Usage	**Nunit**	**TestNG**

Groups of tests	[TestFixture]
	  [Category("LongRunning")]
	  public class LongRunningTests
	  {
	    // ...
	  }
		@Test(groups = { "bonding", "strong_ties" })
		    public void tc01LaunchURL() {
		     //.....
		    }

Preserver Order in Testng	If you want your classes / methods to be run in an unpredictable order, then we should go for preserve-order attribute in testng. In TestNg bydefault the preserve-order attribute will be set to 'true', this means, TestNG will run your tests in the order they are found in the XML file.	<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
		<suite name="Preserve order test runs">
		  <test name="Regression 1" **preserve-order="true"**>
		    <classes>
		      <class name="com.pack.preserve.ClassOne"/>
		      <class name="com.pack.preserve.ClassTwo"/>
		      <class name="com.pack.preserve.ClassThree"/>
		    </classes>
		  </test>
		</suite>

priority/Ordering the test cases	public class MyFixture
	{
	    [Test, Order(1)]
	    public void TestA() { /* ... */ }
	
	    [Test, Order(2)]
	    public void TestB() { /* ... */ }
	
	    [Test]
	    public void TestC() { /* ... */ }
	}
		@Test (priority = 1)
		
		public void func(){
		
		//test code
		
		}

The description for this method.	[TestFixture, Description("Fixture description here")]
	  public class SomeTests
	  {
	    [Test, Description("Test description here")]
	    public void OneTest()
	    { /* ... */ }
	  }
		@Test(description = "Test printing out all the Spring beans.")
		public void printAllBeansTest(Method ngMethod)
		{
		    ...
		}

Indicates that a test shouldn't be run for some reason	[Test]
	[Ignore("Ignore a test")]
	public void IgnoredTest()
	{ /* ... */ }
	-- 	
	[TestFixture]
	[Ignore("Ignore a fixture")]
	public class SuccessTests
	{
	// ...
	}
		@Ignore
		public class TestcaseSample {
		}
		--- 	
		@Test(enabled = false)
		public void testPrintMessage() {
		System.out.println("Inside testPrintMessage()");
		message = "Manisha";
		Assert.assertEquals(message, messageUtil.printMessage());
		}

parameterized test method	[Test, TestCaseSource("DivideCases")]
	public void DivideTest(int n, int d, int q)
	{
	Assert.AreEqual(q, n / d);
	}
	
	static object[] DivideCases =
	{
	new object[] { 12, 3, 4 },
	new object[] { 12, 2, 6 },
	new object[] { 12, 4, 3 }
	};
		public class StaticProvider {
		@DataProvider(name = "create")
		public static Object[][] createData() {
		return new Object[][] {
		new Object[] { new Integer(42) }
		};
		}
		}
		
		public class MyTest {
		@Test(dataProvider = "create", dataProviderClass = StaticProvider.class)
		public void test(Integer n) {
		// ...
		}
		}

run before any test method belonging to the classes inside the <test> tag is run	[OneTimeSetUp]
	public void BaseSetUp() { /* ... */ }
		@BeforeTest
		public void beforeTest() {
		m_dataSource = ...;
		m_jdbcDriver = driver;
		}

run after all the test methods belonging to the classes inside the <test> tag have run.	[OneTimeTearDown]
	public void BaseTearDown() { /* ... */ }
		@AfterTest
		public void afterTest() {
		m_dataSource = ...;
		m_jdbcDriver = driver;
		}

Specifies that the decorated method should be executed multiple times	[Test]
	  [Repeat(25)]
	  public void MyTest( ){
	    // your test logic here
	  }
		@Test(invocationCount = 10)
		public void testServer() {

Causes a test to rerun if it fails, up to a maximum number of times	[Test, Retry(2)]
	public void test(){}
		package Analyzer;
		import org.testng.IRetryAnalyzer;
		import org.testng.ITestResult;
		
		public class RetryAnalyzer implements IRetryAnalyzer {
		   private int retryCnt = 0;
		   private int maxRetryCnt = 2; // count for rerun
		   public boolean retry(ITestResult result) {
		      if (retryCnt < maxRetryCnt) {
		         System.out.println("Retrying " + result.getName() + " again and the count is " + (retryCnt+1));
		         retryCnt++;
		         return true;
		      }
		      return false;
		   }
		}
		--- 	
		 @Test(retryAnalyzer = Analyzer.RetryAnalyzer.class)
		 public void Test1()
		 {
		 Assert.assertEquals(false, true);
		 }

run after each test method.	[TearDown]
	   public void BaseTearDown() { /* ... */ }
		@AfterMethod
		public void afterTest() {
		  m_dataSource = ...;
		  m_jdbcDriver = driver;
		}

Test case method as part of the test.	[Test]
	 public void Add()
	 { /* ... */ }
		@Test
		   public void tc01LaunchURL() {
		    //..... }

The Author Attribute adds information about the author of the tests	[TestFixture]
	[Author("Jane Doe", "jane.doe@example.com")]
	public class MyTests
	{
	 [Test]
	 [Author("Joe Developer")]
	 [Author("Yet Another Developer", "not.my.email@example.com")]
	 public void Test2() { /* ... */ }
	}	

method with parameters	[TestCase(12,3, Result=4)]
	[TestCase(12,2, Result=6)]
	public int DivideTest(int n, int d)
	{
	  return(n / d);
	}
	-- 	
	[TestCase(12, 3, 4)]
	public void DivideTest(int n, int d, int q)
	{
	    Assert.AreEqual(q, n / d);
	}
		@Parameters({ "first-name" })
		@Test
		public void testSingleString(String firstName) {
		  System.out.println("Invoked testString " + firstName);
		  assert "Cedric".equals(firstName);
		}
		-- 	
		<suite name="My suite">
		  <parameter name="first-name" value="Cedric"/>
		  <test name="Simple example">
		  <-- ... -->

