Trường Đại Học Bách Khoa Hà Nội
Viện Công Nghệ Thông Tin &Truyền Thông

Kiểm thử phần mềm

JUnit

TS. Nguyễn Thanh Hùng
Bộ Môn Công Nghệ Phần Mềm
Email: hungnt@soict.hust.edu.vn
Website: http://soict.hust.edu.vn/~hungnt

JUnit

Junit là một nền tảng kiểm thử

- Viết bởi Erich Gamma (Design patterns) và Kent Bech (eXtreme Programming)
- Sử dụng khả năng phản chiếu (Chương trình Java có thể kiểm tra chính mã nguồn cuả nó)
- Cho phép:
  - Định nghĩa và thực hiện kiểm thử và các tập kiểm thử
  - Sử dụng test như một công cụ hiệu quả cho specification
- Hỗ trợ trên IDEs như BlueJ, Jbuilder, và Eclipse có tích hợp sẵn Junit
- Website về Junit: **http://www.junit.org**

Các thuật ngữ về JUnit

- "Test runner" là phần mềm chạy kiểm thử và báo cáo kết quả
- "Test suite" là một tập các trường hợp kiểm thử
- "Test case" kiểm tra phản ứng của một hàm đơn với 1 tập đầu vào
- "Unit test" là một kiểm thử của phần tử mã nguồn nhỏ nhất có thể kiểm thử, thường là một lớp đơn.

Các thuật ngữ về JUnit

- "Test fixture" là môi trường chạy kiểm thử. Một môi trường mới được cài đặt trước mỗi lần thực hiện trường hợp kiểm thử, và được huỷ bỏ sau đó.

  Ví dụ: để thử nghiệm một CSDL, môi trường kiểm thử có thể thiết lập máy chủ ở trạng thái ban đầu chuẩn, sẵn sàng cho khách hàng để kết nối.

- "Integration test" là một kiểm thử cho việc kết hợp hoạt động của nhiều lớp với nhau

  Junit hỗ trợ một phần cho kiểm thử tích hợp

Cấu trúc của một lớp kiểm thử JUnit

- Chúng ta muốn kiểm thử một lớp tên Triangle
  - Đây là kiểm thử đơn vị của lớp Triangle; định nghĩa đối tượng sử dụng trong một hay nhiều kiểm thử

```
public class TriangleTestJ4{
}
```

Đây là hàm tạo mặc định:

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

Cấu trúc của một lớp kiểm thử JUnit

- @Before public void init()
  Tạo một môi trường kiểm thử bằng cách tạo và khởi tạo các đối tượng và các giá trị.
- @After public void cleanUp()
  Giải phóng các tài nguyên hệ thống sử dụng bởi môi trường kiểm thử. Java thường thực hiện giải phóng tự động, nhưng files, kết nối mạng, …, có thể ko được giải phóng hoàn toàn.
- @Test public void noBadTriangles(), @Test public void scaleneOk(), etc.
  Các phương thức chứa các tests cho hàm tạo Triangle và phương thức isScalene().

Thực hiên Tests: Assert

- Trong một test:
  - Gọi phương thức được kiểm thử và thu kết quả thực tế
  - *Assert (xác nhận)* một thuộc tính chứa kết quả kiểm thử
  - Mỗi assert là một thách thức cho kết quả kiểm thử
- Nếu một thuộc tính thất bại, assert sẽ thất bại và một đối tượng AssertionFailedError được sinh ra
  - Junit sẽ nhận được các lỗi, ghi lại các kết quả kiểm thử và hiện thị ra.

Thực hiện kiểm thử: 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).

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 ☞       import static org.junit.Assert.*;

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

no need to inherit from TestCase ☞ public class TestTriangle {

                                           private Triangle t;

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

   ...rather than special names ☞          @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

