**TestNG Framework**

TestNG is a testing framework developed based on the concepts of JUnit and NUnit however it introduces some new features that make it more powerful and easier to use.

TestNG is designed to cover all categories of tests –

* Unit testing
* Functional testing
* End-to-End testing
* Integration testing
* Regression Testing, etc.,
* Basic requirement is **JDK 5 or higher**.

Basically, a framework is supposed to organize the classes, methods, etc so that there is higher reusability, lesser complexity, to work around with big projects, ability to generate detailed report. Therefore, to refer a framework which can handle all these features, one of the most suitable examples is TestNG

Implementation of TestNG

  

TestNG (Independent) TestNG with Cucumber BDD TestNG with any Closed-Source Framework

TestNG (Independent)

* If a framework is implemented only by using TestNG then it is just enough to run the test but a non-technical person may not understand the concepts clearly.
* It is itself enough to generate reports.

TestNG with Cucumber BDD

* If a framework is implemented by using TestNG and Cucumber BDD then it helps to run the test as well as lets a non-technical person to understand the concepts of working functionality very clearly.
* Default reporting strategy of TestNG could be applied here

TestNG with any Closed-Source Framework

* When an organization or third party framework designs a framework keeping in mind to eliminate the possible issues that will be faced by test designers, those frameworks could be included under this category.
* Some of the common problems faced in designing test scripts are organizing Object Repository and Test Reporting.
* With a closed-source framework it becomes easier to locate objects.

Eg: - CRAFT Frameworks, Third Party Frameworks, etc

* Also a framework under this category will provide extra features according to the requirements wanted.

Eg: - Customized reports

**TestNG Features**

* Supports annotations
  + TestNG provides extra annotations, therefore allows us to add features that will execute before and after every test, before and after specific group and so on. In detail all annotations are explained below
* TestNG uses more Java and OO features
  + As Java is the most popular platform, utilizing Java helps in reaching large scale audience
  + Object Oriented features guides in effective utilization of the resources. Reduces line of codes
* Supports testing integrated classes (e.g., by default,
* No need to create a new test class instance for every test method
* Separates compile-time test code from run-time configuration/data info
* Tests will be run without having to change anything in the code resulting in flexibility.
* Introduces ‘test groups’. Once you have compiled your tests, you can just ask TestNG to run which group.
* Front-end tests or
* Fast, Slow, Database tests, etc.
* Supports Dependent test methods, parallel testing, load testing, and partial failure.
* Flexible plug-in API.

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| **@BeforeSuite** | This method will run only once before all tests in this suite have run. |
| **@AfterSuite** | This method will run only once after all tests in this suite have run. |
| **@BeforeClass** | This method will run only once before the first test method in the current class is invoked. |
| **@AfterClass** | This method will run only once after all the test methods in the current class have run. |
| **@BeforeTest** | This method will run before each test method belonging to the classes inside the <test> tag is run. |
| **@AfterTest** | This method will run after each test method belonging to the classes inside the <test> tag has run. |
| **@BeforeGroups** | This method is guaranteed to run shortly before the first test method that belongs to any of the defined group is invoked. |
| **@AfterGroups** | This method is guaranteed to run shortly after the last test method that belongs to any of the defined group has invoked. |
| **@BeforeMethod** | This method will run before each test method. |
| **@AfterMethod** | This method will run after each test method. |
| **@Parameters** | Describes how to pass parameters to a @Test method. |
| **@Test** | Contains the details of a test to be run. (basically a Test case) |
| **@DataProvider** | Marks a method as supplying data for a test method. The annotated method must return an Object[ ][ ], where each Object[ ] can be assigned the parameter list of the test method. The @Test method that wants to receive data from this DataProvider needs to use a dataProvider name equals to the name of this annotation. |
| **@Factory** | Marks a method as a factory that returns objects that will be used by TestNG as Test classes. The method must return Object[ ]. |
| **@Listeners** | Defines listeners on a test class. Utilizes methods of listening class |

**@DataProvider Example:**

@DataProvider

**public** Object[][] ValidDataProvider() {

**return** **new** Object[][]{

{ 'A', 65 },{ 'a', 97 },

{ 'B', 66 },{ 'b', 98 },

{ 'C', 67 },{ 'c', 99 },

{ 'D', 68 },{ 'd', 100 },

{ 'Z', 90 },{ 'z', 122 },

{ '1', 49 },{ '9', 57 }

};

}

@DataProvider Example:

@Test(dataProvider = "ValidDataProvider")

**public** **void** CharToASCIITest(**final** **char** character, **final** **int** ascii) {

**int** result = CharUtils.CharToASCII(character);

Assert.assertEquals(result, ascii);

}

**@Factory Example**

**public** **class** SimpleTest

{

@Test

**public** **void** simpleTest() {

System.out.println("Simple Test Method.");

}

}

**public** **class** SimpleTestFactory

{

@Factory

**public** Object[] factoryMethod() {

**return** **new** Object[] { **new** SimpleTest(), **new** SimpleTest() };

}

}