JUnit 4.0 in 10 minutes

Gunjan Doshi
Instrumental Services Inc

Abstract: JUnit needs no introduction. Originally written by Kent Beck and Erich Gamma, the software is the preferred tool of choice for developer testing. Now, the team of Kent Beck and Erich Gamma is back again with a new version of JUnit – 4.0. This quick reference guide is for programmers and testers looking to migrate to JUnit 4.0. If you have a flight to catch or do not want to spend 10 minutes going through the guide, just jump to the summary section and you will learn enough.

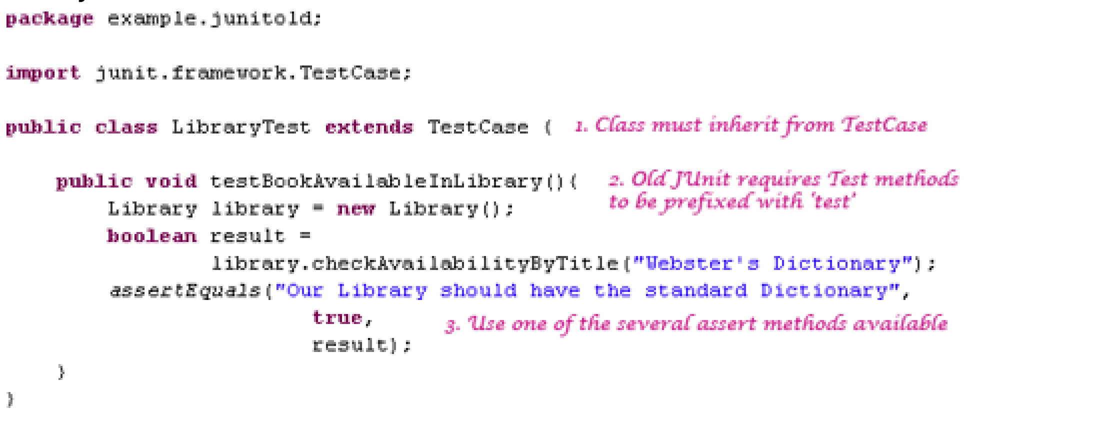
For the purpose of this article, I will call JUnit 3.8.1 and its predecessors as the old JUnit and JUnit 4.0 as the new JUnit.

Table of contents:

This guide contains the following sections:

Old JUnit revisited

Using the old JUnit, let us write a test, which verifies the availability of a book in the library.



```
package example.junitold;

import junit.framework.TestCase;

public class LibraryTest extends TestCase {

    public void testBookAvailableInLibrary() {
        Library library = new Library();
        boolean result =
                library.checkAvailabilityByTitle("Webster's Dictionary");
        assertEquals("Our Library should have the standard Dictionary",
                        true,
                        result);
    }
}
```

1. Class must inherit from TestCase

2. Old JUnit requires Test methods to be prefixed with 'test'

3. Use one of the several assert methods available

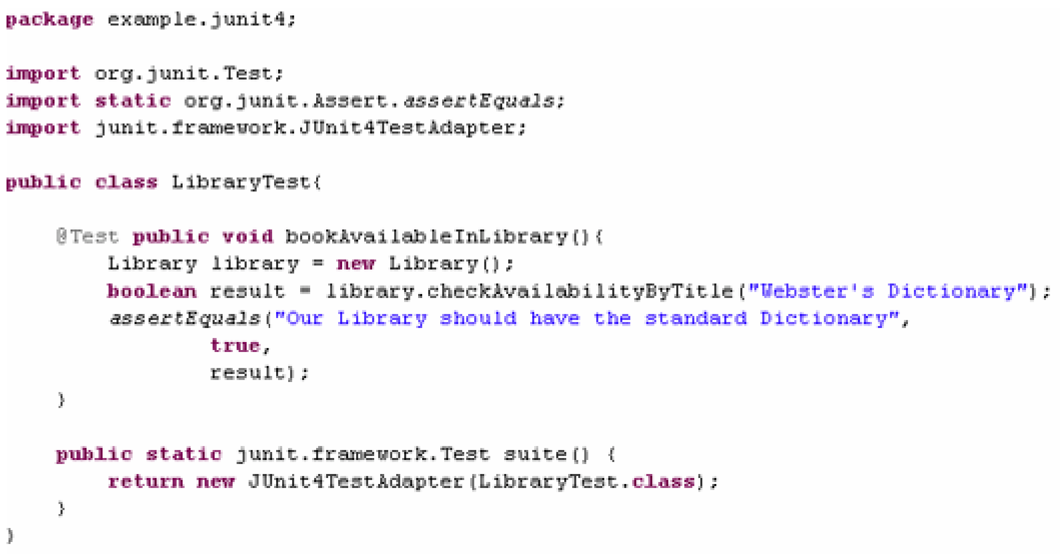
To summarize the steps:

- We extend from junit.framework.TestCase.
- We name the test methods with a prefix of ‘test’.

- We validate conditions using one of the several assert methods.

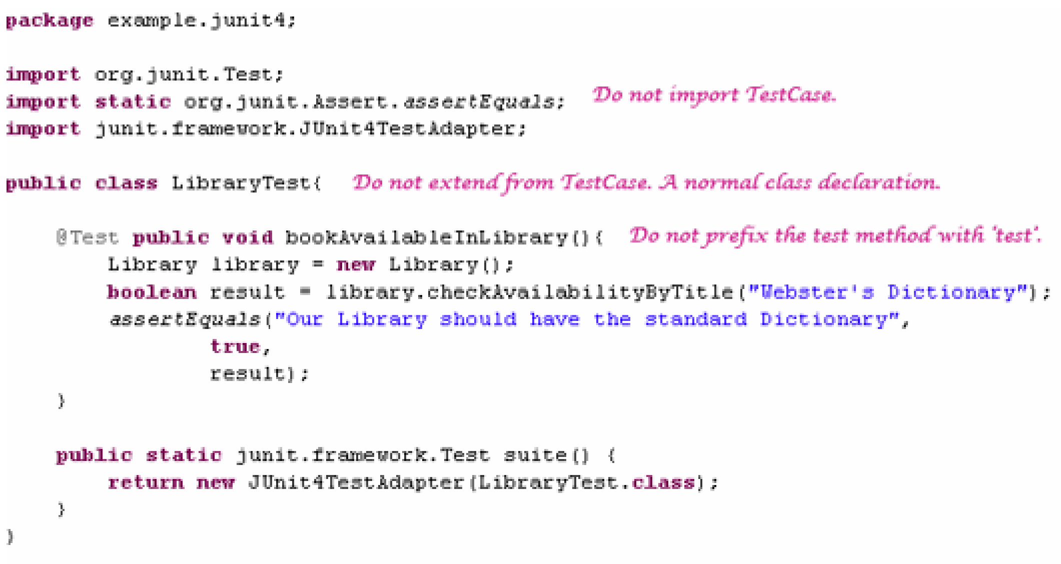
Cut the chase to JUnit 4.0

Let us write the same test using JUnit 4.0.



```java
package example.junit4;

import org.junit.Test;
import static org.junit.Assert.assertEquals;
import junit.framework.JUnit4TestAdapter;

public class LibraryTest{

    @Test public void bookAvailableInLibrary(){
        Library library = new Library();
        boolean result = library.checkAvailabilityByTitle("Webster's Dictionary");
        assertEquals("Our Library should have the standard Dictionary",
                true,
                result);
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(LibraryTest.class);
    }
}
```

When I upgrade to a new version I look for tasks, I do not have to do anymore. Here is the same code with notes telling us what not to do anymore.

