Why TestNg is called “**Testing Framework**“?

TestNG, where NG stands for “Next Generation”. TestNG is a testing framework to perform unit, functional, end-to-end, integration, etc testing. This library helps you in writing automated tests.

Here, TestNG comes into picture, saying you just organize your test methods or add some specific information using our annotations to your test methods and handle all types of customization execution from a single file called “testng.xml”. TestNG allows you to control when to run which tests.

**Writing a test is typically a three-step process:**

1. Write the business logic of your test and insert TestNG annotations in your code.
2. Add the information about your test (e.g., the class name, the groups you wish to run, etc…) in a testng.xml file or in build.xml.
3. Run TestNG.

**Hierarchy of TestNG**

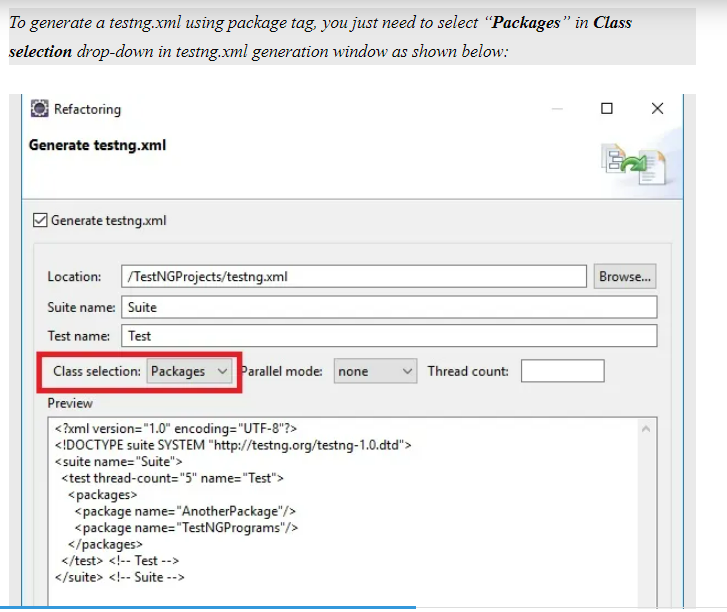


**There are five levels in a testng.xml as marked in red border in above image:**

1. Method
2. Classes
3. Class
4. Test
5. Suite

* A suite (Project) can have multiple Tests(Modules).
* A Test can have multiple Classes (Please note here I am talking about <class> tag not <classes>.
* You can have only single <classes> tag in a <test> tag).

All class names of each package are grouped within separate package tag first and then all package names are combined within <packages> tag. <packages> tag is a container which contains multiple <package> tags. It is similar to nesting of <classes> and <class> tag.



It will include all classes of those packages for testng suite. Remember here we do not have option to select classes to run when we use package tag. You lose that flexibility here.

If you want to include all packages, TestNG allows you to use expressions in stead of multiple <package> tags as below:



**Note:** So, an interface can have TestNG annotated methods in it but you require an implemented class of interface to run it*.*

**Can We Overload Methods in TestNG Class?**

A picture containing diagram

Description automatically generated

You will get above exception in console because TestNG does not allow you to put any type of parameters to a TestNG annotated method.

**Creating overloaded methods in TestNG using Data provider:**

**import** org.testng.annotations.DataProvider;

**import** org.testng.annotations.Test;

**public** **class** OverloadedMethods {

// Data provider which provides one attribute

@Test(dataProvider = "DemoData1")

**public** **void** NormalMethod(String s) {

System.***out***.println("Normal Method");

}

// Data provider which provides two attribute

@Test(dataProvider = "DemoData")

**public** **void** NormalMethod(String s, **int** a) {

System.***out***.println("Overloaded Method");

}

@DataProvider(name = "DemoData")

**public** **static** Object[][] dataProviderMethod() {

**return** **new** Object[][] { { "Amod", 123 } };

}

@DataProvider(name = "DemoData1")

**public** **static** Object[][] dataProviderMethod1() {

**return** **new** Object[][] { { "Amod" } };

}

}

Text, email

Description automatically generated

Creating overloaded methods in TestNG using Parameters:

We can create overloaded methods in TestNG using Parameters.

