Tutorial #3

SFWR ENG / COMP SCI 2S03

Introduction to Test Driven Development (TDD)

Dillon Dixon

Week of September 23, 2013

What is TDD?

- *Test Driven Development* is a software development approach that relies on the repetition of a very short development cycle involving many unit tests.

- **Unit Testing:** The testing of small pieces of code that fit into a larger system.

Steps to TDD

1. Write a *test case*.
2. Run all tests to see if the new one fails.
3. If it fails, write some code to try and make it pass.
4. Go back to step 2 until *all* tests again pass.
5. Refactor the new code.
6. Write another test case and do it again!

Example

- Let's create a *Line* class. A class for representing 2-dimensional lines.
- We initialize it with an *m* and a *b* value.
- For now, we would really like the ability to calculate a *Y* value given an *X* value.

Example

- Class skeleton.

```java
public class Line {

    private int m;
    private int b;

    public Line(int m, int b){

        this.m = m;
        this.b = b;
    }

    /**
     * Find the Y position of a line given x.
     *
     * @param x The X position on the line.
     * @return The corresponding Y.
     */
    public int calculateY(int x){

        // Return 0 until we implement this method.
        return 0;
    }
}
```

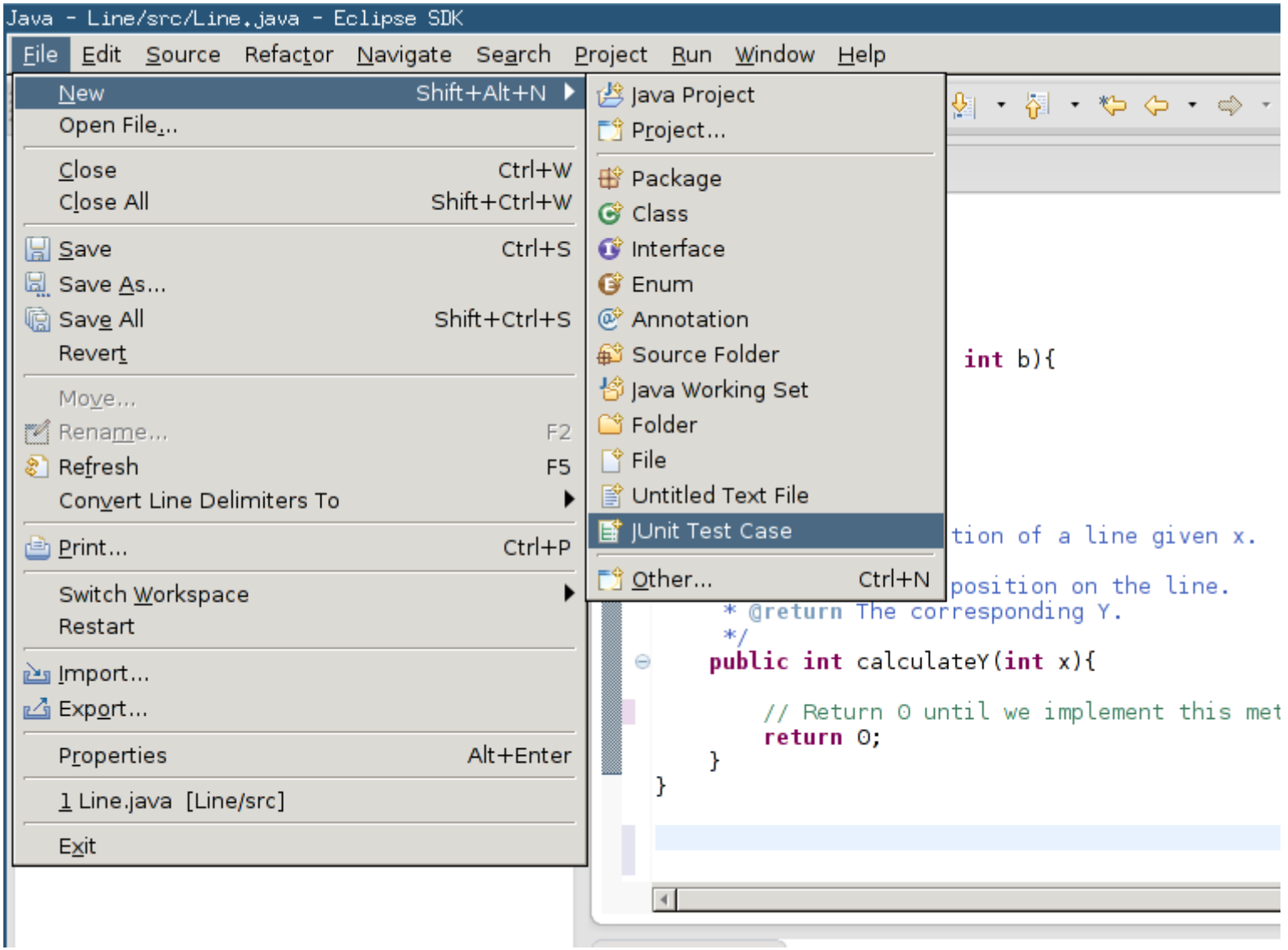
Example

- This class skeleton is sufficient to write our first test case.
- How do we write a test case?
- We use *JUnit.*

What is JUnit?

- *JUnit* is a testing framework for Java.
- The *J* stands for “Java” and *Unit* refers to “Unit Testing” (testing small pieces of a program at a time).
- How do we use it?

Creating a JUnit Test Case



Creating a JUnit Test Case

- A new testing skeleton is created.

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


public class LineTest {

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

}
```

Let's write a simple test!

- The *@Test* means that this method is to be run as a *test case.*
- Lets change it to match our needs.

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


public class LineTest {

    @Test
    public void testCalculateY() {

        int m = 4;
        int b = 5;

        Line line = new Line(m, b);

        assertEquals(line.calculateY(1), 9);
    }

}
```

Let's write a simple test!

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


public class LineTest {

    @Test
    public void testCalculateY() {

        int m = 4;
        int b = 5;

        Line line = new Line(m, b);

