LearnQuest

Presents

Test Driven Development
Working with JUnit

www.LearnQuest.com

JUnit Assumptions

- Assumptions about the code to be developed:
  - ✓ Component is designed according to OOP/OOD best practices
  - ✓ Component functionality is clearly defined during design
- Assumptions about state of the development:
  - ✓ Architecture of the system is defined: dependencies between components are known
  - ✓ Acceptance tests exist at the system level so we know how the component under development should behave
  - ✓ The levels of acceptable risk and quality are defined

The Outside-In Principle

- TDD assumes code development follows the outside-in principle:
  - ✓ Interfaces are designed earlier during the design phase.
  - ✓ Interfaces describe all the functionality of a component
  - ✓ Classes are written to implement interfaces
- Because interfaces are defined during design:
  - ✓ They remain stable and do not change during code development
  - ✓ Interfaces can change if requirements change
  - ✓ Interfaces can change if the application architecture changes
  - ✓ All interactions take place through a component's interface:
    - Therefore, a component can be fully functionally tested through its interface methods

Design by Contract

- Interfaces establish contracts between the component and clients that call the methods
  - ✓ Each interface method has three constraints as part of its contract
    - Preconditions: conditions that must be true before a method can be allowed to execute
    - Postconditions: conditions that must be true after the method executes
    - Invariants: conditions that must not change as a result of executing the method
  - ✓ Tests are easier to implement for code when these constraints are known

LearnQuest

Command-Query Separation

- Interfaces establish contracts between the component and clients that call the methods
  - ✓ Each interface method has three constraints as part of its contract
    - Preconditions: conditions that must be true before a method can be allowed to execute
    - Postconditions: conditions that must be true after the method executes
    - Invariants: conditions that must not change as a result of executing the method
  - ✓ Tests are easier to implement for code when these constraints are known

TDD Testing Assumptions

- TDD assumptions about how testing should be done:
  - ✓ Test execution should always be automated
  - ✓ There should be no test code inside the production code
  - ✓ Testing is not debugging: those tools already exist elsewhere
  - ✓ There should only be one copy of the application code, there should not be a “test” version of the production code
  - ✓ The presence of test code should not impact the design of the production code

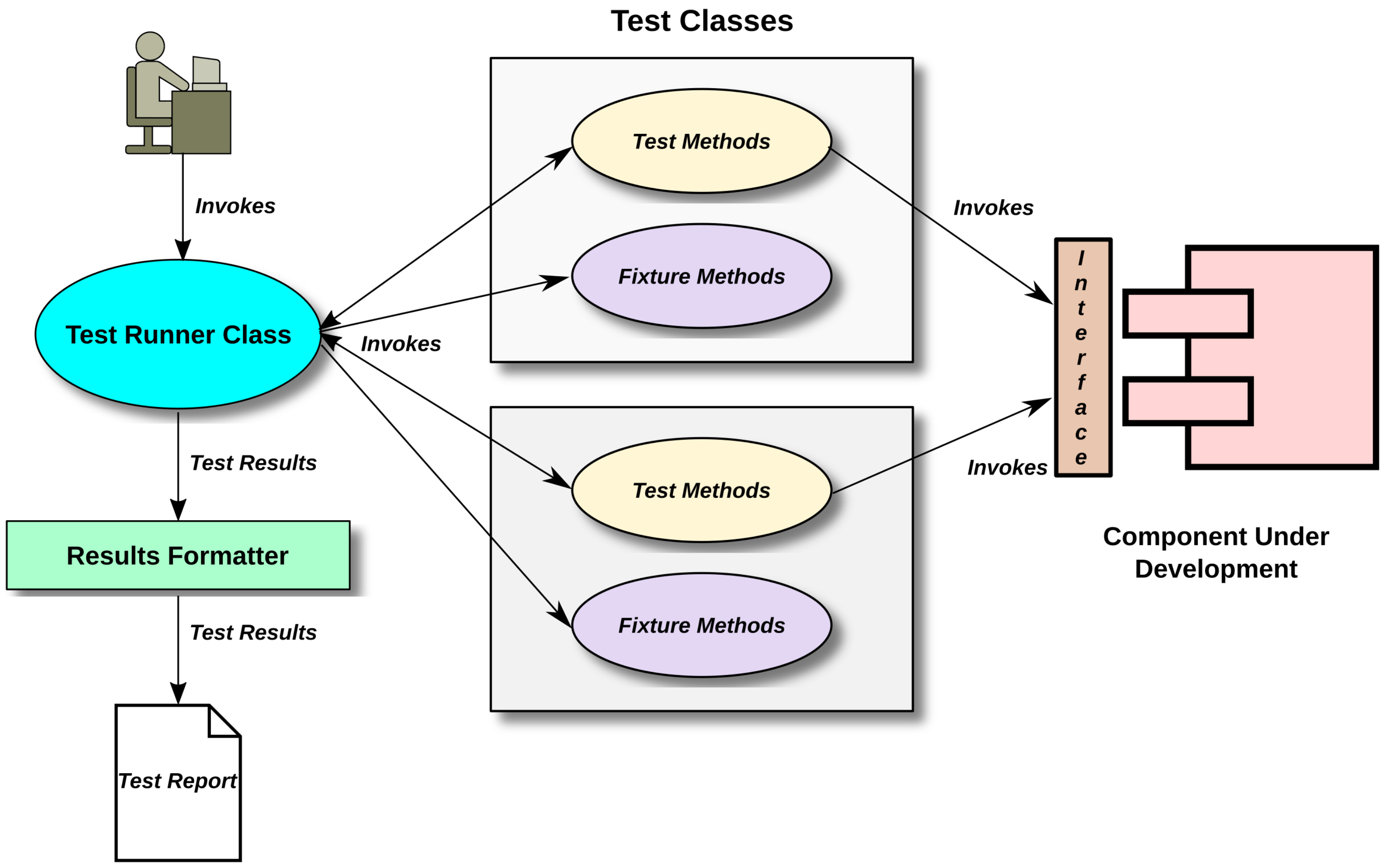
Some JUnit Background

- Kent Beck originally developed SUnit – a testing framework for Smalltalk programmers in the mid 1990s
- Beck and Eric Gamma converted it to a Java framework and called it JUnit on a flight from Zurich to Atlanta in 1997
- The original architecture has come to be known as xUnit and is the basis for many language ports (CppUnit for C++) for example
- The xUnit family of tools shares a characteristic architectural pattern

JUnit Architecture

- JUnit is made up of:
  - ✓ Test Runner: A class or mechanism that is responsible for executing the tests
  - ✓ Test Class: One or more test classes containing the tests for a component under development
  - ✓ Test Method: Each test is implemented by a test method in a test class
  - ✓ Test Fixture: The state the system to be in for a test to be run
  - ✓ Test Suite: A set of tests that all share the same test fixture
  - ✓ Test Execution: The running of the test case along with any fixture methods required to set up and tear down the test fixtures
  - ✓ Test Result Formatter: Responsible for reporting on the results of the tests in a usable format
  - ✓ Assertion Set: Functions that verify the results of a test

