Code Coverage

- **Code coverage** is a measure used to describe the degree to which the source code of a program is tested by a particular test suite.

- High code coverage means
  - □ more thoroughly tested
  - □ has a lower chance of containing software bugs

Coverage criteria

- Method coverage: Have all methods been called?

- Decision/Branch Coverage: Have all decisions been executed in both the true and false paths?

- Condition Coverage: Have all conditionals been executed in both the true and false paths?

- Statement coverage: Have all statements in a method been executed?

Method Coverage

```
String getTriangle (int a, int b, int c) {

if (a == b && b == c) {
    success();
    return "Equilateral";
}

if (a == b || b == c || c == a) {
    return "Isosceles";
}

    return "Scalene";
}
```

```
void success()
{
    printf(“Got it!”);

}
```

```
void testCoverage()
{
    getTriangle(1,1,2);
    getTriangle(2,2,2);
}
```

Condition Coverage

```
String getTriangle (int a, int b, int c) {

if (a == b && b == c) {
    success();
    return "Equilateral";
}

if (a == b || b == c ||  c == a) {
    return "Isosceles";
}

    return "Scalene";
}

void success()
{
    printf(“Got it!”);

}
```

```
void testCoverage()
{
    getTriangle(2,2,3);
    getTriangle(2,2,2);
    getTriangle(1,2,2);
    getTriangle(2,1,2);
}
```

Branch Coverage

```
String getTriangle (int a, int b, int c) {

if (a == b && b == c) {
   success();
   return "Equilateral";
}

if (a == b || b == c || c == a) {
   return "Isosceles";
}

   return "Scalene";
}

void success()
{
   printf(“Got it!”);

}
```

```
void testCoverage()
{
    getTriangle(2,2,3);
    getTriangle(2,2,2);
    getTriangle(1,2,2);
    getTriangle(2,1,2);
    getTriangle(2,3,4);
}
```

Statement Coverage

```
String getTriangle (int a, int b, int c) {

if (a == b && b == c) {
    success();
    return "Equilateral";
}

if (a == b || b == c || c == a) {
    return "Isosceles";
}

    return "Scalene";
}

void success()
{
   printf("Got it!");

}
```

```
void testCoverage()
{
    getTriangle(2,2,3);
    getTriangle(2,2,2);
    getTriangle(1,2,2);
    getTriangle(2,1,2);
    getTriangle(2,3,4);
}
```

Benefits and Limitations

- Benefits:
  - A measure of how complete your test cases are
  - High coverage does not guarantee code correctness!
  - Can identify paths through the code that you may have missed

- Limitations:
  - The assumption that you are done testing if you have high coverage is incorrect
  - Coverage tools only tell you if you’ve covered what’s there
  - There may be requirements that you have missed!
  - Don’t write test cases ONLY to make your coverage tool happy
  - Inspect surrounding code for other potential errors, perhaps requirements you’ve missed

Code Coverage tools

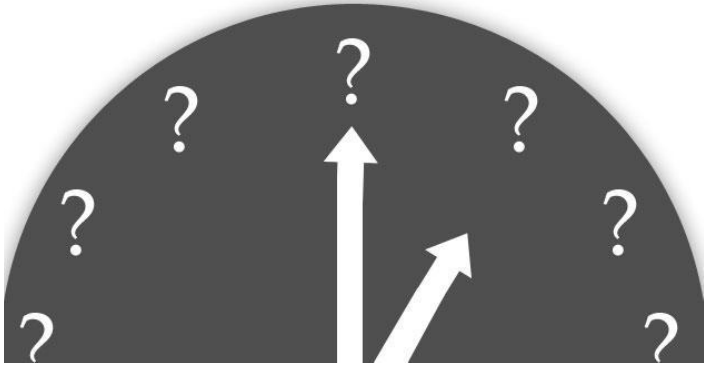
- djUnit- http://works.dgic.co.jp/djunit/
  - □ Eclipse plug-in
  - □ Measures
    - Statement Coverage
    - Branch Coverage
- EclEmma - http://eclemma.org/
  - □ Eclipse plug-in
  - □ Measures (at the bytecode level)
    - Instruction Coverage
    - Block Coverage – roughly corresponds to condition coverage
    - Line Coverage
    - Method Coverage
    - Type Coverage
- Clover - https://www.atlassian.com/software/clover/overview
  - □ Commercial tool
    - Method coverage
    - Statement coverage
    - Decision coverage

What is Unit testing?

Unit tests are whitebox tests written by developers, and designed to verify small units of program functionality.

When to write unit tests?

Unit tests are written by you, the developer, concurrently with implementation.



Good Unit Testing

- Automatic: Run completely by itself, without any human input.
- Atomic: Determine by itself whether the function it is testing has passed or failed, without a human interpreting the results
- Single responsibility: Test exactly one feature
- Independent: Run in isolation, separate from any other test cases (even if they test the same functions)
- Repeatable: Multiple invocations of the test should consistently return the same value.

JUnit

- Unit testing framework for the Java Programming Language.
- Open source (http://Junit.org)
- Framework for both writing and automated execution of unit tests
- Can be integrated with eclipse.

