
Tools for Unit Test — JUnit

Ajitha Rajan



JUnit

JUnit is a framework for writing tests

- Written by Erich Gamma (Design Patterns) and Kent Beck (eXtreme Programming)
- JUnit uses *Java's reflection capabilities* (Java programs can examine their own code) and (as of version 4) *annotations*
- JUnit allows us to:
 - define and execute tests and test suites
 - Use test as an effective means of specification
 - write code and use the tests to support refactoring
 - integrate revised code into a build
- JUnit is available on several IDEs, e.g. BlueJ, JBuilder, and Eclipse have JUnit integration to some extent.

JUnit's Terminology

- A **test runner** is software that runs tests and reports results.

Many implementations: standalone GUI, command line, integrated into IDE

- A **test suite** is a collection of test cases.
- A **test case** tests the response of a single method to a particular set of inputs.
- A **unit test** is a test of the smallest element of code you can sensibly test, usually a single class.

JUnit's Terminology

- A **test fixture** is the environment in which a test is run. A new fixture is set up before each test case is executed, and torn down afterwards.

Example: if you are testing a database client, the fixture might place the database server in a standard initial state, ready for the client to connect.

- An **integration test** is a test of how well classes work together.

JUnit provides some limited support for integration tests.

- *Proper* unit testing would involve **mock objects** – fake versions of the other classes with which the class under test interacts.

JUnit does not help with this. It is worth knowing about, but not always necessary.

Structure of a JUnit (4) test class

We want to test a class named Triangle

- This is the unit test for the Triangle class; it defines objects used by one or more tests.

```
public class TriangleTest{  
  
}
```

- This is the default constructor.

```
public TriangleTest(){ }
```

Structure of a JUnit (4) test class

- `@Before public void init()`

Creates a test fixture by creating and initialising objects and values.

- `@After public void cleanUp()`

Releases any system resources used by the test fixture. Java usually does this for free, but files, network connections etc. might not get tidied up automatically.

- `@Test public void noBadTriangles()`, `@Test public void scaleneOk()`, etc.

These methods contain tests for the `Triangle` constructor and its `isScalene()` method.

Making Tests: Assert

- Within a test,
 - Call the method being tested and get the actual result.
 - *assert* a property that should hold of the test result.
 - Each *assert* is a challenge on the test result.
- If the property fails to hold then `assert` fails, and throws an `AssertionFailedError`:
 - JUnit catches these Errors, records the results of the test and displays them.

Making Tests: Assert

- `static void assertTrue(boolean test)`

`static void assertTrue(String message, boolean test)`

Throws an `AssertionFailedError` if the test fails. The optional *message* is included in the Error.

- `static void assertFalse(boolean test)`

`static void assertFalse(String message, boolean test)`

Throws an `AssertionFailedError` if the test succeeds.

Aside: Throwable

- `java.lang.Error`: a problem that an application would not normally try to handle — does not need to be declared in *throws* clause.
e.g. command line application given bad parameters by user.
- `java.lang.Exception`: a problem that the application might reasonably cope with — needs to be declared in *throws* clause.
e.g. network connection timed out during connect attempt.
- `java.lang.RuntimeException`: application might cope with it, but rarely — does not need to be declared in *throws* clause.
e.g. I/O buffer overflow.

Triangle class

For the sake of example, we will create and test a trivial `Triangle` class:

- The constructor creates a `Triangle` object, where only the lengths of the sides are recorded and the private variable p is the longest side.
- The `isScalene` method returns true if the triangle is scalene.
- The `isEquilateral` method returns true if the triangle is equilateral.
- We can write the test methods before the code. This has advantages in separating coding from testing.

But Eclipse helps more if you create the class under test first: Creates test stubs (methods with empty bodies) for all methods and constructors.

Notes on creating tests

- **Size:** Often the amount of (very routine) test code will exceed the size of the code for small systems.
- **Complexity:** Testing complex code can be a complex business and the tests can get quite complex.
- **Effort:** The effort taken in creating test code is repaid in reduced development time, most particularly when we go on to use the test subject in anger (i.e. real code).
- **Behaviour:** Creating a test often helps clarify our ideas on how a method should behave (particularly in exceptional circumstances).