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;
    }
}
```

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(Objectexpected, Objectactual)

  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, Objectobject)

  Asserts that the object is 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

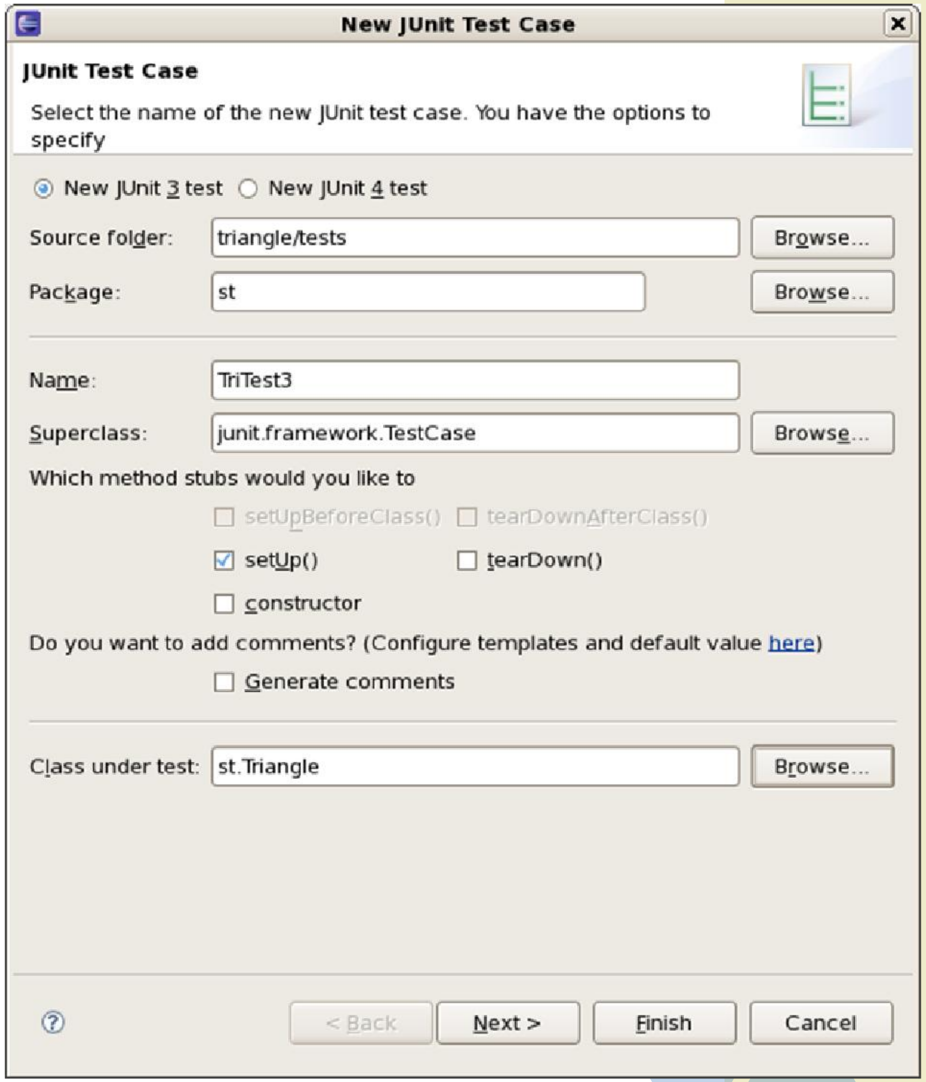
To create a test class, select
