Tools for Unit Test — JUnit

Stuart Anderson

School of informatics

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.

Slide 1: For more info on JUnit

The JUnit site provides a wealth of useful information on JUnit and the host of JUnit-based products.

http://www.junit.org/

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 TriangleTestJ4{

  }
  ```

- 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.

Setup and Teardown

```
@Before
public void name() { ... }
@After
public void name() { ... }
```

- methods to run before/after each test case method is called

```
@BeforeClass
public static void name() { ... }
@AfterClass
public static void name() { ... }
```

- methods to run once before/after the entire test class runs

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.

Assert methods

- Each assert method has parameters like these: *message, expected-value, actual-value*
- Assert methods dealing with floating point numbers get an additional argument, a tolerance
- Each assert method has an equivalent version that does not take a message – however, this use is not recommended because:
  - messages helps documents the tests
  - messages provide additional information when reading failure logs

Assert methods

- assertTrue(String *message*, Boolean *test*)
  - fails if the boolean test is `false`
- assertFalse(String *message*, Boolean *test*)
  - fails if the boolean test is `true`
- assertNull(String *message*, Object *object*)
  - fails if the given value is *not* `null`
- assertNotNull(String *message*, Object *object*)
  - fails if the given value is `null`
- assertEquals(String *message*, Object *expected*, Object *actual*) (uses equals method)
  - fails if the values are not equal
- assertSame(String *message*, Object *expected*, Object *actual*) (uses == operator)
  - fails if the values are not the same (by ==)
- assertNotSame(String *message*, Object *expected*, Object *actual*)
  - fails if the values *are* the same (by ==)

Example Junit test

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

public class TestArrayIntList {
    @Test
    public void testAddGet1() {
        ArrayIntList list = new ArrayIntList();
        list.add(42);
        list.add(-3);
        list.add(15);
        assertEquals(42, list.get(0));
        assertEquals(-3, list.get(1));
        assertEquals(15, list.get(2));
    }

    @Test
    public void testIsEmpty() {
        ArrayIntList list = new ArrayIntList();
        assertTrue(list.isEmpty());
        list.add(123);
        assertFalse(list.isEmpty());
        list.remove(0);
        assertTrue(list.isEmpty());
    }
    ...
```

Another Example

```
package com.vogella.junit.first;
import static org.junit.Assert.assertEquals;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
public class MyClassTest {
  @Test(expected = IllegalArgumentException.class)
  public void testExceptionIsThrown() {
    MyClass tester = new MyClass();
    tester.multiply(1000, 5);
  }
  @Test
  public void testMultiply() {
    MyClass tester = new MyClass();
    assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));
  }
}
```

TestSuites

- TestSuites collect a selection of tests to run them as a unit
- Collections automatically use TestSuites, however to specify the order in which tests are run, write your own:

```
public static Test suite() {
        suite.addTest(new TestBowl("testBowl"));
        suite.addTest(new TestBowl("testAdding"));
        return suite;
}
```

- Should seldom have to write your own TestSuites as each method in your TestCase should be independent of all others
- Can create TestSuites that test a whole package:

```
public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(TestBowl.class);
        suite.addTestSuite(TestFruit.class);
        return suite;
}
```

TestSuite Example

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    WeekdayTest.class,
    TimeTest.class,
    CourseTest.class,
    ScheduleTest.class,
    CourseComparatorsTest.class
})
public class HW2Tests {}
```

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.

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

more imports are necessary ☞

no need to inherit from TestCase ☞

Use annotations... ☞

...rather than special names ☞

```
package st;

import static org.junit.Assert.*;

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

public class TestTriangle {

        private Triangle t;

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

        @Test public void scaleneOk() {
                assertTrue(t.isScalene());
        }
}
```

Slide 13: The `Triangle` class itself

```
public class Triangle {
private int p; // Longest edge
private int q;
private int r;

public Triangle(int s1, int s2, int s3) {
   if (s1>s2) {
      p = s1; q = s2;
   } else {
      p = s2; q = s1;
   }
   if (s3>p) {
      r = p; p = s3;
   } else {
      r = s3;
   }
}

public boolean isScalene() {
   return ((r>0) && (q>0) && (p>0) &&
      (p<(q+r))&& ((q>r) || (r>q)));
}