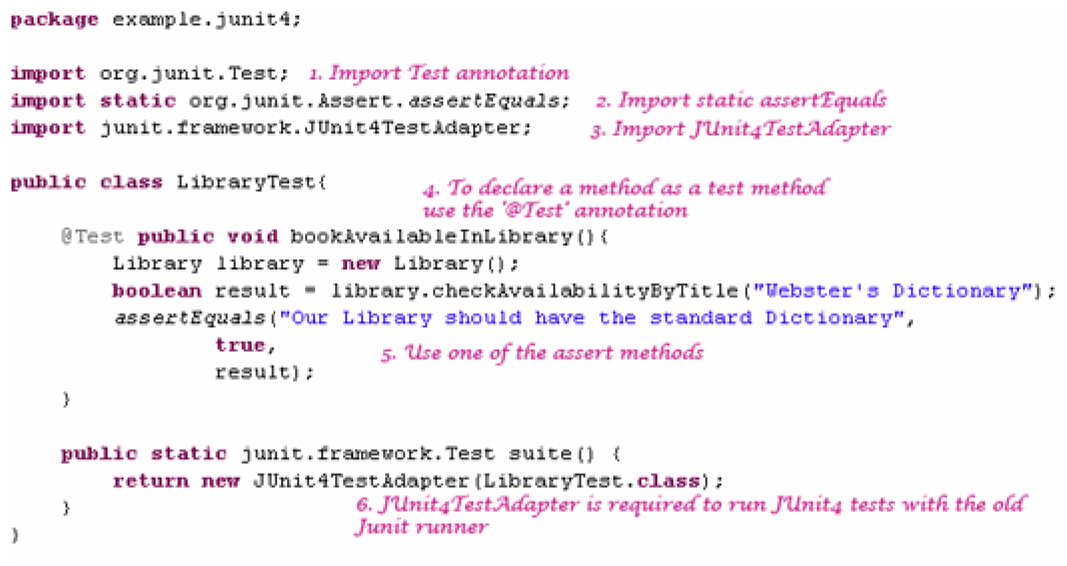


```java
package example.junit4;

import org.junit.Test;
import static org.junit.Assert.assertEquals;   Do not import TestCase.
import junit.framework.JUnit4TestAdapter;

public class LibraryTest{   Do not extend from TestCase. A normal class declaration.

    @Test public void bookAvailableInLibrary(){   Do not prefix the test method with 'test'.
        Library library = new Library();
        boolean result = library.checkAvailabilityByTitle("Webster's Dictionary");
        assertEquals("Our Library should have the standard Dictionary",
                true,
                result);
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(LibraryTest.class);
    }
}
```

To summarize:

- We do not extend from junit.framework.TestCase.
- We do not prefix the test method with 'test'.

Next, I look for new tasks I must always do. The diagram below summarizes what we must do according to the new JUnit standards:


1. Import Test annotation
2. Import static assertEquals
3. Import JUnit4TestAdapter
4. To declare a method as a test method use the '@Test' annotation
5. Use one of the assert methods
6. JUnit4TestAdapter is required to run JUnit4 tests with the old Junit runner

```
package example.junit4;

import org.junit.Test;
import static org.junit.Assert.assertEquals;
import junit.framework.JUnit4TestAdapter;

public class LibraryTest{

    @Test public void bookAvailableInLibrary(){
        Library library = new Library();
        boolean result = library.checkAvailabilityByTitle("Webster's Dictionary");
        assertEquals("Our Library should have the standard Dictionary",
                true,
                result);
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(LibraryTest.class);
    }
}
```

To summarize:

- Use a normal class and not extend from junit.framework.TestCase.
- Use the Test annotation to mark a method as a test method. To use the Test annotation, we need to import org.junit.Test
- Use one of the assert methods. There is no difference between the old assert methods and the new assert methods. An easy way to use the assert method is to do a static import as shown by point 2 in the code above.
- Run the test using JUnit4TestAdapter. If you want to learn more about JUnit4TestAdapter, keep reading ahead.

Run the tests

Unfortunately, our favorite development environments are still unaware of JUnit 4. JUnit4Adapter enables compatibility with the old runners so that the new JUnit 4 tests can be run with the old runners. The suite method in the diagram above illustrates the use of JUnit4Adapter.