**import** org.testng.annotations.Parameters;

**import** org.testng.annotations.Test;

**public** **class** OverloadedMethods2 {

@Parameters({"name"})

@Test

**public** **void** NormalMethod(String name)

{

System.***out***.println("Normal Method");

}

@Parameters({"name1","age"})

@Test

**public** **void** NormalMethod(String name,**int** age)

{

System.***out***.println("Overloaded Method");

}

}

Text

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Inheritance In TestNG Classes

**package** InheritanceInTestNG;

**import** org.testng.annotations.Test;

**public** **class** SuperTestNgClass {

@Test

**public** **void** superTestNgMethod() {

System.***out***.println("Super testng method");

}

}

**package** InheritanceInTestNG;

**import** org.testng.annotations.Test;

**public** **class** SubTestNGClass **extends** SuperTestNgClass

{

@Test

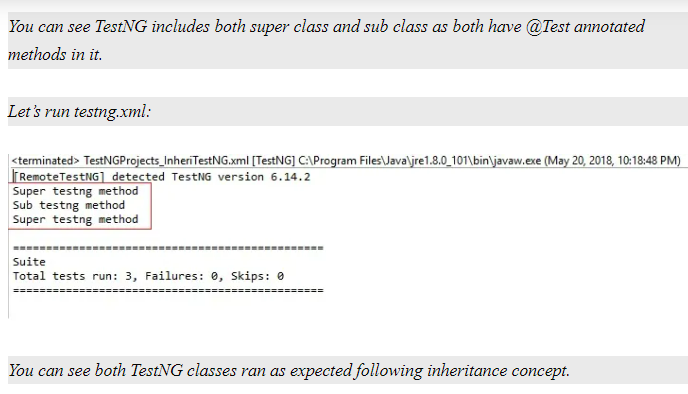
**public** **void** subTestNgMethod()

{

System.***out***.println("Sub testng method");

}

}



Super TestNG class “SuperTestNgClass” has one @Test annotated method which got executed and printed “Super testng method”. Class “SubTestNGClass” extends class “SuperTestNgClass”. So, sub class “SubTestNGClass” will have total two @Test annotated methods: One from super class and another its own. So, it executed both methods and printed output. Here TestNG will not wait for any explicit call to super class testng method.

# Can @Test Annotation Be Used For A Class In TestNG?

Yes, We can use @Test annotation on class.

Suppose, you need to write 10 @Test annotated method in a class. Either you mark all methods as @Test annotated or directly annotate class itself as @Test. All method will be treated as @Test annotated method by default.

**package** TestAnnotation;

**import** org.testng.annotations.Test;

@Test

**public** **class** TestEx1 {

**public** **void** m1()

{

System.***out***.println("M1");

}

**public** **void** m2()

{

System.***out***.println("M2");

}

**public** **void** m3()

{

System.***out***.println("M3");

}

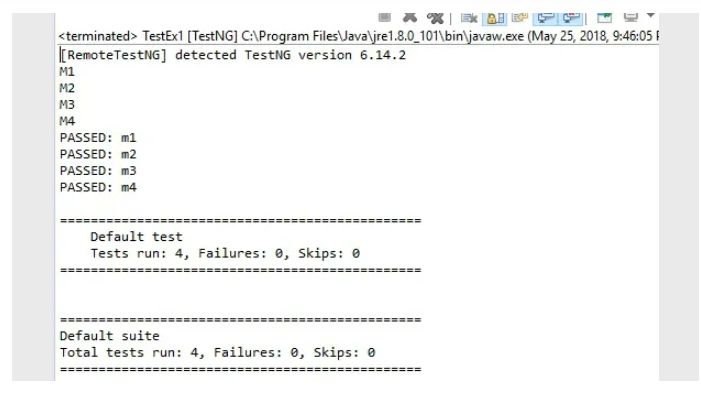
**public** **void** m4()

{

System.***out***.println("M4");

}

}



What will happen if I need to write other annotated method like BeforeMethod or AfterMethod? Not a big deal. Mark required annotation on method. It will override class level @Test annotation.

**Another scenario:** Suppose you have a class which is @Test annotated at class level. You have a sub class extending super class. Sub class is not @Test annotated method. **Because of inheritance, automatically sub class will also become @Test annotated.**

**package** TestAnnotation;

**import** org.testng.annotations.Test;

@Test

**class** superCLass {

}

**public** **class** TestEx1 **extends** superCLass {

**public** **void** m1() {

System.***out***.println("M1");

}

**public** **void** m2() {

System.***out***.println("M2");

}

**public** **void** m3() {

System.***out***.println("M3");