JUnit in Action

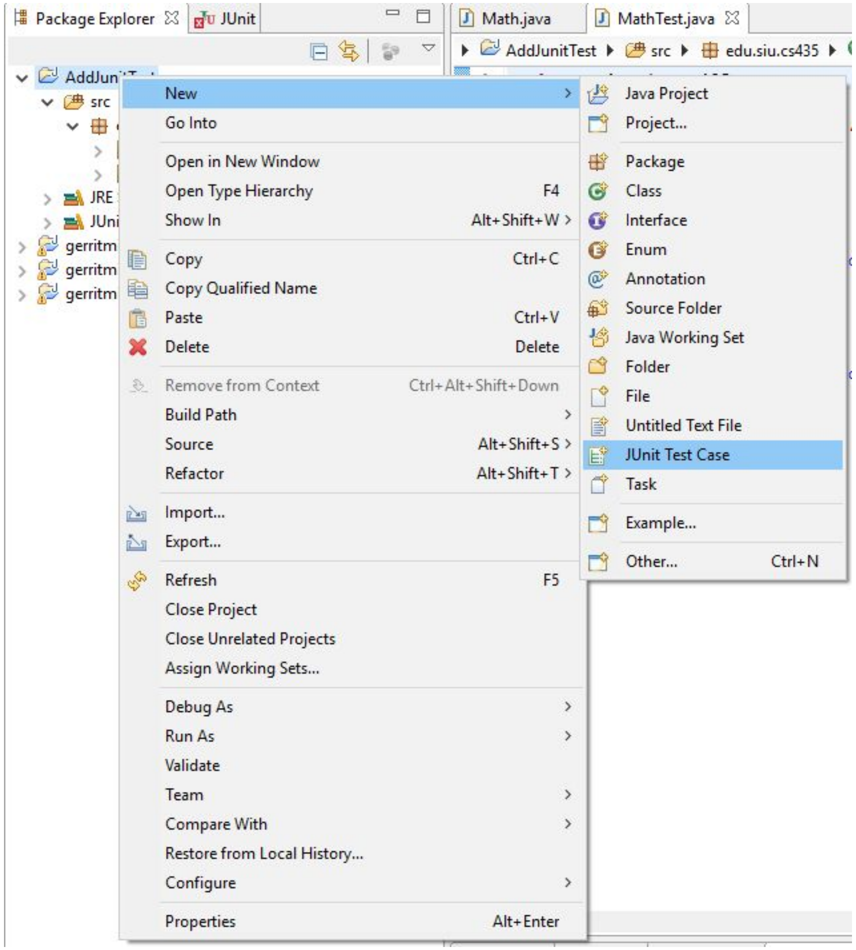
```
package edu.siu.cs435;

public class Math
{
   public int add(int a, int b) {
      return a + b;
   }

   public int sub(int a, int b) {
   return a - b;
   }
}
```

JUnit and Eclipse

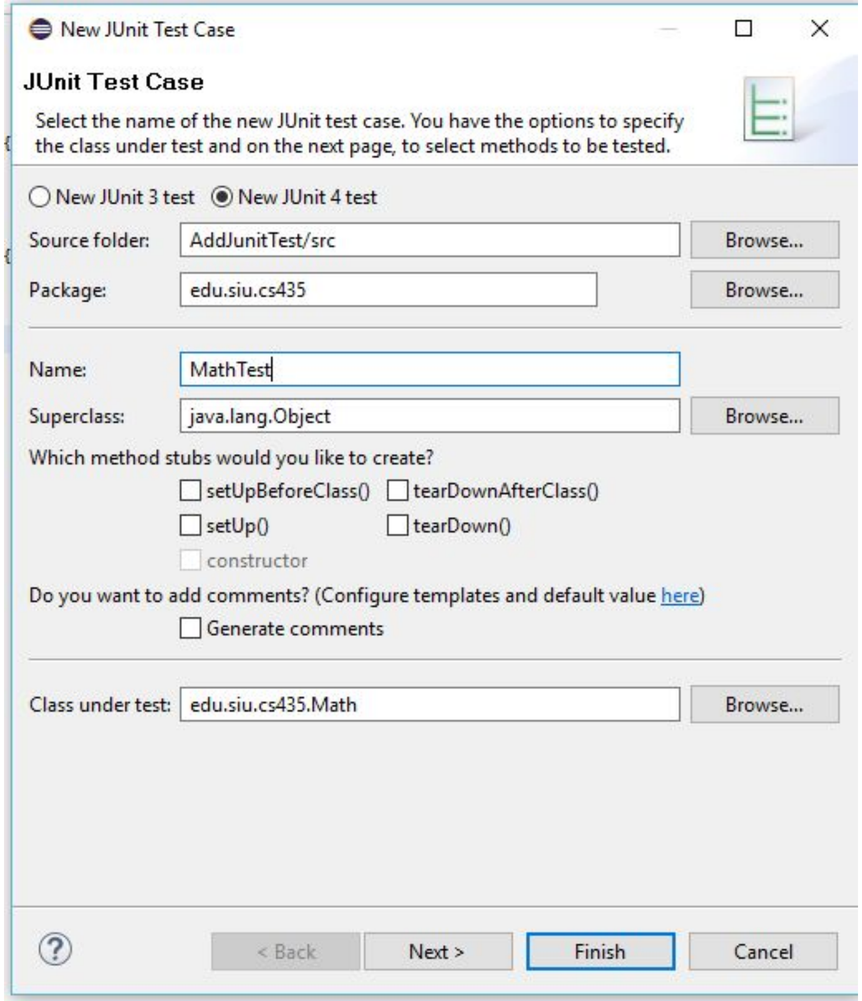
- To add JUnit to an Eclipse project, click:
  - □ **Project → Properties → Build Path → Libraries → Add Library... → JUnit → JUnit 4 → Finish**

- To create a test case:
  - □ right-click a file and choose **New → Test Case**
  - □ or click **File → New → JUnit Test Case**

  - □ Eclipse can create stubs of method tests for you.



Junit in Eclipse



New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

○ New JUnit 3 test ● New JUnit 4 test

Source folder: AddJunitTest/src

Package: edu.siu.cs435

Name: MathTest

Superclass: java.lang.Object

Which method stubs would you like to create?

□ setUpBeforeClass() □ tearDownAfterClass()

