TestNG_Complete_Reference

May 18, 2020

1 TestNG

1.1 1. Introduction

1.1.1 1.1 What is TestNG?

- TestNG, where NG stands for "next generation" is a test automation framework inspired by JUnit (in Java) and NUnit (in C#).
- It can be used for unit, functional, integration, and end-to-end testing.
- TestNG has gained a lot of popularity within a short time and is one of the most widely used testing frameworks among Java developers.
- It mainly uses Java annotations to configure and write test methods.
- TestNG was developed by Cedric Beust. He developed it to overcome a deficiency in JUnit.

1.1.2 1.2 TestNG Features

Multiple Before and After annotation options: TestNG provides multiple kinds of Before/After annotations for support of different setup and cleanup 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 and tell TestNG to include or exclude a particular group in a test.

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 multi-threaded 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.

1.1.3 1.3 TestNG Run Options

Class: Using this option you can provide the class name along with the package to run only the said specific test class.

Method: Using this you can run only a specific method in a test class.

Groups: In case you would like to run specific test methods belonging to a particular TestNG group, you can enter those here for executing them.

Suite: In case you have suite files in the form of testng.xml files, you can select those here for execution.

1.1.4 1.4 TestNG Allows

- 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 multithreaded execution options.

1.2 2. testng.xml

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

1.2.1 2.1 Sample xml file

```
In [ ]: <suite name="First Suite" verbose="1" >
            <test name="First Test" >
                <classes>
                    <class name="test.FirstTest" />
                </classes>
            </test>
        </suite>
```

The preceding XML defines a TestNG suite using the tag name suite. The name of the suite is mentioned using the name attribute (in this case First Suite). It contains a test, declared using the XML tag test and the name of the test is given using the name attribute. The test contains a class (test.FirstTest) to be considered for test execution which is configured using the classes and class tags as mentioned in the XML file.

1.2.2 2.2 How to Run testng.xml

2.2.1 Using Command Prompt

1. Open the command prompt in the system.
2. Go to the Test Java project folder where the new testng.xml is created.
3. Type the following line. java -cp "/opt/testng-6.8.jar:bin" org.testng.TestNG testng.xml "/opt/testng-6.8.jar:bin" -> Class path of jar and compiled java files. org.testng.TestNG -> Consists of the main method that Java will use to execute the testng.xml file testng.xml -> Passed as an argument at the command line.

java -cp "/opt/testng-6.8.jar:bin" org.testng.TestNG -testnames "Second Test" testng.xml -> The preceding command will execute a test with the name Second Test from testng.xml if it exists in the defined suite

2.2.2 Using Eclipse

1. Open Eclipse and go to the project where we have created the testng.xml file.
2. Select the testng.xml file, right-click on it, and select Run As | TestNG suite.
3. Eclipse will execute the XML file as TestNG suite and we can see the report in Eclipse:

You can also use the Run Configuration option provided by Eclipse to customize your TestNG tests in Eclipse.

1. On the top-bar menu of Eclipse, go to Run | Run Configurations.
2. Select TestNG from the set of configurations and click on the New Launch Configuration icon.
3. On the configuration window give a name My Test to the configuration.
4. Go to the Project section, click on Browse and select your project on the project window.
5. Now go to the Suite section and click on Browse. Select the mytestng.xml configuration.
6. Click on Apply, and then click on Run. This will run the selected testng XML configuration file.

1.3 3. Create Tests

1.3.1 3.1 Creating Multiple Tests

1. Below defined different classes with test methods :

```
In [ ]: package test.firstpackage;

        import org.testng.annotations.Test;
        public class FirstTestClass {
          @Test
          public void firstTest(){
            System.out.println("First test method");
          }
        }
```

```
In [ ]: package test.firstpackage;

        import org.testng.annotations.Test;
        public class SecondTestClass {
          @Test
          public void secondTest(){
            System.out.println("Second test method");
          }
        }
```

```
In [ ]: package test.firstpackage;

        import org.testng.annotations.Test;
        public class ThirdTestClass {
          @Test
          public void thirdTest(){
            System.out.println("Third test method");
          }
        }
```

2. Now open the testng.xml file and add the following snippet to it:

```
In [ ]: <suite name="Suite" verbose="1" >
            <test name="FirstTest" >
                <classes>
                    <class name="test.firstpackage.FirstTestClass" />
                </classes>
            </test>
            <test name="SecondTest" >
                <classes>
                    <class name="test.firstpackage.SecondTestClass" />
                </classes>
            </test>
        </suite>
```

The XML file defines a suite with the name Suite. The suite contains two tests with names FirstTest and SecondTest respectively. These tests are configured to execute separate classes test.firstpackage.FirstTestClass and test.firstpackage.SecondTestClass. When the XML file is executed as a suite in TestNG, each class is executed by a separate test section of a suite.

