

# JUnit 4 Tutorial

---

## Objective

After this tutorial you should be able to write unit tests for your own project using the JUnit 4 library.

## Getting Started

### Download demo files:

You should download calculator.zip from the assignment page.

### Import the project:

- Run Eclipse/RTC.
- Select Import from the File menu.
- In the Import dialogue box select “Existing Project into Workspace...” under the General folder and then press Next.
- In Import Projects select “Select archive file” and browse for calculator.zip, then press Finish.
- Switch to the “Java” perspective using the icon at the top-right of the application.

There are total 5 java files located in two packages:

<b>Package</b>	<b>Filename</b>	<b>Description</b>
Calc	Calculator	A simple buggy calculator class that only handles integer operands. The source code is quite self-explanatory.
Test	BasicTests	Basic test cases to demonstrate the simple test classes: assert methods, fixtures such as @before, @test, and @Ignore..
	ParameterizedTest	Demonstrates a new feature in JUnit 4: parameterized test.
	AdvancedTests	Demonstrates advanced features such as timeout tests and advanced fixtures in JUnit 4
	TestSuite	A test suite of all test cases.

If your test cases have compile errors, make sure that you use a JDK version 1.6 or higher. To do so, right click onto the project, select “Properties > Java Compiler”, check “Enable project specific settings”, select 1.6 as the Compiler compliance level (if 1.6 is not available, you will have to install a JDK 1.6 on your computer first) and press “Ok”. After a rebuilt of the code (“Project > Clean...”), there should be no compile errors left.

## Basics of JUnit 4 testing

The test case file must import the necessary classes from *org.junit* and the classes that will be tested:

For example,

```
import static org.junit.Assert.assertEquals;

import org.junit.After;

import org.junit.Before;

import org.junit.Ignore;

import org.junit.Test;


import calc.Calculator; // Class being tested
```

With JUnit 4, you use annotations to indicate which methods are for setup/teardown and which methods are test cases. There are three basic annotations in a JUnit 4 test class: `@Before`, `@After` and `@Test`. Methods annotated with `@Before` will be used for setup (eg, initializing variables) methods annotated with `@After` will be used for teardown (eg. releasing any permanent resources allocated in methods annotated with `@Before`). A JUnit 4 class needs at least one `@Test` annotated method; otherwise, you will get a runtime error: *java.lang.Exception: No runnable methods*. Note: a test method must return void and have no parameters!

All the methods annotated with `@Before` will be run before each test case, and all the methods annotated with `@After` will be run after each test case.

## Assertions

In each test case, you need to check that the method you are testing behaves as expected. Often times, this involves checking a return value. You can check that the return value is what you expect with an assert. The most commonly used assertion methods are *assertEquals()*, *assertTrue()* and *assertFalse()*.

For example, if you were testing a method *getPrice()* and you expected the return value to be 4.4, you could use *assertEquals(obj.getPrice(), 4.4)* in your test method.

For details, please refer to the JUnit 4.0 API: [http://junit.sourceforge.net/javadoc\\_40/index.html](http://junit.sourceforge.net/javadoc_40/index.html)

## Ignoring Tests

Adding the `@Ignore` annotation in front or after `@Test` will ignore the test. This is a useful feature if there are some unimplemented features which have test cases available.

```
@Ignore("not implemented yet")

@Test

public void testMultiply() {

    calculator.add(10);

    calculator.multiply(10);

    assertEquals(calculator.getResult(), 100);

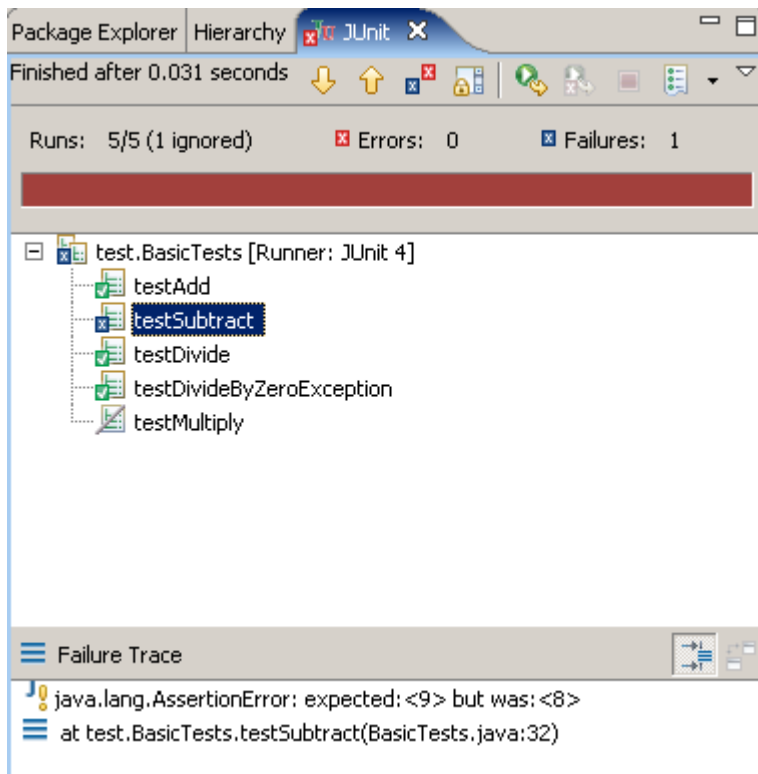
}
```

## Running the sample tests

Now you can run the BasicTests by right clicking and selecting Run As → JUnit Test.



JUnit will run the test, and report the test results. Here we have three passed tests, one failure and one ignored test.