Example

A JUnit 3 test for Triangle

```
import junit.framework.TestCase;

public class TriangleTest extends TestCase {

    private Triangle t;

    // Any method named setUp will be executed before each test.
    protected void setUp() {
        t = new Triangle(5,4,3);
    }

    protected void tearDown() {} // tearDown will be executed afterwards

    public void testIsScalene() { // All tests are named test[Something]
        assertTrue(t.isScalene());
    }

    public void testIsEquilateral() {
        assertFalse(t.isEquilateral());
    }

}
```

A JUnit 4 test for Triangle

```
package st;

more imports are necessary R import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

no need to inherit from TestCase R public class TestTriangle {

    private Triangle t;

    Use annotations... R @Before public void setUp() throws Exception {
                        t = new Triangle(3, 4, 5);
    }

    ...rather than special names R @Test public void scaleneOk() {
                                assertTrue(t.isScalene());
    }
}
```

The Triangle class itself

- Is JUnit too much for small programs?
- Not if you think it will reduce errors.
- Tests on this scale of program often turn up errors or omissions – construct the tests working from the specification
- Sometimes you can omit tests for some particularly straightforward parts of the system

Assert methods II

- `assertEquals(expected, actual)`
`assertEquals(String message, expected, actual)`
This method is heavily overloaded: *expected* and *actual* must be both objects or both of the same primitive type. For objects, uses your equals method, if you have defined it properly, as `public boolean equals(Object o)` — otherwise it uses `==`
- `assertSame(Object expected, Object actual)`
`assertSame(String message, Object expected, Object actual)`
Asserts that two objects refer to the same object (using `==`)
- `assertNotSame(Object expected, Object actual)`
`assertNotSame(String message, Object expected, Object actual)`
Asserts that two objects do not refer to the same object

Assert methods III

- `assertNull(Object object)`
`assertNull(String message, Object object)`
Asserts that the object is null
- `assertNotNull(Object object)`
`assertNotNull(String message, Object object)`
Asserts that the object is not null
- `fail()`
`fail(String message)`
Causes the test to fail and throw an `AssertionFailedError` — Useful as a result of a complex test, when the other assert methods are not quite what you want

The assert statement in Java

- Earlier versions of JUnit had an `assert` method instead of an `assertTrue` method — The name had to be changed when Java 1.4 introduced the `assert` statement
- There are two forms of the `assert` statement:
 - `assert boolean_condition ;`
 - `assert boolean_condition : error_message ;`

Both forms throw an `AssertionFailedError` if the *boolean_condition* is false. The second form, with an explicit *error_message*, is seldom necessary.

The assert statement in Java

When to use an assert statement:

- Use it to document a condition that you *'know'* to be true
- Use `assert false`; in code that you *'know'* cannot be reached (such as a default case in a switch statement)
- Do not use `assert` to check whether parameters have legal values, or other places where throwing an `Exception` is more appropriate
- **Can be dangerous:** customers are not impressed by a library bombing out with an assertion failure.

JUnit in Eclipse

To create a test class, select
File → New → JUnit Test Case
and enter the name of your test case

Package R

Test class R

Decide what stubs you want to create R

Identify the class under test R

New JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify

New JUnit 3 test New JUnit 4 test

Source folder: Browse...

Package: Browse...

Name:

Superclass: Browse...

Which method stubs would you like to

setUpBeforeClass() tearDownAfterClass()
 setUp() tearDown()
 constructor

Do you want to add comments? (Configure templates and default value [here](#))