3. Now run the testng.xml file using Eclipse.

1.3.2 3.2 Creating a Test with Classes

```
In [ ]: <suite name="Class Suite" verbose="1">
            <test name="Test">
                <classes>
                    <class name="test.firstpackage.FirstTestClass" />
                    <class name="test.secondpackage.FirstTestClass" />
                    <class name="test.thirdpackage.FirstTestClass" />
```

```
            </classes>
        </test>
    </suite>
```

The preceding testng XML suite defines a test with three classes (one from each package). To add a class to your test suite just use a combination of classes and class tag as shown. Use the class tag with the attribute name having a value of the class name along with the package name (for example, test.firstpackage. FirstTestClass) to add a test class to your test.

1.3.3 3.3 Creating a Test with Packages

```
In [ ]: <suite name="Package Suite" verbose="1">
            <test name="Package Test">
                <packages>
                    <package name="test.firstpackage" />
                    <package name="test.secondpackage" />
                </packages>
            </test>
        </suite>
```

The preceding testng XML suite defines a test with two packages (test. firstpackage and test.secondpackage, respectively). To add a package to test suite just use a combination of the packages and package tag as shown in the previous code. Use the package tag with the attribute name having a value of the package name (for example, test.firstpackage) under the tag packages to add packages to the tests.

```
In [ ]: <suite name="Package Suite" verbose="1">
            <test name="Package Test">
                <packages>
                    <package name="test.*" />
                </packages>
            </test>
        </suite>
```

This will execute all the subpackages present under the package test.

1.3.4 3.4 Creating a Test with Methods

```
In [ ]: <suite name="Method Suite" verbose="1">
            <test name="Method Test">
                <classes>
                    <class name="test.firstpackage.FirstTestClass">
                        <methods>
                            <include name="firstTest" />
                        </methods>
                    </class>
                </classes>
            </test>
        </suite>
```

The preceding testng XML suite defines a class that needs to be considered for test execution and the test method that needs to be included for execution. To add methods to your test suite we have to use the tags methods and include/exclude under them to add or remove particular methods from a test class.

1.3.5 3.5 Creating a Test Suite with Package, Class, and Test Method

```
In [ ]: <suite name="Combine Suite" verbose="1">
            <test name="Combine Test">
                <packages>
                    <package name="test.firstpackage" />
                </packages>
                <classes>
                    <class name="test.secondpackage.FirstTestClass" />
                    <class name="test.thirdpackage.FirstTestClass">
                        <methods>
                            <include name="firstTest" />
                        </methods>
                    </class>
                </classes>
            </test>
        </suite>
```

The preceding testng XML suite defines a test with a package (test.firstpackage), a particular class(test.secondpackage.FirstTestClass), and a particular test method (firstTest under the class test.thirdpackage.FirstTestClass) as part of the test suite.

1.4 4. Including and Excluding

1.4.1 4.1 Include/Exclude Packages

4.1.1 Test Suite to Include a Particular Package

```
In [ ]: <suite name="Include Package Suite" verbose="1">
            <test name="Include Package Test">
                <packages>
                    <package name="test.*">
                        <include name="test.secondpackage" />
                    </package>
                </packages>
            </test>
        </suite>
```

The preceding test suite defines a test, which includes all subpackages of the test package (defined by using regular expression test.*) and includes only a particular package from all the packages for test execution. This is done by using the include tag with the name attribute value as the package name that needs to be included (that is, test.secondpackage). This informs TestNG to include classes belonging to the included package for test execution.

4.1.2 Test Suite to Exclude a Particular Package

```
In [ ]: <suite name="Exclude Package Suite" verbose="1">
            <test name="Exclude Package Test">
                <packages>
                    <package name="test.*">
                        <exclude name="test.secondpackage" />
                    </package>
                </packages>
            </test>
        </suite>
```

The preceding test suite defines a test by including all subpackages of the test package (defined by using the regular expression test.*) and excluding only a particular package from all the packages for test execution. This is done by using the exclude tag with the name attribute value as the package name (that is, test.secondpackage) that needs to be excluded. This informs TestNG to exclude classes belonging to the package from test execution.

1.4.2 4.2 Include/Exclude Methods

4.2.1 Test Suite to Exclude a Particular Method

```
In [ ]: <suite name="Exclude Method Suite" verbose="1">
            <test name="Exclude Method Test">
                <classes>
                    <class name="test.firstpackage.FirstTestClass">
                        <methods>
                            <exclude name="firstTest" />
                            <include name="secondTest">
                        </methods>
                    </class>
                </classes>
            </test>
        </suite>
```

The preceding testng XML suite defines a class that needs to be considered for test execution and the test method that needs to be excluded from execution. To exclude a method from your test suite we have to use the tags methods and exclude under them to exclude a particular method from a test class.

