SEG 3103 Introduction to JUnit

JUnit

- JUnit (http://www.junit.org)
  - Open source regression testing framework built by Erich Gamma (author of the most popular Design Patterns book and one main designer of Eclipse) and Kent Beck (creator of Extreme Programming (XP) and one author of the Agile Manifesto).
  - Used for unit testing in Java. Other xUnit frameworks exist for other languages.
  - The core Eclipse distribution includes a JUnit plugin
- You may be used to debugging through your code to test it or simply inserting output statements in your code to verify that it is working properly. The main problem with this approach is that it relies on human eyes to verify the correctness of the tested software. Furthermore, if you change the code, all your manually inserted tests will have to be repeated.
- JUnit automates testing. You write your tests once and can run them as often as you like.

Concepts

Test case

- A test case is a small unit of code that tests a specific scenario.
- A test case includes one or more assertions that must be true for the test case to pass.
- For example, we could test that the `java.util.Collection.clear()` method actually removes all element from a `Collection` by verifying that the number of elements in the `Collection` is zero afterward.
- We are comparing *expected results* with *actual results*.
  - If they are the same, we say the test was successful.
  - If they are not the same, we say the test failed.
  - If an uncaught exception occurred, we say an error occurred.

Test fixture

- When you have many similar test cases, which work with the same set of objects for example, you can avoid redundant code by creating a test fixture.
- A test fixture helps you provide a uniform initialization and cleanup mechanism, so you don't have to repeat it in every test case.

Test runner

- A test runner is the software framework that runs the tests and displays the results.
- The base JUnit comes with several textual runners. However, since we will be using Eclipse as our development environment, the test runner we will use is the one bundled in the Eclipse JUnit plug-in.

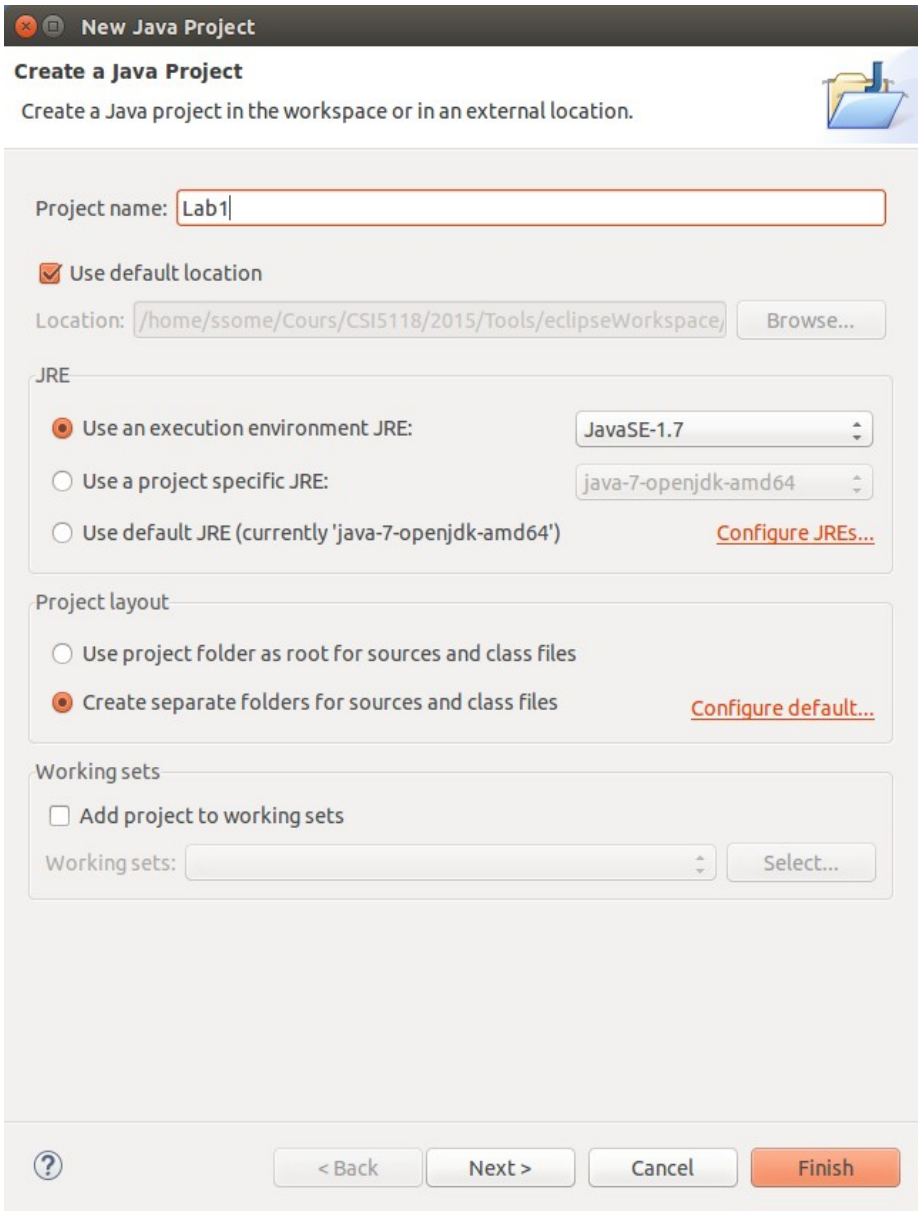
Lab

We are going to create a small class and experiment with testing it. At this point, the intention will be to try out JUnit rather than to do a proper test of a class. In this spirit, we are going to deliberately create some tests