        assertEquals(line.calculateY(1), 9);
    }

}
```

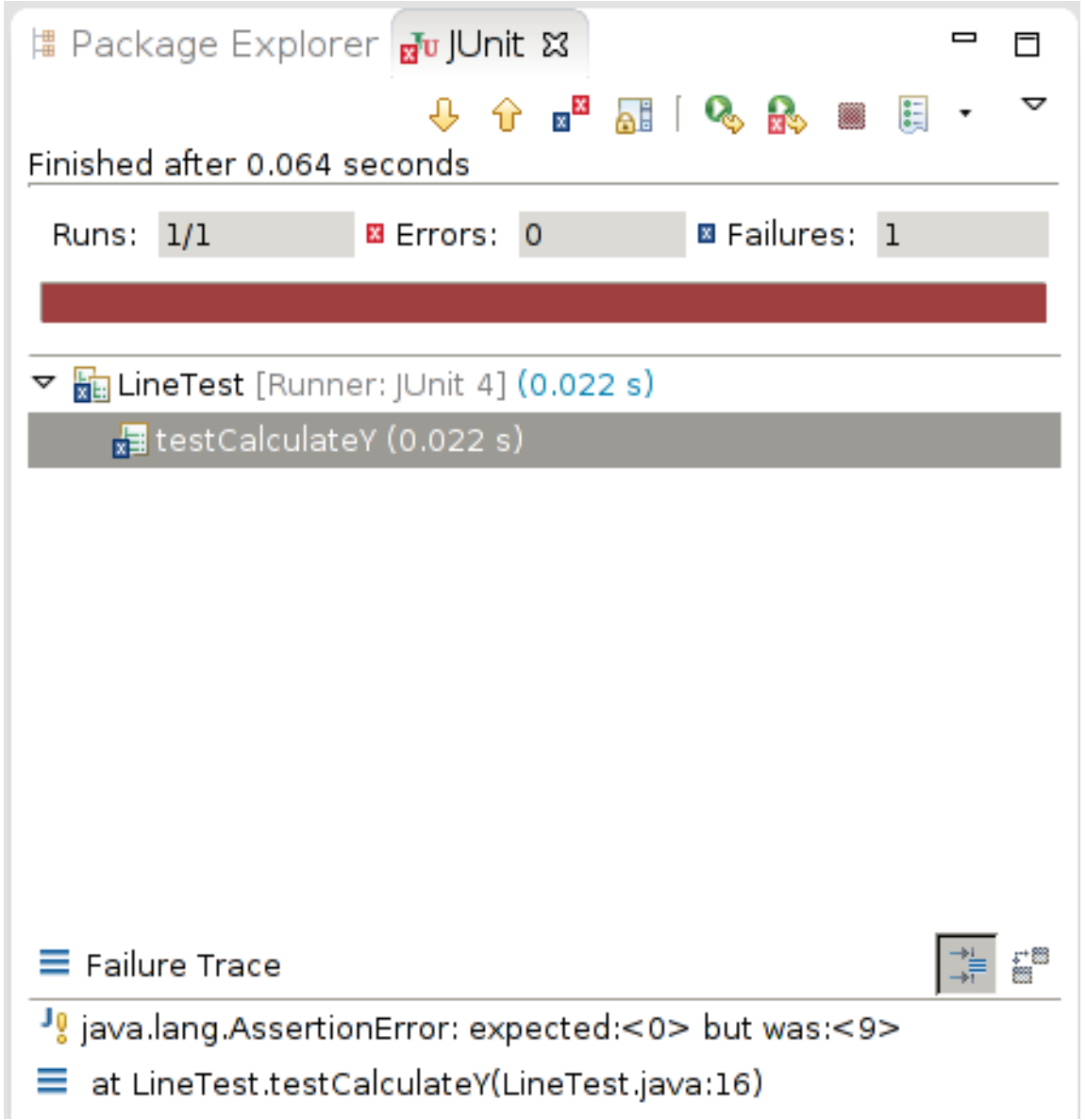
Marks the method as a test case

Line variable declarations

Creates a new line instance

Assert statement. Checks whether the *left* argument