<http://javarevisited.blogspot.gr/2012/06/junit4-annotations-test-examples-and.html>

JUnit4 Annotations : Test Examples and Tutorial

**JUnit4** **Annotations** are single big change from JUnit 3 to JUnit 4 which is introduced in Java 5. With annotations creating and running a JUnit test becomes more easy and more readable, but you can only take full advantage of JUnit4 if you know the **correct meaning of  annotations** used on this version and how to use them while writing tests. In this

**Junit tutorial** we will not only understand meaning of those annotations but also we will see examples of JUnit4 annotations. By the way this is my first post in unit testing but if you are new here than you may like post 1[0 tips to write better code comments](http://javarevisited.blogspot.sg/2011/08/code-comments-java-best-practices.html) and [10 Object oriented design principles for Programmer](http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html) as well.

JUnit 4 Annotations : Overview

Following is a list of ***frequently used Annotation***s , which is available when you include junit4.jar in your Classpath:

@Before  
@BeforeClass  
@After  
@AfterClass  
@Test  
@Ignore  
@Test(timeout=500)  
@Test(expected=**IllegalArgumentException**.**class**)

**@Before and @After**

In Junit4 there is no setup() or tearDown() method and instead of that we have @Before and @After annotations.

By using @Before you can make any method as setup() and by using @After you can make any method as teardown(). What is most important point to remember is ***@Before and @After annotated method will be invoked before and after each test case***. So in case you have five test cases in your JUnit test file than just like setup() and tearDown() method annotated with @Before and @After will be called five times. Here is an example of using

@Before and @After Annotation :

    @Before  
    **public** **void** setUp() {  
        **System**.out.println("@Before method will execute before every JUnit4 test");  
    }  
    
    @After  
    **public** **void** tearDown() {  
        **System**.out.println("@After method will execute after every JUnit4 test");  
    }

**@BeforeClass and @AfterClass**

@BeforeClass and @AfterClass JUnit4 Annotations are similar to @After and @Before with only exception that they

are *called on per TestClass basis and not on per test basis*. They can be used as one time setup and tearDown

method and can be used to [initialize class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) level resources. here is an example of using @BeforeClass and @AfterClass Annotations in JUnit4, here is an **example of @BeforeClass and @AfterClass Junit 4 annotation**

    @BeforeClass  
    **public** **static** **void** setUpClass() **throws** **Exception** {  
        **System**.out.println("@BeforeClass method will be executed before JUnit test for"  
                + "a Class starts");  
    }  
  
    @AfterClass  
    **public** **static** **void** tearDownClass() **throws** **Exception** {  
         **System**.out.println("@AfterClass method will be executed after JUnit test for"  
                + "a Class Completed");  
    }

**@Test**

@Test is a replacement of both TestCase class and convention "test" which we prefix to every test method. for example to test a method  called calculateInterest() we used to create method testCalcuatedInterest() and our class needs to be extended from **org.junit.TestCase** class. Now with @Test annotation that is not required any more. You just need to annotate your test method with @Test Junit4 annotation and done. no need to extend from TestCase class and no need to prefix "test" to your method, here is an **example of  JUnit 4 @Test annotation**

 @Test  
    **public** **void** testCalculateInterest() {  
        **System**.out.println("calculateInterest");  
        fail("An Example of @Test JUnit4 annotation");  
    }

**@Ignore**

Some time we add test method in JUnit test class but hasn't implemented that is causing your [build](http://javarevisited.blogspot.sg/2010/10/ant-basics.html) to fail if JUnit testcase are integrated or embedded into build process. you can avoid that problem by marking your test method as @Ignore in Junit4. *JUnit4 ignores method annotated with @Ignore* and doesn't run during test. Here is an **example of using @Ignore annotation in JUnit4** to exclude a particular Test from running:

 @Ignore("Not yet implemented")  
    @Test  
    **public** **void** testGetAmount() {  
        **System**.out.println("getAmount");  
        fail("@Ignore method will not run by JUnit4");  
    }

**@Test(timeout=500)**

Now with JUnit4 writing testcases based on timeout is extremely easy. You just need to pass a parameter timeout with value in millisecond to @Test annotation. remember timeout values are specified in millisecond and your JUnit4 timeout test case will help if it doesn't complete before timeout period. This works great if you have SLA(Service Level Agreement)  and an operation need to complete before predefined timeout.

  @Test(timeout = 500)  
    **public** **void** testTimeout() {  
        **System**.out.println("@Test(timeout) can be used to enforce timeout in JUnit4 test case");  
        **while** (1 == 1) {  
            
        }  
    }

This JUnit4 test will fail after 500 millisecond.