## Advanced JUnit 4 test techniques:

JUnit 4 offers many new advanced features compared to the older versions. In this tutorial, we will examine the following features: advanced fixtures, parameterized tests, and timeout tests.

### Advanced fixtures

Advanced fixtures (A *fixture* is the set of objects that a test case uses) such as `@BeforeClass` and `@AfterClass` can be useful for some test cases. The differences between `@BeforeClass/@AfterClass` and `@Before/@After` are listed in the following table:

<b><code>@BeforeClass/@AfterClass</code></b>	<b><code>@Before/@After</code></b>
Only one method per class can be annotated.	Multiple methods can be annotated. Order of execution is unspecified. Overridden methods are not run.
Method names are irrelevant	Method names are irrelevant
Runs once per class	Runs before/after each test method
<code>@BeforeClass</code> methods of superclasses will be run before those of the current class. <code>@AfterClass</code> methods declared in superclasses will be run after those of the current class.	<code>@Before</code> in superclasses are run before those in subclasses. <code>@After</code> in superclasses are run after those in subclasses.
Must be public and static.	Must be public and non static.
All <code>@AfterClass</code> methods are guaranteed to run even if a <code>@BeforeClass</code> method throws an exception.	All <code>@After</code> methods are guaranteed to run even if a <code>@Before</code> or <code>@Test</code> method throws an exception.

`@BeforeClass` and `@AfterClass` can be very useful if you need to allocate and release expensive resources only once.

### Parameterized test cases

For boundary test cases, we often need to repeat the same test with different parameters. In older versions of JUnit, this was done by copying and pasting test methods. Parameterized test cases (introduced in JUnit 4) can handle this scenario in a much simpler way. A Parameterized test case uses two new annotations: annotation `@RunWith` specifies which runner class to invoke and annotation `@Parameters` tells the runner which parameters to use:

```
@RunWith(Parameterized.class)
```

...

```

@Parameters

public static List param() {

    return Arrays.asList(new Object[][] {

        ... ..

    })
}

```

A public constructor which accepts these parameters is also required:

```

public ParameterizedTest(int param, int result) {

    this.param = param;

    this.result = result;

}

```

See ParameterizedTest.java in the Calculator package for an example.

## Testing for exceptions

Sometime we need to test whether the program throws proper exceptions. We can use the first optional parameter of `@Test` to test for exceptions:

```

@Test(expected = ArithmeticException.class)

public void testDivideByZeroException() {

    calculator.divide(0);

}

```

## Testing for infinite loops

Some bugs can trap the program in an infinite loop. JUnit 4 introduces timeout as the second optional parameter of the `@Test` annotation to handle these cases:

```

@Test(timeout = 1000)

public void squareRoot() {

    calculator.squareRoot(2);

}

```

## Test Suite

A test suite is a simple way of running multiple test cases at one time. In JUnit 4, annotating a class with `@RunWith` and `@Suite.SuiteClasses` will change this class to a test suite:

```
@RunWith(Suite.class)

@Suite.SuiteClasses({

    AdvancedTests.class,

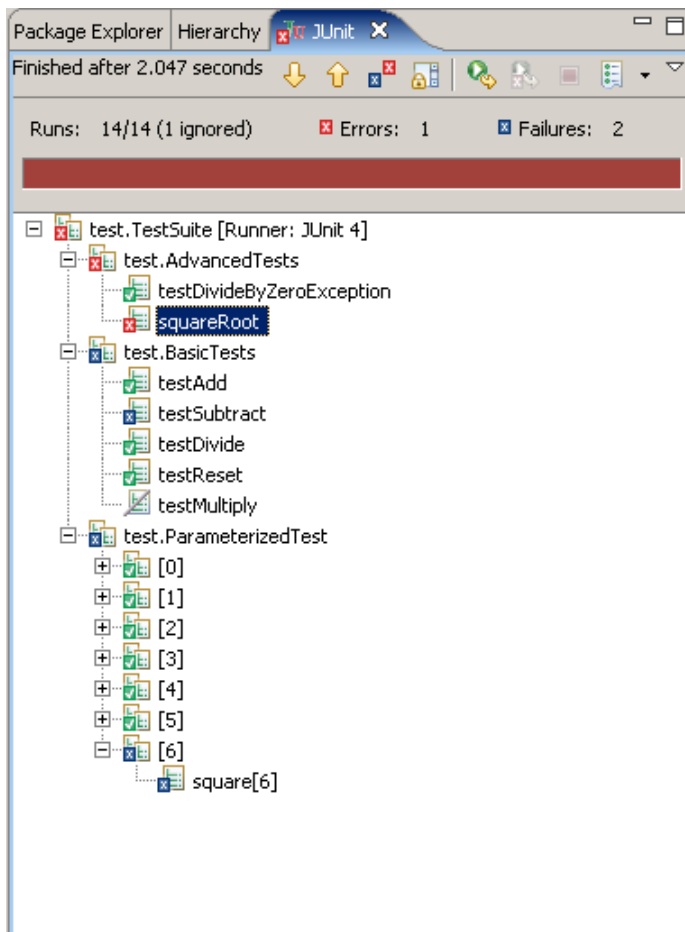
    BasicTests.class,

    ParameterizedTest.class

})
```

To include more test cases/classes, just add them as parameters of `Suite.SuiteClasses`.

Running the test suite will show the results of all test cases:



See `TestSuite.java` in the Calculator package for an example.

## Tasks (for Assignment 1)

1. Find and fix one bug in the source code (calculator.java) and one bug in the test cases. (This is a warm up exercise, so you needn't fix all the bugs in the source code. Ignore the features not implemented)
2. Write a parameterized test for the multiply method in the Calculator class.