Alternatively, you can use the JUnitCore class in the org.junit.runner package. JUnit 4 runner can also run tests written using the old JUnit. To run the tests using the JUnitCore class via the command line, type:

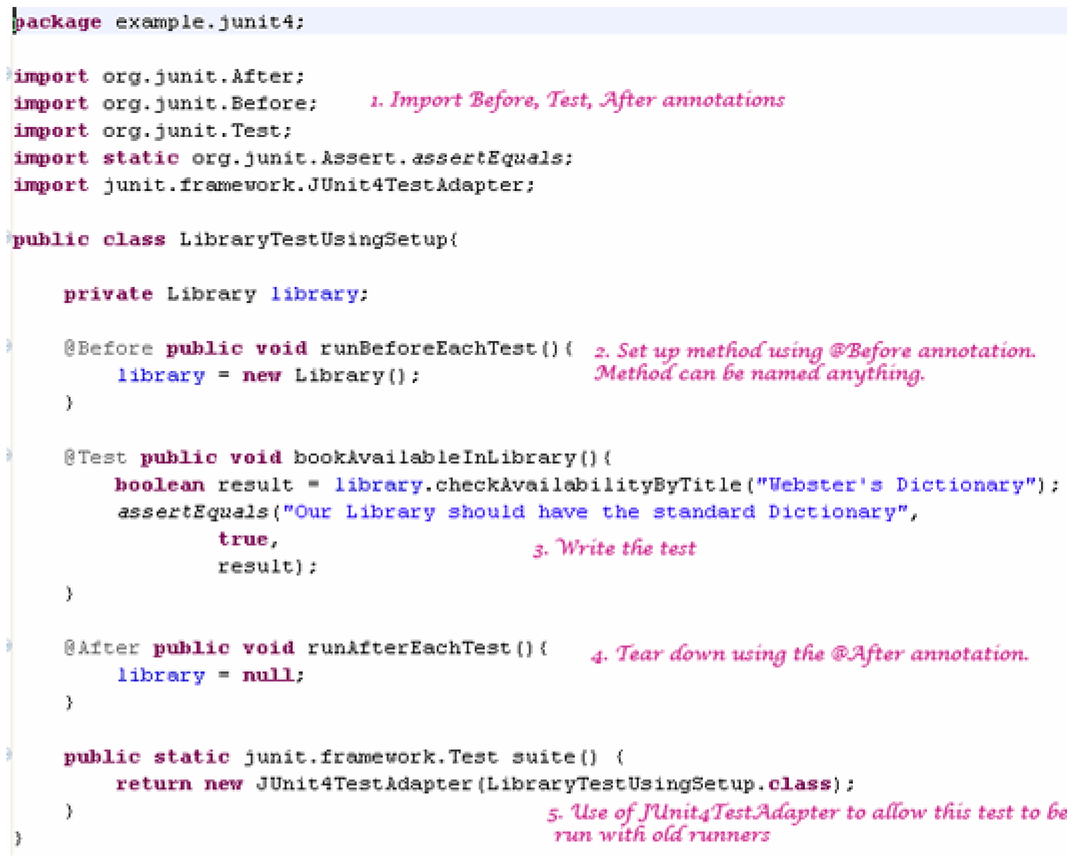
```
java org.junit.runner.JUnitCore LibraryTest
```

Set up and tear down

The new JUnit provides two new annotations for set up and tear down:

- @Before: Method annotated with @Before executes before every test.
- @After: Method annotated with @After executes after every test.

Here is the code that demonstrates the use of @Before and @After:



```
package example.junit4;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import junit.framework.JUnit4TestAdapter;

public class LibraryTestUsingSetup{

    private Library library;

    @Before public void runBeforeEachTest(){
        library = new Library();
    }

    @Test public void bookAvailableInLibrary(){
        boolean result = library.checkAvailabilityByTitle("Webster's Dictionary");
        assertEquals("Our Library should have the standard Dictionary",
                true,
                result);
    }

    @After public void runAfterEachTest(){
        library = null;
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(LibraryTestUsingSetup.class);
    }
}
```

1. Import Before, Test, After annotations
2. Set up method using @Before annotation. Method can be named anything.
3. Write the test
4. Tear down using the @After annotation.
5. Use of JUnit4TestAdapter to allow this test to be run with old runners