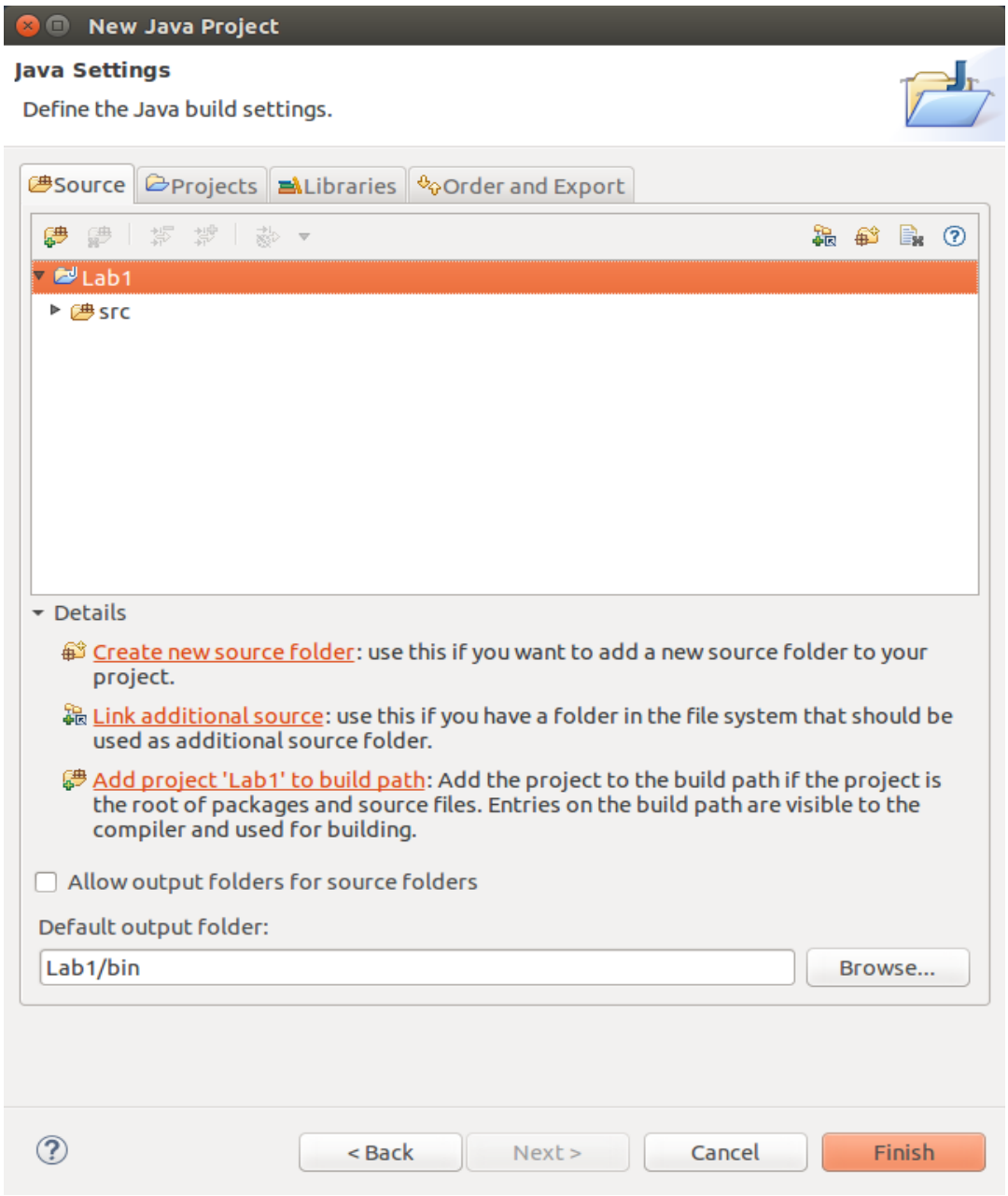
Initial project setup

1. Create a new Java project. When creating the project, be sure the following is set up in the new project wizard:

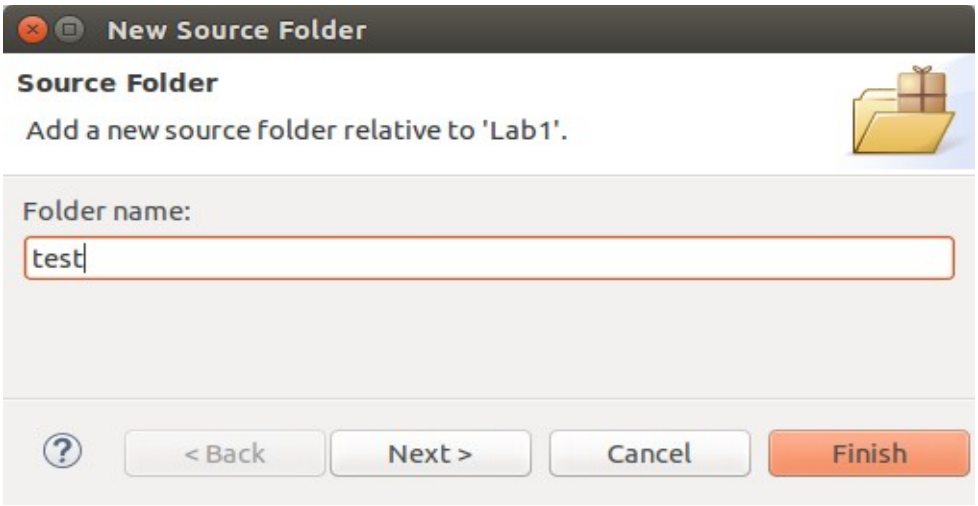
    - On the "Create a Java Project" screen, enter a project name, select the option "Create separate source and output folders." The exact version of the JRE is not important, but it should be version 1.5.0 or greater. Then, click Next.