JUnit Architecture

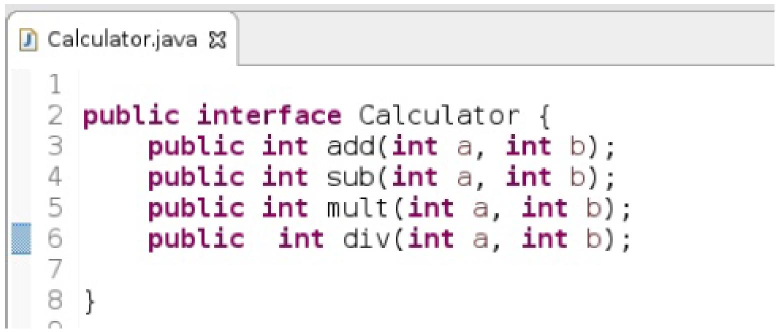


LearnQuest

The Calculator Project

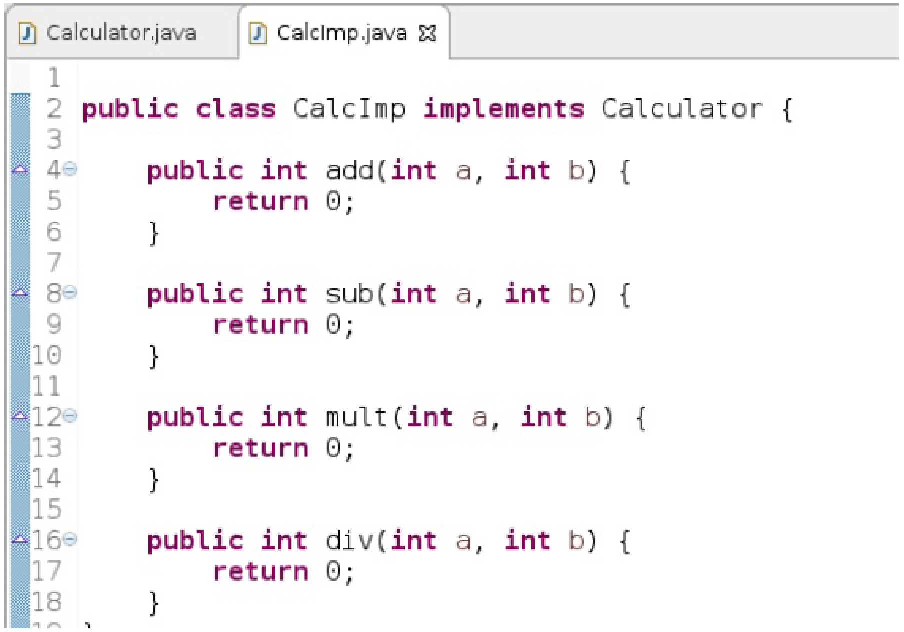
- To demonstrate the functionality of JUnit, we will implement a trivial example of a calculator that does basic arithmetic.
  - ✓ This is not Test Driven Development, this is just experimenting with Junit
  - ✓ We will be using JUnit (JUnit 5 can have some hiccups in eclipse we want to avoid) – the functionality is the same though
  - ✓ We will be working within Eclipse
  - ✓ Initial steps:
    - Create the Java project
    - Define the Calculator interface
    - Create the implementing class CalcImp
    - Create the JUnit test class

The Calculator Project



Calculator.java

```java

public interface Calculator {
     public int add(int a, int b);
     public int sub(int a, int b);
     public int mult(int a, int b);
     public  int div(int a, int b);

}
```



Calculator.java | CalcImp.java

```java

public class CalcImp implements Calculator {

     public int add(int a, int b) {
          return 0;
     }

     public int sub(int a, int b) {
          return 0;
     }

     public int mult(int a, int b) {
          return 0;
     }

     public int div(int a, int b) {
          return 0;
     }
```

The Project Code

The first image shows the Calculator interface which describes the calculator functionality – which is quite trivial. The second image shows the implementing class where we have to create the code that implements the functionality defined by the interface.

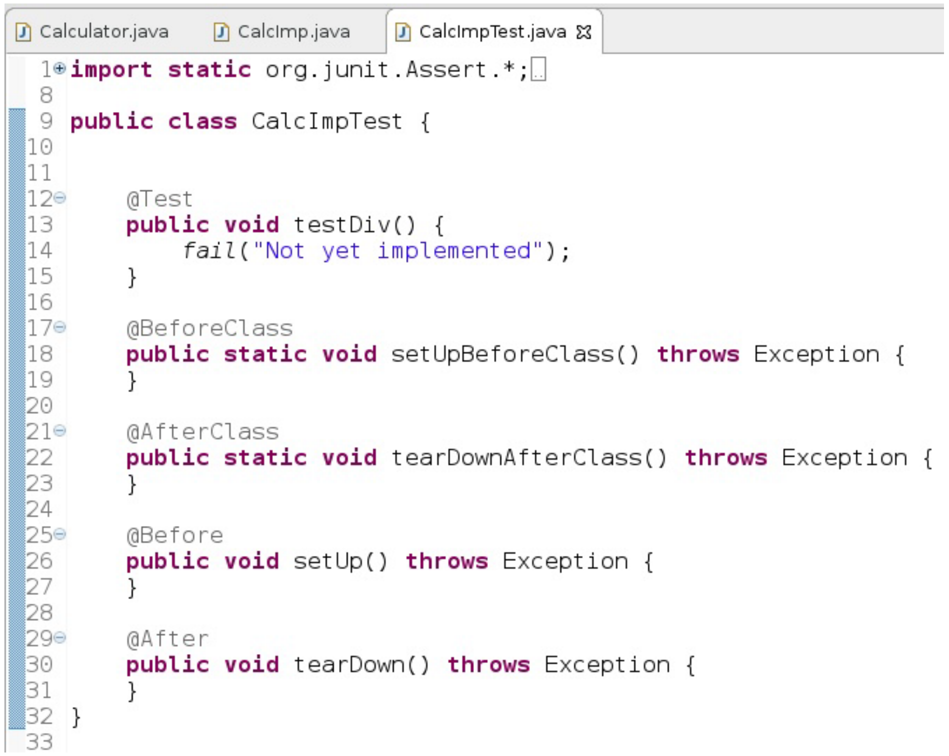
This is an example of "outside-in" development where the interface is defined before we start to write any code. This is the state we want our code in just before we start to do our Test Driven Development.

Also notice that these methods are all queries since they do not change anything inside a calculator object nor do they create any side effects.

Adding the Test Class

- Eclipse is used to add a JUnit test class to the project using a builtin wizard, although we could hand code it
- The test code is in a separate class from the production code
- The test class will be created that creates test method "stubs" that we will use to implement the tests and fixture methods