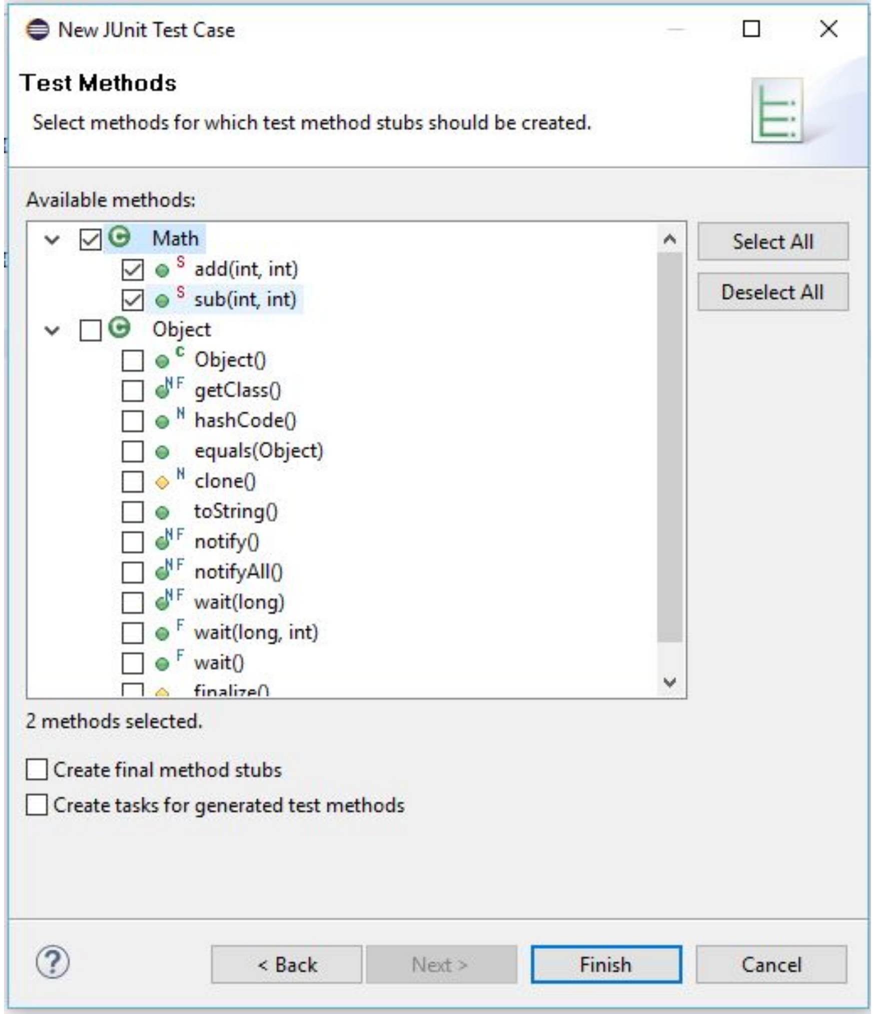
□ setUp() □ tearDown()

□ constructor

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

□ Generate comments

Class under test: edu.siu.cs435.Math



New JUnit Test Case

Test Methods

Select methods for which test method stubs should be created.

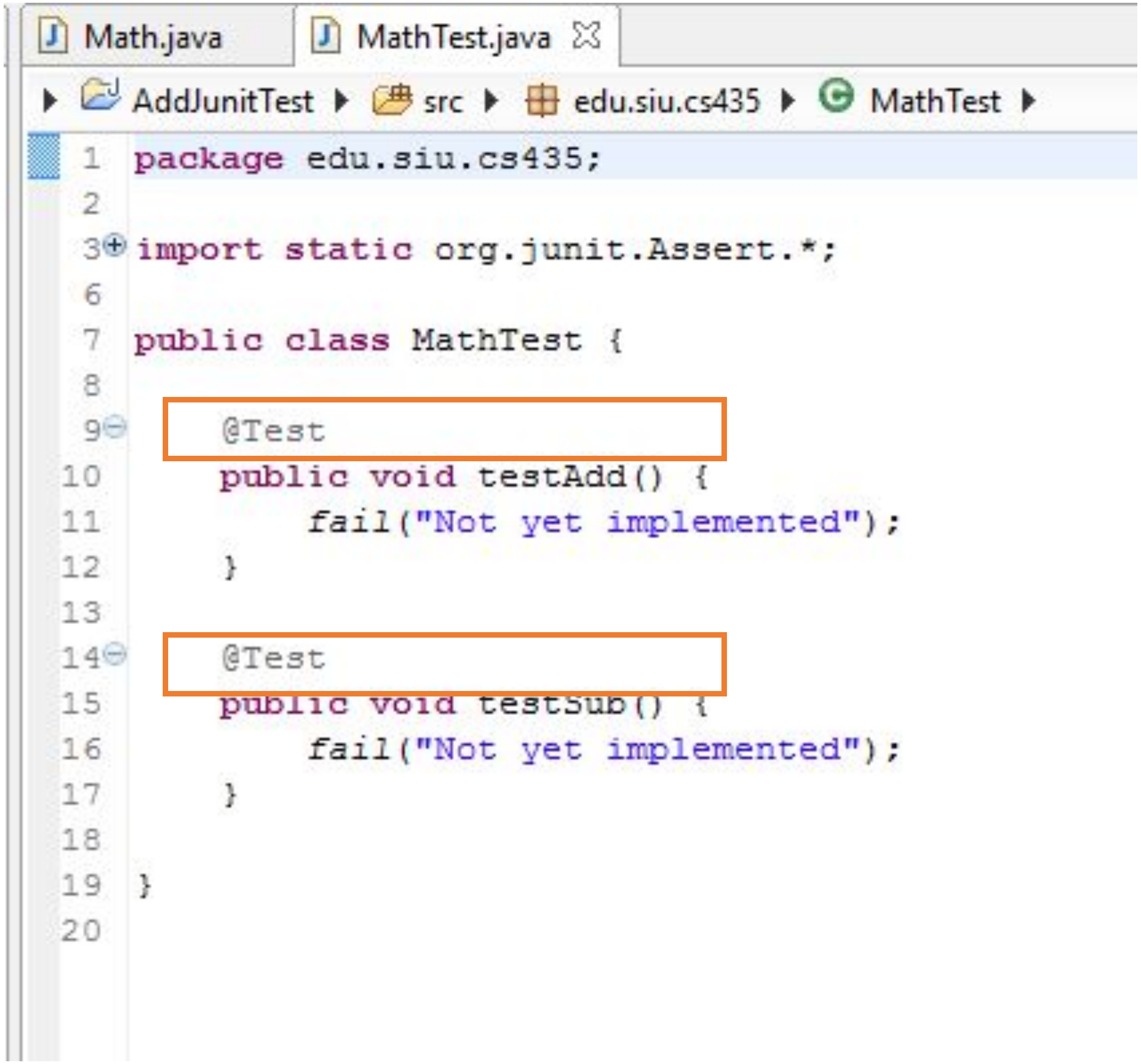
Available methods:

☑ Math
  ☑ add(int, int)
  ☑ sub(int, int)
□ Object
  □ Object()
  □ getClass()
  □ hashCode()
  □ equals(Object)
  □ clone()
  □ toString()
  □ notify()
  □ notifyAll()
  □ wait(long)
  □ wait(long, int)
  □ wait()
  □ finalize()

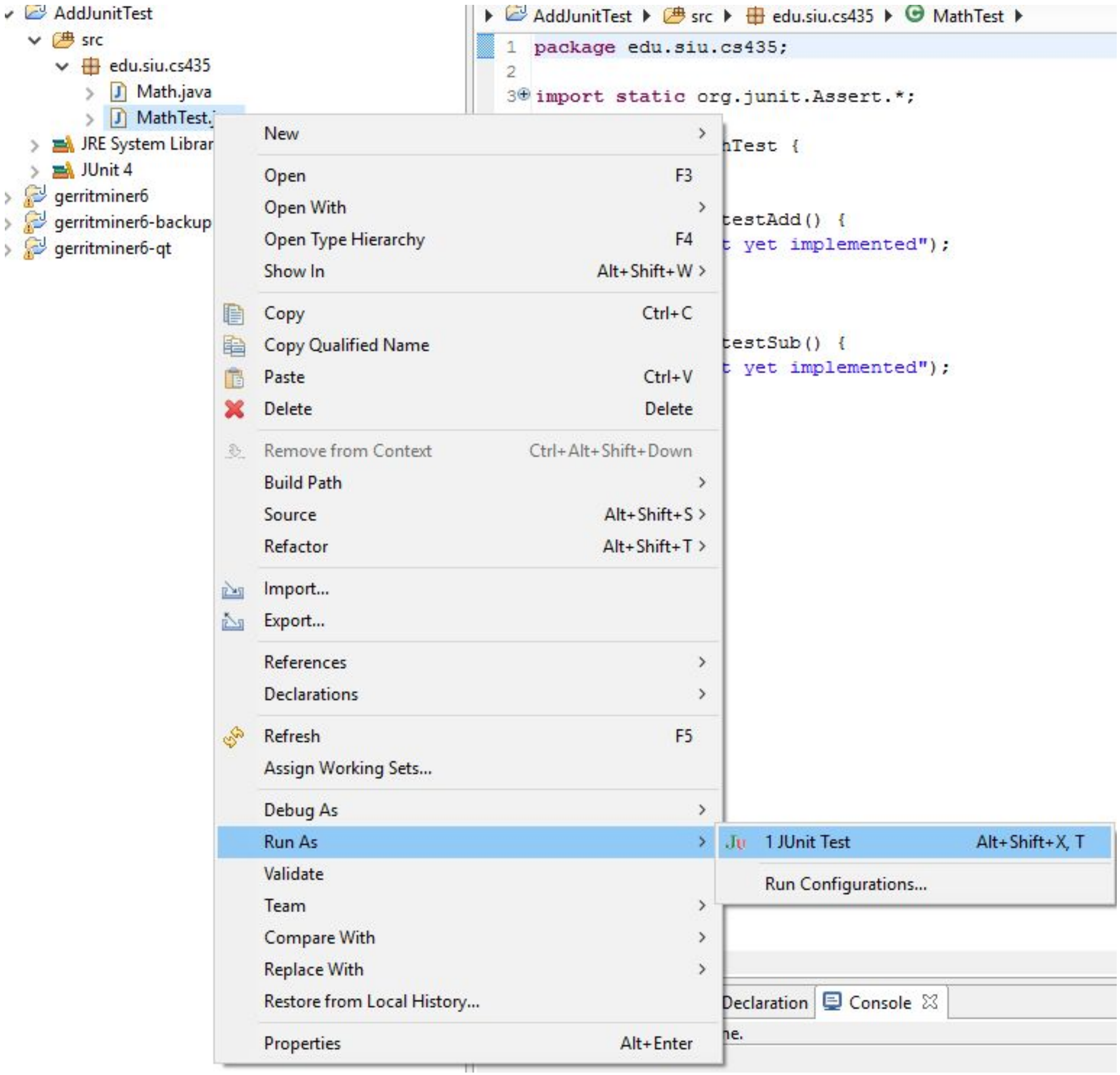
2 methods selected.