    - On the "Java Settings" screen, click on "Create a new source folder".

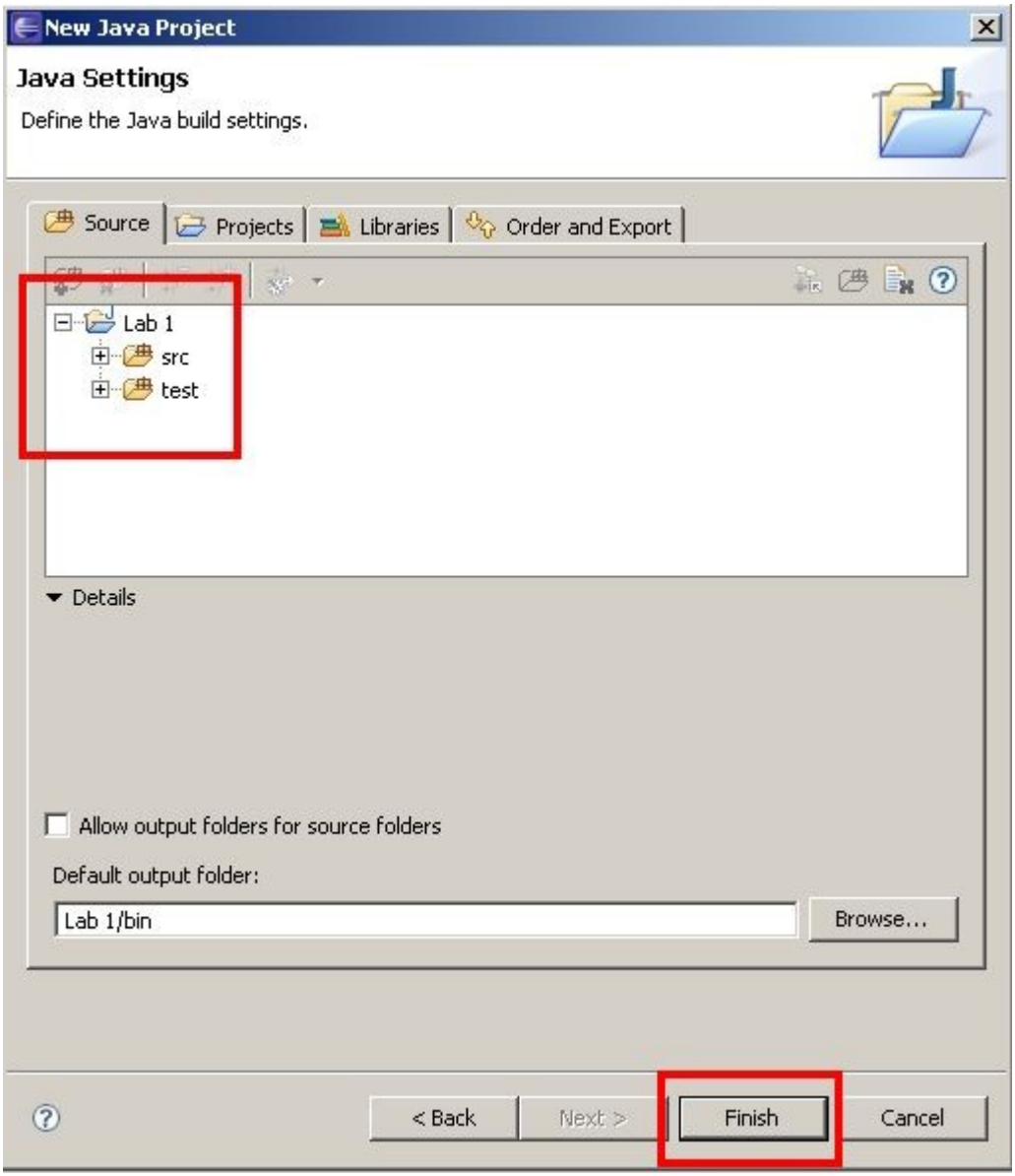


- Name the new source folder `test`. Click Finish.



The result should be two source folders, `src` and `test`. We are going to store the

code to be tested in the `src` folder, and the test cases in the `test` folder.