1.4.3 4.3 Using Regular Expressions to Include/Exclude

```
In [ ]: <suite name="Regular Exp Suite" verbose="1">
            <test name="Regular Exp Test">
                <classes>
                    <class name="test.regularexppackage.RegularExpClass">
                        <methods>
                            <include name=".*Test.*" />
                        </methods>
```

```
            </class>
        </classes>
    </test>
</suite>
```

The preceding testng XML suite is configured to consider only those test methods from a particular class whose name contains the word Test in it. The regular expression is considered by use of .* before and after the text.

1.5 5. Annotations

@BeforeSuite : The annotated method will be executed before any tests declared inside a TestNG suite.

@AfterSuite : The annotated method will be executed after any tests declared inside a TestNG suite.

@BeforeTest : The annotated methods will be executed before each test section declared inside a TestNG suite.

@AfterTest : The annotated methods will be executed after each test section declared inside a TestNG suite.

@BeforeGroups : BeforeGroups annotated method will run before any of the test method of the specified group is executed.

@AfterGroups : AfterGroups annotated method will run after any of the test method of the specified group gets executed.

@BeforeClass : BeforeClass annotated method is executed before any of the test method of a test class.

@AfterClass : AfterClass annotated method is executed after the execution of every test methods of a test class are executed.

@BeforeMethod : These annotated methods are executed before the execution of each test method.

@AfterMethod : These annotated methods are executed after the execution of each test method.

@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.

@Factory : Marks a 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.

@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.

@Parameters : This annotation is used to pass parameters to a test method. These parameter values are provided using the testng.xml configuration file at runtime.

@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.

```
In [ ]: public class TestClass {
            @BeforeSuite
            public void beforeSuite(){
                System.out.println("Before Suite method");
            }
```

```
@AfterSuite
public void afterSuite(){
    System.out.println("After Suite method");
}

@BeforeTest
public void beforeTest(){
    System.out.println("Before Test method");
}

@AfterTest
public void afterTest(){
    System.out.println("After Test method");
}

@BeforeClass
public void beforeClass(){
    System.out.println("Before Class method");
}

@AfterClass
public void afterClass(){
    System.out.println("After Class method");
}

@BeforeGroups(groups={"testOne"})
public void beforeGroupOne(){
    System.out.println("Before Group Test One method");
}

@AfterGroups(groups={"testOne"})
public void afterGroupOne(){
    System.out.println("After Group Test One method");
}

@BeforeMethod
public void beforeMethod(){
    System.out.println("Before Method");
}

@AfterMethod
public void afterMethod(){
    System.out.println("After Method");
}

@Test(groups={"testOne"})
public void testOneMethod(){
    System.out.println("Test one method");
```

```
    }

    @Test(groups={"testTwo"})
    public void testTwoMethod(){
        System.out.println("Test two method");
    }
```

```
In [ ]: <suite name="First Suite" verbose="1" >
    <test name="First Test" >
        <classes>
            <class name="test.beforeafter.TestClass" >
                <methods>
                    <include name="testOneMethod"/>
                </methods>
            </class>
        </classes>
    </test>
    <test name="Second Test" >
        <classes>
            <class name="test.beforeafter.TestClass" >
                <methods>
                    <include name="testTwoMethod"/>
                </methods>
            </class>
        </classes>
    </test>
</suite>
```

The preceding testng.xml file contains two tests containing the same test class but with different test methods

1.5.1 5.1 Test Annotation

One of the basic annotations of TestNG is the Test annotation. This annotation marks a method or a class as part of the TestNG test. If applied at class level this annotation will mark all the public methods present inside the class as test methods for TestNG test. It supports lot of attributes which you can use along with the annotation, which will enable you to use the different features provided by TestNG. The following is a list of attributes supported by the Test annotation:

alwaysRun : Takes a true or false value. If set to true this method will always run even if its depending method fails.

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

dataProviderClass : The class where TestNG should look for the data-provider method mentioned in the dataProvider attribute. By default its the current class or its base classes.

dependsOnGroups : Specifies the list of groups this method depends on.

dependsOnMethods : Specifies the list of methods this method depends on.

description : The description of this method.

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.

expectedExceptions : This attribute is used for exception testing. This attribute specifies the list of exceptions this method is expected to throw. In case a different exception is thrown.

groups : List of groups the said method or class belongs to.

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

5.1.1 Disabling a Test

```
In [ ]: @Test(enabled=false)
        public void testMethodTwo(){
            System.out.println("Test method two.");
        }
```

5.1.2 Exception Test

```
In [ ]: import java.io.IOException;
        import org.testng.annotations.Test;

        public class ExceptionTest {
            @Test(expectedExceptions={IOException.class})
            public void exceptionTestOne() throws Exception{
                 throw new IOException();
            }

            @Test(expectedExceptions={IOException.class, NullPointerException.class})
            public void exceptionTestTwo() throws Exception{
                throw new Exception();
            }

            @Test(expectedExceptions={IOException.class}, expectedExceptionsMessageRegExp=".* M
            public void exceptionMsgTestTwo() throws Exception{
                throw new IOException("Pass Message test");
            }
        }
```

The preceding class contains two test methods, each throwing one particular kind of exception, exceptionTestOne throws IOException where as exceptionTestTwo throws Exception. The expected exception to validate while running these tests is mentioned using the expectedExceptions attribute value while using the Test annotation.