File ! New ! JUnit Test Case
and enter the name of your test case

Package ☞

Test class ☞

Decide what stubs you want to create ☞

Identify the class under test ☞



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: triangle/tests

Package: st

Name: TriTest3

Superclass: junit.framework.TestCase

Which method stubs would you like to

☐ setUpBeforeClass() ☐ tearDownAfterClass()

☑ setUp() ☐ tearDown()

☐ constructor

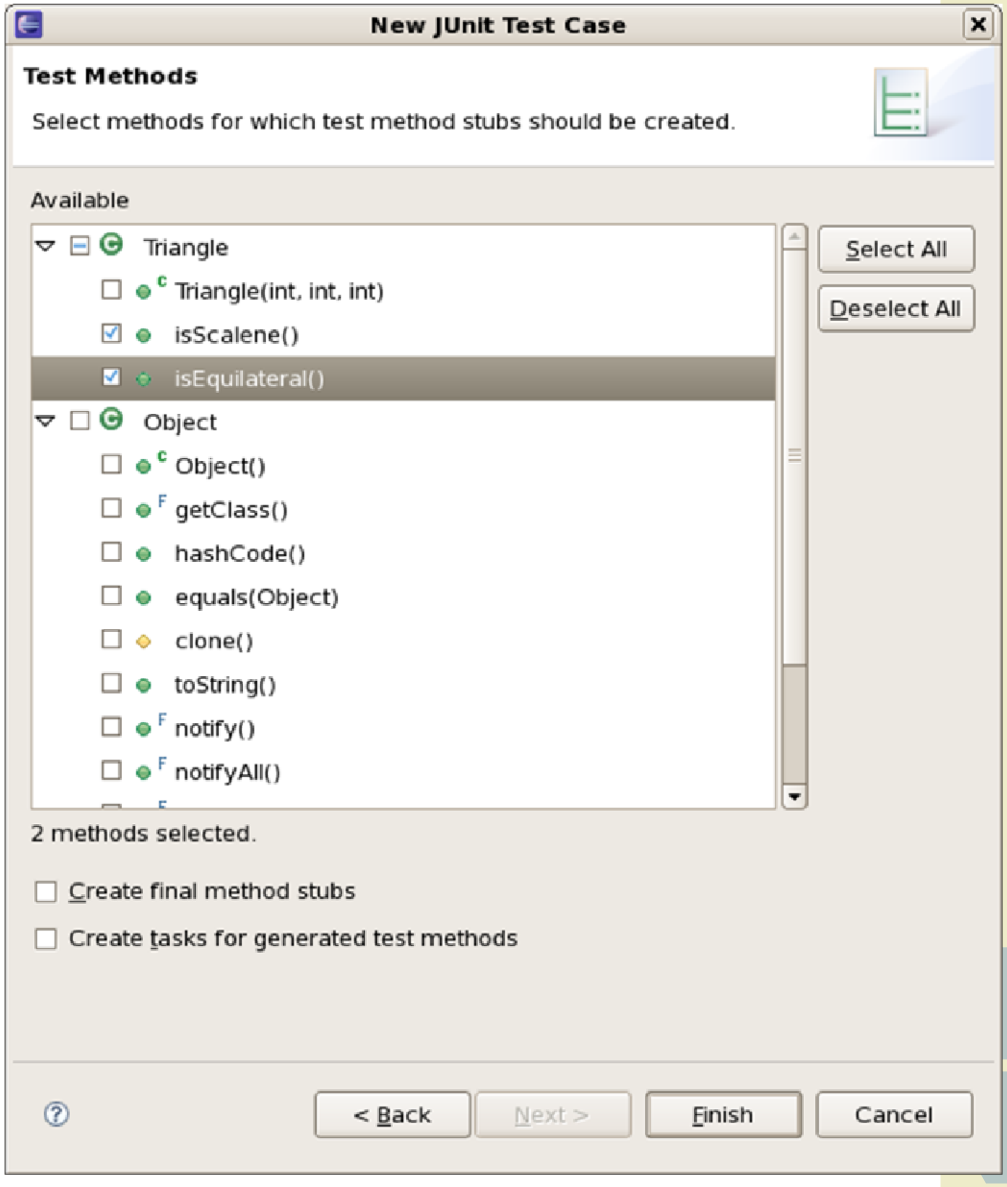
Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Class under test: st.Triangle

Creating a Test

Decide what you want to test ☞



New JUnit Test Case

Test Methods

Select methods for which test method stubs should be created.

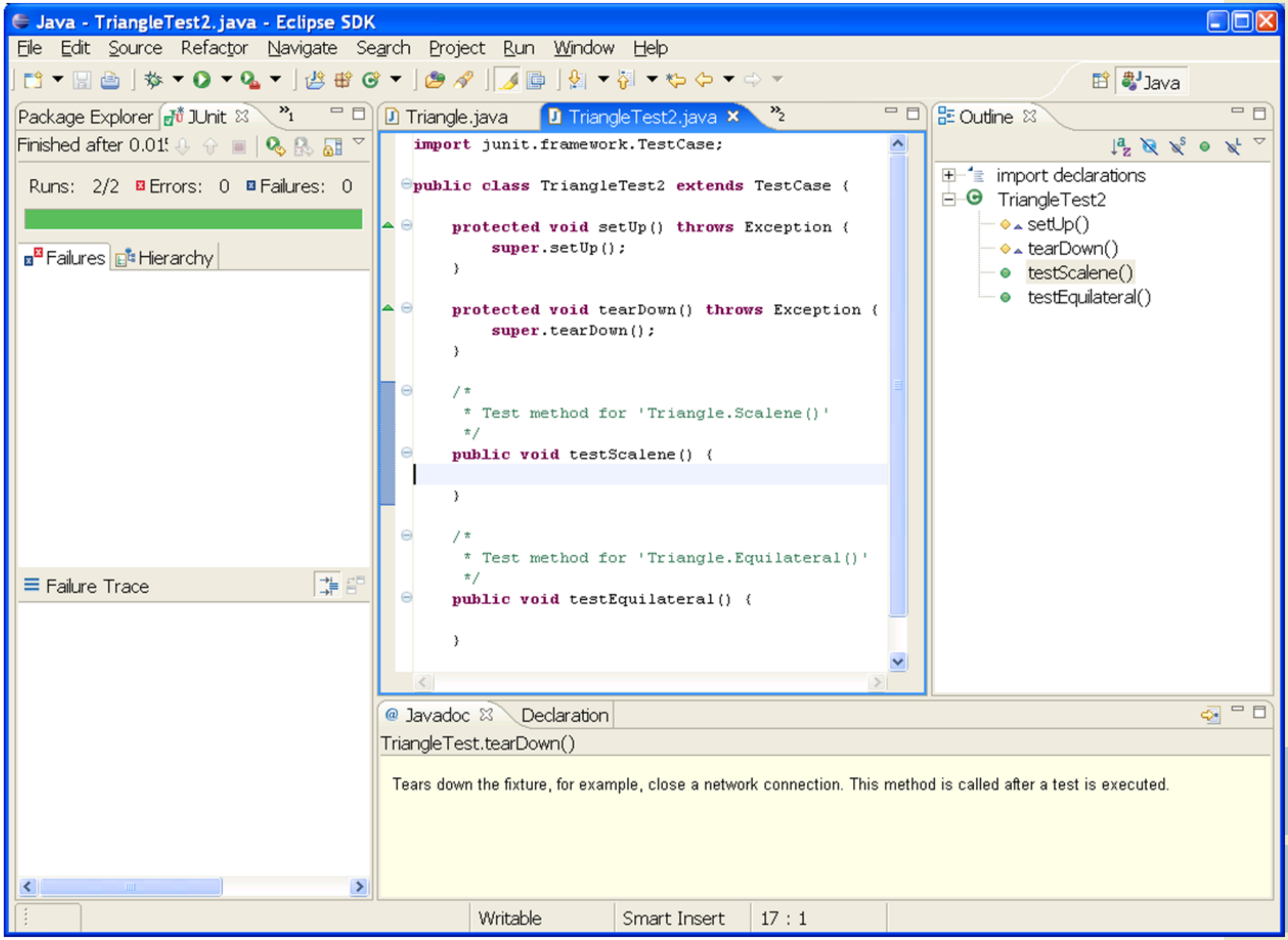
Available

- [ ] Triangle
  - [ ] Triangle(int, int, int)