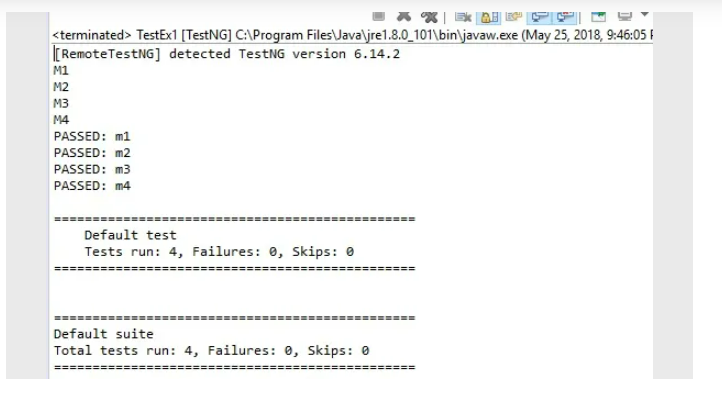
}

**public** **void** m4() {

System.***out***.println("M4");

}

}



# Default Priority Of @Test Methods In TestNG

**How TestNG will priorities these method? It follows below rule based on ASCII value.**

1. ‘$’ has highest priority.
2. Capital alphabets will be second priority.
3. Underscore will have third priority.
4. Small letter alphabets will be fourth priority.

Note : When a TestNG class inherits another class, @Test methods of superclass also become part of subclass. But when you run it, superclass methods and sub class methods will be prioritized and executed separately . Test methods from both the classes will not be sorted and prioritised combined. It will be individual process for both superclass and subclass.

First subclass test methods got executed followed by superclass methods.

**Overriding Of Test Methods in TestNG Class**

We learnt that TestNG prioritized and execute methods of super class and subclass separately. But if some methods are overridden, only overridden method will be executed and parent method of super class will be ignored. If you have better understanding of overriding concept in java, then it is similar to calling super class method using sub class reference.

**@Test Annotation – Games Of Priority Of Methods In TestNG**

1. You need to write this element in small letters as “priority”.
2. Priority is an element applicable only for @Test annotated methods.
3. Priority should be an integer value.
4. It can be negative , zero or positive number. If you write it is decimal, you must need to cast it into integer.
5. TestNG will execute test methods from lowest to highest priority. Remember Lower priorities will be scheduled first.
6. TestNG ignore default priority based on ASCII if priority value is provided.
7. You can pass duplicate priority to test methods. In case of tie, TestNG will decide priority based on ASCII value.
8. You can create a TestNG class with some test methods with priority and some without priority in same class.
9. Test methods without priority will have default priority of Zero and execution sequence will be decided based on ASCII value.
10. Suppose if you have three test methods with first two test methods say M1, M2 with priority as -1 and 2 respectively and third test methods say M3 without any priority. In this case, TestNG will have priority as M1=-1, M3=0 and M2=2. So, FIrst M1 will be executed followed by M3 and M2.
11. You cannot define multiple priority element for a test method.
12. You cannot pass priority to methods through testng.xml.

Diagram

Description automatically generated with low confidence

 TestNG provides you below annotations:

1. @BeforeSuite
2. @AfterSuite
3. @BeforeTest
4. @AfterTest
5. @BeforeGroups
6. @AfterGroups
7. @BeforeClass
8. @AfterClass
9. @BeforeMethod
10. @AfterMethod
11. @DataProvider
12. @Factory
13. @Listeners
14. @Parameters
15. @Test

* A testng xml can have single <suite> tag.
* You can have multiple <test> tags in a <suite> tag.
* You can have single <classes> tag in a <test> tag.
* You can have multiple <class> tag inside a <classes> tag.
* You can have multiple test method (@Test annotated) names inside <methods> tag inside a <class> tag.

**Understand Usage Of AlwaysRun Attribute With Test Method Of TestNG**

Text

Description automatically generated

**Usage Of AlwaysRun Attribute With Configuration Methods Of TestNG**

Since BeforeSuite method is failed, it skips all other configuration method and test method as well that you can see in console log above:

SKIPPED CONFIGURATION: @BeforeTest beforeTestMethod

SKIPPED CONFIGURATION: @BeforeMethod beforeMethodMethod

SKIPPED CONFIGURATION: @AfterMethod afterMethodMethod

SKIPPED CONFIGURATION: @AfterClass afterClassMethod

SKIPPED CONFIGURATION: @AfterTest afterTestMethod

SKIPPED: testMethod

TestNG shows final result of suite as below:

===============================================

Default suite: Total tests run: 1, Failures: 0, Skips: 1

Configuration Failures: 1, Skips: 7

===============================================

You can see TestNG displays result of “Configuration” and “Test” execution separately. We have total 8 configuration methods. Out of 8, 1 failed and 7 skipped which is displayed by TestNG.