**@Test(expected=IllegalArgumentException.class)**

Another useful enhancement is Exception handling testcases of JUnit4. Now to test [Exception](http://javarevisited.blogspot.com/2011/12/checked-vs-unchecked-exception-in-java.html) is become very easy and you just need to specify Exception class inside @Test annotation to check whether a method throws a particular exception or not. here is an example which test behaviour of a method to verify whether it [throws](http://javarevisited.blogspot.com/2012/02/difference-between-throw-and-throws-in.html) Exception or not,  when run with invalid input:

    @Test(expected=**IllegalArgumentException**.**class**)  
    **public** **void** testException(**int** input) {  
        **System**.out.println("@Test(expected) will check for specified exception during its run");  
        
    }

These were **list of frequently used JUnit 4 annotations** and there meanings. In the course we have also learn how to use @Before , @After in place of setup() and teardown(). [Code review](http://javarevisited.blogspot.sg/2011/09/code-review-checklist-best-practice.html) and Unit testing is one of the best development practices to follow and we must try our best to incorporate that in our daily coding and development cycle.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<http://javarevisited.blogspot.com/2013/03/how-to-write-unit-test-in-java-eclipse-netbeans-example-run.html>

Steps to Create JUnit Test in Eclipse and Netbeans IDE

Writing Junit tests for Java classes in Eclipse and Netbeans IDE are super easy, and I will show you with that later in this JUnit tutorial. Before that, let’s revise what is unit test and why should you write them. Unit test is to test smaller unit of code, e.g. methods. Writing unit test to test individual unit of code is one of the best development practice and helps to find bug earlier in development cycle. Though there are other unit testing framework available in Java e.g. TestNG, JUnit has it’s own place among Java developers. IMHO [code review](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html) and unit testing are two most important practices for improving code quality and should always be followed during software development. Sad thing is that not every developer follows it; some programmer don’t write unit test due to ignorance and others due to laziness. Any way, it’s just start which take time, once you start writing unit tests, you will automatically start enjoying it. I have seen Java developers testing there code with [main() method](http://javarevisited.blogspot.com/2011/12/main-public-static-java-void-method-why.html), but now they prefer to test them with JUnit testcases. I agree few initial tests are difficult because of knowledge and inertia and best way to approach is to start with simplest of JUnit tests. In this **JUnit tutorial**, I will show you *how to write and execute JUnit test from Eclipse and Netbeans*, two popular Java IDE. By the way, if you are looking for any good book on JUnit and unit testing, you should look Pragmatic Unit Testing in Java with JUnit, it's an amazing book and teaches a lot about both JUnit and unit testing in Java.

## JUnit 3 and JUnit 4 testing framework

JUnit frameworks is popular from quite a sometime now and there are two popular versions available in form of JUnit 3.8, known as JUnit 3 and JUnit 4. Working of both versions are same, as you create testcases, testsuite and execute them. Main difference between JUnit 4 and JUnit 3 is that, JUnit4 is based on annotation feature of Java 1.5 and easy to write, while JUnit 3 uses “test” keyword, to identify test methods. What is bonus in terms of writing JUnit tests for Java program is that, two of most popular Java IDE, [Eclipse](http://javarevisited.blogspot.com/2013/02/must-override-superclass-method-java-eclipse.html) and [Netbeans](http://javarevisited.blogspot.com/2012/03/how-to-create-and-execute-jar-file-in.html) has inbuilt support and provides easy interface and infrastructure to create and execute JUnit tests for Java classes. In this JUnit 4 tutorial, we will see step by step guide for **writing JUnit test in Eclipse and Netbeans** with simple example of Calculator. Our Calculator class has methods add() and multiply() for addition and multiplication, I have used variable arguments of Java 1.5 to implement these methods, so that they can accept any number of parameter.

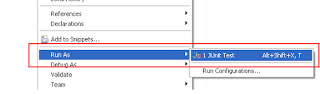
## How to write JUnit tests in Eclipse

1. Create a New Java Project called JUnitExample.

2. Create a [Java class](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html) Calculator in project which should have add() and multiply() method.