Adding the Test Class



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

public class CalcImpTest {


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

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
    }

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }
}
```

The Generated Stubs

This is what the final result of the using the wizard looks like, although the fixture methods have been moved to end of the file for readability.

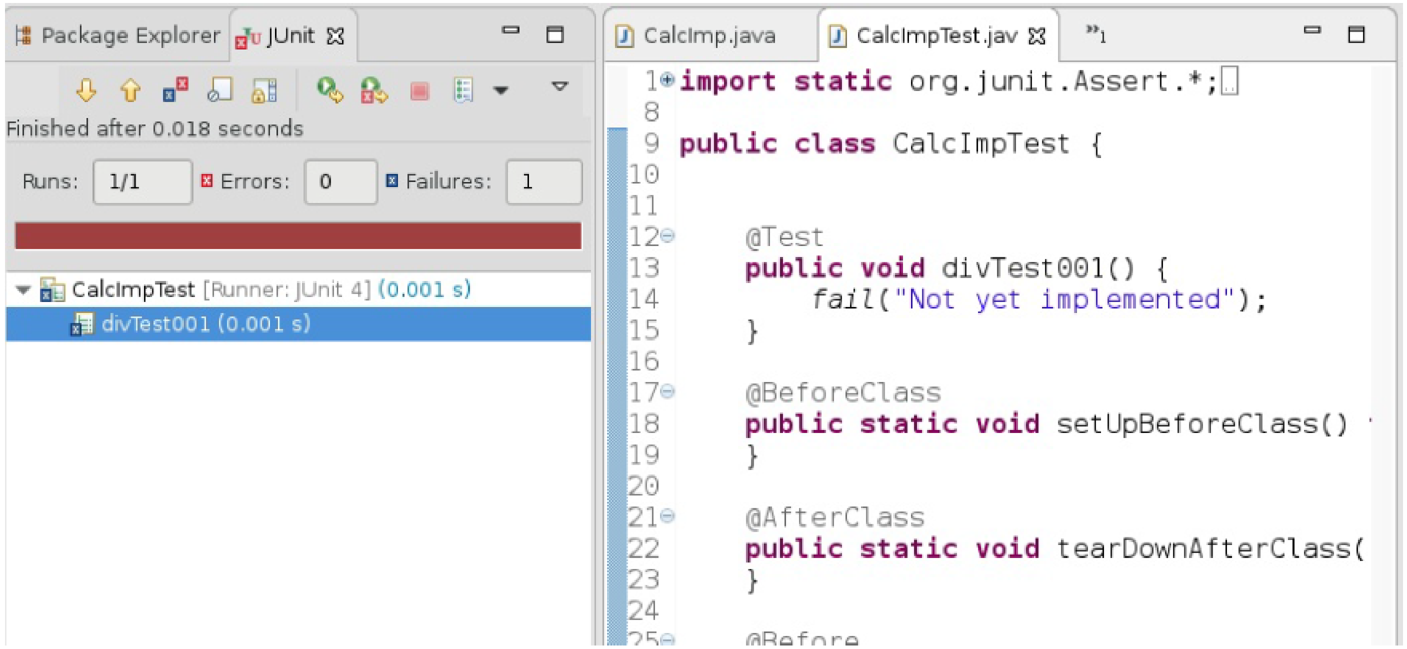
Adding the Test Class

- Eclipse is used to add a JUnit test class to the project using a builtin wizard, although we could hand code it
- The test code is in a separate class from the production code
- The test class will be created that creates test method “stubs” that we will use to implement the tests and fixture methods

The Test Method

- The JUnit runner uses the annotations to find the test methods
  - ✓ All methods with the @Test annotation are test methods
  - ✓ The autogenerated names on the methods should be changed to something that more meaningful
  - ✓ Test methods must return void and take no arguments
  - ✓ The other methods with other annotations (@Before, @After, etc) are fixture method stubs
- At this point we can run the test method using the built in Eclipse JUnit runner

The Test Method



Finished after 0.018 seconds

Runs: 1/1 Errors: 0 Failures: 1

CalcImpTest [Runner: JUnit 4] (0.001 s)
divTest001 (0.001 s)

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

public class CalcImpTest {


    @Test
    public void divTest001() {
        fail("Not yet implemented");
    }

    @BeforeClass
    public static void setUpBeforeClass()
    }

    @AfterClass
    public static void tearDownAfterClass(
    }

    @Before
```

Running the Tests

By selecting the “Run as JUnit test” option, Eclipse calls the JUnit default runner class which then looks through the test class and runs all of the @Test methods.

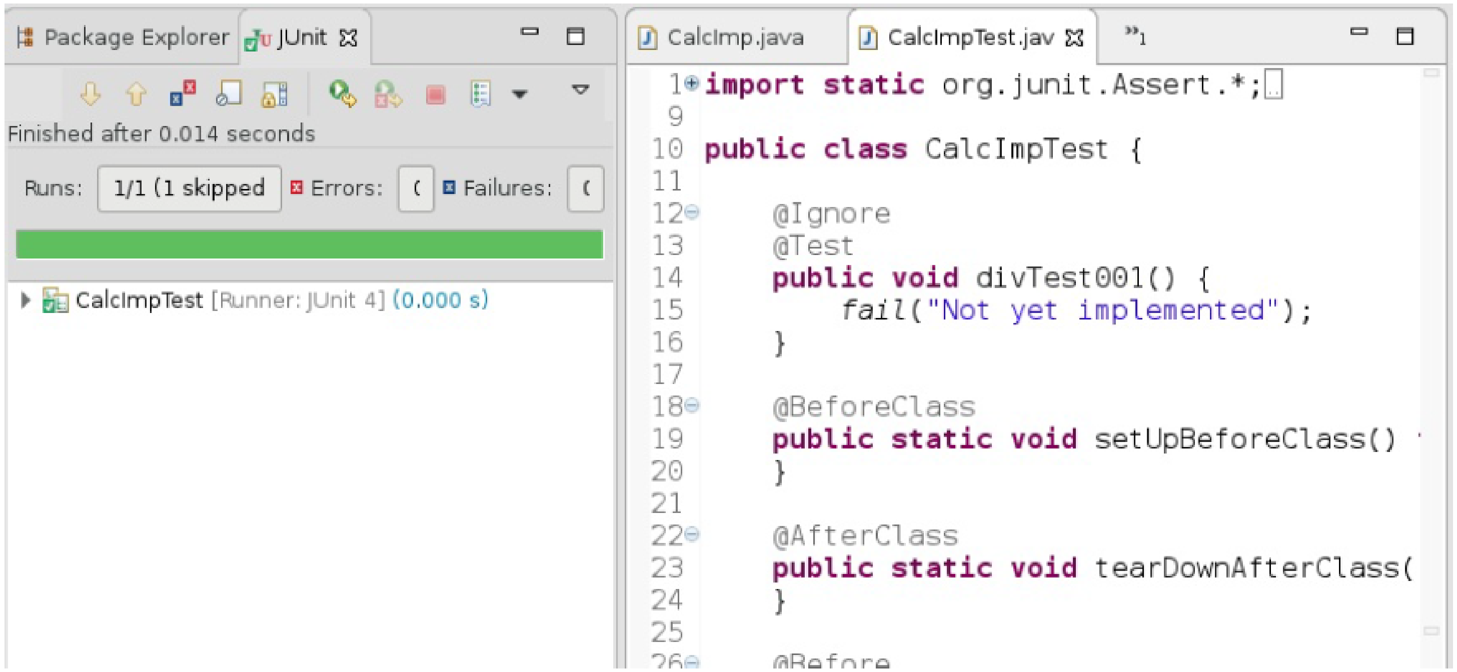
The Eclipse built in results formatter reports that 1 test was run and 1 test failed. The test failed because when the stub was generated, a fail assertion was placed in the body as a reminder to add the code to test method.

The test methods are not guaranteed to run in any particular order.

The Test Results

- JUnit cannot understand what a test “means” so it relies on us to tell it whether a test has passed or failed
  - ✓ If an assertion exception is thrown, JUnit marks the test as failed
  - ✓ The fail() method throws an exception every time it executes
- Tests that don’t throw assertion exceptions are considered to have passed
  - ✓ Best practice: always have a fail() method in a test method until the test code is written

Ignoring Tests



Finished after 0.014 seconds

Runs: 1/1 (1 skipped Errors: (Failures: (

CalcImpTest [Runner: JUnit 4] (0.000 s)

