JUnit Annotations

Annotations were introduced in Junit4, which makes Java code more readable and simple. This is the big difference between Junit3 and Junit4 that Junit4 is annotation based.

With the knowledge of annotations in Junit4, one can easily learn and implement a JUnit test.

Below is the list of important and frequently used annotations:

1. @Before
2. @BeforeClass
3. @After
4. @AfterClass
5. @Test
6. @Ignore
7. @Test(timeout=500)
8. @Test(expected=IllegalArgumentException.class)

see below table to understand more on annotation :

S.No.	Annotations	Description
1.	@Test	This annotation is a replacement of org.junit.TestCase which indicates that public void method to which it is attached can be executed as a test Case.
2.	@Before	This annotation is used if you want to execute some statement such as preconditions before each test case.
3.	@BeforeClass	This annotation is used if you want to execute some statements before all the test cases for e.g. test connection must be executed before all the test cases.
4.	@After	This annotation can be used if you want to execute some statements after each Test Case for e.g resetting variables, deleting temporary files ,variables, etc.

5.	@AfterClass	This annotation can be used if you want to execute some statements after all test cases for e.g. Releasing resources after executing all test cases.
6.	@Ignores	This annotation can be used if you want to ignore some statements during test execution for e.g. disabling some test cases during test execution.
7.	@Test(timeout=500)	This annotation can be used if you want to set some timeout during test execution for e.g. if you are working under some SLA (Service level agreement), and tests need to be completed within some specified time.
8.	@Test(expected=IllegalArgumentException.class)	This annotation can be used if you want to handle some exception during test execution. For, e.g., if you want to check whether a particular method is throwing specified exception or not.

@Test: The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case. To run the method,

```
public class MyTestClass {
   @Test
   public void myTestMethod() {
      /**
       * Use Assert methods to call your methods to be tested.
       * A simple test to check whether the given list is empty or not.
       */
     org.junit.Assert.assertTrue( new ArrayList().isEmpty() );
   }
 }
```

@Test (expected = Exception.class): Sometimes we need to test the exception to be thrown by the test. @Test annotation provides a parameter called 'expected', declares that a test method should throw an exception.

```
public class MyTestClass {
   @Test(expected=IOException.class)
   public void myTestMethod() {
      /**
       * this test performs some IO operations, sometimes we may not
```

```
        * get access to the resources, then the method should through
        * declared exception.
        */
       ....
       ....
    }
 }
```

@Test(timeout=100): Somethimes we need to mesure the performance interms of time. The @Test annotations provides an optional parameter called 'timeout', which causes a test to fail if it takes longer than a specified amount of clock time (measured in milliseconds).

```
public class MyTestClass {
    @Test@Test(timeout=100)
    public void myTestMethod() {
        /**
        * The IO operation has to be done with in 100 milli seconds. If not,
        * the test should fail.
        */
       ....
       ....
    }
 }
```

@Before: When writing tests, it is common to find that several tests need similar objects created before they can run. Annotating a public void method with @Before causes that method to be run before the Test method. The @Before methods of super classes will be run before those of the current class.

```
public class MyTestClass {

    List<String> testList;

    @Before
    public void initialize() {
        testList = new ArrayList<String>();
    }

    @Test
    public void myTestMethod() {
        /**
        * Use Assert methods to call your methods to be tested.
        * A simple test to check whether the given list is empty or not.
        */
       org.junit.Assert.assertTrue( testList.isEmpty() );
    }
 }
```

@After: If you allocate external resources in a Before method you need to release them after the test runs. Annotating a public void method with @After causes that method to be run after the Test method. All @After

methods are guaranteed to run even if a Before or Test method throws an exception. The @After methods declared in superclasses will be run after those of the current class.

```
public class MyTestClass {

   OutputStream stream;

   @Before
   public void initialize() {
      /**
       * Open OutputStream, and use this stream for tests.
       */
      stream = new FileOutputStream(...);
   }

   @Test
   public void myTestMethod() {
      /**
       * Now use OutputStream object to perform tests
       */
     ...
     ...
   }


   @After
   public void closeOutputStream() {
        /**
         * Close output stream here
         */
        try{
          if(stream != null) stream.close();
        } catch(Exception ex){

        }
   }
 }
```

@BeforeClass: Sometimes several tests need to share computationally expensive setup (like logging into a database). While this can compromise the independence of tests, sometimes it is a necessary optimization. Annotating a public static void no-arg method with @BeforeClass causes it to be run once before any of the test methods in the class. The @BeforeClass methods of superclasses will be run before those the current class.

The annotations @BeforeClass and @Before are same in functionality. The only difference is the method annotated with @BeforeClass will be called once per test class based, and the method annotated with @Before will be called once per test based.

```
public class MyTestClass {

   @BeforeClass
   public void initGlobalResources() {
      /**
```

```
     * This method will be called only once per test class.
     */
    }

    @Before
    public void initializeResources() {
        /**
         * This method will be called before calling every test.
         */
    }

    @Test
    public void myTestMethod1() {
        /**
         * initializeResources() method will be called before calling this method
         */
    }

    @Test
    public void myTestMethod2() {
        /**
         * initializeResources() method will be called before calling this method
         */
    }
}
```

@AfterClass: If you allocate expensive external resources in a BeforeClass method you need to release them after all the tests in the class have run. Annotating a public static void method with @AfterClass causes that method to be run after all the tests in the class have been run. All @AfterClass methods are guaranteed to run even if a BeforeClass method throws an exception. The @AfterClass methods declared in superclasses will be run after those of the current class.

The annotations @AfterClass and @After are same in functionality. The only difference is the method annotated with @AfterClass will be called once per test class based, and the method annotated with @After will be called once per test based.

```
public class MyTestClass {

    @BeforeClass
    public void initGlobalResources() {
        /**
         * This method will be called only once per test class. It will be called
         * before executing test.
         */
    }

    @Test
    public void myTestMethod1() {
        // write your test code here...
        ...
        ...
    }
```

```
  @AfterClass
  public void closeGlobalResources() {
     /**
      * This method will be called only once per test class. It will be called
      * after executing test.
      */
  }
}
```

@Ignore: Sometimes you want to temporarily disable a test or a group of tests. Methods annotated with Test that are also annotated with @Ignore will not be executed as tests. Also, you can annotate a class containing test methods with @Ignore and none of the containing tests will be executed. Native JUnit 4 test runners should report the number of ignored tests along with the number of tests that ran and the number of tests that failed.

You can also use @Ignore annotation at class level.

```
public class MyTestClass {
  @Ignore
  @Test
  public void myTestMethod() {
     /**
      * This test will be ignored.
      */
    org.junit.Assert.assertTrue( new ArrayList().isEmpty() );
  }
}
```

1. Simple JUnit test using @Test annotation.
2. List of JUnit annotations.
3. Assertion method Assert.assertArrayEquals() example.
4. How to do JUnit test for comapring two list of user defined objects?
5. Assertion method Assert.assertEquals() example.
6. Assertion method Assert.assertFalse() example.
7. Assertion method Assert.assertTrue() example.
8. Assertion method Assert.assertNotNull() example.
9. Assertion method Assert.assertNull() example.
10. Assertion method Assert.assertNotSame() example.
11. Assertion method Assert.assertSame() example.