equals the *right* argument. If this is not true, the test fails.

Let's run our test!



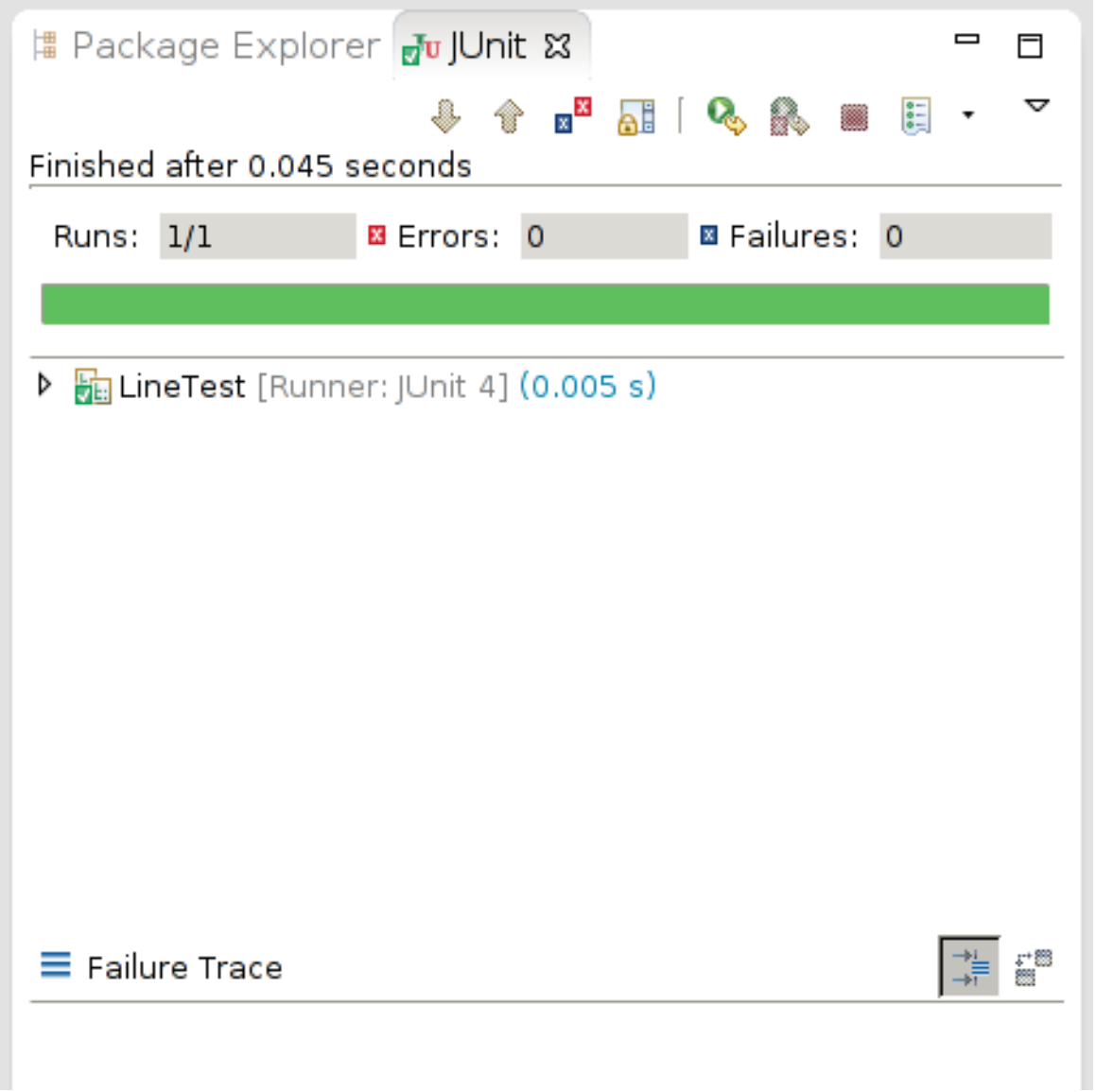
- When you run this test in Eclipse, a *JUnit* tab comes up.
- It's red because our test failed.

Let's fix our test!