3. Right click on Java class and click on create Junit testcase

## [How to write JUnit test case in Eclipse JAva](http://4.bp.blogspot.com/-c0RIkLQC__s/UTC-M9ilt0I/AAAAAAAAAkE/4flRpMqu6lc/s1600/JUnit+Test+Eclipse+Java+Example.PNG)How to execute JUnit tests in Eclipse

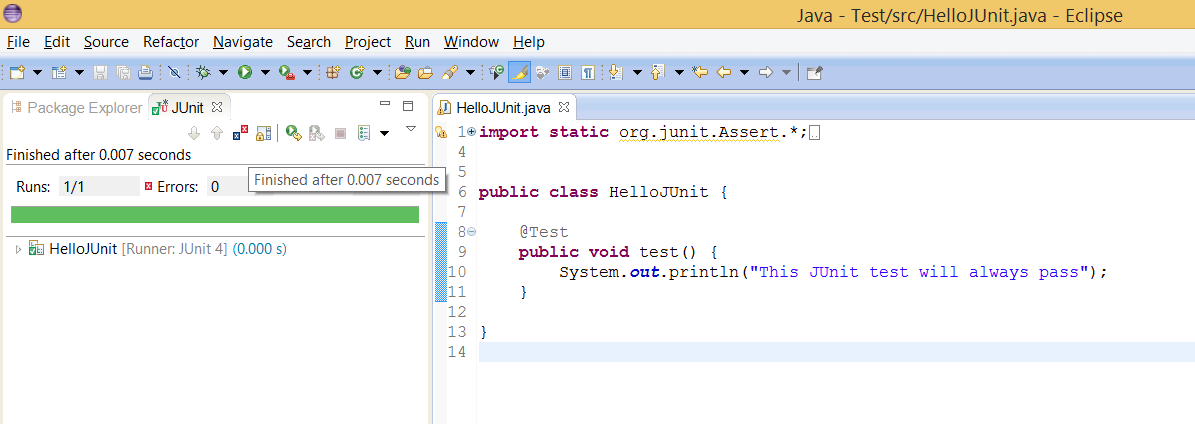
[](http://1.bp.blogspot.com/-Up4OpHNSq9I/UTC-QODXIjI/AAAAAAAAAkM/QIWUiAv9SCg/s1600/how+to+run+JUnit+test+in+Eclipse+Java.PNG)Now you have your Junit test created, you can execute it just like you run any Java application:

Right Click --> Run As --> Junit Test

This will run all the *JUnit tests* declared in this class and will pass if all the test run successfully and pass the condition tested by various [assert statement](http://javarevisited.blogspot.com/2012/01/what-is-assertion-in-java-java.html) and fail if any of JUnit tests failed. Eclipse will print stack trace and hyper link to the failed test and you can go and fix the problem.

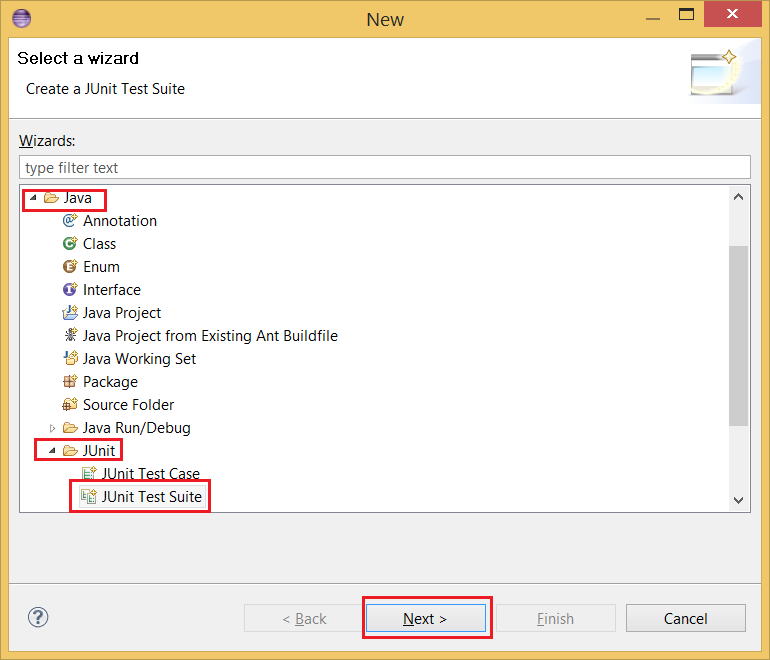
### Why are my JUnit results not showing up?

Some time when your all JUnit tests passed you may not see any result, because of an option called "Show Failures Only". For more satisfying results, go to the downward-pointing triangle at the right end of the JUnit pane, and make sure Show Failures Only is not checked, as shown in following screenshot. Some time your JUnit test will not succeed i.e. it will fail or run into error. In JUnit, there is difference between a "failure" and an "error"? A failure is when one of your assertions fails--that is, your program does something wrong, and your JUnit test notices and reports the fact. An error is when some other Exception occurs due to programming mistakes, such as a NullPointerException or an ArrayIndexOutOfBoundsException. Both error and failure are not good for your code. A good Junit test case should be able to bring failure and error a like. You should also ensure that your JUnit testcase should pass always and doesn't throw any error or failures.