5.1.3 Time Test

- At suite level: This will be applicable for all the tests in the said TestNG test suite
- At each test method level: This will be applicable for the said test method and will override the time period if configured at the suite level

```
In [ ]: <suite name="Time test Suite" time-out="500" verbose="1" >
            <test name="Timed Test" >
                <classes>
```

```
            <class name="test.timetest.TimeSuite" />
        </classes>
    </test>
</suite>
```

The preceding testng.xml contains a suite with a single test considering a test class for test execution. You'll notice that the suite tag contains a new attribute named time-out which is set with a value 500. This attribute applies a time-out period for test methods for the whole suite. That means if any test method in the said suite takes more than the specified time period (in this case 500 milliseconds) to complete execution it will be marked as failed.

```
In [ ]: @Test(timeOut=500)
        public void timeTestOne() throws InterruptedException{
            System.out.println("Time test method one");
        }
```

1.5.2 5.2 Parameterization of Test

- Through testng XML configuration file
- Through DataProviders

5.2.1 Parameterization through testng.xml

```
In [ ]: @Parameters({ "suite-param" })
        @Test
        public void prameterTestOne(String param) {
            System.out.println("Test one suite param is: " + param);
        }

        @Parameters({ "suite-param", "test-three-param" })
        @Test
        public void prameterTestThree(String param, String paramTwo) {
            System.out.println("Test three suite param is: " + param);
            System.out.println("Test three param is: " + paramTwo);
        }
```

```
In [ ]: <suite name="Parameter test Suite" verbose="1">
            <parameter name="suite-param" value="suite level parameter" />
            <test name="Parameter Test one">
                <classes>
                    <class name="test.parameter.ParameterTest">
                        <methods>
                            <include name="prameterTestOne" />
                        </methods>
                    </class>
                </classes>
            </test>
            <test name="Parameter Test two">
                <parameter name="test-two-param" value="Test two parameter" />
```

```
            <classes>
                <class name="test.parameter.ParameterTest">
                    <methods>
                        <include name="prameterTestTwo" />
                    </methods>
                </class>
            </classes>
        </test>
        <test name="Parameter Test three">
            <parameter name="suite-param" value="overiding suite parameter" />
            <parameter name="test-three-param" value="test three parameter" />
            <classes>
                <class name="test.parameter.ParameterTest">
                    <methods>
                        <include name="prameterTestThree" />
                    </methods>
                </class>
            </classes>
        </test>
    </suite>
```

The preceding XML file contains three tests in it, each explains a different way of passing the parameters to the test methods. The parameter is declared in testng XML file using the parameter tag. The name attribute of the tag defines name of the parameter whereas the value attribute defines the value of the said parameter. The tag can be used at suite level as well as at test level, as you can see from the preceding XML file.

1.5.3 5.3 Data Provider

One of the important features provided by TestNG is the DataProvider feature. It helps the user to write data-driven tests, that means same test method can be run multiple times with different datasets. DataProvider is the second way of passing parameters to test methods. It helps in providing complex parameters to the test methods as it is not possible to do this from XML.

```
In [ ]: package test.dataprovider;

        import org.testng.annotations.DataProvider;
        import org.testng.annotations.Test;

        public class SameClassDataProvider {
            @DataProvider(name = "data-provider")
            public Object[][] dataProviderMethod() {
                return new Object[][] { { "data one" }, { "data two" } };
            }

            @Test(dataProvider = "data-provider")
            public void testMethod(String data) {
                System.out.println("Data is: " + data);
```

```
    }
}
```

The preceding test class contains a test method which takes one argument as input and prints it to console when executed. A DataProvider method is also available in the same class by using the DataProvider annotation of TestNG. The name of the said DataProvider method is mentioned using the name attribute of the DataProvider annotation. The DataProvider returns a double Object class array with two sets of data, data one and data two.

The DataProvider to provide parameter values to a test method is defined by giving the name of the data provider using the dataProvider attribute while using the Test annotation.