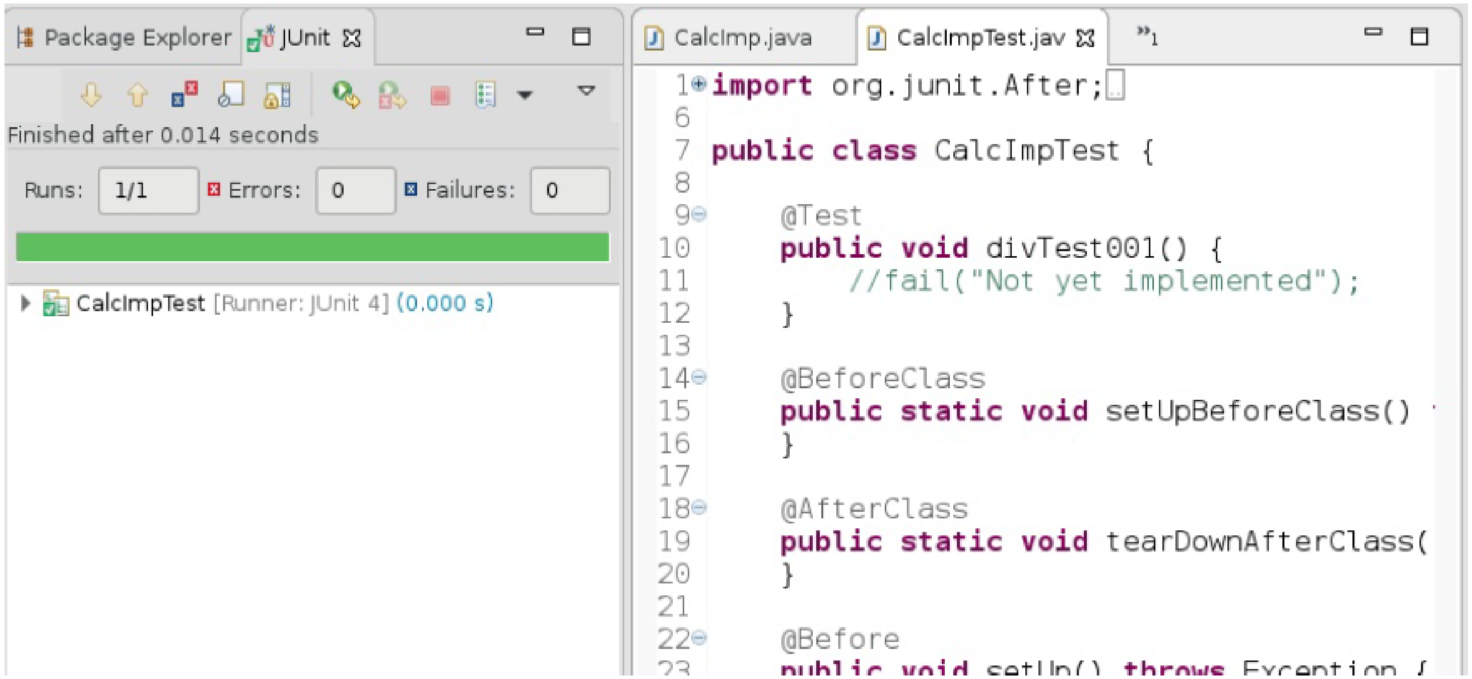
```
import static org.junit.Assert.*;

public class CalcImpTest {

    @Ignore
    @Test
    public void divTest001() {
        fail("Not yet implemented");
    }

    @BeforeClass
    public static void setUpBeforeClass() 
    }

    @AfterClass
    public static void tearDownAfterClass(
    }

    @Before
```

Ignoring Tests

The @Ignore annotation has been added to the test methods. This causes JUnit to skip the test and not count it as a pass or failure. The @Ignore annotation is used to suppress reporting about a test until we actually want to run it. Ignoring a test method is a lot safer than editing out the fail() annotation.

Because no test has failed, JUnit reports that all of the tests passed.

Passing Tests



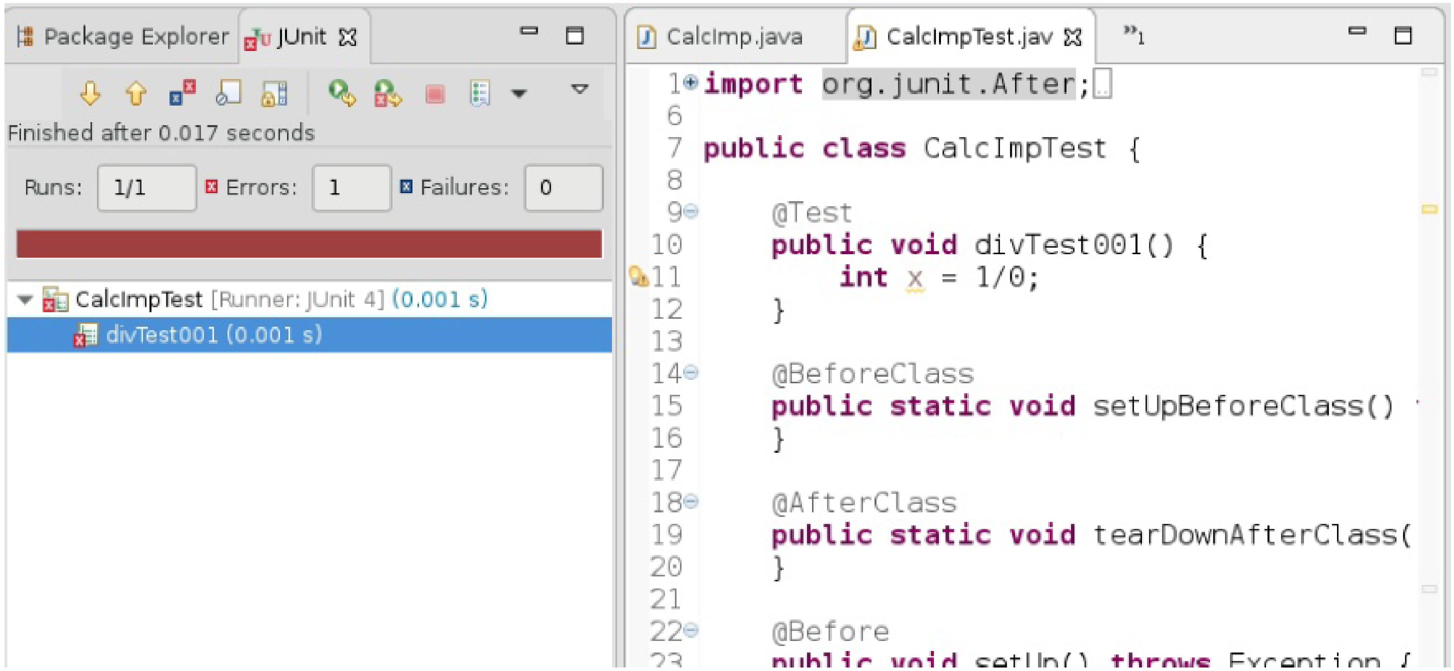
```
Finished after 0.014 seconds
Runs: 1/1    Errors: 0    Failures: 0
CalcImpTest [Runner: JUnit 4] (0.000 s)
```

```java
import org.junit.After;

public class CalcImpTest {

    @Test
    public void divTest001() {
        //fail("Not yet implemented");
    }

    @BeforeClass
    public static void setUpBeforeClass()
    }

    @AfterClass
    public static void tearDownAfterClass(
    }

    @Before
    public void setUp() throws Exception {
```

Passing Tests

In the example, the @Ignore annotation has been removed and the fail() assertion commented out. Now the test passes vacuously since it doesn't do anything that informs JUnit a failure has taken place. Notice the only different between this output and the previous slide is that in this case, no tests are reported as being skipped.

Errors are not Failures



Package Explorer | JUnit

Finished after 0.017 seconds

Runs: 1/1 Errors: 1 Failures: 0

CalcImpTest [Runner: JUnit 4] (0.001 s)

divTest001 (0.001 s)

CalcImp.java | CalcImpTest.jav

```
import org.junit.After;

public class CalcImpTest {

    @Test
    public void divTest001() {
        int x = 1/0;
    }

    @BeforeClass
    public static void setUpBeforeClass()
    }

    @AfterClass
    public static void tearDownAfterClass(
    }

    @Before
    public void setUp() throws Exception {
```

Errors are NOT Failures

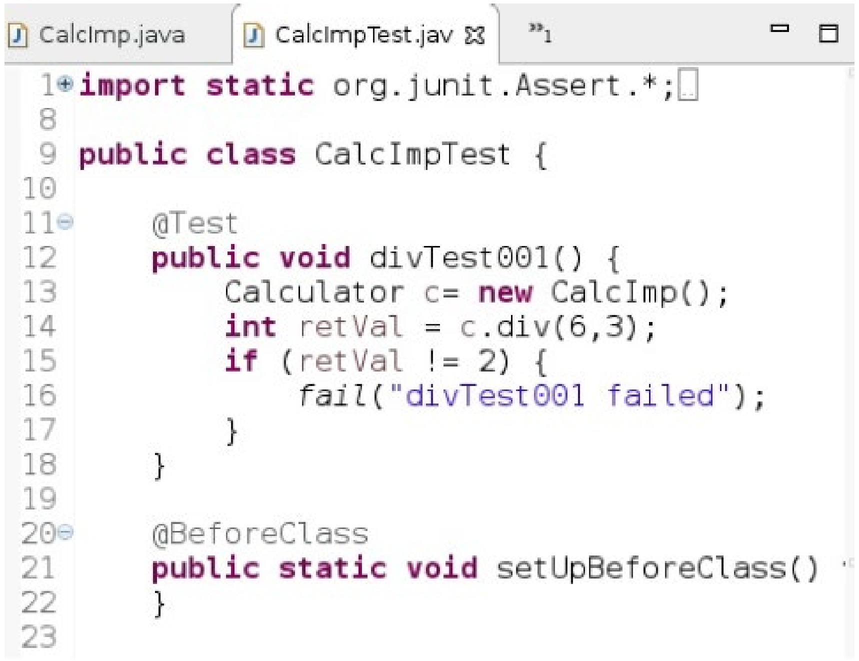
In the example above, a Java error (divide by zero) has been added to the code. This caused an exception to be thrown that was not generated by an assertion statement. Because the divide by zero exception is not an assertion exception, this test has not failed or passed from a testing point of view because it could not be run.