In above case, you will expect at least some configuration methods like @AfterSuite method should be run for cleanup or report generation irrespective of suite result. To achieve this, you need to use alwaysRun attribute as below:

You can see @AfterSuite annotated method gets executed as we set alwaysRun to true for it. This help you in properly wrap up suite execution.

In the same way, you can mark alwaysRun attribute to true for configuration methods which you would like to get executed in any case.

**Grouping Test Methods In TestNG**

A picture containing text

Description automatically generated

**Groups Attribute At Class Level In TestNG**

***Solution is marking a class to group/s instead of methods****.*

TestNG allows you to mention “groups” attribute at class level also which will be inherited automatically to @Test annotated methods. It will eliminate writing same lines of code for each methods.

**package** GroupsExample;

**import** org.testng.annotations.Test;

// Marking a class with @Test annotation.

// Assign a group at class level itself.

// It will be inherited to methods of class

@Test(groups = { "G1" })

**public** **class** GroupClassA {

// Note here I am not writing @Test as it will inherited from class level.

// Group information will also be inherited

**public** **void** G1Method1() {

System.***out***.println("G1Method1");

}

// Note here I am not writing @Test as it will inherited from class level.

// Group information will also be inherited

**public** **void** G1Method2() {

System.***out***.println("G1Method2");

}

// Note here I am not writing @Test as it will inherited from class level.

// Group information will also be inherited

**public** **void** G1Method3() {

System.***out***.println("G1Method3");

}

}



### **Overriding “groups” attribute at test method level from class level:**

Value of “groups” attribute can be overridden at test level from class level. Calling it “overridden” will not be perfect conceptually because when you declare a group for a test method at test level, that method will belong to group mentioned at class level and method level both. It is as good as multiple groups concept in TestNG.

**package** GroupsExample;

**import** org.testng.annotations.Test;

// Marking a class with @Test annotation.

// Assign a group at class level itself.

// It will be inherited to methods of class

@Test(groups = { "G1" })

**public** **class** GroupClassA {

// This method will inherited group info from class level.

// this means this method will belong to group "G1".

**public** **void** G1Method1() {

System.***out***.println("G1Method1");

}

// This method will inherited group info from class level.

// this means this method will belong to group "G1".

**public** **void** G1Method2() {

System.***out***.println("G1Method2");

}

// This method will belong to groups "G1" and "G2" both.

// It is as good as multiple groups concept.

@Test(groups = { "G2" })

**public** **void** G1Method3() {

System.***out***.println("G1Method3");

}

}

A picture containing text

Description automatically generated

A picture containing text

Description automatically generated

Icon

Description automatically generated

# Customizing TestNG Xml For Methods Belong To Multiple Group Names

**package** GroupsExample;

**import** org.testng.annotations.Test;

**public** **class** MultipleGroups {

// A test method belongs to Group G1 and G2

@Test(groups = { "G1", "G2" })

**public** **void** MethodsBelongToG1G2() {

System.***out***.println("MethodsBelongToG1G2");

}

// A test method belongs to Group G1, G2 and G3

@Test(groups = { "G1", "G2", "G3" })

**public** **void** MethodsBelongToG1G2G3() {

System.***out***.println("MethodsBelongToG1G2G3");

}

// A test method belongs to Group G1

@Test(groups = { "G1" })

**public** **void** MethodsBelongToG1() {

System.***out***.println("MethodsBelongToG1");

}

// A test method belongs to Group G2

@Test(groups = { "G2" })

**public** **void** MethodsBelongToG2() {

System.***out***.println("MethodsBelongToG2");

}

// A test method belongs to Group G3

@Test(groups = { "G3" })

**public** **void** MethodsBelongToG3() {

System.***out***.println("MethodsBelongToG3");

}

// A test method belongs to Group G1 and G3

@Test(groups = { "G1", "G3" })

**public** **void** MethodsBelongToG1G3() {

System.***out***.println("MethodsBelongToG1G3");

}

}



Graphical user interface, text, application

Description automatically generated

Graphical user interface

Description automatically generated with low confidence

**Can A Test Method Return A Value In TestNG?**

“**Can a test method return a value? If yes, how can you use it”**

Yes, a @Test annotated method can return a value but that method will not be considered as a test method for TestNG anymore i.e. TestNG will ignore that method. Let’s see an example:

**package** TestAnnotation;

**import** org.testng.annotations.Test;

**public** **class** ValueReturningTest {

// A @Test method which is returning a value will be ignored by TestNG

@Test

**public** String returnMethod()

{

System.***out***.println("Returning Method");

**return** "AmodMahajan";

}

@Test

**public** **void** normalMethod()

{

System.***out***.println("Normal Method");

}

}

A picture containing text

Description automatically generated

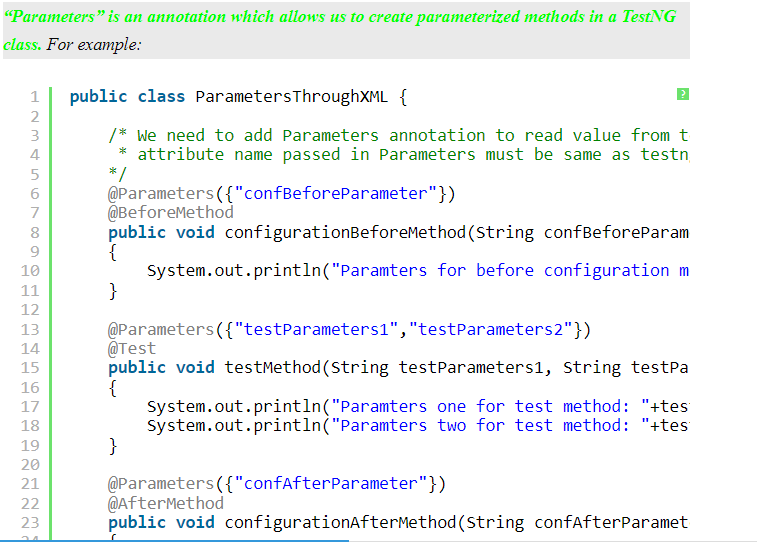
Since TestNG follows best practices for unit testing, which means a unit test method should not have a return value. This is the reason a @Test annotated method which is returning a value is ignored by TestNG default run behaviour.

But you can force TestNG to include test methods which are returning values to be included in normal run. You need to set allow-return-values as true at suite level. Update your testng xml as below:

Graphical user interface, text

Description automatically generated

**Difference Between “Parameters” And “Parameter” In TestNG**



Graphical user interface, text, application

Description automatically generated

1. “Parameters” is an annotation which allows us to create parameterized methods in a TestNG class.
2. “Parameter” is a tag which is used to pass values to parameterized methods of a TestNG class.

**Constructor With @Parameter Annotation**

The approach of using @Parameters annotations with each methods is not so good.

Can’t we do something so that all parameters will be initialized at once with values provided at run time which will be used by methods instead of individual Parameter annotation for each method ?

Yes, we can. We can achieve that using Constructor concept of Java.

**package** Parameters;

**import** org.testng.annotations.AfterMethod;

**import** org.testng.annotations.BeforeMethod;

**import** org.testng.annotations.Parameters;

**import** org.testng.annotations.Test;

**public** **class** ParametersExampleAtClonstructor {

// List of parameters

String p1;

String p2;

String p3;

String p4;

// Accept all parameters in Constructor from testng.xml

@Parameters({"confBeforeParameter","testParameters1","testParameters2","confAfterParameter"})

**public** ParametersExampleAtClonstructor(String s1, String s2, String s3, String s4)

{

// Initialize all parameters

System.***out***.println("in constructor");

p1=s1;

p2=s2;

p3=s3;

p4=s4;

}

// Use whatever parameters are needed in methods

// Note here I am not using any parameter in method signature and Parameters annotation

@BeforeMethod

**public** **void** configurationBeforeMethod()

{

System.***out***.println("Paramters for before configuration method: "+p1);

}

@Test

**public** **void** testMethod()

{

System.***out***.println("Paramters one for test method: "+p2);

System.***out***.println("Paramters two for test method: "+p3);

}

@AfterMethod

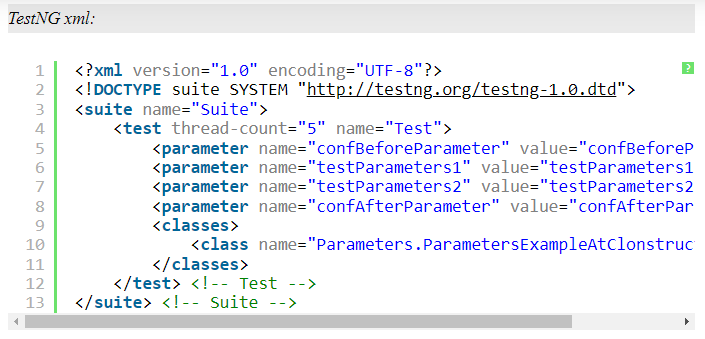
**public** **void** configurationAfterMethod()

{

System.***out***.println("Paramters for after configuration method: "+p4);

}

}



**Passing Parameters At Test Method Level In TestNG**

TestNG allows us to create parameterized methods in a TestNG class. A TestNG class may contain more than one @Test annotated methods and every test method may accept zero or more parameters. We can pass those parameters as a whole at “suite” level or “test” level.