Data Provider in Different Class :

```
In [ ]: public class TestClass {
            @Test(dataProvider = "data-provider", dataProviderClass=DataProviderClass.class)
            public void testMethod(String data) {
                System.out.println("Data is: " + data);
            }
        }

In [ ]: public class DataProviderClass {
            @DataProvider(name="data-provider")
            public static Object[][] dataProviderMethod(){
                return new Object[][] { { "data one" }, { "data two" } };
            }
        }
```

1.6 6. Groups

1.6.1 6.1 Grouping Tests

```
In [ ]: package test.groups;
        import org.testng.annotations.Test;

        public class TestGroup {
            @Test(groups={"test-group"})
            public void testMethodOne(){
                System.out.println("Test method one belonging to group.");
            }

            @Test
            public void testMethodTwo(){
                System.out.println("Test method two not belonging to group.");
            }

            @Test(groups={"test-group"})
            public void testMethodThree(){
                System.out.println("Test method three belonging to group.");
            }
        }
```

```
In [ ]: <suite name="Time test Suite" verbose="1">
            <test name="Timed Test">
                <groups>
                    <run>
                        <include name="test-group" />
                    </run>
                </groups>
                <classes>
                    <class name="test.groups.TestGroup" />
                </classes>
            </test>
        </suite>
```

The preceding XML file contains only one test inside a suite. This contains the groups section defined by using the groups tag as shown in the code. The run tag represents the group that needs to be run. The include tag represents the name of the group that needs to be executed.

Tests with Multiple Groups :

```
In [ ]: package test.groups;
        import org.testng.annotations.Test;

        public class MultiGroup {
            @Test(groups={"group-one"})
            public void testMethodOne(){
                System.out.println("Test method one belonging to group.");
            }

            @Test(groups={"group-one","group-two"})
            public void testMethodTwo(){
                System.out.println("Test method two belonging to both group.");
            }

            @Test(groups={"group-two"})
            public void testMethodThree(){
                System.out.println("Test method three belonging to group.");
            }
        }
```

```
In [ ]: <suite name="Multi Group Suite" verbose="1">
            <test name="Group Test one">
                <groups>
                    <run>
                        <include name="group-one" />
                    </run>
                </groups>
                <classes>
                    <class name="test.groups.MultiGroup" />
                </classes>
```

```
        </test>
        <test name="Group Test two">
            <groups>
                 <run>
                     <include name="group-two" />
                </run>
            </groups>
            <classes>
                <class name="test.groups.MultiGroup" />
            </classes>
        </test>
    </suite>
```

The preceding testng XML suite contains two tests, each of them executing test methods belonging to a particular group.

1.6.2 6.2 Include/Exclude Groups using testng.xml file

```
In [ ]: <suite name="Exlude Group Suite" verbose="1">
        <test name="Exclude Group Test">
            <groups>
                <run>
                    <include name="include-group" />
                    <exclude name="exclude-group" />
                </run>
            </groups>
            <classes>
                <class name="test.groups.ExcludeGroup" />
            </classes>
        </test>
    </suite>
```

The preceding XML contains a simple test in which the group include-group is included in the test using the include XML tag and the group exclude-group is being excluded from the test execution by using the exclude tag.

Using Regular Expressions :

```
In [ ]: <suite name="Regular Exp. Group Suite" verbose="1">
        <test name="Regular Exp. Test">
            <groups>
                <run>
                    <include name="include.*" />
                    <exclude name=".*exclude" />
                </run>
            </groups>
            <classes>
                <class name="test.groups.RegularExpressionGroup" />
            </classes>
```

```
        </test>
    </suite>
```

1.6.3 6.3 Default Group

Sometimes we may need to assign a default group to a set of test methods that belong to a class. This can be achieved by using the @Test annotation at class level and defining the default group in the said @Test annotation. This way all the public methods that belong to the said class will automatically become TestNG test methods and become part of the said group.

```
In [ ]: package test.groups;
        import org.testng.annotations.Test;

        @Test(groups={"default-group"})
        public class DefaultGroup {

            public void testMethodOne(){
                System.out.println("Test method one.");
            }

            public void testMethodTwo(){
                System.out.println("Test method two.");
            }

            @Test(groups={"test-group"})
            public void testMethodThree(){
                System.out.println("Test method three.");
            }
        }
```

The preceding class contains three methods that print a message onto console when executed. All of the methods are considered as test methods by the use of the @Test annotation on the class. All of the methods belong to the group default-group by mentioning the group name at the class level. One of the test methods also belong to the group test-group, this is done by using the @Test annotation at the method level as shown in the preceding code.

```
In [ ]: <suite name="Default Group Suite" verbose="1">
            <test name="Default Group Test one">
                <groups>
                    <run>
                        <include name="default-group" />
                    </run>
                </groups>
                <classes>
                    <class name="test.groups.DefaultGroup" />
                </classes>
            </test>
            <test name="Default Group Test two">
                <groups>
```

```
                <run>
                    <include name="test-group" />
                </run>
            </groups>
            <classes>
                <class name="test.groups.DefaultGroup" />
            </classes>
        </test>
    </suite>
```

The preceding XML contains two separate tests, which execute two separate groups, default-group and test-group respectively. Both tests consider the same test class to search for the test method that belongs to the group.