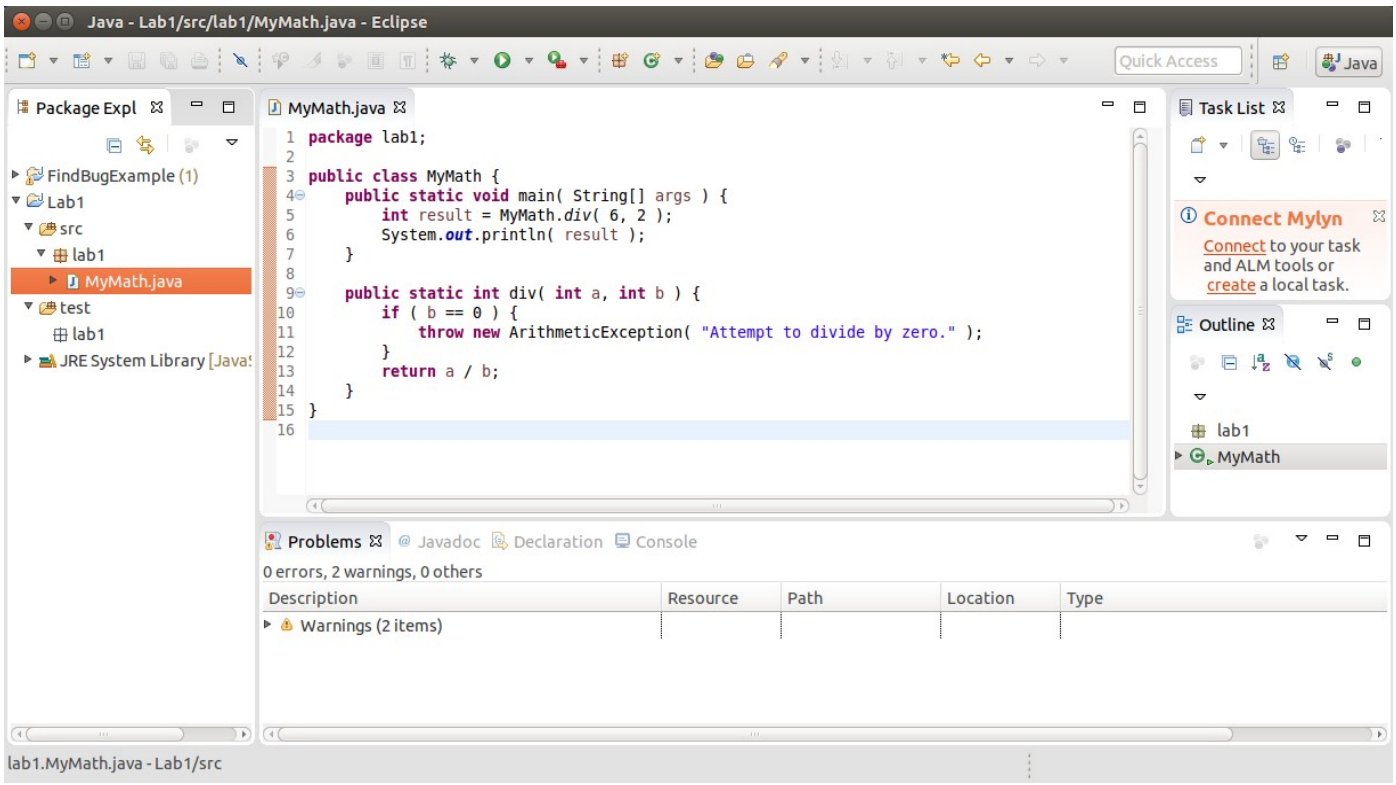
2. Inside each of those folders, create a package named `lab1`. Even though the files will be stored in different folders, Java considers that classes in either folder will be in the `lab1` package.

    - The result is that our test case classes will be effectively in the same package as the classes to be tested for the purposes of access and imports, while still being stored separately.

Create a class to test

3. Create the class **MyMath** within the **lab1** package of the **src** folder. Enter the following code:

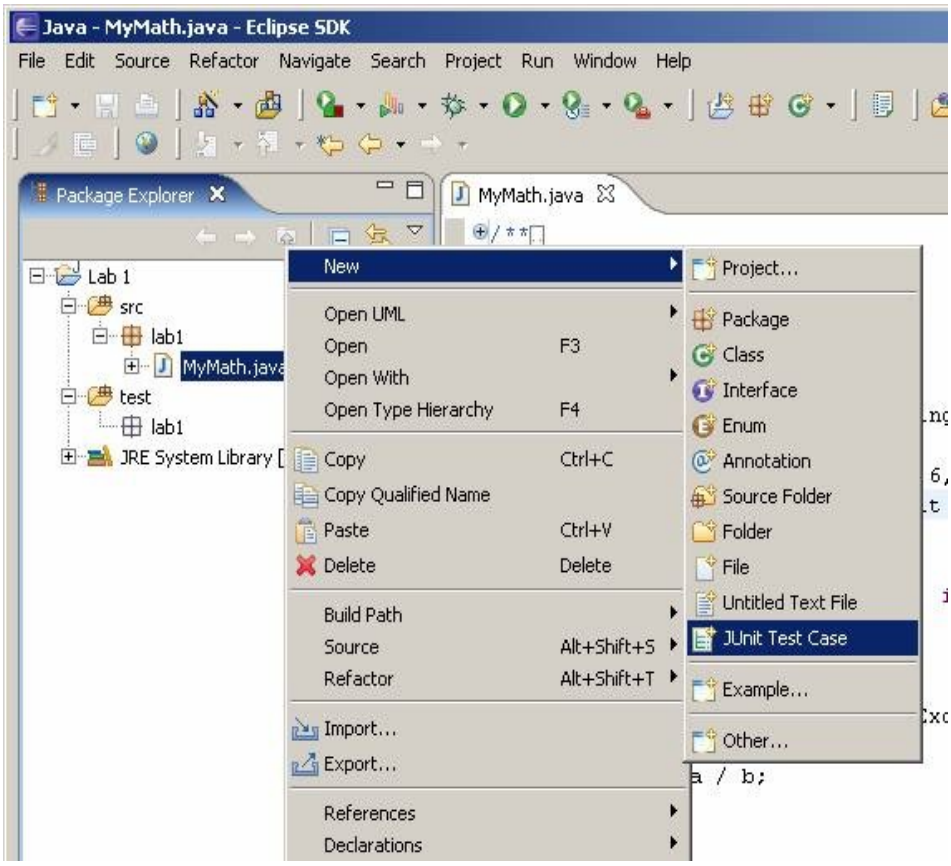
```
package lab1;

   public class MyMath {
      public static void main( String[] args ) {
            int result = MyMath.div( 6, 2 );
            System.out.println( result );
   }

   public static int div( int a, int b ) {
      if ( b == 0 ) {
            throw new ArithmeticException( "Attempt to divide by zero." );
      }
      return a / b;
   }
}
```