**package** Parameters;

**import** org.testng.annotations.Parameters;

**import** org.testng.annotations.Test;

**public** **class** ParametersAtMethodLevel {

/\*

\* Two test methods accepting parameters/arguments

\*/

@Parameters({"testParameters1","testParameters2"})

@Test

**public** **void** testMethod1(String testParameters1, String testParameters2)

{

System.***out***.println("Paramters one for test method 1: "+testParameters1);

System.***out***.println("Paramters two for test method 1: "+testParameters2);

}

@Parameters({"testParameters3","testParameters4"})

@Test

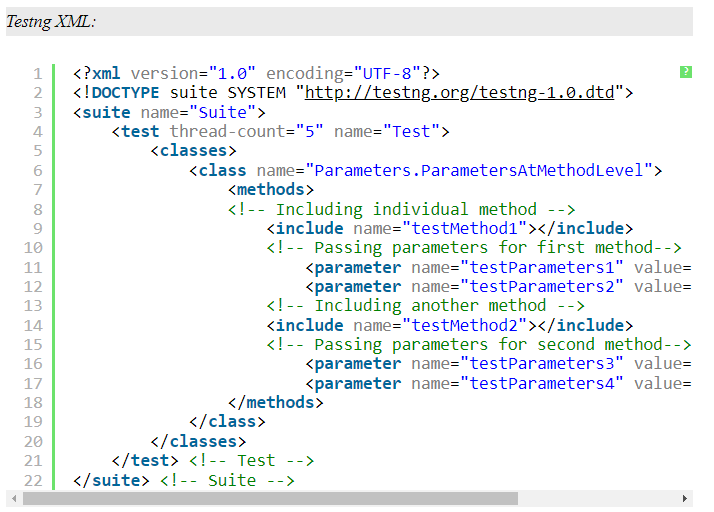
**public** **void** testMethod2(String testParameters1, String testParameters2)

{

System.***out***.println("Paramters one for test method 2: "+testParameters1);

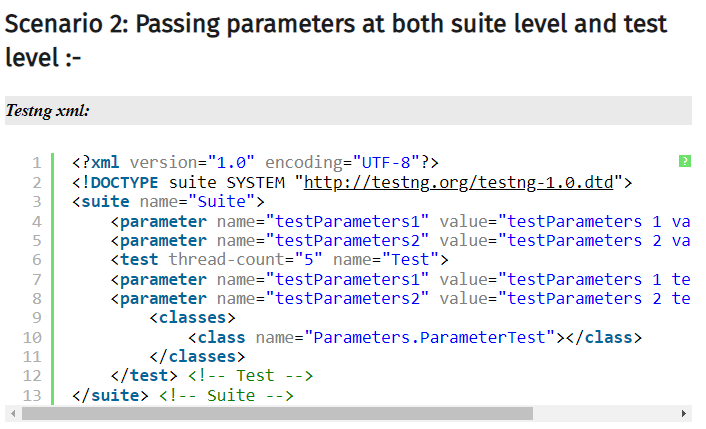
System.***out***.println("Paramters two for test method 2: "+testParameters2);

}

}

**Overriding Parameters In TestNG**

Parameter value at higher level will be overridden by value at lower level if same parameter is passed. For example: If we pass a parameter say “Param” with value say “value1” at suite level and also we pass value “value2” for same parameter “Param” at test level, then parameter “Param” will have value as “value2” at that test level.



Text

Description automatically generated

# Marking A Parameter As Optional In TestNG

How to set a parameter as Optional?

TestNG provides you flexibility of declaring parameters as optional. When we declare a parameter as optional, TestNG first looks in testng xml **if** parameter value is provided. If provided, it will use that value otherwise it will assign **default** value to parameter. It will not **throw** any exception.