Notice that this is reported in the results window as an error and not as a failure.

Writing Test Methods

- Each test method is a single test case consisting of
  - ✓ A test input
  - ✓ A description of the system state required for test execution
  - ✓ The expected correct output
- The test method:
  - ✓ Acquires an instance of the component under development
  - ✓ Invokes the method being tested using the test input
  - ✓ Compares the actual value returned with the expected value
  - ✓ If they two values match then the test passes, otherwise it fails
- Putting the system into the required test state is done with the fixture methods

Implementing Test Case



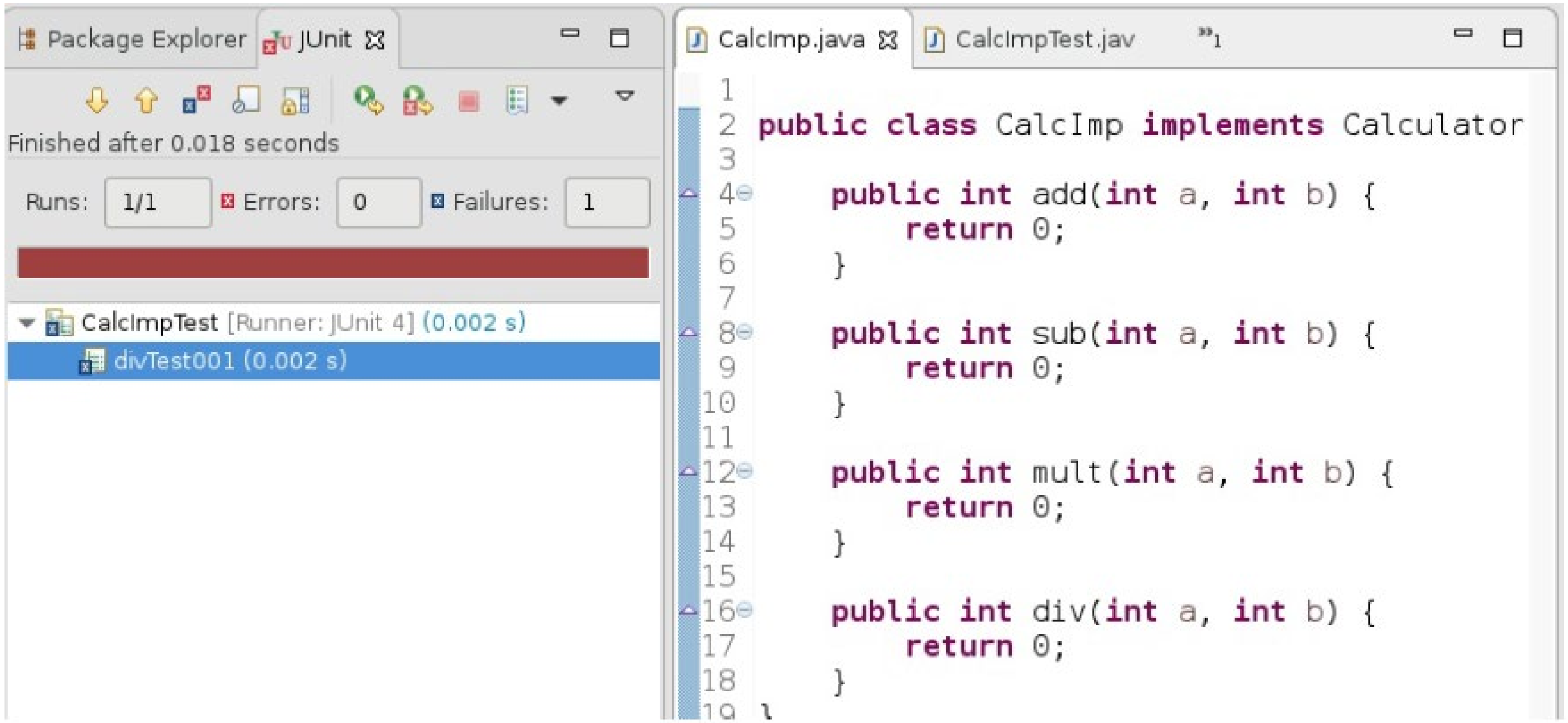
```
import static org.junit.Assert.*;

public class CalcImpTest {

    @Test
    public void divTest001() {
        Calculator c= new CalcImp();
        int retVal = c.div(6,3);
        if (retVal != 2) {
            fail("divTest001 failed");
        }
    }

    @BeforeClass
    public static void setUpBeforeClass()
    }

```

Implementing a Test Case

Consider test case divTest001() for the Calculator div() method. The test case specifies inputs of 6 and 3 and an expected value of 2. An instance "c" of the calculator object is created and the result of invoking c.div(6,3) is stored in retVal

If retVal is not 2, a fail() exception is thrown and the test fails

Failing Test Case



Package Explorer | JUnit

Finished after 0.018 seconds

Runs: 1/1 Errors: 0 Failures: 1

CalcImpTest [Runner: JUnit 4] (0.002 s)
divTest001 (0.002 s)

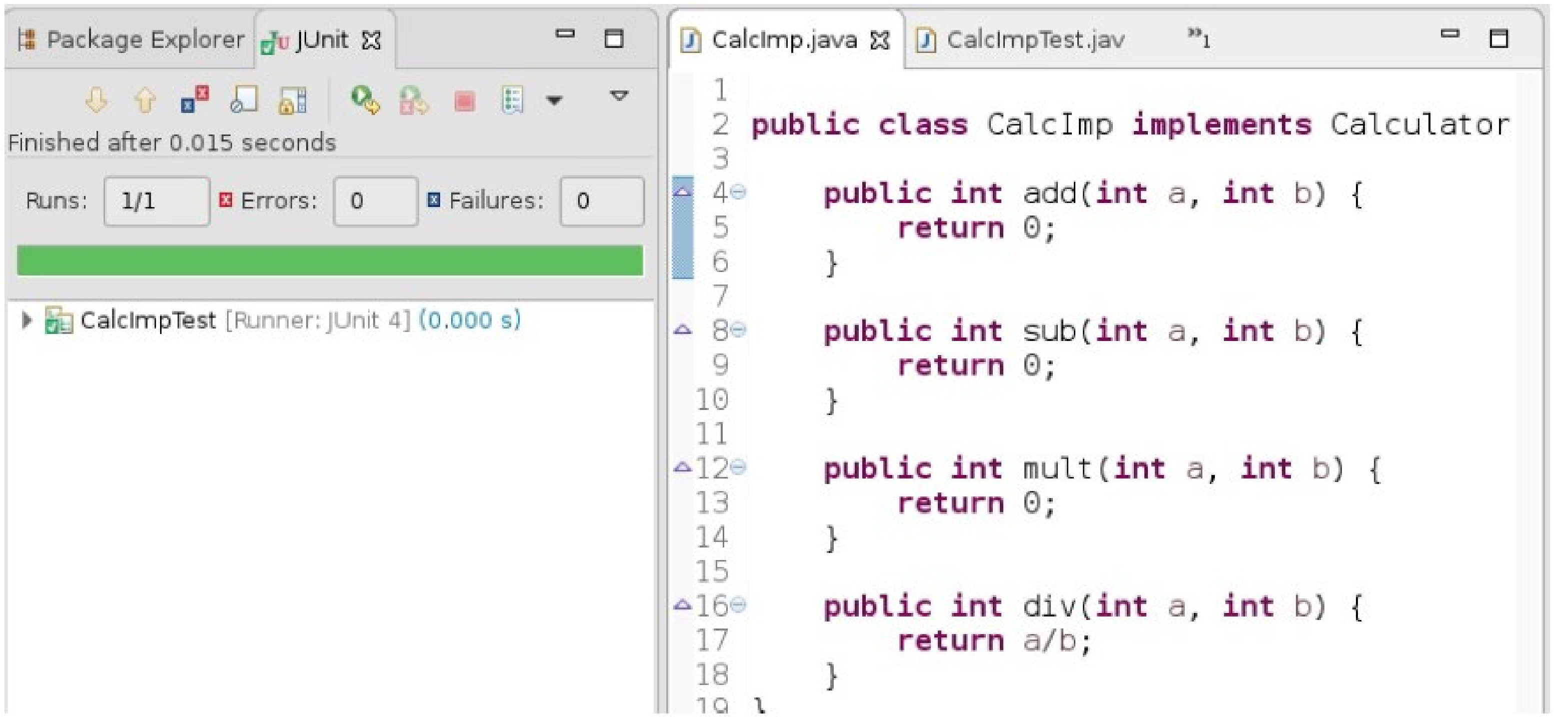
CalcImp.java | CalcImpTest.jav

```java

public class CalcImp implements Calculator

    public int add(int a, int b) {
        return 0;
    }

    public int sub(int a, int b) {
        return 0;
    }

    public int mult(int a, int b) {
        return 0;
    }

    public int div(int a, int b) {
        return 0;
    }
}
```