1.6.4 6.4 Group of Groups

TestNG allows users to create groups out of existing groups and then use them during the creation of the test suite. You can create new groups by including and excluding certain groups and then use them.

```
In [ ]: <suite name="Group of group Suite" verbose="1">
        <test name="Group of group Test">
            <groups>
                <define name="include-group">
                    <include name="include-test-one" />
                    <include name="include-test-two" />
                </define>
                <define name="exclude-group">
                    <include name="test-one-exclude" />
                    <include name="test-two-exclude" />
                </define>
                <run>
                    <include name="include-group" />
                    <exclude name="exclude-group" />
                </run>
            </groups>
            <classes>
                <class name="test.groups.RegularExpressionGroup" />
            </classes>
        </test>
    </suite>
```

The preceding XML contains one test in it. Two groups of groups have been defined inside the test, and then these groups are used for test execution. The MetaGroup is created using the define tag inside the groups tag as shown in the previous code. The name of the new group is defined using the name attribute under the define tag. Groups are included and excluded from the new group by using the include and exclude tags.

1.7 7. Dependencies

1.7.1 7.1 Single Test Dependency

```
In [ ]: package test.depends.method;
        import org.testng.annotations.Test;

        public class SimpleDependencyTest {
            @Test(dependsOnMethods={"testTwo"})
            public void testOne(){
                System.out.println("Test method one");
            }

            @Test
            public void testTwo(){
                System.out.println("Test method two");
            }
        }
```

The preceding test class contains two test methods which print a message name onto the console when executed. Here, test method testOne depends on test method testTwo. This is configured by using the attribute dependsOnMethods while using the Test annotation as shown is the preceding code.

1.7.2 7.2 Multiple Test Dependency

```
In [ ]: package test.depends.method;
        import org.testng.annotations.Test;

        public class MultiDependencyTest {
             @Test(dependsOnMethods={"testTwo","testThree"})
             public void testOne(){
                  System.out.println("Test method one");
             }

             @Test
             public void testTwo(){
                  System.out.println("Test method two");
             }

             @Test
             public void testThree(){
                  System.out.println("Test method three");
             }
        }
```

The preceding test class contains three test methods which print a message name onto the console when executed. Here test method testOne depends on test methods testTwo and testThree. This is configured by using the attribute dependsOnMethods while using the Test annotation as shown in the preceding code.

1.7.3 7.3 Inherited Test Dependency

```
In [ ]: package test.depends.method;
        import org.testng.annotations.Test;

        public class InheritedTest extends SimpleDependencyTest{
             @Test(dependsOnMethods={"testOne"})
             public void testThree(){
                  System.out.println("Test three method in Inherited test");
             }

             @Test
             public void testFour(){
                  System.out.println("Test four method in Inherited test");
             }
        }
```

The preceding test class contains two test methods which print a message name onto the console when executed. Here test method testThree depends on test method testOne. This is configured by using the attribute dependsOnMethods while using the Test annotation as shown in the preceding code.

1.7.4 7.4 Group Dependency

```
In [ ]: package test.depends.groups;
        import org.testng.annotations.Test;

        public class SimpleGroupDependency {
             @Test(dependsOnGroups={"test-group"})
             public void groupTestOne(){
                  System.out.println("Group Test method one");
             }

             @Test(groups={"test-group"})
             public void groupTestTwo(){
                  System.out.println("Group test method two");
             }

             @Test(groups={"test-group"})
             public void groupTestThree(){
                  System.out.println("Group Test method three");
            }
        }
```

The preceding test class contains three test methods which print a message onto the console when executed. Two of the test methods belong to a group named test group whereas the third method named groupTestOne depends on the group test group. The dependency on the group is configured using the attribute dependsOnGroups while using the Test annotation, as shown in the preceding code.

1.7.5 7.5 Depending on Methods from different Classes

```
In [ ]: package test.depends.groups;
        import org.testng.annotations.Test;

        public class DifferentClassDependency {
             @Test(dependsOnGroups={"test-group","same-class"})
             public void testOne(){
                  System.out.println("Different class test method one");
             }

             @Test(groups={"same-class"})
             public void testTwo(){
                  System.out.println("Different class test method two");
             }

             @Test(groups={"same-class"})
             public void testThree(){
                  System.out.println("Different class test method three");
             }
        }
```

The preceding test class contains three test methods which print a message onto the console when executed. Two of the test methods belong to group named sameclass whereas the third method, named testOne, depends on the groups named test-group and same-class. The group test-group refers to the test that belongs to the earlier created class named SimpleGroupTest.

1.7.6 7.6 Using Regular Expressions

```
In [ ]: import org.testng.annotations.Test;

        public class RegularExpressionTest {
             @Test(dependsOnGroups={"starts-with.*"})
             public void regularExpMethod(){
                  System.out.println("Dependent method");
             }

             @Test(groups={"starts-with-one"})
             public void startsWithMethodOne(){
                  System.out.println("Starts with method one");
             }

             @Test(groups={"starts-with-two"})
             public void startsWithMethodTwo(){
                  System.out.println("Starts with method two");
             }
        }
```

The preceding test class contains three test methods which print a message onto the console

when executed. Two of the test methods belong to different groups named starts-with-one and starts-with-two, respectively, whereas the third method named regularExpMethod depends on all the groups whose names start with the text starts-with.