- The class has one static method, **div**, which performs integer division. The **main** method shows how the **div** method could be called to divide 6 by 3.

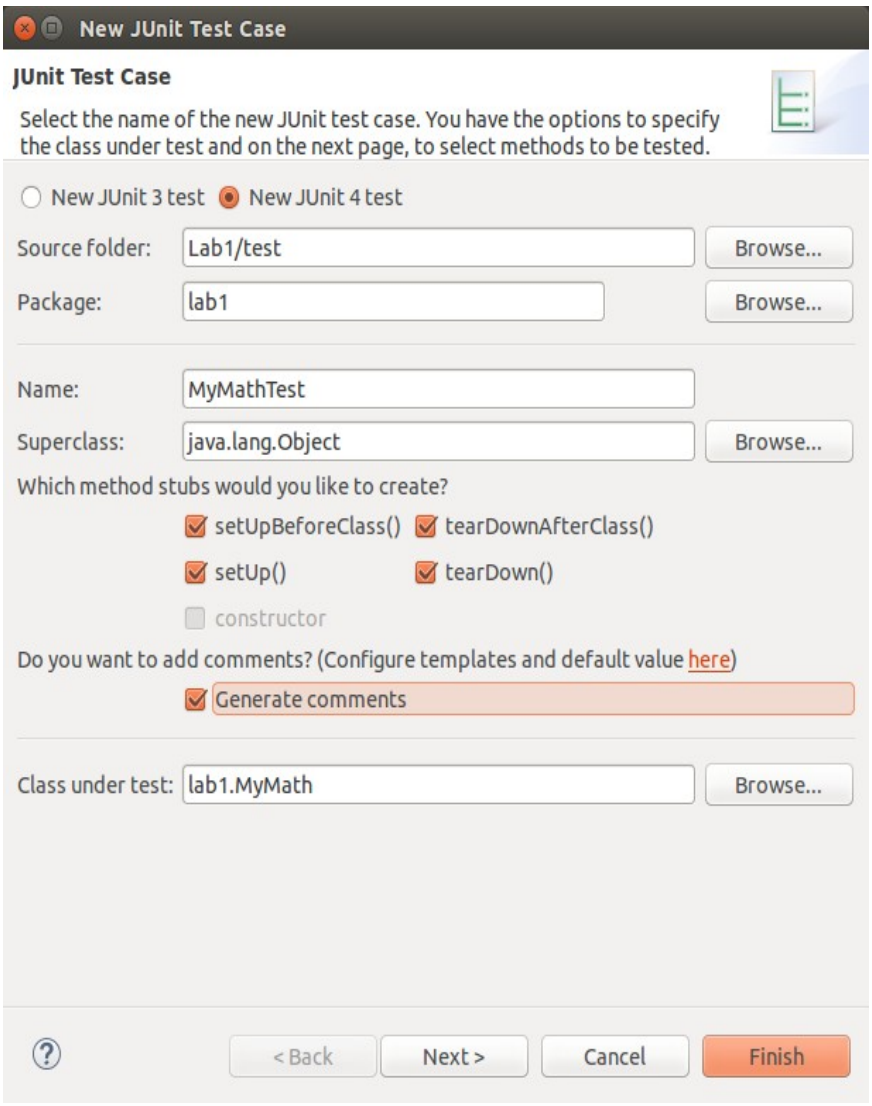
Create a test case class

4. Select the class **`MyMath`**, and right-click to get the popup menu. Select New > JUnit Test Case.

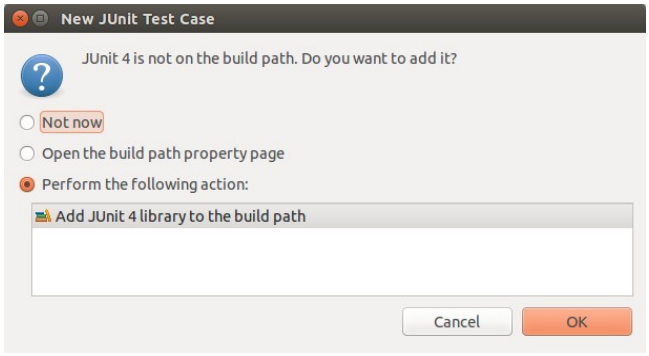


5. When the new test case wizard appears:

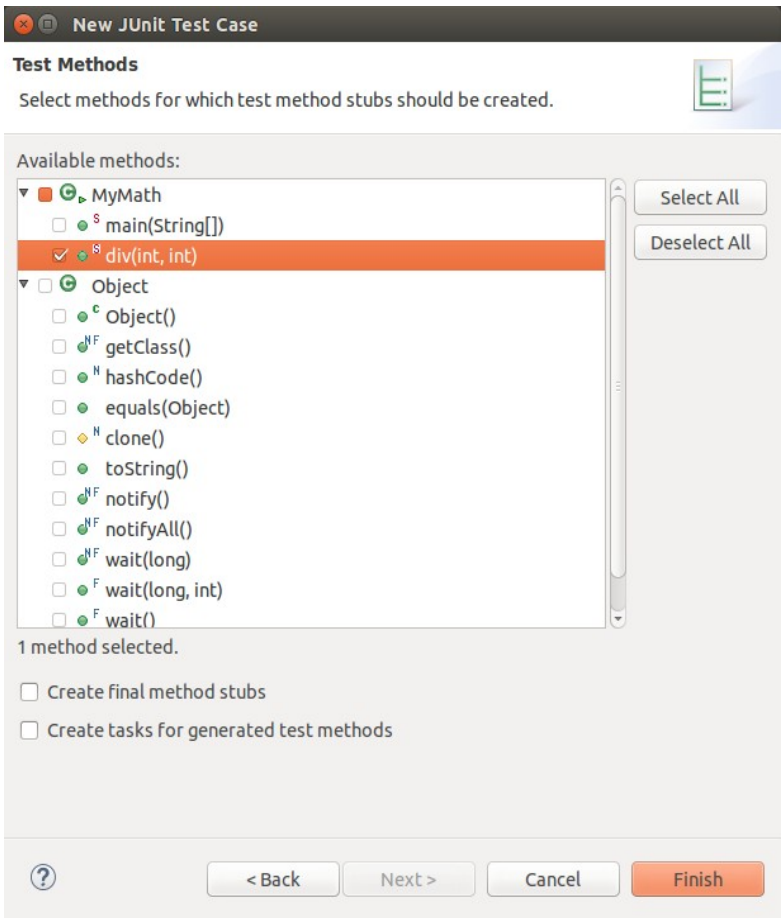
   - Select New JUnit 4 test
   - Select the source folder of the test case to be **`lab1/test`**. To do this, click on the Browse... button on the same line and select the **`test`** folder.
   - In the section "Which method stubs would you like to create?" check all of **`setUpBeforeClass()`**, **`tearDownAfterClass()`**, **`setUp()`**, and **`tearDown()`**. This will create methods with the annotations **`@BeforeClass`**, **`@AfterClass`**, **`@Before`**, and **`@After`**, respectively.
   - If you would like automatically generated comments, check the "Generate comments" box.



- Click **Finish** and **OK** to add the JUnit4 library to the project. This will ensure the JUnit framework is on the class path.



- Click Next



6. Select the method `div(int,int)`, and then click Finish.

7. A new file `MyMathTest.java` will be created with a new class `MyMathTest`. The file will be opened in a new editor window, and the contents should resemble the following code:

```
package lab1;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

/**
 * @author ssome
 *
 */
public class MyMathTest {

	/**
	 * @throws java.lang.Exception
	 */
	@BeforeClass
	public static void setUpBeforeClass() throws Exception {
	}

	/**
```

```
     * @throws java.lang.Exception
     */
    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }

    /**
     * @throws java.lang.Exception
     */
    @Before
    public void setUp() throws Exception {
    }

    /**
     * @throws java.lang.Exception
     */
    @After
    public void tearDown() throws Exception {
    }

    /**