  - [x] isScalene()
  - [x] isEquilateral()
- [ ] Object
  - [ ] Object()
  - [ ] getClass()
  - [ ] hashCode()
  - [ ] equals(Object)
  - [ ] clone()
  - [ ] toString()
  - [ ] notify()
  - [ ] notifyAll()

2 methods selected.

- [ ] Create final method stubs
- [ ] Create tasks for generated test methods

Template for New Test

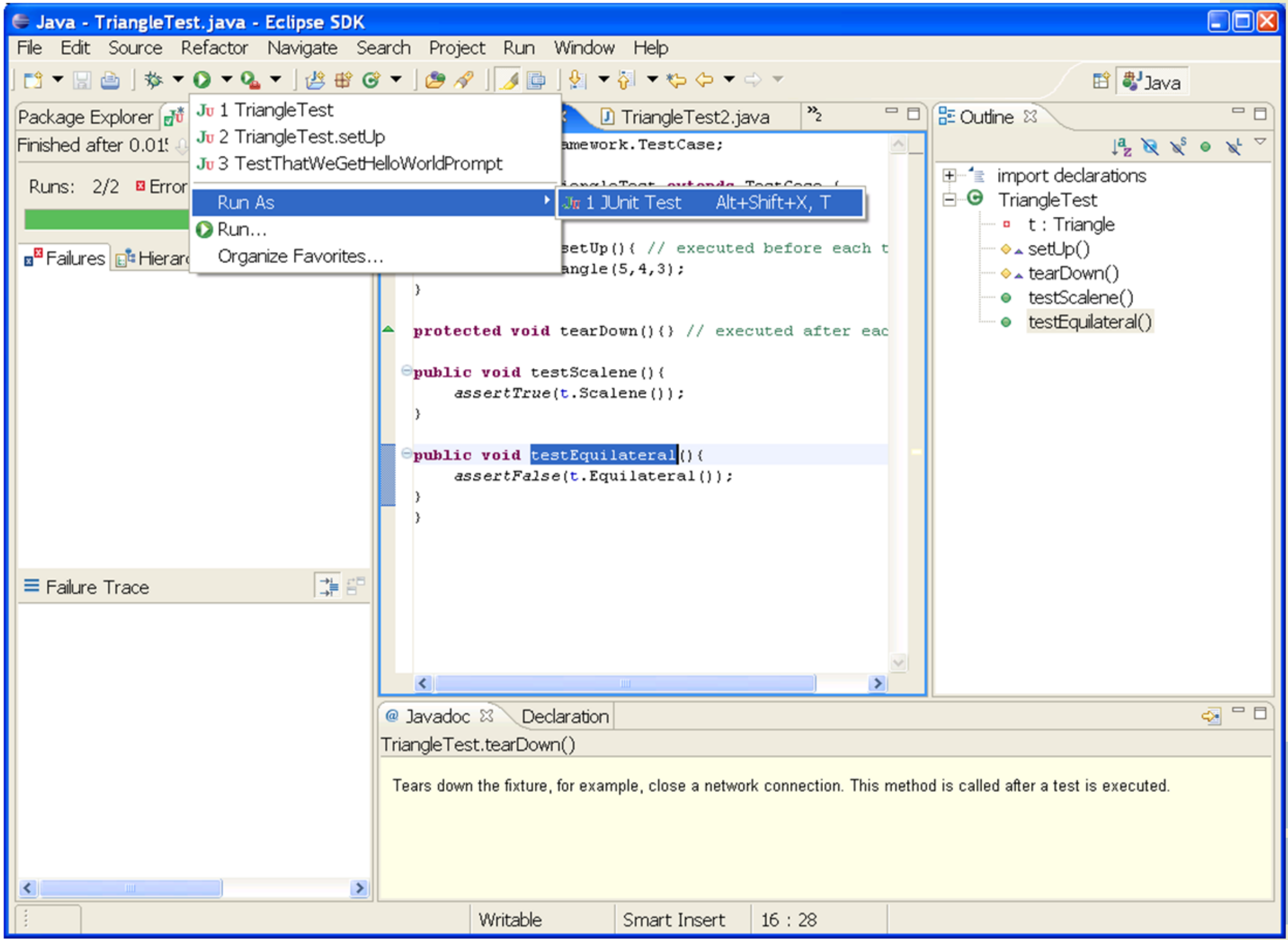


