Junit Vs TestNG

Functionality - JUnit 4 vs TestNG

	Annotation Support	Exception Test	Ignore Test	Timeout Test	Suite Test	Group Test	Parameterized (primitive value)	Parameterized (object)	Dependency Test
TestNG	✓	✓	✓	✓	✓	✓	✓	✓	✓
JUnit 4	✓	✓	✓	✓	✓	✗	✓	✗	✗

Feature	JUnit 4	TestNG
test annotation	@Test	@Test
run before all tests in this suite have run	–	@BeforeSuite
run after all tests in this suite have run	–	@AfterSuite
run before the test	–	@BeforeTest
run after the test	–	@AfterTest
run before the first test method that belongs to any of these groups is invoked	–	@BeforeGroups
run after the last test method that belongs to any of these groups is invoked	–	@AfterGroups
run before the first test method in the current class is invoked	@BeforeClass	@BeforeClass
run after all the test methods in the current class have been run	@AfterClass	@AfterClass
run before each test method	@Before	@BeforeMethod
run after each test method	@After	@AfterMethod
ignore test	@ignore	@Test(enbale=false)
expected exception	@Test(expected = ArithmeticException.class)	@Test(expectedExceptions = ArithmeticException.class)
timeout	@Test(timeout = 1000)	@Test(timeout = 1000)

SHAILESH KUMAR(940

Some additional Description about TestNG with example

2. Exception Test

The “exception testing” means what exception throws from the unit test, this feature is implemented in both JUnit 4 and TestNG.

JUnit 4

```
@Test(expected = ArithmeticException.class)
  public void divisionWithException() {
    int i = 1/0;
  }
```

TestNG

```
@Test(expectedExceptions = ArithmeticException.class)
  public void divisionWithException() {
    int i = 1/0;
  }
```

3. Ignore Test

The “Ignored” means whether it should ignore the unit test, this feature is implemented in both JUnit 4 and TestNG .

JUnit 4

```
@Ignore("Not Ready to Run")
@Test
public void divisionWithException() {
  System.out.println("Method is not ready yet");
  }
```

TestNG

```
@Test(enabled=false)
public void divisionWithException() {
  System.out.println("Method is not ready yet");
  }
```

4. Time Test

The “Time Test” means if an unit test takes longer than the specified number of milliseconds to run, the test will terminated and mark as fails, this feature is implemented in both JUnit 4 and TestNG .

JUnit 4

SHAILESH KUMAR(9404067990)

```
@Test(timeout = 1000)
public void infinity() {
        while (true);
}
```

TestNG

```
@Test(timeOut = 1000)
public void infinity() {
        while (true);
}
```

5. Suite Test

The "Suite Test" means bundle a few unit test and run it together. This feature is implemented in both JUnit 4 and TestNG. However both are using very different method to implement it.

JUnit 4

The "@RunWith" and "@Suite" are use to run the suite test. The below class means both unit test "JunitTest1" and "JunitTest2" run together after JunitTest5 executed. All the declaration is define inside the class.

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
        JunitTest1.class,
        JunitTest2.class
})
public class JunitTest5 {
}
```

TestNG

XML file is use to run the suite test. The below XML file means both unit test "TestNGTest1" and "TestNGTest2" will run it together.

```
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="My test suite">
  <test name="testing">
    <classes>
      <class name="com.fsecure.demo.testng.TestNGTest1" />
      <class name="com.fsecure.demo.testng.TestNGTest2" />
    </classes>
  </test>
</suite>
```

TestNG can do more than bundle class testing, it can bundle method testing as well. With TestNG unique "Grouping" concept, every method is tie to a group, it can categorize tests according to features. For example,

Here is a class with four methods, three groups (method1, method2 and method3)

SHAILESH KUMAR(9210406 7990)

```
@Test(groups="method1")
public void testingMethod1() {
  System.out.println("Method - testingMethod1()");
}

@Test(groups="method2")
public void testingMethod2() {
        System.out.println("Method - testingMethod2()");
}

@Test(groups="method1")
public void testingMethod1_1() {
        System.out.println("Method - testingMethod1_1()");
}

@Test(groups="method4")
public void testingMethod4() {
        System.out.println("Method - testingMethod4()");
}
```

With the following XML file, we can execute the unit test with group "method1" only.

```
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="My test suite">
  <test name="testing">
        <groups>
      <run>
        <include name="method1"/>
      </run>
    </groups>
    <classes>
       <class name="com.fsecure.demo.testng.TestNGTest5_2_0" />
    </classes>
  </test>
</suite>
```

With "Grouping" test concept, the integration test possibility is unlimited. For example, we can only test the "DatabaseFuntion" group from all of the unit test classes.

6. Parameterized Test

The "Parameterized Test" means vary parameter value for unit test. This feature is implemented in both JUnit 4 and TestNG. However both are using very different method to implement it.

JUnit 4

The "@RunWith" and "@Parameter" is use to provide parameter value for unit test, @Parameters have to return List[], and the parameter will pass into class constructor as argument.