     * Test method for {@link lab1.MyMath#div(int, int)}.
     */
    @Test
    public void testDiv() {
          fail("Not yet implemented");
    }

}
```

- Empty fixture methods are created, with the appropriate headers and annotations.
- An empty test case method `testDiv()` is created. It will have one line, which automatically causes an unconditional failure. The idea is that if you haven't written the test case yet, it should fail to remind you that it is empty and isn't testing anything useful.

At this point, a warning is in order. The "new JUnit test case" wizard will create one empty test case method for each method in the class to be tested. Do **NOT** assume that there should be one test case method per original method. In fact, you may need several test case methods for adequate testing of each original method. Conversely, a `getXXX()` method that just returns the value of a private instance variable `xxx` is really too simple to break – especially if you used the Eclipse "generate getters and setters" function to write the method automatically – and is unlikely to discover any bugs. Such methods do not typically require having a corresponding test case. The code will likely get used as part of some other test case anyway.

In Summary: do not assume a one-to-one correspondence between methods and test cases.

For example, if you are testing a method that does the mathematical absolute value method, you would likely have (at least...) 3 test methods: one where the input is greater than zero (strictly positive), one where the input is less than zero (strictly negative), and one where the input is exactly zero. These are three different cases of interest for such a method.

Create some tests for MyMath:

Now that you have seen how to set up a project, and create the skeleton of a JUnit test case class, the next step is to create some tests. Since the test case wizard created a template method for you, create three additional copies of the method, for a total of four methods. You will have to rename the methods eventually to have different names.

