Parameters of @Test

1. **Enabled**: Boolean
    a. If this parameter is set false, then test method will not take part in execution.
    b. E.g. myTestMethod will not take part in execution.

```
@Test(enabled=false)

public void myTestMethod() {
```

2. **Priority**= - int to +int (-2148473638 to + 2,14,84,73,647)
    a. If we want to execute test methods according to some order, then we can use priority.
    b. By default all test method have 0 priority.
    c. If priority is nor t mentioned, then test methods will execute according to the alphabetical order of their name.
    d. E.g yourTestMethod will execute first as it have highest priority.

```
@Test(priority=1)
public void myTestMethod() {
  System.out.println("Hello");
}
@Test(priority=0)
public void yourTestMethod() {
  System.out.println("Hello");
```

3. **alwayRun**: Boolean
    a. If this is set to true, then test method will execute regardless of any dependency.
    b. In below example, myTestMethod will execute even if it depends on 'yourTestMethod'. See dependsOnMethods for reference.
    c. It can also be used to achieve soft-dependency.

```
@Test(alwaysRun=true)
public void myTestMethod() {
  System.out.println("Hello");
}
@Test(priority=0)
public void yourTestMethod() {
  System.out.println("Hello");
}
```

4. **invocationCount**: +ve int
    a. It is used to execute test method multiple times
    b. E.g @Test(invocationCount=4): Now myTestMethod will execute 4 times and 4 different tests will appear in report.

```
@Test(invocationCount=4)
public void myTestMethod() {
  System.out.println("Hello");
}
```

c. If invocationCount is set to 0, then test method will not execute even once.

5. invocationTimeOut: long
   a. It is used to set the maximum time a test should take to execute.
   b. If the specified time is crossed by test case, then it will be marked as fail in Report.
   c. If the test method is executed multiple times, then invocationTimeOut specifies cumulative time taken by all threads to execute it.
   d. It takes time in milliseconds.
   e. E.g. @Test(invocationTimeOut=4000):

```
@Test(invocationTimeOut=4000)
public void myTestMethod() {
   Thread.sleep(6000);
   System.out.println("Hello");
}
```

6. timeOut: long
   a. Max permitted time to run this test. Otherwise it will be marked as failed
   b. In this example myTestMethod will fail

```
@Test(timeOut=4000)
public void myTestMethod() {
   Thread.sleep(6000);
   System.out.println("Hello");
}
```

7. successPercentage: +Ve int
   a. It is used to specify success percentage of the test case.
   b. Lets say, we have set the successPercentage to 70 and our test method have 10 checks. 7 checks are pass whereas 3 checks are fail. Still we will consider our test case as PASS.
   c. E.g @Test(successPercentage=50). Below method will pass as 50% checks are passing.

```
@Test(successPercentage=50)
        public void myTestMethod() {
                System.out.println("Hello");
                Assert.assertEquals(1, 1);
                Assert.assertEquals(2, 2);
```

   d. Default value of successPercentage is 100 for all test methods even if you don't write.

8. expectedExceptions: List {A.class,B.class,C.class}
   a. It is used to specify list of exceptions expected from a test method.
   b. If any of the exceptions mentioned, is thrown by test method, then it will be considered as PASS.
   c. If none of the exceptions mentioned is thrown, then test method will be considered as FAIL.
   d. Below test will pass

```
@Test(expectedExceptions={ArithmeticExceptions.class})
        public void myTestMethod() {
                System.out.println("Hello");
```

9. singleThreaded: Boolean
    a. It is used to specify that this test method will be executed by only one thread.
    b. It should be used at class level only. It will be ignored if written at method level.

```
@Test(singleThreaded=true)
public class TestCases{
    public void myTestMethod() {
            System.out.println("Hello");
            throw new ArithmeticException();
```

10. threadPoolSize=+ int
    a. It is used to mention number of threads which will invoke this method.
    b. It should be written along with invocationCount.
    c. It is useless if not used with invocationCount.
    d. E.g. @Test(invocationCount=50): This test method will be executed by single thread for 50 times.
    e. E.g. @Test(invocationCount=50, threadPoolSize=5): This test method will be executed by 5 threads 50 times.

```
@Test(invocationCount=50, threadPoolSize=10)
public class TestCases{
    public void myTestMethod() {
            System.out.println("Hello");
```

11. dependsOnMethods=List of methods in String
    a. It is used to create dependency of my test method on another test method/s
    b. It takes list of names of methods separated by comma.
    c. E.g. @Test(dependsOnMethods={"A","B","Test.C"}): This test method is dependent on test methods, "A", "B" and "C" from class Test.
    d. If the method on which my test method is dependent, fails, then my test method will be marked as SKIPPED in testing Report.

```
public class A {
    @Test(dependsOnMethods= {"tc_01","tc_03"})
    public void tc_02() {
        System.out.println("Second Test Case");
    }
    @Test
    public void tc_01() {
        System.out.println("First Test Case");
        Assert.assertEquals(1, 2);
    }

    @Test
    public void tc_04() {
        System.out.println("Fourth Test Case");
```

12. Groups: List of logical groups in String
    a. It is used to divide test cases in logical groups.
    b. E.g. @Test(groups={"Smoke"}): My test method belongs to Smoke group.
    c. To execute test cases belonging to Smoke group, we should write following in testing.xml and execute the xml file.

```
<groups>
        <run>
                <include name="Smoke"/>
        </run>
</groups>
```

13. dependsOnGroups: List of groups in String separated by comma
    a. It is used to create dependency of my test method on entire group or groups.
    b. Group of test cases can be created using 'groups' parameter of @Test
    c. E.g. @Test(dependsOnGroups={"Smoke","Sanity"})

```
public class A {
        @Test(dependsOnGroups= {"Smoke"})
        public void tc_02() {
                System.out.println("Second Test Case");
        }
        @Test(groups="Smoke")
        public void tc_01() {
```