We just need to append parameter name with “@Optional” (from org.testng.annotations.Optional) annotation. Example:

**package** Parameters;

**import** org.testng.annotations.Optional;

**import** org.testng.annotations.Parameters;

**import** org.testng.annotations.Test;

**public** **class** ParameterTest {

@Test

@Parameters({"testParameters1","testParameters2"})

// I used Optional annotation with parameter declaration

**public** **void** testMethod**(@Optional** String testParameters1, **@Optional** String testParameters2)

{

System.***out***.println("Paramters one for test method: "+testParameters1);

System.***out***.println("Paramters two for test method: "+testParameters2);

}

}

# How To Pass Parameters Of Different Datatypes In TestNG

1. **We can pass parameters in whatever data types we required. TestNG will convert them automatically to required data type.**
2. **TestNG is intelligent in case of boolean. If you pass anything true, it will take true and false in case of false. If you pass anything other than this, it will take false by default.**

**package** Parameters;

**import** org.testng.annotations.Parameters;

**import** org.testng.annotations.Test;

**public** **class** ParametersWithDifferentDatatypes {

/\* TestNG will convert automatically passed parameter value sin desired data types.\*/

@Parameters({"StringParam", "intParam", "booleanParam"})

@Test

/\*

\* A test method accepting argument of different datatypes

\*/

**public** **void** sampleTest(String stringParaValue, **int** intParamValue, **boolean** booleanParamValue)

{

System.***out***.println("String parameter: "+stringParaValue);

System.***out***.println("Int parameter: "+intParamValue);

System.***out***.println("Boolean parameter: "+booleanParamValue);

}

}

Graphical user interface, text, application

Description automatically generated

# Dependency In TestNG – Creating Dependency Among Test Methods – DependsOnMethod

**import** org.testng.annotations.BeforeTest;

**import** org.testng.annotations.Test;

**public** **class** MultiLevelDependency {

@BeforeTest

**public** **void** launchBrowser()

{

System.***out***.println("browser Launched.");

}

@Test

**public** **void** loadFacebookURL()

{

System.***out***.println("Facebook URL loaded.");

}

/\*

\* registerOnFacebook depends on loadFacebookURL

\*/

@Test(dependsOnMethods = {"loadFacebookURL"})

**public** **void** registerOnFacebook()

{

System.***out***.println("Register on Facebook.");

}

/\*

\* postStatusOnFacebook depends on registerOnFacebook

\*/

@Test(dependsOnMethods = {"registerOnFacebook"})

**public** **void** postStatusOnFacebook()

{

System.***out***.println("Post an status on Facebook.");

}

}

# Dependency In TestNG – Creating Dependency Among Test Methods – DependsOnGroup

You have created automation scripts for Integration testing. We know that in integration testing, data flows from one module to another. You divide your test methods as per group. Say Group 1 and group 2. You run Group 1 methods first followed by Methods of Group 2. We can achieve the same in TestNG using DependsOnGroup.

**import** org.testng.annotations.BeforeTest;

**import** org.testng.annotations.Test;

**public** **class** DependsOnGroupsExample {

// Test method belong to preSetupTestA

@Test(groups= {"preSetupTestA"})

**public** **void** methodA()

{

System.***out***.println("MethodA");

}

//Test method belong to preSetupTestA

@Test(groups= {"preSetupTestA"})

**public** **void** methodB()

{

System.***out***.println("MethodB");

}

// Test method belong to preSetupTestB

@Test(groups= {"preSetupTestB"})

**public** **void** methodC()

{

System.***out***.println("MethodC");

}

// Test method belong to preSetupTestB

@Test(groups= {"preSetupTestB"})

**public** **void** methodD()

{

System.***out***.println("MethodD");

}

// Test method which is dependent of other groups

@Test(dependsOnGroups = {"preSetupTestB", "preSetupTestA"})

**public** **void** finalTest()

{

System.***out***.println("Final Test.");

}

}

**package** Dependecy;

**import** org.testng.annotations.Test;

**public** **class** DependsOnGroupsExample {

// Test method belong to preSetupTestA

@Test(groups= {"preSetupTestA"}, priority = 2)

**public** **void** methodA()

{

System.***out***.println("MethodA");

}

//Test method belong to preSetupTestA

@Test(groups= {"preSetupTestA"}, priority= 1)

**public** **void** methodB()

{

System.***out***.println("MethodB");

}

// Test method belong to preSetupTestB

@Test(groups= {"preSetupTestB"}, priority = 4)

**public** **void** methodC()

{

System.***out***.println("MethodC");

}

// Test method belong to preSetupTestB

@Test(groups= {"preSetupTestB"}, priority = 3)

**public** **void** methodD()

{

System.***out***.println("MethodD");

}

// Test method which is dependent of other groups

@Test(dependsOnGroups = {"preSetupTestB", "preSetupTestA"})

**public** **void** finalTest()

{

System.***out***.println("Final Test.");

}

}

So you can use dependsOnGroup attribute, when you need all test methods of a specific group or multiple groups to be executed before.