Create four tests for the `div(int,int)` method. The test purposes should be as follows:

1. Create a test that is intended to pass, and name it `testDivPass()`. For example, you could create a test that is the equivalent of running the main method that is provided with the class: expect that if 6 is divided by 3, the result is 2.

   ```
   @Test

   public void testDivPass() {

         int number1 = 6;
         int number2 = 3;
         int expected = 2;
         Assert.assertEquals(expected, MyMath.div(number1,number2));
   }
   ```

   Make sure the imported class is `org.junit.Assert;`

2. Create a test case that is intended to fail, and name it `testDivFail()`.[Normally, you would write a test case with the intention that it should pass if the implementation being tested is correct, but for exploring the tool, we are going to waive this assumption.]. A test case fails if a JUnit Assert method is called, and the assertion is not true. The test should perform a legitimate division, but have an incorrect expected result in the test case.

3. Create a test case that attempts to divide by zero, and expects a result of zero. Name this method `testDivError()`. What will happen is that an `ArithmeticException` will be thrown during the execution of the `div(int,int)` method. Since the test case would not contain an assertion failure, when the test case runs, it should result in an Error verdict. The intention of an Error verdict is to show that the intended purpose of the test was not achieved. This distinguishes a test case from one that **did** achieve its test purpose, but **did not** have the expected result – a failure. Typically, a verdict of Error could mean that a test could not be set up properly, or that something unexpected happened during the test that should be investigated.

4. Create a test case that checks to see if an arithmetic exception is thrown when attempting to divide by zero, and name the method `testDivException()`. The intention of this test case is that it should **pass** if an arithmetic exception is thrown within the `div()` method. The test case should **fail** if the call to `div()` returns a value.

   This type of test case is often confusing. A typical user wouldn't want to see an exception, and it would be considered as a problem. As a test person investigating unusual situations, sometimes an exception is the desired outcome of a test case because it shows that the software under test is detecting unusual situations.

In this case, we want the method we are testing to throw an exception if there is an attempt to divide by zero. In fact, the software would not be functioning properly if it didn't throw an exception, and that is what we want to discover with a test failure.

5. We are going to add some `println()` statements so that the order of execution is observed in the console window. For each of the methods annotated with **`@BeforeClass`**, **`@AfterClass`**, **`@Before`**, and **`@After`**, add a statement to print a message that indicates the annotation on the method that has been called. For example:

```
System.out.println( "@BeforeClass" );
```

NOTE: Using `println()` statements in test cases should only be done for the purposes of debugging test cases or logging events during test cases. They should **never** be part of a test in the sense that someone has to read the console output to determine or verify the test verdict.

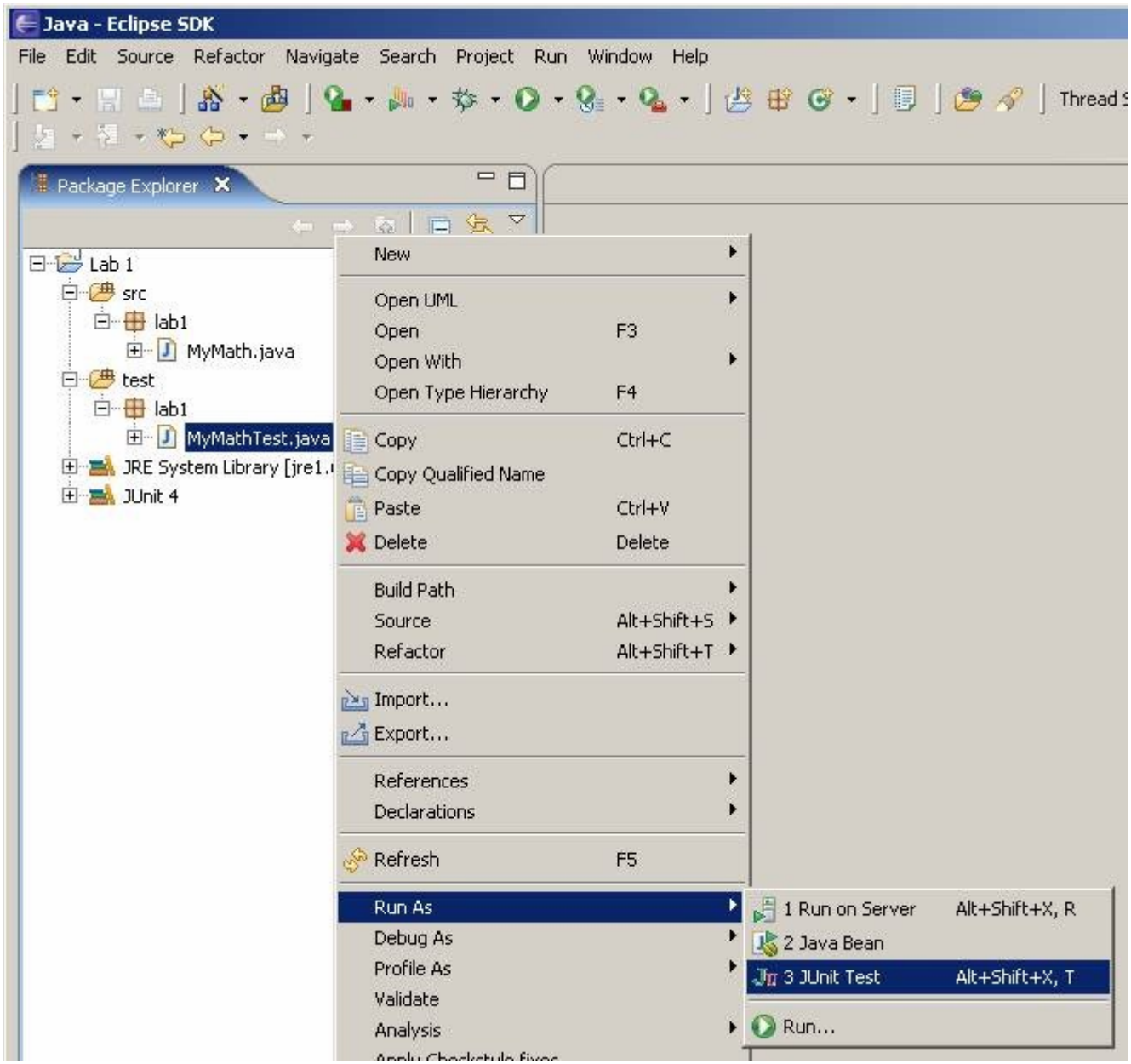
6. Add a `println()` statement for each test case that shows the annotation and name of the test. The statement should be the first statement in the method. For example:

```
System.out.println( "@Test: testDivPass" );
```