Test Case Failure

Running the tests now reports a failure because we haven't yet implemented the production code that makes the test pass.

Passing Test Case



Package Explorer | JUnit

Finished after 0.015 seconds

Runs: 1/1 Errors: 0 Failures: 0

CalcImpTest [Runner: JUnit 4] (0.000 s)

CalcImp.java | CalcImpTest.jav

```

public class CalcImp implements Calculator

    public int add(int a, int b) {
        return 0;
    }

    public int sub(int a, int b) {
        return 0;
    }

    public int mult(int a, int b) {
        return 0;
    }

    public int div(int a, int b) {
        return a/b;
    }
}
```

Passing the Test

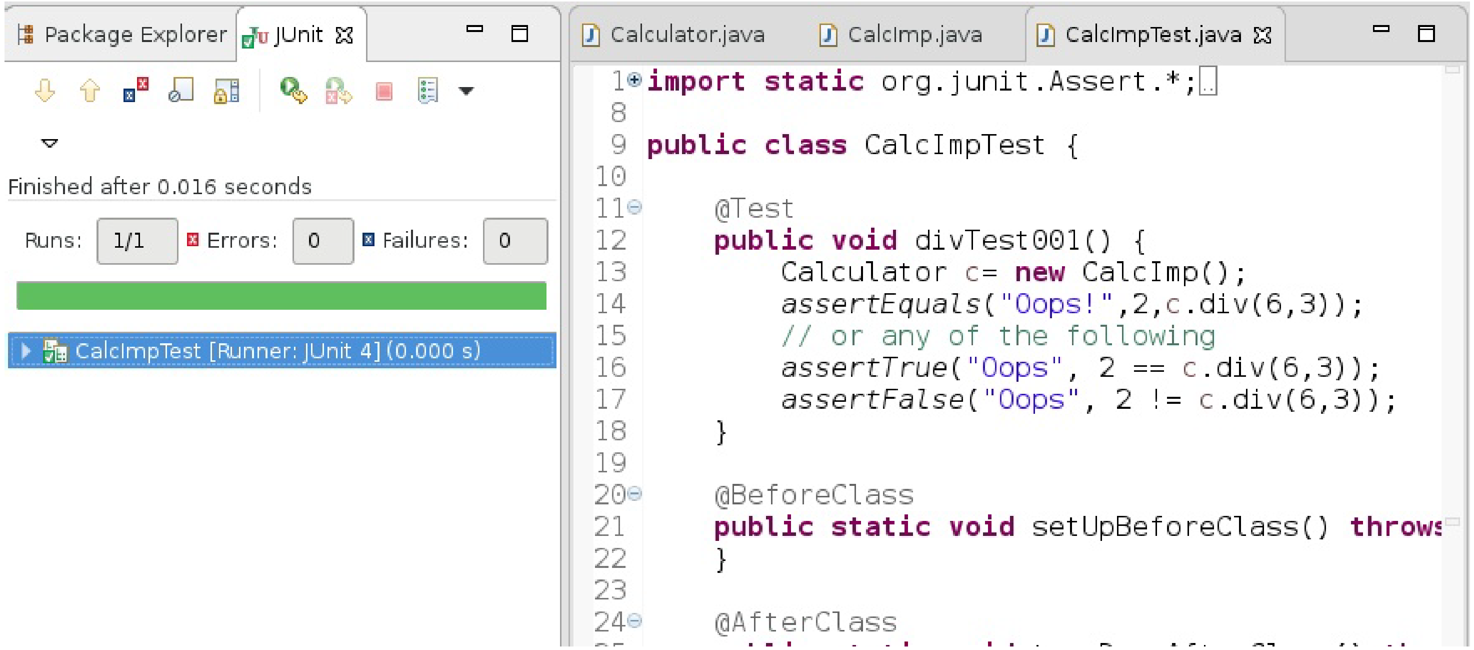
The production code has been added to the div() method and the test now pases.

JUnit Assertions

- Assertions are statements that evaluate to true or false
  - ✓ If an assertion is false then an exception is thrown, otherwise no action is taken
- Assertions express test conditions in a readable form, for example:
  - ✓ assertEquals([msg],expected, actual) compares two values, if theyare not equal, the assertion fails ("msg" is an optional message to be printed if the assertion fails)
  - ✓ assertEquals(expected, actual, delta) is a form used for floating point numbers where two values are "equal" if |expectedactual|< delta
  - ✓ assertTrue(val) where val is a boolean predicate

LearnQuest

Adding Assertions



Finished after 0.016 seconds

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

public class CalcImpTest {

    @Test
    public void divTest001() {
        Calculator c= new CalcImp();
        assertEquals("Oops!",2,c.div(6,3));
        // or any of the following
        assertTrue("Oops", 2 == c.div(6,3));
        assertFalse("Oops", 2 != c.div(6,3));
    }