```
/**
 * Find the Y position of a line given x.
 *
 * @param x The X position on the line.
 * @return The corresponding Y.
 */
public int calculateY(int x){

    int y = 0;

    y = m * x;
    y = y + b;

    return y;
}
```

- We can quickly write some code to calculate Y in our *Line* class.

Let's run our test!



- The same window as before comes up when we run our test again.
- It's green because our test passed.

Time to refactor!

```java
/**
 * Find the Y position of a line given x.
 *
 * @param x The X position on the line.
 * @return The corresponding Y.
 */
public int calculateY(int x){

    return m * x + b;
}
```

- Lets clean up our code and run the test one more time to make sure it still works.

(**SPOILER ALERT:** It does)

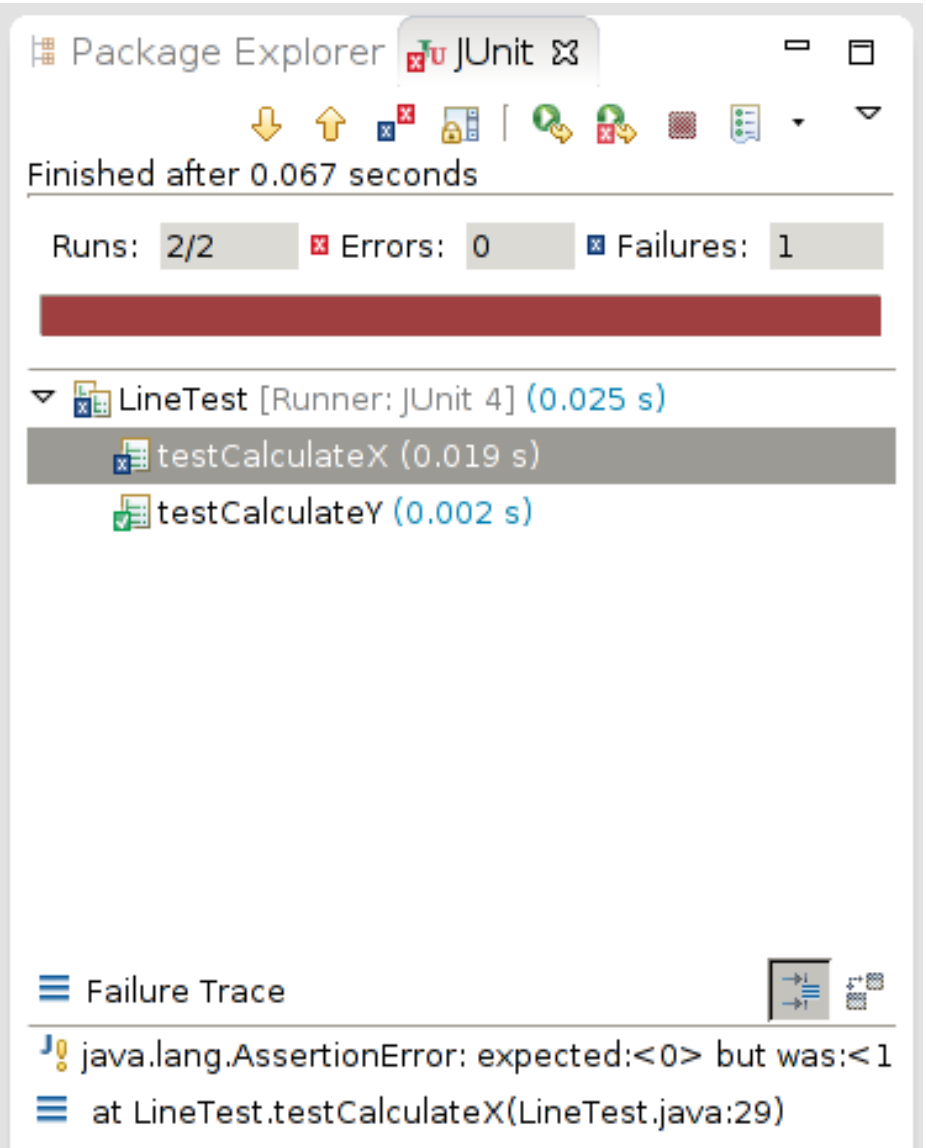
What is next?

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

import org.junit.Test;


public class LineTest {

    @Test
    public void testCalculateY() {

        int m = 4;
        int b = 5;

        Line line = new Line(m, b);

        assertEquals(line.calculateY(1), 9);
    }

    @Test
    public void testCalculateX(){

        int m = 4;
        int b = 5;

        Line line = new Line(m, b);

        assertEquals(line.calculateX(9), 1);
    }
}
```

- Maybe we want a function to calculate X from Y!
- Let's add another test.

(Assume we have created the associated method skeleton for calclateX in the *Line* class)

Running our tests



- We can now see both of our tests run.
- The ***x*** and ✓ in the lower left hand corner of each test indicate those that have passed and failed.
- Our unimplemented calculate x test has failed.

Your turn!

- Try fixing calculateX and checking it with *JUnit* yourself!

Making tests more robust