□ Create final method stubs

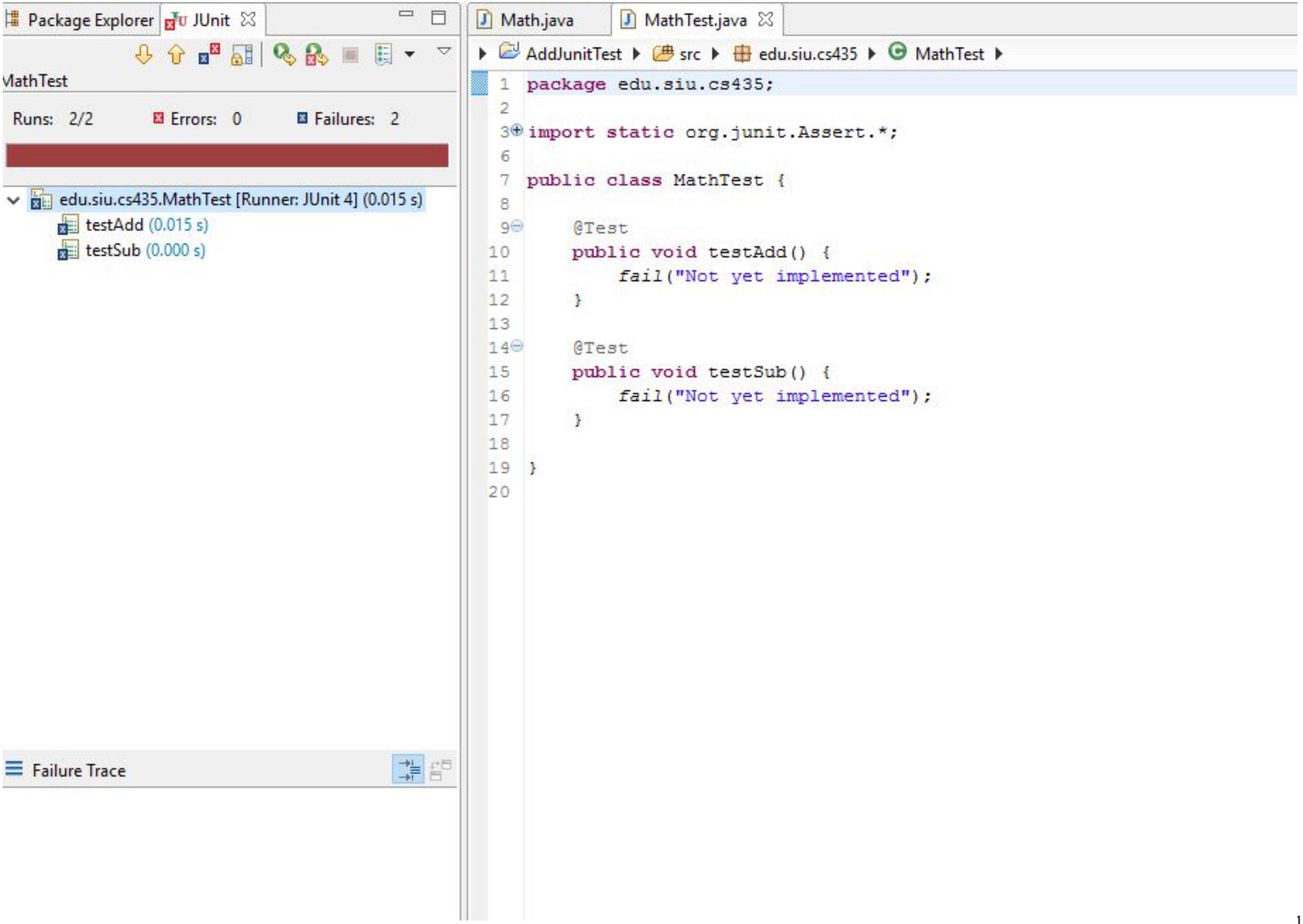
□ Create tasks for generated test methods



```java
package edu.siu.cs435;

import static org.junit.Assert.*;

public class MathTest {

	@Test
	public void testAdd() {
		fail("Not yet implemented");
	}

	@Test
	public void testSub() {
		fail("Not yet implemented");
	}

}

```



```
package edu.siu.cs435;

import static org.junit.Assert.*;

nTest {

testAdd() {
t yet implemented");

testSub() {
t yet implemented");
```

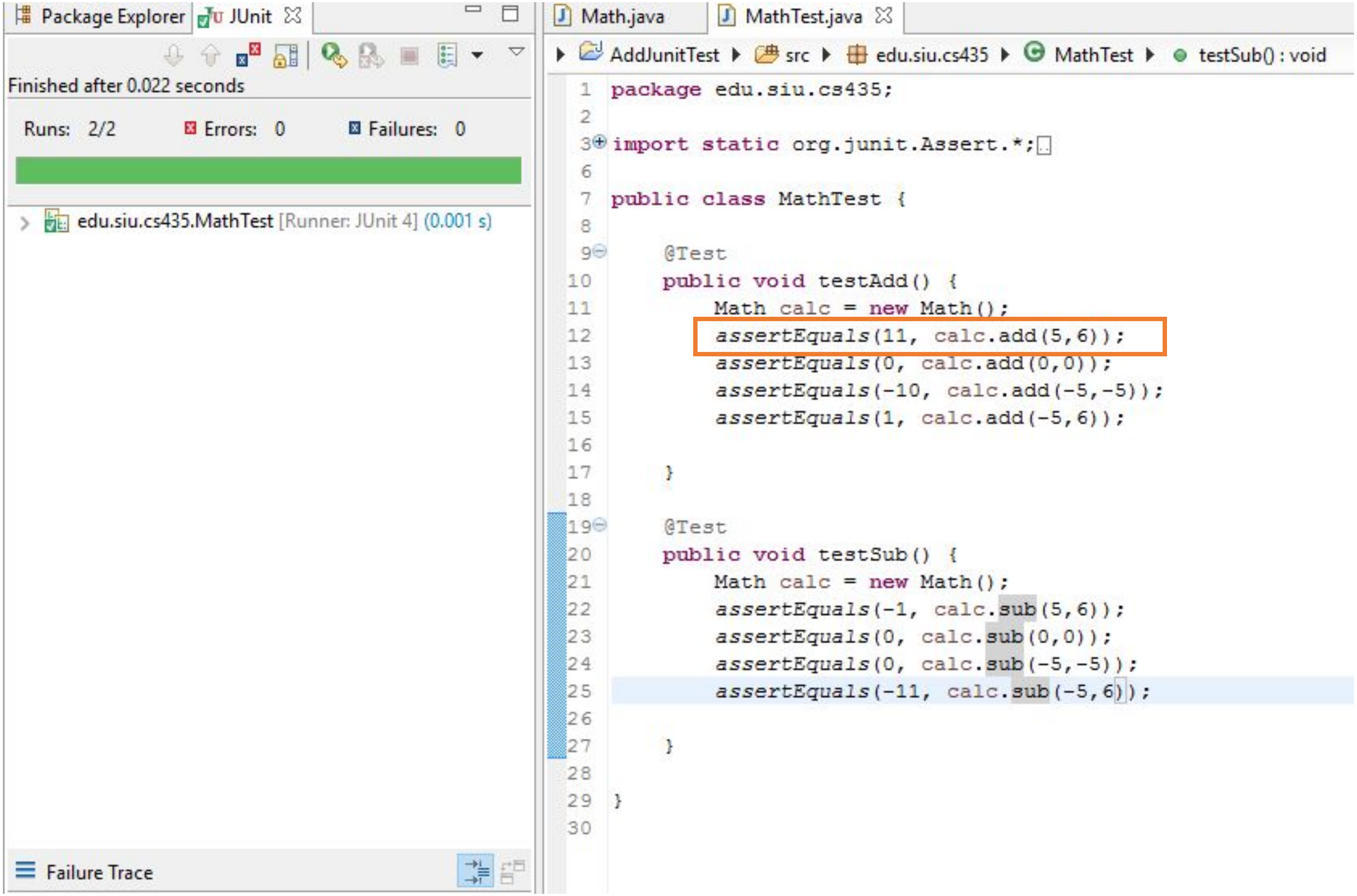


MathTest

Runs: 2/2   Errors: 0   Failures: 2

```
edu.siu.cs435.MathTest [Runner: JUnit 4] (0.015 s)
    testAdd (0.015 s)
    testSub (0.000 s)
```

```java
package edu.siu.cs435;

import static org.junit.Assert.*;

public class MathTest {

    @Test
    public void testAdd() {
        fail("Not yet implemented");
    }

    @Test
    public void testSub() {
        fail("Not yet implemented");
    }

}