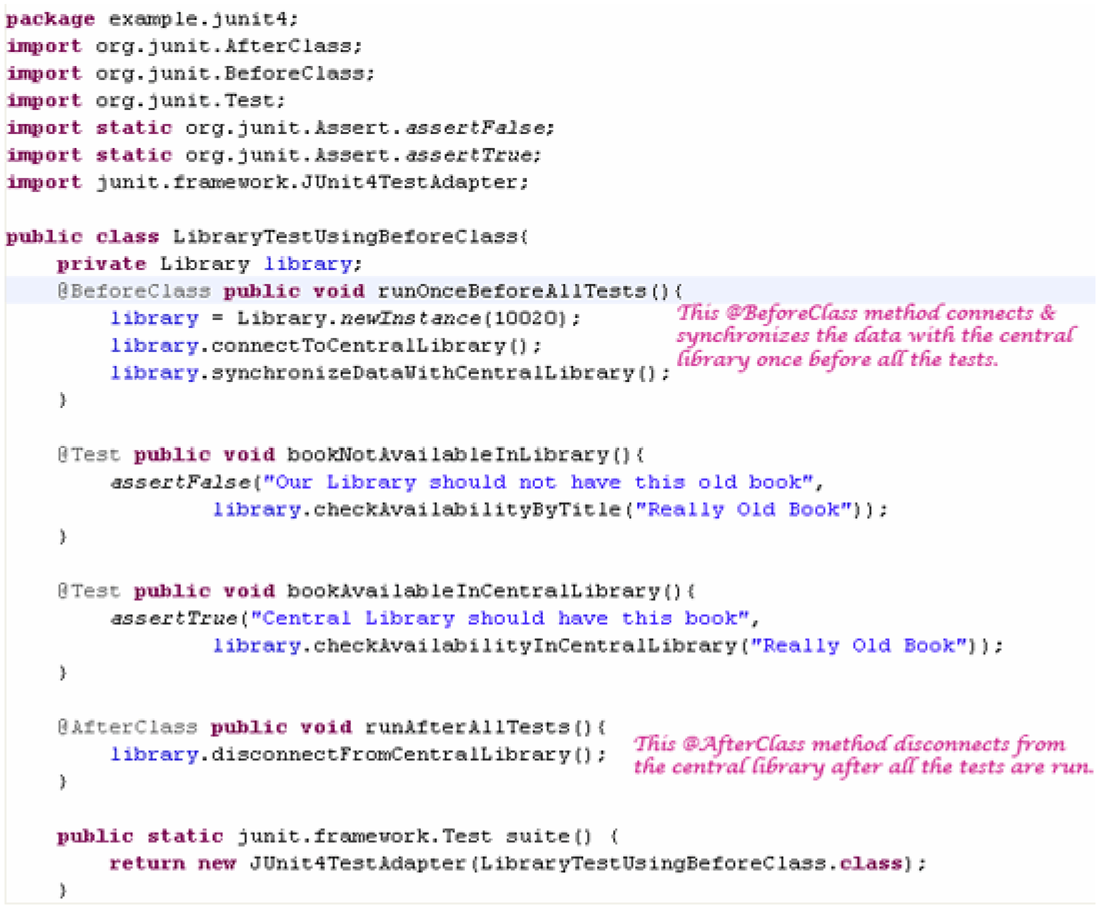
Two features of @Before and @After annotations that are helpful to learn:

- You can have any number of @Before and @After as you need.
- It is possible to inherit the @Before and @After methods. New JUnit executes @Before methods in superclass before the inherited @Before methods. @After methods in subclasses are executed before the inherited @After methods.

One-time set up and tear down

The new JUnit4 provides @BeforeClass and @AfterClass annotations for one-time set up and tear down. This is similar to the TestSetup class in the old junit.extensions package, which ran setup code once before all the tests and cleanup code once after all the tests.

Here is the code that demonstrates @BeforeClass and @AfterClass:



```java
package example.junit4;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import junit.framework.JUnit4TestAdapter;

public class LibraryTestUsingBeforeClass{
    private Library library;
    @BeforeClass public void runOnceBeforeAllTests(){
        library = Library.newInstance(10020);
        library.connectToCentralLibrary();
        library.synchronizeDataWithCentralLibrary();
    }

    @Test public void bookNotAvailableInLibrary(){
        assertFalse("Our Library should not have this old book",
                library.checkAvailabilityByTitle("Really Old Book"));
    }

    @Test public void bookAvailableInCentralLibrary(){
        assertTrue("Central Library should have this book",
                library.checkAvailabilityInCentralLibrary("Really Old Book"));
    }

    @AfterClass public void runAfterAllTests(){
        library.disconnectFromCentralLibrary();
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(LibraryTestUsingBeforeClass.class);
    }
```

