this test method is dependent.

If any of these methods are not executed successfully, this test method will not be run and will be flagged as a SKIP.

Syntax:

    @Test(dependsOnMethods = {"Method1", "Method2". . .})

Let’s take an example program where we will use dependsOnMethods attribute with test method. Let’s see the following source code.

**Program source code 5:**

package testAnnotation;

import org.junit.Assert;

import org.testng.annotations.Test;

public class dependsOnMethodsEx

{

@Test

public void m1()

{

System.out.println("Test method one");

}

@Test(dependsOnMethods = "m1")

public void m2()

{

System.out.println("Test method two");

}

@Test

public void m3()

{

Assert.fail();

System.out.println("Test method three");

}

@Test(dependsOnMethods = {"m3", "m1"})

public void m4()

{

System.out.println("Test method four");

}

}

Output:

Test method one

Test method two

FAILED: m3

SKIPPED: m4

**6. description:** It is the description of this method. The syntax of description attribute is given below.

Syntax:

     @Test(description = "test method")

**7. enabled:** This enabled attribute is used to disable the execution of a particular test or set of tests in TestNG. It can be achieved by setting enabled attribute of the Test annotation to false.

If you set this attribute for the Test annotation at class level, all the public methods declared inside class will be disabled. The general syntax is as follows:

Syntax:

      @Test(enabled = ture)

      @Test(enabled = false)

Let’s understand it with the help of a scenario.

Suppose a scenario where a serious bug exists in a certain test belonging to a certain scenario that cannot be executed. As the issue has already been identified. Therefore, we may need to disable that certain test from being executed.

Let’s understand with the help of a simple program to see how this feature works. In this example program, we will disable the test method from getting executed.

**Program source code 6:**

package testAnnotation;

import org.testng.annotations.Test;

public class DisableTestClass

{

@Test(enabled = true)

public void m1()

{

System.out.println("Hello");

}

@Test(enabled = false)

public void m2()

{

System.out.println("Hi");

}

@Test

public void m3()

{

System.out.println("Hello-Hi");

}

}

Output:

Hello

Hello-Hi

Default test

Tests run: 2, Failures: 0, Skips: 0

In the preceding program, a class DisableTestClass contains three methods m1, m2, and m3. Out of three, two contain attributes enabled with values true and false respectively.

As you can see in the output, only two methods have been executed by TestNG. The method m2() with attribute enabled value as false has been ignored by test execution.

The test method m3() has been also executed by TestNG even when the attribute value is not specified. This is because by default the attribute value of enabled is true.

**8. expectedExceptions:** The attribute expectedExceptions is used for exception testing. It specifies the type of exceptions that are expected to be thrown by a test method during execution.

If the exception thrown by a test method does not match with the exception list entered by user, the test method will be marked as failed.

TestNG also supports multiple expected exceptions for verification while executing a particular test. The general syntax is as follows:

Syntax:

    @Test(expectedExceptions = IOException.class)

Let’s understand it by making a simple test and learn how the exception test works in TestNG.

**Program source code 7:**

package testAnnotation;

import java.io.IOException;

import org.testng.annotations.Test;

public class ExceptionTestEx

{

@Test(expectedExceptions = IOException.class)

public void exceptionTestOne() throws Exception

{

throw new IOException();

}

@Test(expectedExceptions = {IOException.class, ArithmeticException.class})

public void exceptionTestTwo() throws Exception

{

throw new Exception();

}

}

Output:

PASSED: exceptionTestOne

FAILED: exceptionTestTwo

In the above program, a class contains two test methods exceptionTestOne and exceptionTestTwo which throws IOException and Exception respectively.

As you can see from the test output, the test method exceptionTestTwo() has been marked as failed by TestNG during execution. This is because exception thrown by test method does not match with exception provided in the expectedExceptions list.

**9. groups:** It specifies the list of groups this method or class belongs to. For more detail, go to this tutorial: [TestNG Groups](https://www.scientecheasy.com/2020/07/testng-groups.html/)

**10. invocationCount:** This attribute is used to execute a method the number of times. It acts as a loop.

For example:

@Test(invocationCount = 5) // This method will execute 5 times.