```



```
Finished after 0.022 seconds

Runs: 2/2     Errors: 0     Failures: 0

edu.siu.cs435.MathTest [Runner: JUnit 4] (0.001 s)
```

```java
package edu.siu.cs435;

import static org.junit.Assert.*;

public class MathTest {

    @Test
    public void testAdd() {
        Math calc = new Math();
        assertEquals(11, calc.add(5,6));
        assertEquals(0, calc.add(0,0));
        assertEquals(-10, calc.add(-5,-5));
        assertEquals(1, calc.add(-5,6));

    }

    @Test
    public void testSub() {
        Math calc = new Math();
        assertEquals(-1, calc.sub(5,6));
        assertEquals(0, calc.sub(0,0));
        assertEquals(0, calc.sub(-5,-5));
        assertEquals(-11, calc.sub(-5,6));

    }

}

```

JUnit assertion methods

assertTrue(**test**)	fails if the boolean test is false
assertFalse(**test**)	fails if the boolean test is true
assertEquals(**expected**, **actual**)	fails if the values are not equal
assertSame(**expected**, **actual**)	fails if the values are not the same (by ==)
assertNotSame(**expected**, **actual**)	fails if the values *are* the same (by ==)
assertNull(**value**)	fails if the given value is *not* null
assertNotNull(**value**)	fails if the given value is null
fail()	causes current test to immediately fail

- Each method can also be passed a string to display if it fails:
  - e.g. assertEquals(**"message"**, **expected**, **actual**)

Junit Framework

Provides following important features

1. Fixtures
2. Test suites
3. Test runners
4. JUnit classes

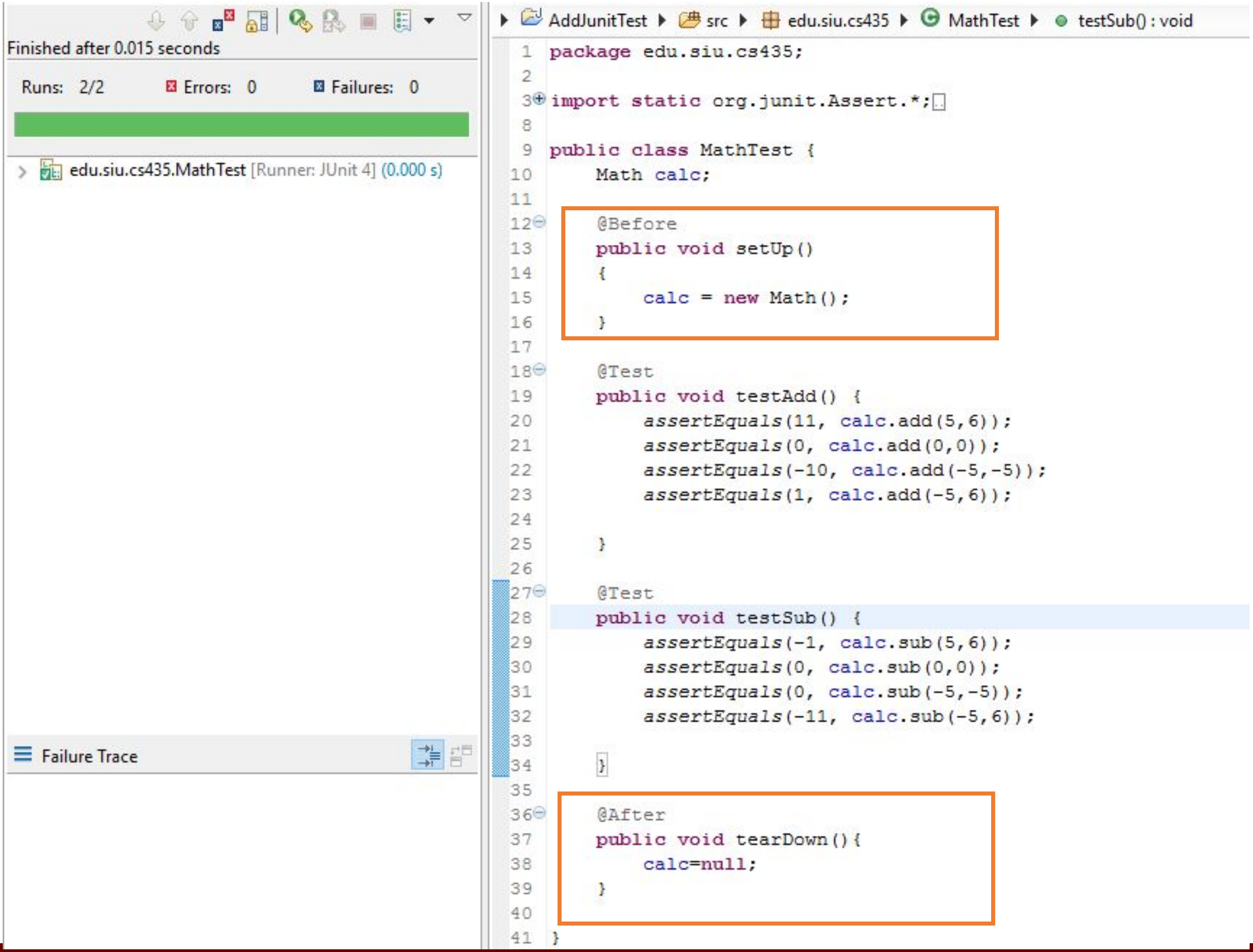
Fixtures

- **Fixtures** is a fixed state of a set of objects used as a baseline for running tests. The purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable.