[](http://1.bp.blogspot.com/-xjAttuhAbNo/U2H_XQzeaeI/AAAAAAAABd0/w8t7b8qNyso/s1600/How+to+See+JUnit+Result+in+Eclipse.png)

### How to create JUnit Test suit in Eclipse

Like individual unit tests, you can also create a test suite for creating tests for more than one classes in Java. In order to create JUnit test suite in Eclipse  Go to File → New → Other... → Java → JUnit → TestSuite, and click Next>. Select all the classes, and click Finish. Once created, You can run this test suite the same way you run other JUnit tests. Result of JUnit test suite will also show in JUnit console like previous run.

[](http://1.bp.blogspot.com/-R5zRrJiXZII/U2H_XYGwL-I/AAAAAAAABd4/1pDGY9Z5rZw/s1600/How+to+create+JUnit+Test+Suite+in+Eclipse+IDE.png)

## How to write JUnit tests in Netbeans

Junit support in Netbeans is also great and seamless. Here is the steps to create JUnit test in Netbeans

1. Create a New Java Project called JUnitExample.

2. Create a Java Class Calculator in project which should have add() and multiply() method.

3. Now Select a Java Class --> Right click --> Tools --> Create Junit tests

this will create Junit test class for all the methods of selected Java class.

## How to execute Junit tests in Netbeans

Executing JUnit tests in Netbeans is much simpler than it was in Eclipse. Go to your Junit test class and right click and select run File option. This will execute all the JUnit tests on File and show the result in [console](http://javarevisited.blogspot.com/2012/10/java-program-to-get-input-from-user.html). As earlier test will be pass if all test method passes otherwise it will fail. Netbeans also shows complete stack trace and hyperlink of failed test cases.

**Code**

Here is complete code example of, *How to write unit test in Java using JUnit framework*. In this example, we have a Java class called Calculator, which has two methods add() and multiply() and takes variable arguments. In this JUnit tutorial, we will write JUnit testcases, to test these two methods.

/\*\*

\* Simple Java Calculator with add and multiply method

\*/

public class ***Calculator*** {

public int **add**(int... *number*) {

int total **=** *0*;

**for** (int i **:** number) {

total **+=** i;

}

**return** total;

}

public int **multiply**(int... *number*) {

int product **=** *0*;

**for** (int i **:** number) {

product **\*=** i;

}

**return** product;

}

}

Following class CalculatorTest is our JUnit test class, it contains two methods testAdd() and testMultiply(). Since we are using JUnit 4, we don’t need to use prefix test, but I have used that to make test methods explicit. @Test annotation is used by JUnit 4 framework to identify all test cases. If you are new to JUnit 4, then see this post to learn more about [JUnit 4 annotations](http://www.blogger.com/javarevisited.blogspot.com/2012/06/junit4-annotations-test-examples-and.html). By the way when we run this class as JUnit Test, it will show how many test cases pass and how many failed. If all the test cases pass then it will show green bar, otherwise red bar to indicate failure.

**import static** org.junit.Assert.\*;

**import** org.junit.Test;

/\*\*

\* JUnit Test class for testing methods of Calculator class.

\*/

public class ***CalculatorTest*** {

@Test

public void **testAdd**() {

Calculator calc **=** **new** Calculator();

assertEquals(*60*, calc**.**add(*10*,*20*,*30*));

}

@Test

public void **testMultiply**() {

Calculator calc **=** **new** Calculator();

assertEquals(*6000*, calc**.multiply**(*10*,*20*,*30*));

}

}

That’s all on **How to write unit test in Java using JUnit framework in Eclipse and Netbeans**. JUnit testing framework is is very useful for writing and executing unit test for Java classes and it can also be integrated into build system by using [ANT and Maven](http://www.blogger.com/javarevisited.blogspot.sg/2011/08/increase-heap-size-maven-ant.html), which means you can all your tests automatically at build time. By the way since many project have there JUnit test running during build process, it’s good to keep unit test short and quickly executable to avoid lengthy build. See [JUnit best practices](http://javarevisited.blogspot.com/2012/08/best-practices-to-write-junit-test.html) guide more unit testing practices. Also start using Junit 4 annotations, they have made job of writing unit tests much easier.   
  
**Recommended Book: Pragmatic Unit Testing in Java with JUnit**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*