Provides a timeout value in milliseconds for test cases	[Test, Timeout(2000)]
	public void
	PotentiallyLongRunningTest()
	{
	    /* ... */
	}
	--------------------------- 	
	[Test, MaxTime(2000)]
	public void TimedTest()
	{
	    /* ... */
	}
		@Test(timeOut=1000) // specify time in milliseconds
		 public void executetimeOut() throws InterruptedException{
		  Thread.sleep(3000);
		            // Thread.sleep(500);
		 }

PropertyAttribute provides a generalized approach to setting named properties on any test case or fixture, using a name/value pair.	[Test, Property("Severity", "Critical")]
	    public void AdditionTest()
	    { /* ... */ }	

run before each test method.	[SetUp]
	   public void BaseSetUp() { /* ... */ }
		@BeforeMethod
		public void beforeTest() {
		  m_dataSource = ...;
		  m_jdbcDriver = driver;
		}

[TestFixture]	This is the attribute that marks a class that contains tests and, optionally, setup or teardown methods	[TestFixture]
		  public class SuccessTests
		  {
		    // ...
		  }
		-- 	
		[TestFixture(42, 42, 99)]
		public class ParameterizedTestFixture
		{
		    private string eq1;
		    private string eq2;
		    private string neq;
		
		    public ParameterizedTestFixture(int eq1, int eq2, int neq)
		    {
		        this.eq1 = eq1.ToString();
		        this.eq2 = eq2.ToString();
		        this.neq = neq.ToString();
		    }

TestCaseData	The TestCaseData class provides extended test case information for a parameterized test, although any object deriving from TestCaseParameters may be used. Unlike NUnit 2, you cannot implement ITestCaseData, you must derive from TestCaseParameters.	[TestFixture]
		public class MyTests
		{
		   [TestCaseSource(typeof(MyDataClass), "TestCases")]
		   public int DivideTest(int n, int d)
		   {
		      return n / d;
		   }
		}
		
		public class MyDataClass
		{
		   public static IEnumerable TestCases
		   {
		      get
		      {
		         yield return new TestCaseData(12, 3).Returns(4);
		         yield return new TestCaseData(12, 2).Returns(6);
		         yield return new TestCaseData(12, 4).Returns(3);
		      }
		   }
		}

[TestFixtureSource]	TestFixtureSourceAttribute is used on a parameterized fixture to identify the source from which the required constructor arguments will be provided. The data is kept separate from the fixture itself and may be used by multiple fixtures.	[TestFixtureSource(typeof(AnotherClass), "FixtureArgs")]
		public class MyTestClass
		{
		   public MyTestClass(string word, int num) { ... }
		
		   ...
		}
		
		class AnotherClass
		{
		   static object [] FixtureArgs = {
		      new object[] { "Question", 1 },
		      new object[] { "Answer", 42 }
		   };
		}

Order of Annotation Execution	[OneTimeSetUp] [SetUp] [Test] {Testcase 1} [TearDown] [SetUp] [Test] {Testcase 2} [TearDown] [OneTimeTearDown]	@BeforeSuite @BeforeTest @BeforeClass @BeforeMethod @Test {Testcase 1} @AfterMethod @BeforeMethod @Test {Testcase 2} @AfterMethod @AfterClass @AfterTest @AfterSuite

Parallel Execution	[TestFixture]
	[Parallelizable]
	public class MyClassTests1 {
	}
	[TestFixture]
	[Parallelizable]
	public class MyClassTests2 {
	}
	-- 	
	Add this into the test project AssemblyInfo.cs class for nunit3 or greater:
	
	// Make all tests in the test assembly run in parallel
	[assembly: Parallelizable(ParallelScope.Fixtures)] or [assembly: Parallelizable(ParallelScope.Children)]
	//Number of iternations
	[assembly: LevelOfParallelism(5)]
		<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd" >
		<suite name = "Parallel Testing Suite">
		  <test name = "Parallel Tests" parallel = "methods" thread-count = "2">
		    <classes>
		      <class name = "ParallelTest" />
		    </classes>
		  </test>
		</suite>
		
		Methods: This will run the parallel tests on all @Test methods in TestNG.
		Tests: All the test cases present inside the <test> tag will run with this value.
		Classes: All the test cases present inside the classes that exist in the XML will run in parallel.
		Instances: This value will run all the test cases parallelly inside the same instance.

Get Test Run status	TestContext.CurrentContext.Result.Outcome.Status	Reporters implement the interface org.testng.IReporter and are notified when all the suites have been run by TestNG
		
		   ITestResult result = Reporter.getCurrentTestResult();
		
		   if(result.getStatus()==ITestResult.SUCCESS) {}
		   Else if(result.getStatus()==ITestResult.FAILURE) {}

[SetUpFixture]	This is the attribute that marks a class that contains the one-time setup or teardown methods for all the test fixtures under a given namespace. The class may contain at most one method marked with the OneTimeSetUpAttribute and one method marked with the OneTimeTearDownAttribute.	using System;
		using NUnit.Framework;
		
		namespace NUnit.Tests
		{
		  [SetUpFixture]
		  public class MySetUpClass
		  {
		    [OneTimeSetUp]
		    public void RunBeforeAnyTests()
		    {
		      // ...
		    }
		
		    [OneTimeTearDown]
		    public void RunAfterAnyTests()
		    {
		      // ...
		    }
		  }
		}