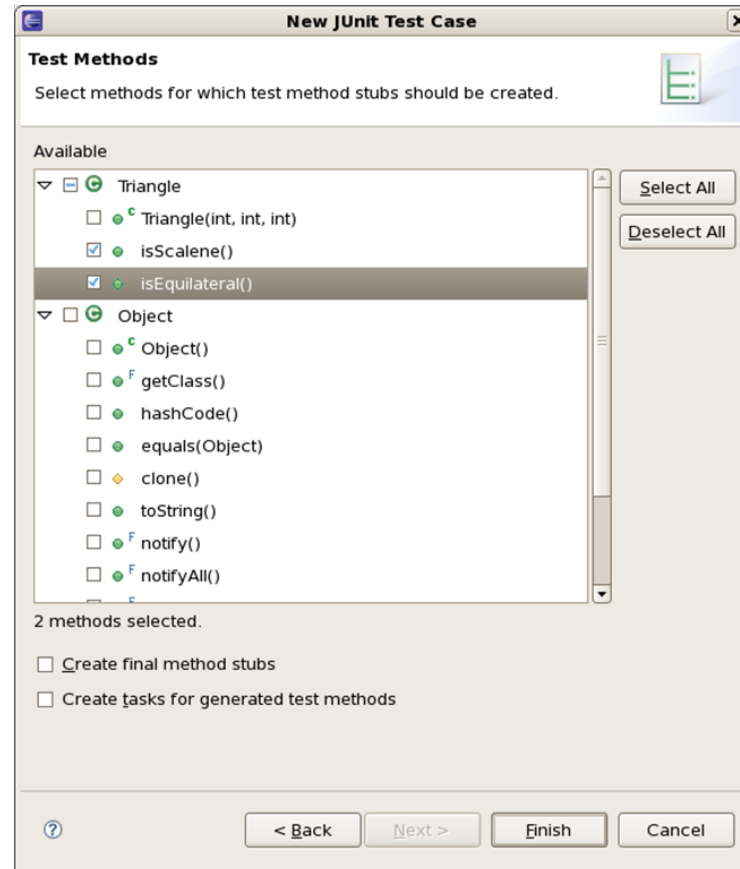
Generate comments

Class under test: Browse...