    @BeforeClass
    public static void setUpBeforeClass() throws
    }

    @AfterClass
```

Readable Forms of Assertions

The use of the different forms of assertions allows us to state the test conditions in a much more natural and readable manner. JUnit does not care what form of the assertion is used which means that we choose the one that reads most naturally.

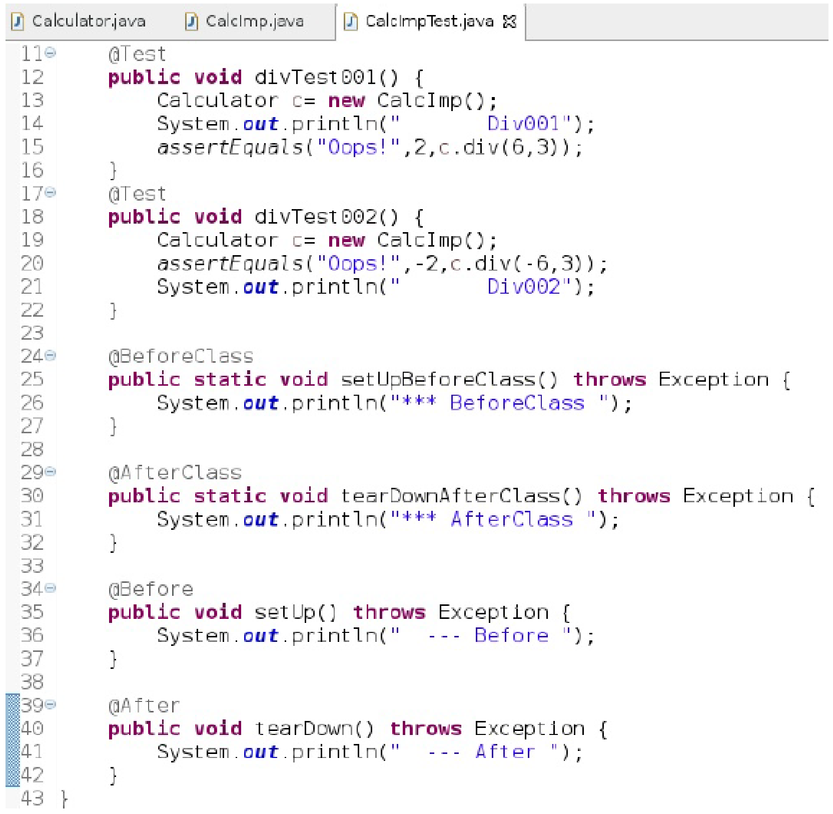
For example, there are three forms used in the test method and JUnit will accept any one of them, but the first assertion used would be preferable because it communicates more clearly what the method is testing to those reading or maintaining the code.

The use of the different forms allows us to write more streamlined and compact code as well.

Fixture Methods

- Fixture Methods are run to set up and tear down tests
  - ✓ Set up methods prepare the test environment and system state
  - ✓ Tear down methods undo what set up methods do
- Fixture methods are identified by annotations
  - ✓ @BeforeClass are executed once before any tests are run
  - ✓ @AfterClass are executed once after all tests have run
  - ✓ @Before are all executed before each test is run
  - ✓ @After are all executed after each tests is run
- Fixture methods with the same annotation are not guaranteed to run in the order they appear in the code

Fixture Method Example

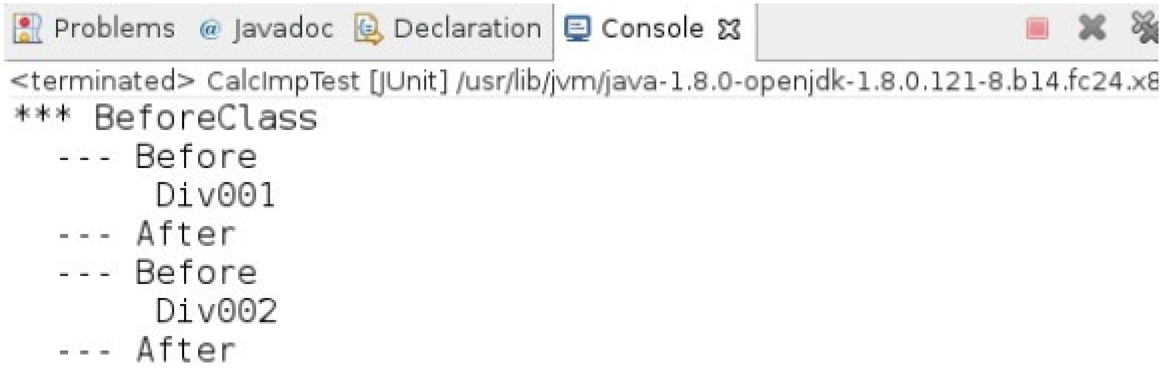


```java
    @Test
    public void divTest001() {
        Calculator c= new CalcImp();
        System.out.println("       Div001");
        assertEquals("Oops!",2,c.div(6,3));
    }
    @Test
    public void divTest002() {
        Calculator c= new CalcImp();
        assertEquals("Oops!",-2,c.div(-6,3));
        System.out.println("       Div002");
    }

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        System.out.println("*** BeforeClass ");
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
        System.out.println("*** AfterClass ");
    }

    @Before
    public void setUp() throws Exception {
        System.out.println("  --- Before ");
    }

    @After
    public void tearDown() throws Exception {
        System.out.println("  --- After ");
    }
}
```

Fixture Methods

In the code, we have added some dummy fixture methods and a second test method.

Fixture Method Example



```
<terminated> CalcImpTest [JUnit] /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.121-8.b14.fc24.x8
*** BeforeClass
  --- Before
       Div001
  --- After
  --- Before
       Div002
  --- After
```

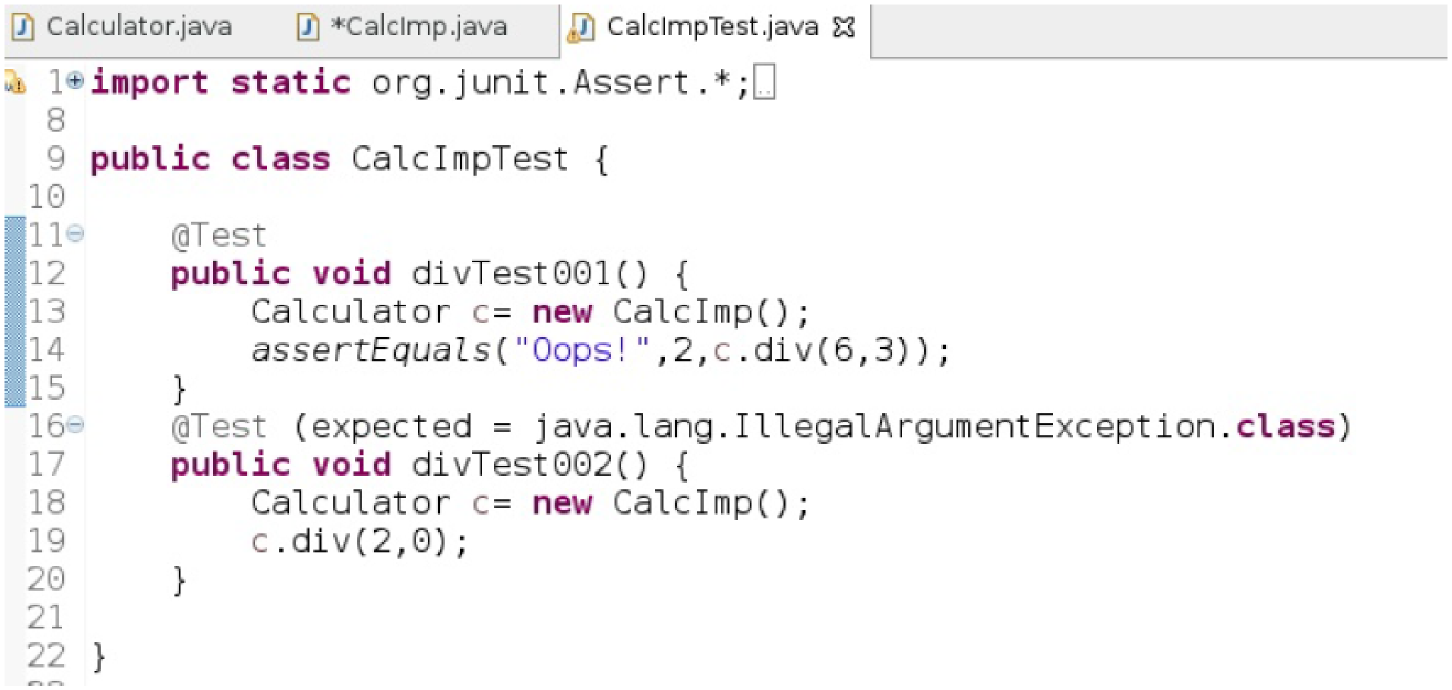
Fixture Method Output

Running the test and looking at the console output, the sequence of execution of the fixtures and the test methods is clearly seen.

Testing for Exceptions

- Sometimes a test expects an exception to be thrown in order to pass
  - ✓ The standard assertions do not allow us to check this case
  - ✓ Instead we identify the exception to be thrown in the annotation
- The assertion syntax is:
  - ✓ @Test(expected = <java exception class>)
- The test will pass only if the specified exception is thrown
  - ✓ The test will fail if no exception is thrown
  - ✓ The test will fail if any exception other than the specifiedone is thrown

Testing for Exceptions

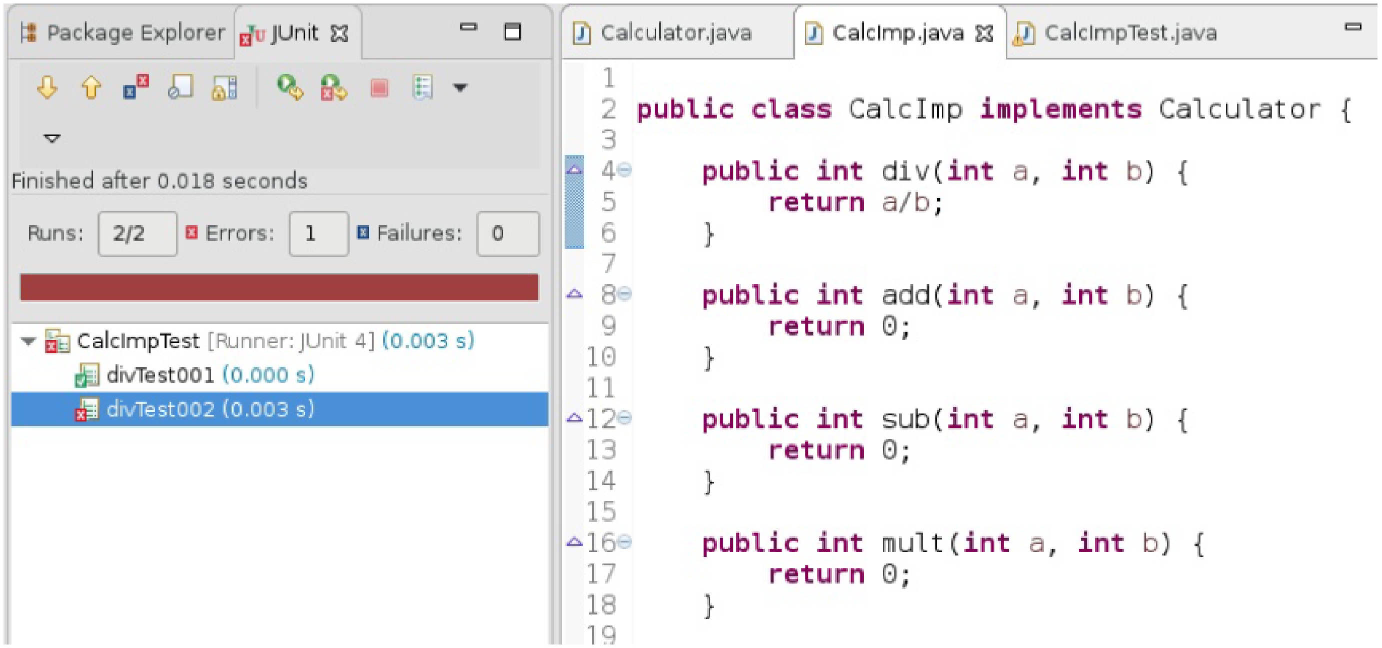