- @Before
  - □ Performs this function before each test case
  - □ Since each of the tests are independent, each test will receive *its own instance of whatever is created in the @Before methods*

- @Test
  - □ Indicates individual test cases
  - □ Must be present to tell JUnit which methods are tests and which are helpers

- @After
  - □ The teardown after each test case
  - □ Usually need to worry about this only if you have created external resources


AddJunitTest ▸ src ▸ edu.siu.cs435 ▸ MathTest ▸ testSub() : void
Failure Trace

Finished after 0.015 seconds

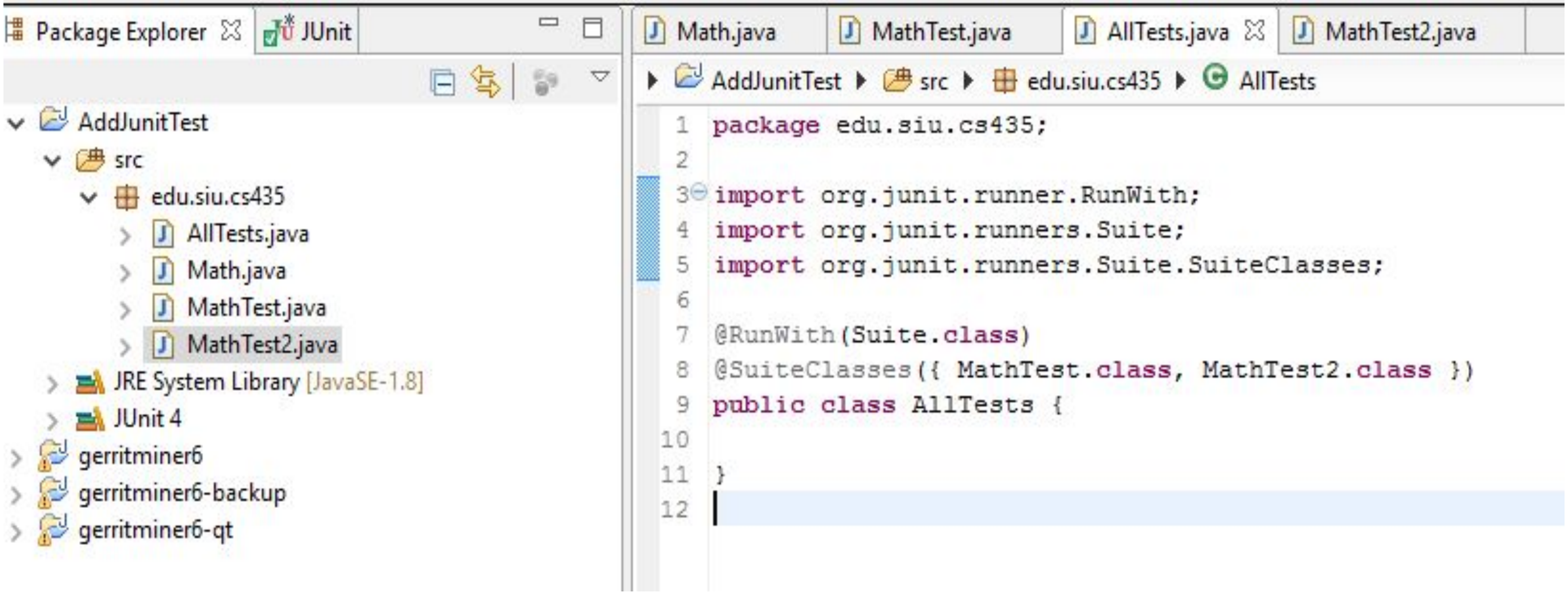
Runs: 2/2 Errors: 0 Failures: 0

edu.siu.cs435.MathTest [Runner: JUnit 4] (0.000 s)

```java
package edu.siu.cs435;

import static org.junit.Assert.*;

public class MathTest {
    Math calc;

    @Before
    public void setUp()
    {
        calc = new Math();
    }

    @Test
    public void testAdd() {
        assertEquals(11, calc.add(5,6));
        assertEquals(0, calc.add(0,0));
        assertEquals(-10, calc.add(-5,-5));
        assertEquals(1, calc.add(-5,6));

    }

    @Test
    public void testSub() {
        assertEquals(-1, calc.sub(5,6));
        assertEquals(0, calc.sub(0,0));
        assertEquals(0, calc.sub(-5,-5));
        assertEquals(-11, calc.sub(-5,6));

    }

    @After
    public void tearDown(){
        calc=null;
    }

}
```

Test Suite

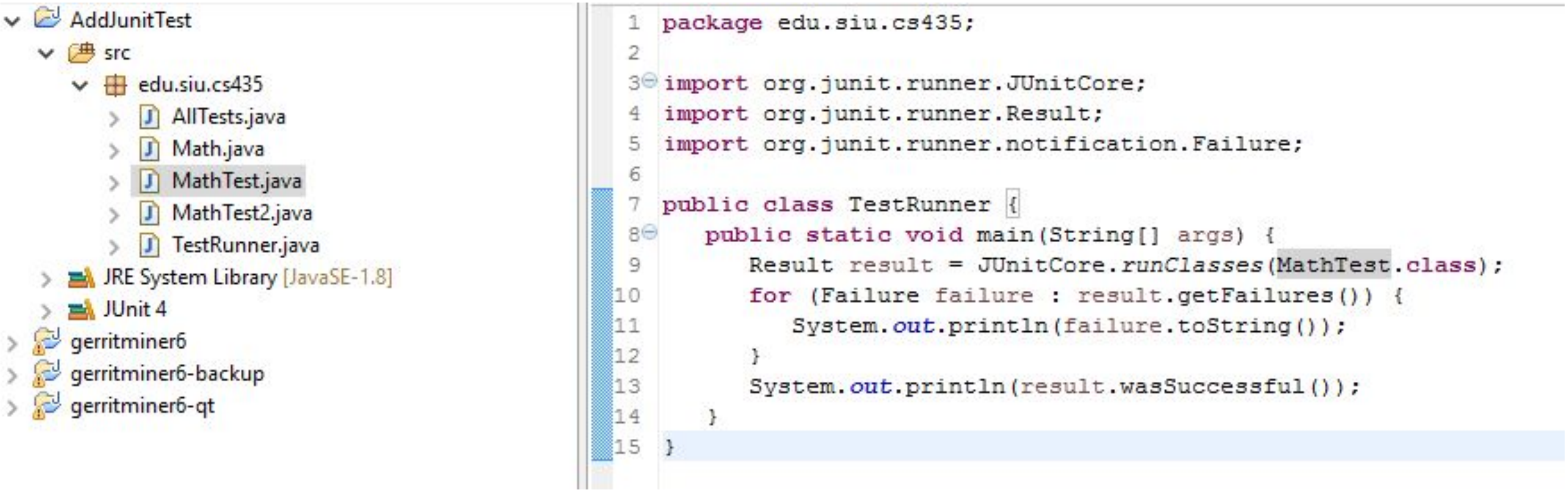
- **Test suite** means bundle a few unit test cases and run it together. In JUnit, both @RunWith and @Suite annotation are used to run the suite test.



```java
package edu.siu.cs435;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ MathTest.class, MathTest2.class })
public class AllTests {

}
```