```
@RunWith(value = Parameterized.class)
public class JunitTest6 {
```

SHAILESH KUMAR(9404067990)

```
    private int number;

    public JunitTest6(int number) {
       this.number = number;
    }

    @Parameters
    public static Collection<Object[]> data() {
      Object[][] data = new Object[][] { { 1 }, { 2 }, { 3 }, { 4 } };
      return Arrays.asList(data);
    }

    @Test
    public void pushTest() {
      System.out.println("Parameterized Number is : " + number);
    }
}
```

It has many limitations here; we have to follow the "JUnit" way to declare the parameter, and the parameter has to pass into constructor in order to initialize the class member as parameter value for testing. The return type of parameter class is "List []", data has been limited to String or a primitive value for testing.

TestNG

XML file or "@DataProvider" is use to provide vary parameter for testing.

XML file for parameterized test.
Only "@Parameters" declares in method which needs parameter for testing, the parametric data will provide in TestNG's XML configuration files. By doing this, we can reuse a single test case with different data sets and even get different results. In addition, even end user, QA or QE can provide their own data in XML file for testing.

Unit Test

```
public class TestNGTest6_1_0 {

    @Test
    @Parameters(value="number")
    public void parameterIntTest(int number) {
        System.out.println("Parameterized Number is : " + number);
    }
}
```

XML File

```
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="My test suite">
  <test name="testing">

    <parameter name="number" value="2"/>
```

SHAILESH KUMAR(9404067990)

```
    <classes>
       <class name="com.fsecure.demo.testng.TestNGTest6_0" />
    </classes>
  </test>
</suite>
```

@DataProvider for parameterized test.

While pulling data values into an XML file can be quite handy, tests occasionally require complex types, which can't be represented as a String or a primitive value. TestNG handles this scenario with its @DataProvider annotation, which facilitates the mapping of complex parameter types to a test method.

@DataProvider for Vector, String or Integer as parameter

```
        @Test(dataProvider = "Data-Provider-Function")
        public void parameterIntTest(Class clzz, String[] number) {
           System.out.println("Parameterized Number is : " + number[0]);
           System.out.println("Parameterized Number is : " + number[1]);
        }

        //This function will provide the patameter data
        @DataProvider(name = "Data-Provider-Function")
        public Object[][] parameterIntTestProvider() {
                return new Object[][]{
                                {Vector.class, new String[]
{"java.util.AbstractList",
"java.util.AbstractCollection"}},
                                {String.class, new String[] {"1", "2"}},
                                {Integer.class, new String[] {"1", "2"}}
                                };

        }
```

@DataProvider for object as parameter
P.S "TestNGTest6_3_0" is an simple object with just get set method for demo.

```
        @Test(dataProvider = "Data-Provider-Function")
        public void parameterIntTest(TestNGTest6_3_0 clzz) {
           System.out.println("Parameterized Number is : " + clzz.getMsg());
           System.out.println("Parameterized Number is : " +
clzz.getNumber());
        }

        //This function will provide the patameter data
        @DataProvider(name = "Data-Provider-Function")
        public Object[][] parameterIntTestProvider() {

                TestNGTest6_3_0 obj = new TestNGTest6_3_0();
                obj.setMsg("Hello");
                obj.setNumber(123);

                return new Object[][]{
                                {obj}
```

SHAILESH KUMAR(9404067990)

```
                };
        }
```

TestNG's parameterized test is very user friendly and flexible (either in XML file or inside the class). It can support many complex data type as parameter value and the possibility is unlimited. As example above, we even can pass in our own object (TestNGTest6_3_0) for parameterized test

7. Dependency Test

The "Parameterized Test" means methods are test base on dependency, which will execute before a desired method. If the dependent method fails, then all subsequent tests will be skipped, not marked as failed.

JUnit 4

JUnit framework is focus on test isolation; it did not support this feature at the moment.

TestNG

It use "dependOnMethods " to implement the dependency testing as following

```
@Test
public void method1() {
   System.out.println("This is method 1");
}

@Test(dependsOnMethods={"method1"})
public void method2() {
        System.out.println("This is method 2");
}
```

The "method2()" will execute only if "method1()" is run successfully, else "method2()" will skip the test.

Conclusion

After go thought all the features comparison, i suggest to use TestNG as core unit test framework for Java project, because TestNG is more advance in parameterize testing, dependency testing and suite testing (Grouping concept). TestNG is meant for high-level testing and complex integration test. Its flexibility is especially useful with large test suites. In addition, TestNG also cover the entire core JUnit4 functionality. It's just no reason for me to use JUnit anymore.

SHAILESH KUMAR(9404067990)