1.7.7 7.7 XML-based Dependency Configuration

```
In [ ]: <suite name="Simple xml dependency Suite" verbose="1">
            <test name="Simple xml dependency Test">
                    <groups>
                        <dependencies>
                            <group name="dependent-group" depends-on="test-group" />
                        </dependencies>
                        <run>
                             <include name="dependent-group" />
                        </run>
                    </groups>
                    <classes>
                         <class name="test.depends.xml.SimpleXmlDependency" />
                    </classes>
            </test>
        </suite>
```

The preceding testng XML configuration file contains a single test inside the suite. Group dependency is defined using the dependencies attribute under the groups block. The group tag is used with the group name and the names of the group that the said group depends on, as shown in the previous XML file.

1.7.8 7.8 Multigroup Dependency in XML

```
In [ ]: <suite name="Multi group xml dependency Suite" verbose="1">
            <test name="Multi group xml dependency Test">
                     <groups>
                        <dependencies>
                            <group name="dependent-group" depends-on="test-group-one test-grou
                        </dependencies>
                        <run>
                             <include name="dependent-group" />
                        </run>
                    </groups>
                    <classes>
                         <class name="test.depends.xml.MultiGrpXmlDependency" />
                    </classes>
            </test>
        </suite>
```

The preceding testng XML configuration contains a single test inside the suite. Group dependency is defined using the dependencies attribute under the groups block. The group tag is used with the group name and the names of the groups that the said group depends on, as shown in the

preceding XML file. In case the group is dependent upon multiple groups, each group is separated by a space, as shown in the preceding XML file.

1.7.9 7.9 Regular Expressions for defining Dependency in XML

```
In [ ]: <suite name="Regexpxmldependency Suite" verbose="1">
                <test name="Regexp xml dependency Test">
                    <groups>
                        <dependencies>
                            <group name="test" depends-on="starts-with.*" />
                        </dependencies>
                        <run>
                             <include name="test" />
                        </run>
                    </groups>
                    <classes>
                         <class name="test.depends.xml.RegularExpressionXmlTest" />
                    </classes>
                </test>
        </suite>
```

The preceding testng XML configuration contains a single test inside the suite. Group dependency is defined using the dependencies attribute under the groups block. The group tag is used with the group name. Dependent group is defined using the regular expressions.

1.8 8. Factory Annotation

```
In [ ]: package test.classes;
        import org.testng.annotations.Test;

        public class SimpleTest {
             @Test
             public void simpleTest(){
                  System.out.println("Simple Test Method.");
             }
        }
```

```
In [ ]: package test.factory;
        import org.testng.annotations.Factory;
        import test.classes.SimpleTest;

        public class SimpleTestFactory {
             @Factory
             public Object[] factoryMethod(){
                  return new Object[]{ new SimpleTest(), new SimpleTest() };
             }
        }
```

The preceding class defines a factory method inside it. A factory method is defined by declaring @Factory above the respective test method. It's mandatory that a factory method should return an array of Object class (Object []) as mentioned in the preceding code snippet.

1.8.1 8.1 Passing Parameters to Test Classes

```
In [ ]: package test.classes;
        import org.testng.annotations.Test;

        public class ParameterTest {
            private int param;
            public ParameterTest(int param){
            this.param = param;
            }

            @Test
            public void testMethodOne(){
            int opValue=param+1;
            System.out.println("Test method one output: "+ opValue);
            }
        }
```

```
In [ ]: package test.factory;
        import org.testng.annotations.Factory;
        import test.classes.ParameterTest;
        public class ParameterFactory {
            @Factory
            public Object[] paramFactory(){
            return new Object[]{ new ParameterTest(0), new ParameterTest(1) };
            }
        }
```

The preceding class defines a factory method inside it. This factory method returns an array of the Object class containing two objects of ParameterTest class. Select the previous factory test class in Eclipse and run it as a TestNG test.

1.8.2 8.2 Using DataProvider along with the @Factory Annotation

```
In [ ]: package test.classes;
        import org.testng.annotations.DataProvider;
        import org.testng.annotations.Factory;
        import org.testng.annotations.Test;
        public class DataProviderConsTest {
            private int param;
            @Factory(dataProvider="dataMethod")
            public DataProviderConsTest(int param){
            this.param=param;
            }
```

```
    @DataProvider
    public static Object[][] dataMethod(){
    return new Object[][]{ {0}, {1} };
    }

    @Test
    public void testMethodOne(){
    int opValue=param+1;
    System.out.println("Test method one output: "+ opValue);
    }

    @Test
    public void testMethodTwo(){
    int opValue=param+2;
    System.out.println("Test method two output: "+ opValue);
    }
}
```

The preceding class is similar to the test class, which we used earlier. The constructor of this test class takes one argument as integer, which is assigned to a local variable param. This variable then is used in the two test methods present in the test class. Each of the test methods adds a value to param and prints it to the console on execution. The constructor of the test class is annotated with the @Factory annotation. This annotation uses a DataProvider method named dataMethod for providing values to the constructor of the test class. The DataProvider method returns a double object array in which the first array represents the dataset, which decides the number of times the test will be iterated, whereas the second array is the actual parameter value that will be passed to the test method per iteration. The said double object array contains two datasets with values 0 and 1.