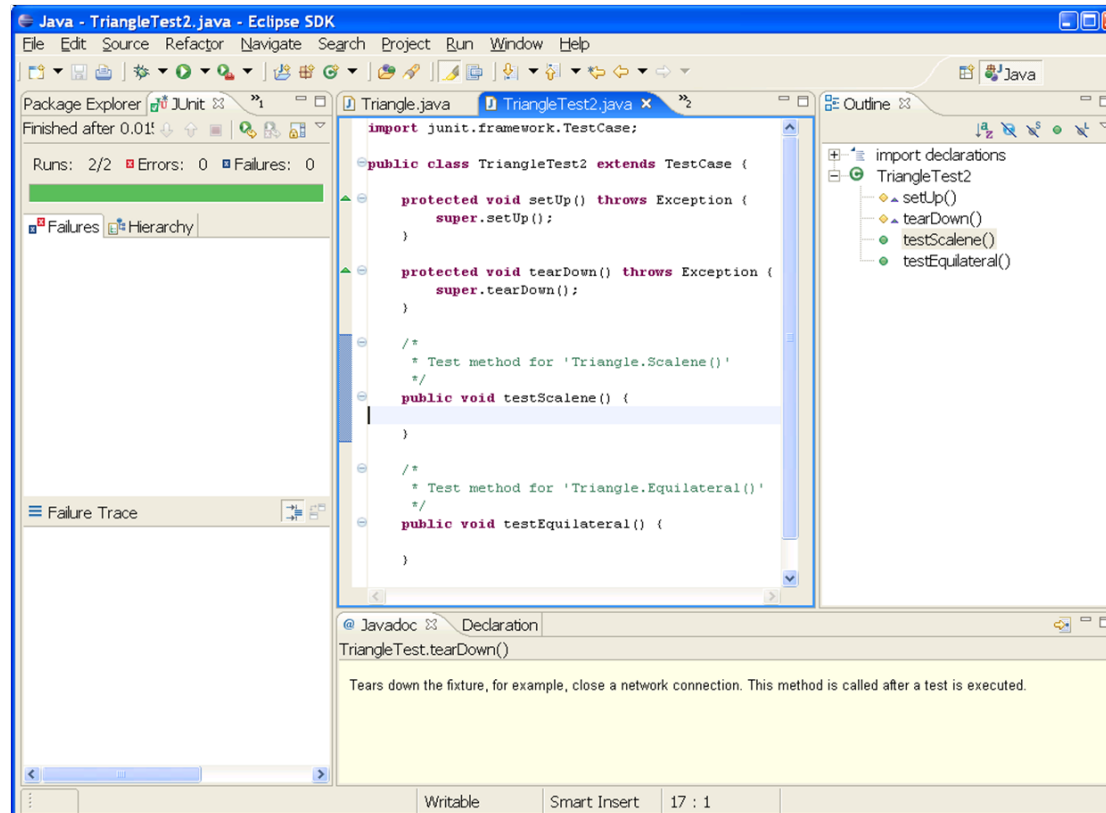
< Back Next > Finish Cancel

Creating a Test

Decide what you want to test R



Template for New Test



The screenshot shows the Eclipse IDE interface with a Java test class named `TriangleTest2.java` open. The code is as follows:

```
import junit.framework.TestCase;

public class TriangleTest2 extends TestCase {

    protected void setUp() throws Exception {
        super.setUp();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    /*
     * Test method for 'Triangle.Scalene()'
     */
    public void testScalene() {

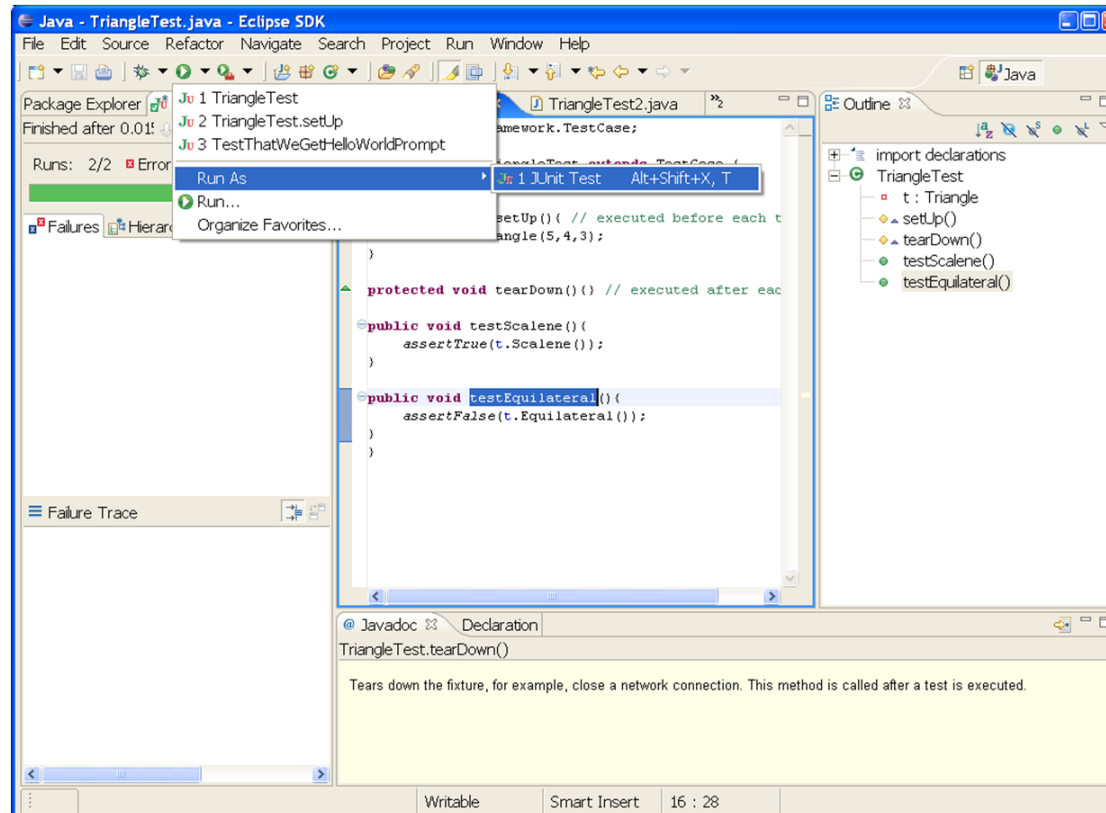
    }

    /*
     * Test method for 'Triangle.Equilateral()'
     */
    public void testEquilateral() {

    }
}
```

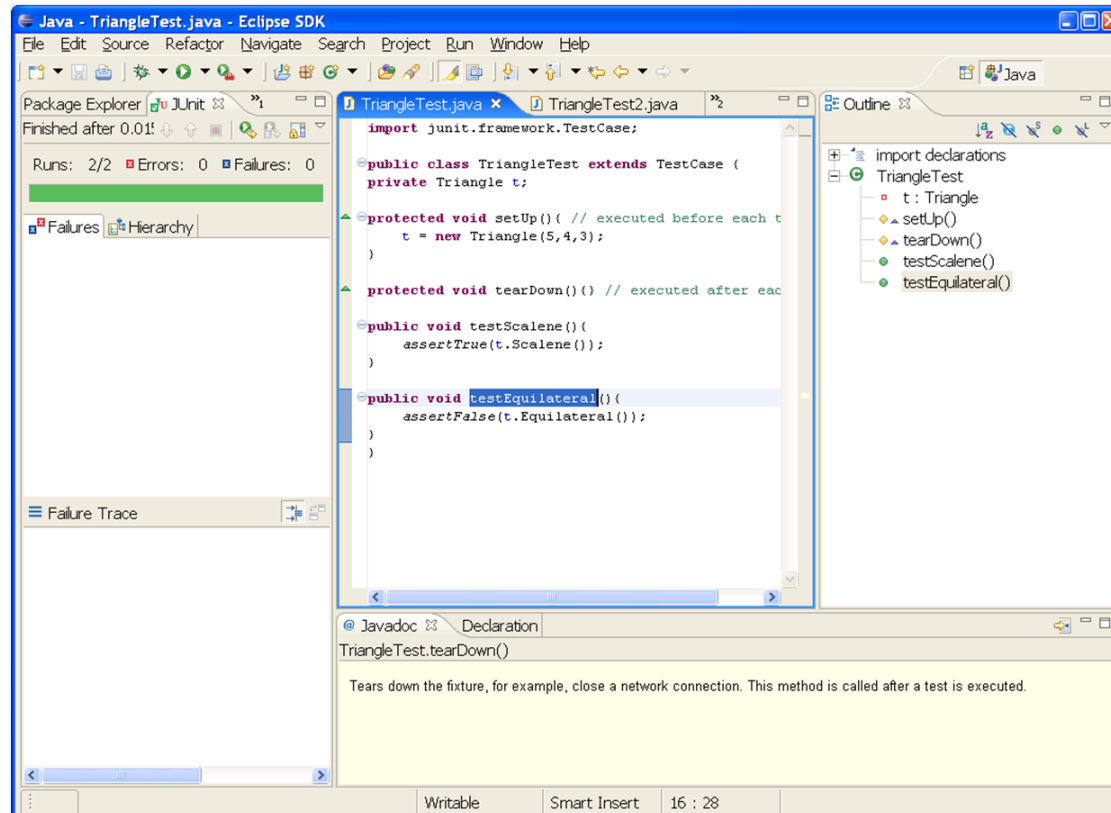
The Package Explorer on the left shows the test results: "Runs: 2/2", "Errors: 0", and "Failures: 0". The Outline view on the right shows the class structure with methods: `setUp()`, `tearDown()`, `testScalene()`, and `testEquilateral()`. The Javadoc view at the bottom shows the declaration for `TriangleTest.tearDown()` with the description: "Tears down the fixture, for example, close a network connection. This method is called after a test is executed."