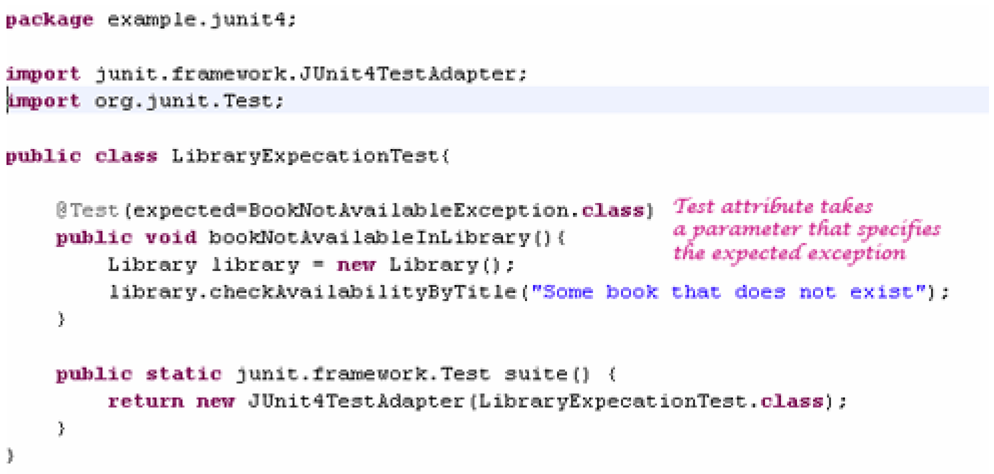
This @BeforeClass method connects & synchronizes the data with the central library once before all the tests.

This @AfterClass method disconnects from the central library after all the tests are run.

Unlike @Before and @After annotations, only one set of @BeforeClass and @AfterClass annotations are allowed.

Expecting exceptions

The new JUnit makes checking for exceptions very easy. The @Test annotation takes a parameter, which declares the type of Exception that should be thrown. The code below demonstrates this:



```java
package example.junit4;

import junit.framework.JUnit4TestAdapter;
import org.junit.Test;

public class LibraryExpecationTest{

    @Test(expected=BookNotAvailableException.class)
    public void bookNotAvailableInLibrary(){
        Library library = new Library();
        library.checkAvailabilityByTitle("Some book that does not exist");
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(LibraryExpecationTest.class);
    }
}
```

Test attribute takes a parameter that specifies the expected exception

In the code above, bookNotAvailableInLibrary is a test, which passes only if BookNotAvailableException is thrown. The test fails if no exception is thrown. Test also fails if a different exception is thrown.

Other Annotations

Ignoring a test

The @Ignore annotation tells the runner to ignore the test and report that it was not run. You can pass in a string as a parameter to @Ignore annotation that explains why the test was ignored. E.g. The new JUnit will not run a test method annotated with @Ignore("Database is down") but will only report it. The version of JUnit4Adapter, I used, did not work with @Ignore annotation. Kent Beck has informed me that the next version of JUnitAdapter will fix this problem.

Timing out a test

You can pass in a timeout parameter to the test annotation to specify the timeout period in milliseconds. If the test takes more, it fails. E.g. A method annotated with @Test (timeout=10) fails if it takes more than 10 milliseconds.

Finally, I would like to thank Kent Beck for taking the time to demonstrate and teach the new JUnit to me.

Summary

To summarize the new JUnit style:

1. It Requires JDK 5 to run.
2. Test classes do not have to extend from junit.framework.TestCase.
3. Test methods do not have to be prefixed with 'test'.
4. There is no difference between the old assert methods and the new assert methods.
5. Use @Test annotations to mark a method as a test case.
6. @Before and @After annotations take care of set up and tear down.
7. @BeforeClass and @AfterClass annotations take care of one time set up and one time tear down.
8. @Test annotations can take a parameter for timeout. Test fails if the test takes more time to execute.
9. @Test annotations can take a parameter that declares the type of exception to be thrown.
10. JUnit4Adapter enables running the new JUnit4 tests using the old JUnit runners.
11. Old JUnit tests can be run in the new JUnit4 runner.