Running JUnit tests.

1. To run a JUnit test, select the file(s) containing the tests you would like to run, and then right-click to see the pop-up menu. Select Run As > JUnit Test.

    - Eclipse will detect that the file contains a JUnit test case from the annotations, and provide an option to run the class as a JUnit test as opposed to a Java application.

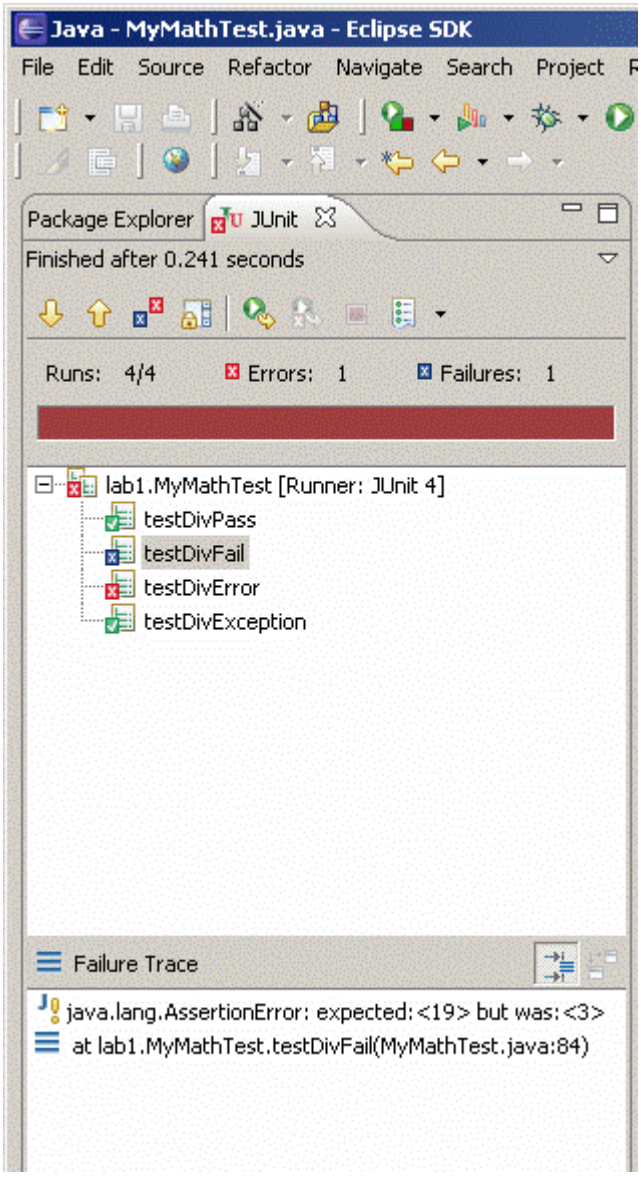


2. A new view will open with the JUnit execution results. If all the tests pass, the famous JUnit results bar would appear as a green bar. If any test has either a failure or error verdict, the bar will be red. Since we have set up both a failure and an error, the bar should be red.

    - To quote the JUnit.org web site: "Keep the bar green to keep the code clean!"
    - The time taken for the test run is shown at the top of the view panel.
    - The number of tests run, and the numbers of errors and failures appear above the bar.
    - For each test, the verdict is shown with an icon. The green check mark indicates a pass. The red and blue X marks failures or errors.
    - You can select each test to see the failure trace. The first test that fails is pre-selected.
        - In the example shown below, the failure message is `expected <19> but was <3>`. These values are taken from the first and second values respectively from

the `assertEquals( expected, actual )` statement that was not valid, so that the method was called with the values `assertEquals( 19, 3 )`. If you swap the order of your expected and actual values, you will get misleading error messages.

- The line number in the Java file where the test failed is also provided. You can use this to locate the assertion that failed by double-clicking on this window entry.



3. In the console window, you should see output similar to the following, if you added the `println()` statements.

```
@BeforeClass
@Before
@Test:  testDivPass
@After
@Before
@Test:  testDivFail
@After
@Before
@Test:  testDivError
```

```
@After
@Before
@Test:  testDivException
@After
@AfterClass
```