**11. timeOut:** The attribute timeOut is used for a time out test. It specifies a time period (in milliseconds) to wait for a test for complete execution.

For example, during running tests, there can be cases where certain tests may take much more time than expected. In such a case we need to mark the test case as fail and then continue. It can be achieved in two ways:

**a) At suit level:** It will be applicable for all tests in the TestNG suite.

**b) At each test method level:** It will be applicable for each test method. It will override the time period if you have configured it at the suite level.

Let’s create a test program and understand how this feature works.

**Program source code 8:** Time test at the suite level.

package testAnnotation;

import org.testng.annotations.Test;

public class TimeOutAtSuiteLevel

{

@Test

public void m1() throws InterruptedException

{

Thread.sleep(1000);

System.out.println("Hello");

}

@Test

public void m2() throws InterruptedException

{

Thread.sleep(400);

System.out.println("Java");

}

}

In the preceding program, a test class contains two test methods m1() and m2() which print messages on the console after successful execution.

Both test methods also contain Thread.sleep() method which pauses test execution for a specified amount of time in milliseconds.

Now create a testng.xml file and put the following code to it.

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

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

<suite name="TimeOut at Suite level" time-out='500' verbose='1'>

<test name="Test">

<classes>

<class name="testAnnotation.TimeOutAtSuiteLevel"/>

</classes>

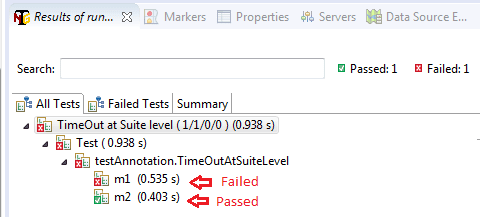
</test> <!-- Test -->

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

In the preceding testng.xml file, a suite tag contains an attribute named time-out which has been set with a value 500. This attribute will set a time-out period for test methods in the whole suite.

That means if any test method in this suite takes more time than the specified time period (in this case 500 milliseconds) for complete execution, it will be marked as failed.

Now run testng.xml file in Eclipse and see the following test results as shown in the screenshot.

[](https://www.scientecheasy.com/2020/07/testng-test-annotation.html/)

As you can see in the test result that TestNG executed tests and failed the first test because the first test took more time than time mentioned in the time-out section to complete execution. This kind of feature is generally useful for doing time testing.

**Program source code 9:** Example program for setting time-out at test method level.

package testAnnotation;

import org.testng.annotations.Test;

public class TimeOutAtTestMethodLevel

{

@Test(timeOut = 500)

public void m1() throws InterruptedException

{

Thread.sleep(1000);

System.out.println("Test method one");

}

@Test(timeOut = 500)

public void m2() throws InterruptedException

{

Thread.sleep(400);

System.out.println("Test method two");

}

}

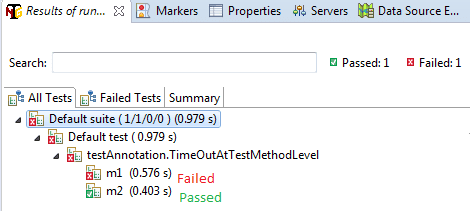
Output:

Test method two

PASSED: m2

FAILED: m1

Look at the following test results in TestNG Results window in Eclipse.

[](https://www.scientecheasy.com/2020/07/testng-test-annotation.html/)

As you can see from the above test result, TestNG executed two tests and failed the first test because the first test has taken more time to complete execution than time specified in the timeOut attribute of Test annotation.

This feature is useful for specifying the predefined execution time limit for a specific method.

**12. priority:** This attribute sets the priority at the test method level. It tells TestNG which priority order has to follow to run the test method.

Lower priorities will be scheduled first. It has the following general form.

Syntax:

     @Test(priority =1 )

     @Test(priority = 2)