- We can add *more* assert statements to our test cases to give them better coverage.
- You should always strive to make your tests *complete* in the sense that they cover *every* case.
- Here we cover positive, zero and negative values of x.

```
@Test
public void testCalculateY() {

    assertEquals(line.calculateY(1), 9);
    assertEquals(line.calculateY(0), 5);
    assertEquals(line.calculateY(-1), 1);
}
```

Refactoring Test Cases

- Right now, we define our *line* variable instantiation multiple times for each test case, which is redundant.
- JUnit lets us create a single *Line* instantiation we can use in all our tests with the *@BeforeClass* annotation.
- The methods tagged with @BeforeClass are run exactly once before all test cases.

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

public class LineTest {

    private static Line line;

    @BeforeClass
    public static void setUp(){

        int m = 4;
        int b = 5;

        line = new Line(m, b);
    }

    @Test
    public void testCalculateY() {

        assertEquals(9, line.calculateY(1));
        assertEquals(5, line.calculateY(0));
        assertEquals(1, line.calculateY(-1));
    }

    @Test
    public void testCalculateX() {

        assertEquals(1, line.calculateX(9));
    }
}
```

What's the point?

- Test Driven Development promotes:
  - Correct code, as everything's tested.
  - More modularized and maintainable code, since everything needs to be tested in isolation.
  - Better programmers who are mindful of all possible use cases.

JUnit

- There is much more to *JUnit* than what was used in this tutorial.
- What you have learned here is sufficient to get you started.
- To explore more of what *JUnit* has to offer, you can visit this tutorial here.

Last Slide

The End