```
import junit.framework.TestCase;

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() {

    }
```

TriangleTest.tearDown()

Tears down the fixture, for example, close a network connection. This method is called after a test is executed.

Running JUnit



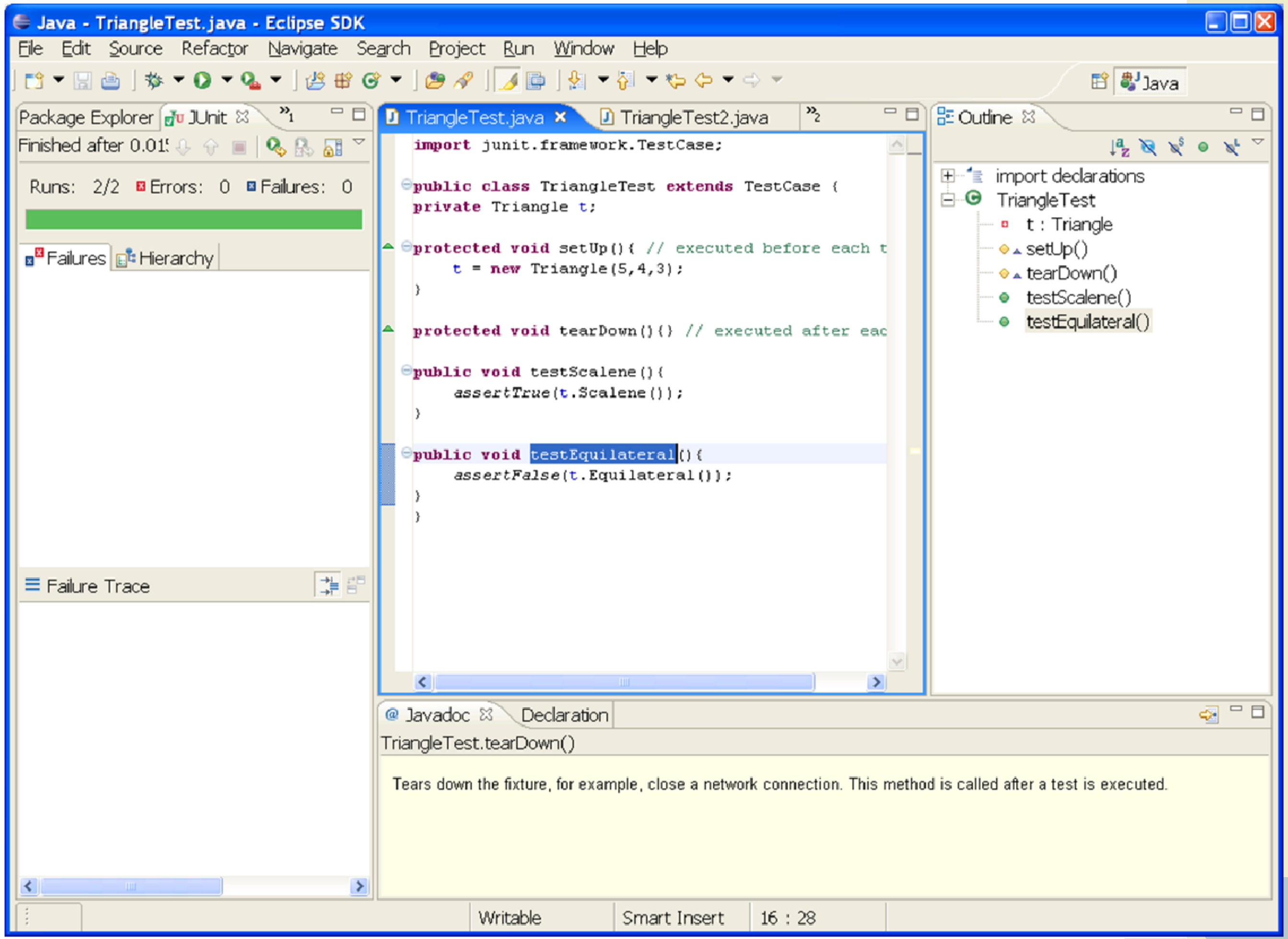
```
amework.TestCase;

iangleTest extends TestCase {

setUp(){ // executed before each t
angle(5,4,3);
}

protected void tearDown(){} // executed after eac

public void testScalene(){
    assertTrue(t.Scalene());
}

public void testEquilateral(){
    assertFalse(t.Equilateral());
}
}
```

TriangleTest.tearDown()

Tears down the fixture, for example, close a network connection. This method is called after a test is executed.

Kết quả

Results are here ☞



```
Finished after 0.01
Runs: 2/2   Errors: 0   Failures: 0
```

```
import junit.framework.TestCase;

public class TriangleTest extends TestCase {
private Triangle t;

protected void setUp(){ // executed before each t
    t = new Triangle(5,4,3);
}

protected void tearDown(){} // executed after eac

public void testScalene(){
    assertTrue(t.Scalene());
}

public void testEquilateral(){
    assertFalse(t.Equilateral());
}
}
```

Outline:
- import declarations
- TriangleTest
  - t : Triangle
  - setUp()
  - tearDown()
  - testScalene()
  - testEquilateral()

TriangleTest.tearDown()

Tears down the fixture, for example, close a network connection. This method is called after a test is executed.

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.