public boolean isEquilateral() {
   return p == q && q == r;
}
}
```

JUnit in Eclipse

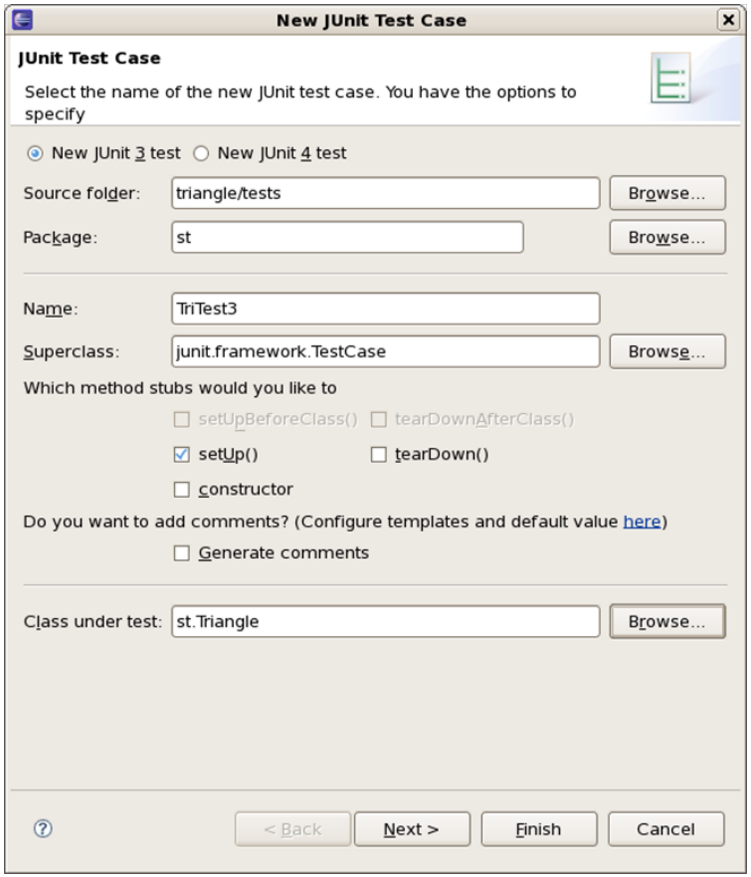
To create a test class, select
`File` → `New` → `JUnit Test Case`
and enter the name of your test case

Package ☞

Test class ☞

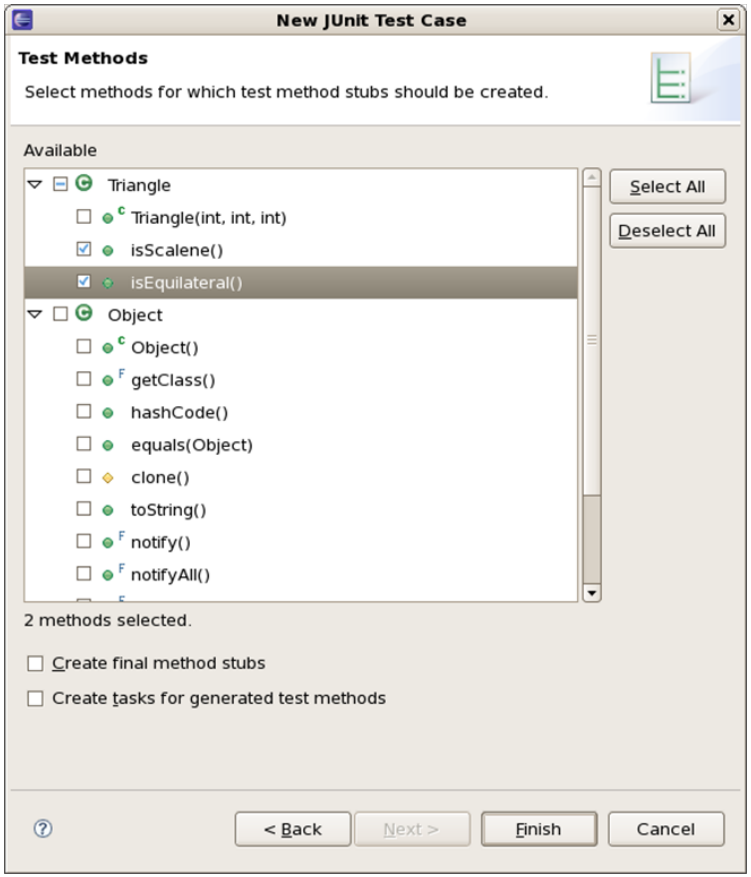