```
import static org.junit.Assert.*;

public class CalcImpTest {

    @Test
    public void divTest001() {
        Calculator c= new CalcImp();
        assertEquals("Oops!",2,c.div(6,3));
    }
    @Test (expected = java.lang.IllegalArgumentException.class)
    public void divTest002() {
        Calculator c= new CalcImp();
        c.div(2,0);
    }

}
```

The Exception Test Method

In the second test method, we are checking to see if the production code throws an IllegalArgumentException when the divide method divides by zero.

Testing for Exceptions



Package Explorer | JUnit | Calculator.java | CalcImp.java | CalcImpTest.java

Finished after 0.018 seconds

Runs: 2/2 Errors: 1 Failures: 0

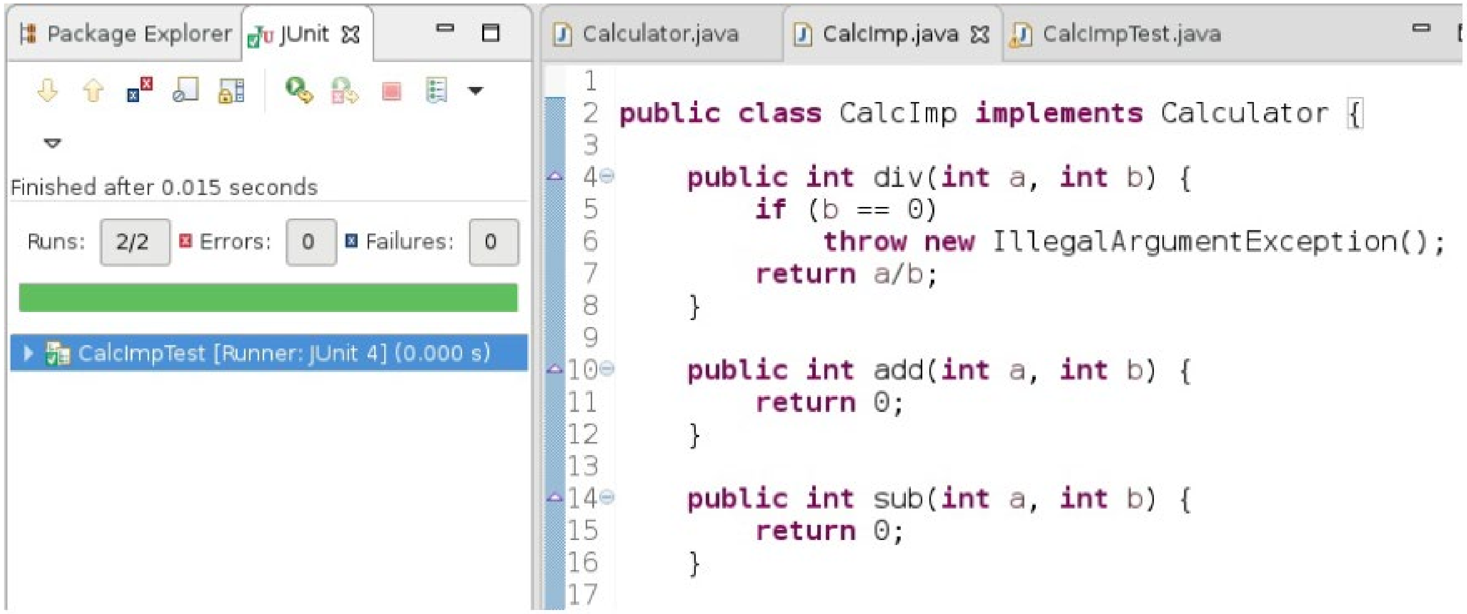
```
CalcImpTest [Runner: JUnit 4] (0.003 s)
    divTest001 (0.000 s)
    divTest002 (0.003 s)
```

```java

public class CalcImp implements Calculator {

    public int div(int a, int b) {
        return a/b;
    }

    public int add(int a, int b) {
        return 0;
    }

    public int sub(int a, int b) {
        return 0;
    }

    public int mult(int a, int b) {
        return 0;
    }

```

Running the Exception Test

Since the code to check for a division by zero does not yet exist, the test fails. Notice that this is one time when an error and failure are the same thing. The error occurred because Java threw an ArithmeticException however the test failed because it was expecting an exception but the wrong type of exception was thrown.

Testing for Exceptions



Finished after 0.015 seconds

Runs: 2/2 Errors: 0 Failures: 0

CalcImpTest [Runner: JUnit 4] (0.000 s)

```

public class CalcImp implements Calculator {

    public int div(int a, int b) {
        if (b == 0)
            throw new IllegalArgumentException();
        return a/b;
    }

    public int add(int a, int b) {
        return 0;
    }

    public int sub(int a, int b) {
        return 0;
    }

```

Running the Exception Test

Once the production code is added to perform the check for zero and throw the correct exception, the tests pass.

Questions

LearnQuest