1.8.3 8.3 DataProvider or Factory

DataProvider: A test method that uses DataProvider will be executed a multiple number of times based on the data provided by the DataProvider. The test method will be executed using the same instance of the test class to which the test method belongs. Factory: A factory will execute all the test methods present inside a test class using a separate instance of the respective class.

1.8.4 8.4 DataProvider Test

```
In [ ]: import org.testng.annotations.BeforeClass;
        import org.testng.annotations.DataProvider;
        import org.testng.annotations.Test;

        public class DataProviderClass {
            @BeforeClass
            public void beforeClass(){
            System.out.println("Before class executed");
            }
```

```
    @Test(dataProvider="dataMethod")
    public void testMethod(String param){
    System.out.println("The parameter value is: "+param);
    }

    @DataProvider
    public Object[][] dataMethod(){
    return new Object[][]{ {"one"}, {"two"} };
    }
}
```

The preceding class contains the testMethod and beforeClass methods. testMethod takes a String argument and the value of the argument is provided by the DataProvider method, dataMethod. The beforeClass method prints a message onto the console when executed, and the same is the case with testMethod. testMethod prints the argument passed onto it to the console when executed.

1.9 9. Parallelism

Parallelism or multithreading in software terms is defined as the ability of the software, operating system, or program to execute multiple parts or subcomponents of another program simultaneously. This ability is provided by TestNG too, it allows the tests to run in parallel or multithreaded mode. This means that based on the test suite configuration, different threads are started simultaneously and the test methods are executed in them. This gives a user a lot of advantages over normal execution, mainly reduction in execution time and ability to verify a multithreaded code.

1.9.1 9.1 Running Methods in Parallel

```
In [ ]: <suite name="Simple Suite" parallel="methods" thread-count="2" >
            <test name="Simple test">
                <classes>
                   <class name="test.parallelism.SimpleClass" />
                </classes>
            </test>
    </suite>
```

The preceding XML defines a simple test suite which contains only a single test inside it. The test considers the class SimpleClass for test execution. Multithreading or parallelism is configured using the attribute parallel and thread-count at the suite level as shown in the previous XML file. The parallel be done for each class, each method, or for each test in the suite. The thread-count attribute is used to configure the maximum number of threads to be spawned for each suite.

1.9.2 9.2 Running Classes in Parallel

```
In [ ]: <suite name="Test-class Suite" parallel="classes" thread-count="2" >
            <test name="Test-class test" >
                <classes>
                   <class name="test.parallelism.SampleTestClassOne" />
                   <class name="test.parallelism.SampleTestClassTwo" />
```

```
            </classes>
        </test>
    </suite>
```

The preceding XML defines a simple test suite, which contains two classes in it: SampleTestClassOne and SampleTestClassTwo.

1.9.3 9.3 Running Tests inside a Suite in Parallel

```
In [ ]: <suite name="Parallel tests" parallel="tests" thread-count="2" >
            <test name="Test One">
                <parameter name="test-name" value="Test One"/>
                <classes>
                   <class name="test.parallelism.SampleTestSuite" />
                </classes>
            </test>
            <test name="Test Two">
                <parameter name="test-name" value="Test Two"/>
                <classes>
                     <class name="test.parallelism.SampleTestSuite" />
                </classes>
            </test>
    </suite>
```

The preceding XML contains two tests in it, each including the SampleTestSuite class for test execution. Each test passes the parameter value Test One and Test Two to test-name. This is passed to the beforeTest method, which sets the value of testName in the test class. This value is then used by the test methods to identify the test that is being executed.

1.9.4 9.4 Running Independent Test in Threads

```
In [ ]: package test.parallelism;
        import org.testng.annotations.Test;

        public class IndependentTestThreading {
             @Test(threadPoolSize=3,invocationCount=6,timeOut=1000)
             public void testMethod(){
             Long id = Thread.currentThread().getId();
             System.out.println("Test method executing on thread with id: "+id);
             }
        }
```

The preceding test class contains a test method, which prints a message onto the console when executed. The ID of the thread on which the current method is being executed is printed along with the said message. The method is configured to run in multithreaded mode by using the threadPoolSize attribute along with the Test annotation. The value of the threadPoolSize is set to 3 in the previous class; this configures the test method to be run in three different threads. The other two attributes, invocationCount and timeOut, configures the test to be invoked a multiple number of times and fail if the execution takes more time. The value of these two attributes is set

to 6 and 1000 respectively, configuring the test method to be run six times and fail if the execution takes more than 1000 milliseconds.