Decide what stubs you want to create ☞

Identify the class under test ☞

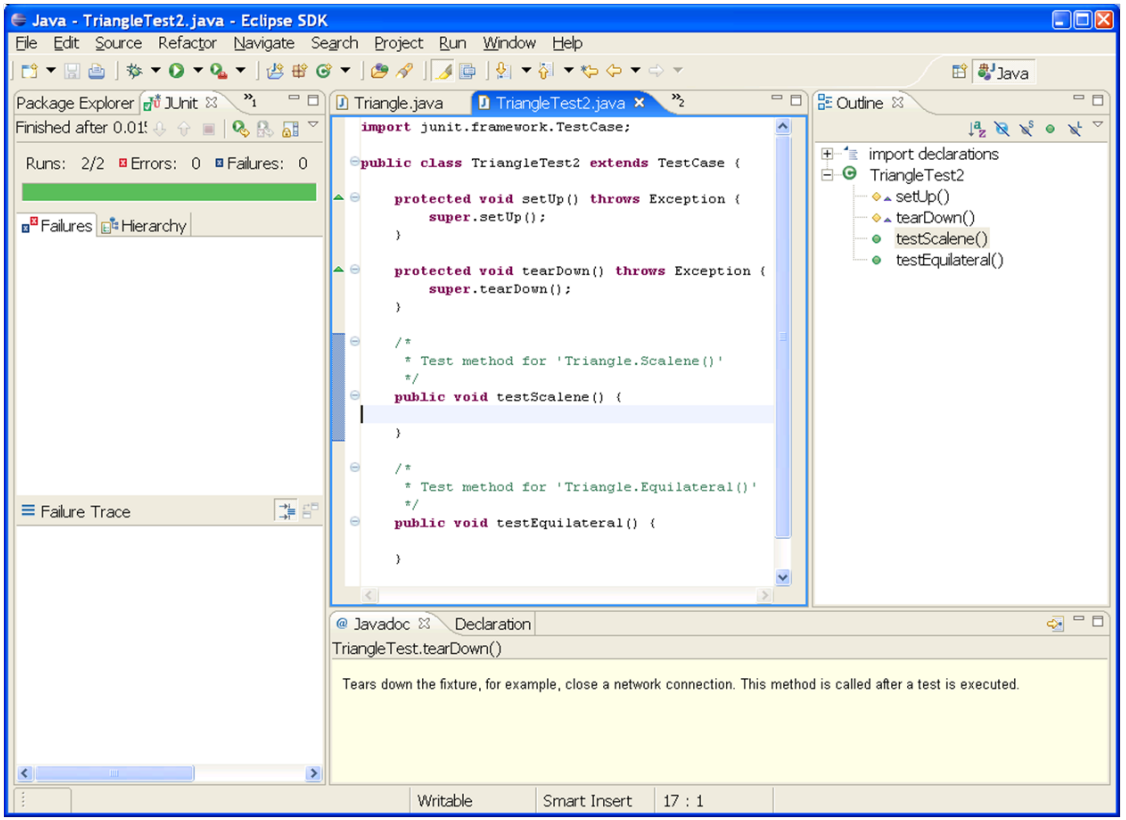


Creating a Test

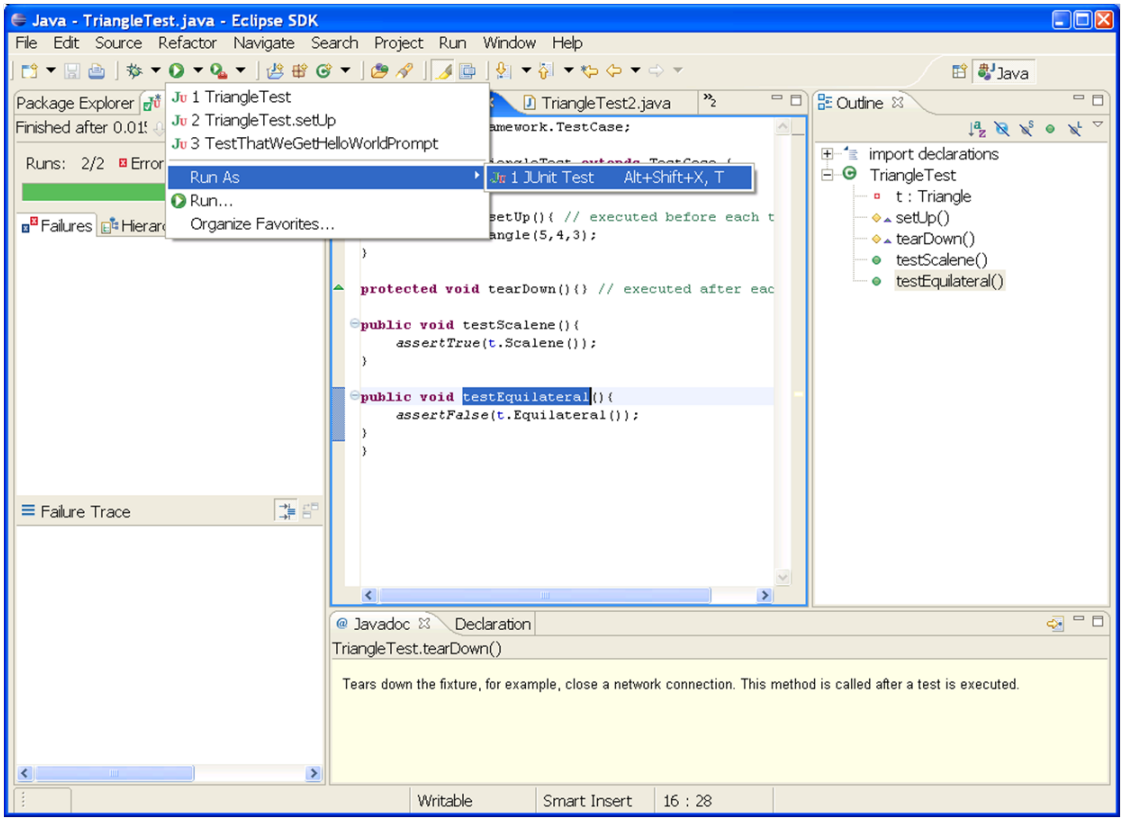
Decide what you want to test ☞



Template for New Test

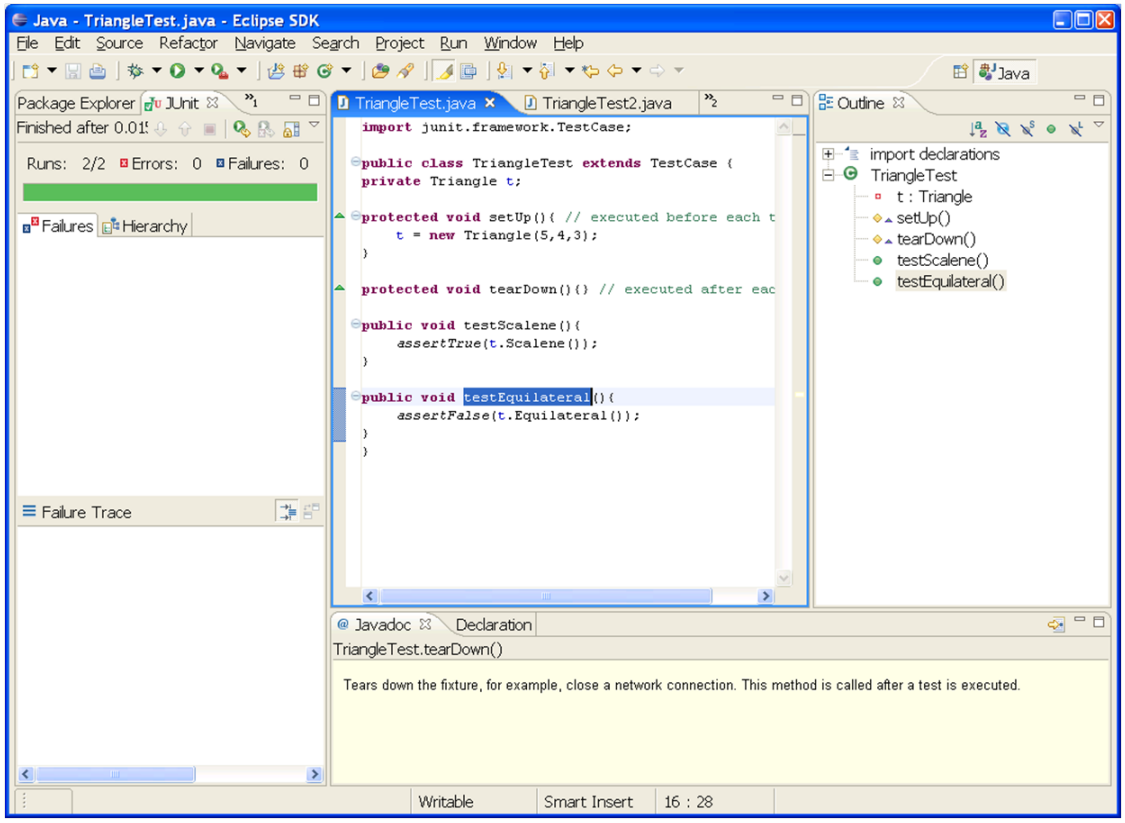


Running JUnit



Results

Results are here ☞



Resources

Getting started with Eclipse and JUnit

Activity: to start using JUnit within Eclipe 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!
7. Follow the video from the Junit lecture's resources for more details.