# Dependency In TestNG – IgnoreMissingDependencies – Another Way Of Achieving Soft Dependencies

**import** org.testng.annotations.Test;

**public** **class** HardDependeny {

@Test(enabled= **false**)

**public** **void** Test1()

{

System.***out***.println("I am Test1");

}

/\*

\* This test method depends on another test method named "Test1" which is not enabled.

\* Setting ignoreMissingDependencies as true will make it soft dependency.

\*/

@Test(dependsOnMethods= {"Test1"}, **ignoreMissingDependencies**= **true**)

**public** **void** Test2()

{

System.***out***.println("I am Test2");

}

}

# Sharing Data Among Test Methods In TestNG

Suppose in a TestNG class, you have two @Test annotated methods. In first method you have created some data and same data you want to use in second method. This is called sharing data between tests. We can do in multiple approaches. General way to use a static variable to store and retrieve as required. We will see an example below:

**import** org.testng.ITestContext;

**import** org.testng.annotations.Test;

**public** **class** SharingStateBetweenUsingStaticVariable {

// Creating a static variable so that updated value can be used across run

**public** **static** String *fullName* = "";

// Priority is provided to run test method in a specific sequence

@Test(priority =1 )

**public** **void** generateData()

{

String firstName = "Amod";

// Setting the data

*fullName* = firstName;

}

@Test(priority =2 )

**public** **void** useData(ITestContext context)

{

String lastName = "Mahajan";

// Modifying the data updated by first method

*fullName* = *fullName* +" "+lastName;

System.***out***.println("Full Name is : "+*fullName*);

}

}

**import** org.testng.annotations.Test;

**public** **class** SharingStateBetweenUsingStaticVariable2 {

@Test(priority =1 )

**public** **void** useDataFromOtherClass()

{

String profession = "Blogger";

// Modifying the data updated by second method

SharingStateBetweenUsingStaticVariable.fullName = SharingStateBetweenUsingStaticVariable.fullName +" "+profession;

System.***out***.println("Full Name with profession is : "+SharingStateBetweenUsingStaticVariable.fullName);

}

}

Text

Description automatically generated

# Sharing Data Among Test Methods In TestNG Using ITestContext

ITestContext interface defines a test context which contains all the information for a given test run. **An instance of this context is passed to the test listeners** so they can query information about their environment.

ITestContext is a powerful interface which provides many useful methods. In this post, we will see two important methods setAttribute​(java.lang.String name, java.lang.Object value) and getAttribute​(java.lang.String name).

setAttribute(atttributeName, attributeValue) :- Set a custom attribute. It is similar to adding at element in a Map as key-value pair. Kindly pay attention here that attribute value can be of any type. This is the reason this method accepts Object type as value.

getAttribute(attributeName) :- Get the value of given attribute name. Remember return type is an Object.

**import** org.testng.ITestContext;

**import** org.testng.annotations.Test;

**public** **class** SharingStateBetweenTestsUsingITestContext {

// ITestContext reference is created once for the duration of test run.

// Passing ITestContext as a parameter to @Test method to use it.

@Test

**public** **void** generateData(ITestContext context)

{

String firstName = "Amod";

// Setting an attribute with name and its value

context.setAttribute("FirstName", firstName);

}

@Test

**public** **void** useData(ITestContext context)

{

String lastName = "Mahajan";

context.setAttribute("LastName", lastName);

// Retrieving attribute value set in ITestContext

String FN = (String) context.getAttribute("FirstName");

String fullName = FN +" "+lastName;

System.***out***.println("Full Name is : "+fullName);

context.setAttribute("FullName", fullName);

}

}

**import** org.testng.ITestContext;

**import** org.testng.annotations.Test;

**public** **class** SharingStateBetweenTestsUsingITestContext2 {

@Test

**public** **void** useDataInOtherClass(ITestContext context)

{

String profession = "Blogger";

String fullName = (String) context.getAttribute("FullName");

String nameWithProfession = fullName +" "+profession;

System.***out***.println("Full Name is : "+fullName);

System.***out***.println("Full Name with profession is "+ nameWithProfession);

}

}

**Note: Actually scope of added variable in ITestContext is within current test. Outside of test, it will give you null.**