Test Runner

- **Test runner** is used for executing the test cases.



```java
package edu.siu.cs435;

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(MathTest.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        System.out.println(result.wasSuccessful());
    }
}
```

JUnit exercise

Given a `Date` class with the following methods:

```
public Date(int year, int month, int day)
public Date()                    // today
public int getDay(), getMonth(), getYear()
public void addDays(int days)    // advances by days
public int daysInMonth()
public String dayOfWeek()        // e.g. "Sunday"
public boolean equals(Object o)
public boolean isLeapYear()
public void nextDay()            // advances by 1 day
public String toString()
```

- Come up with unit tests to check the following:
  - That no `Date` object can ever get into an invalid state.
  - That the `addDays` method works properly.
    - It should be efficient enough to add 1,000,000 days in a call.

What's wrong with this?

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 2);
        assertEquals(d.getDay(), 19);
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 3);
        assertEquals(d.getDay(), 1);
    }
}
```

Well-structured assertions

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(2050, d.getYear());   // expected
        assertEquals(2, d.getMonth());     // value should
        assertEquals(19, d.getDay());      // be at LEFT
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals("year after +14 days", 2050, d.getYear());
        assertEquals("month after +14 days", 3, d.getMonth());
        assertEquals("day after +14 days", 1, d.getDay());
    }   // test cases should usually have messages explaining
}       // what is being checked, for better failure output
```

Expected answer objects

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals(expected, d);  // use an expected answer
    }                               // object to minimize tests

                                    // (Date must have toString
    @Test                           //  and equals methods)
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }
}
```

Naming test cases

```
public class DateTest {
    @Test
    public void test_addDays_withinSameMonth_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days", expected, actual);
    }
    // give test case methods really long descriptive names

    @Test
    public void test_addDays_wrapToNextMonth_2() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
    // give descriptive names to expected/actual values
}
```

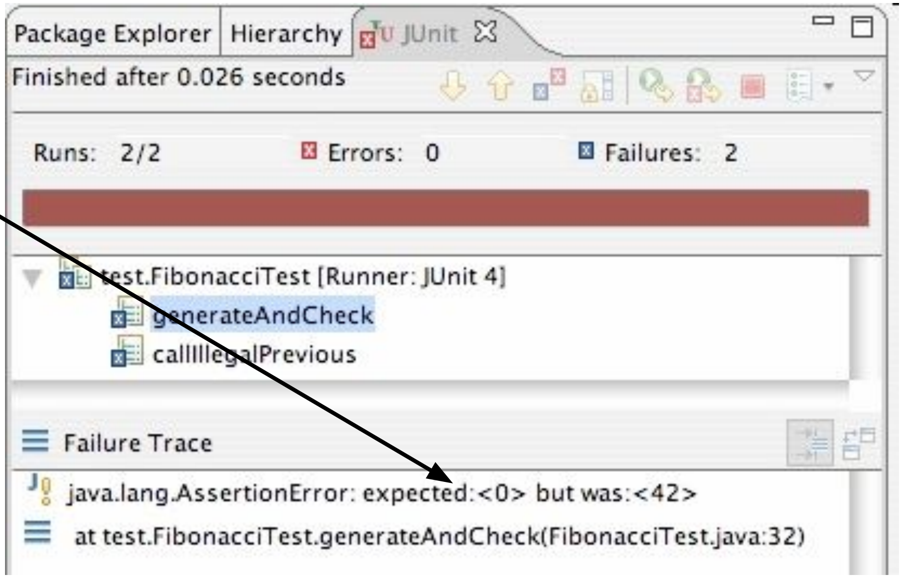
What's wrong with this?

```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals(
            "should have gotten " + expected + "\n" +
            " but instead got " + actual\n",
            expected, actual);
    }
    ...
}
```

Good assertion messages

```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals("adding one day to 2050/2/15",
            expected, actual);
    }
    ...
}
```

```
// JUnit will already show
// the expected and actual
// values in its output;
//
// don't need to repeat them
// in the assertion message
```



Finished after 0.026 seconds

Runs: 2/2   Errors: 0   Failures: 2

```
test.FibonacciTest [Runner: JUnit 4]
    generateAndCheck
    callIllegalPrevious
```

```
java.lang.AssertionError: expected:<0> but was:<42>
at test.FibonacciTest.generateAndCheck(FibonacciTest.java:32)
```

Tests with a timeout

```
@Test(timeout = 5000)
public void name() { ... }
```

□ The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 2000;
...

@Test(timeout = TIMEOUT)
public void name() { ... }
```

□ Times out / fails after 2000 ms

Pervasive timeouts

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_withinSameMonth_1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days", expected, d);
    }


    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_wrapToNextMonth_2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }

    // almost every test should have a timeout so it can't
    // lead to an infinite loop; good to set a default, too
    private static final int DEFAULT_TIMEOUT = 2000;
}
```

Testing for exceptions

```
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```

- Will pass if it *does* throw the given exception.
  - If the exception is *not* thrown, the test fails.
  - Use this to test for expected errors.

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayIntList list = new ArrayIntList();
    list.get(4);   // should fail
}
```

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

Other Unit Testing Frameworks

- NUnit (http://www.nunit.org/)
  - .NET languages

- PHP Unit (https://phpunit.de/)
  - PHP