Running JUnit



Results

Results are here **R**



Issues with JUnit

JUnit has a model of calling methods and checking results against the expected result. **Issues** are:

- **State:** objects that have significant internal state (e.g. collections with some additional structure) are harder to test because it may take many method calls to get an object into a state you want to test. **Solutions:**
 - Write long tests that call some methods many times.
 - Add additional methods in the interface to allow observation of state (or make private variables public?)
 - Add additional methods in the interface that allow the internal state to be set to a particular value
 - “Heisenbugs” can be an issue in these cases (changing the observations changes what is observed).

Issues with JUnit

- Other effects, e.g. output can be hard to capture correctly.
- JUnit tests of GUIs are not particularly helpful (recording gestures might be helpful here?)

Positives

- Using JUnit encourages a ‘*testable*’ style, where the result of a calling a method is easy to check against the specification:
 - Controlled use of state
 - Additional observers of the state (testing interface)
 - Additional components in results that ease checking
- It is well integrated into a range of IDEs (e.g. Eclipse)
- Tests are easy to define and apply in these environments.
- JUnit encourages frequent testing during development — e.g. XP (eXtreme Programming) ‘*test as specification*’
- JUnit tends to shape code to be easily testable.
- JUnit supports a range of extensions that support structured testing (e.g. coverage analysis) – we will see some of these extensions later.

Another Framework for Testing

- Framework for Integrated Test (FIT), by Ward Cunningham (inventor of wiki)
- Allows closed loop between customers and developers:
 - Takes HTML tables of expected behaviour from customers or spec.
 - Turns those tables into test data: inputs, activities and assertions regarding expected results.
 - Runs the tests and produces tabular summaries of the test runs.
- Only a few years old, but lots of people seem to like it — various practitioners seem to think it is revolutionary.

Readings

Required Readings

- [JUnit Test Infected: Programmers Love Writing Tests](#)
an introduction to JUnit.
- [Using JUnit With Eclipse IDE](#)
an O'Reilly article
- [Unit Testing in Jazz Using JUnit](#)
an NCSU Open Lab article on using JUnit with Eclipse

Suggested Readings

- Michael Olan. 2003. [Unit testing: test early, test often](#). J. Comput. Small Coll. 19, 2 (December 2003), 319-328.

Resources

Getting started with Eclipse and JUnit

Activity: to start using JUnit within Eclipse review and try the example of defining tests for a Triangle class.

[\[link to Activity\]](#)

Video: this video tutorial shows how to create a new Eclipse project and start writing JUnit tests first.

[\[link to Video\]](#)

Get testing!

Start up Eclipse and:

1. Create a new Java project
2. Add a new package, ‘‘st’’
3. Create `st.Triangle`; grab the source from the JUnit lecture’s Activity in the resources
4. Create a new source folder called ‘‘tests’’ if you like (with a new ‘‘st’’ package)
5. Create a new JUnit test for `st.Triangle`
6. And get testing!