

CS 211

Testing

JUnit

Debuggers



Testing

Goals of Testing

"The goal of testing is just to find errors" – WRONG!

The goal of testing is to gain assurance about the quality of your programs

The testing you do for your class assignments is more likely to be debugging oriented

→ How do you catch bugs?

→ How do you design a good test case?

Testing

Testing can mean many different things:

- running a program on various inputs
- any human or computer assessment of quality
- evaluations before writing code

The earlier we find an problem, the easier and cheaper it is to fix

When are we done testing?

Facts about testing

Maintenance activities consume 70-90% of the total cost of software

→ spend most of our time testing and debugging code

Some software engineering approaches advocate writing test cases BEFORE you even write any code

- What does this accomplish?

Software Testing

The earlier you find a defect, the cheaper it is to fix

It is up to you to figure out how much testing is enough and when

		Time detected				
		Requirements	Architecture	Construction	System test	Post-release
Time introduced	Requirements	1×	3×	5–10×	10×	10–100×
	Architecture	-	1×	10×	15×	25–100×
	Construction	-	-	1×	10×	10–25×

Software Testing

Modern day testing is preventive

The testing you do for your class assignments is more likely to be debugging oriented

→ How do you catch bugs?

→ How do you design a good test case?

Reviews

A **review** is a meeting in which several people examine a design document or section of code

It is a common and effective form of human-based testing

Presenting a design or code to others:

- makes us think more carefully about it
- provides an outside perspective

Reviews are sometimes called *inspections* or *walkthroughs*

Test Cases

A **test case** is a set of input and user actions, coupled with the expected results

Often test cases are organized formally into *test suites* which are stored and reused as needed (like JUnit)

For medium and large systems, testing must be a carefully managed process

Many organizations have a separate Quality Assurance (QA) department to lead testing efforts

Defect and Regression Testing

Defect testing is the execution of test cases to find errors

The act of fixing an error may introduce new errors

After fixing a set of errors we should perform

regression testing – running previous test suites to ensure new errors haven't been introduced

not generally possible to test all possible inputs

→ design tests to maximize their ability to find problems

Creating test cases

Common behavior (expected inputs / usage)

Uncommon behavior (Border cases, very complex inputs)

How do you identify such properties?

- Black box testing – not looking at the code, just intent
- White box testing – looking at the code/implementation
- Practice, looking at other projects

Black-Box Testing

In *black-box testing*, test cases are developed without considering the internal logic

They are based on the input and expected output

Input can be organized into *equivalence categories*

Two input values in the same equivalence category would produce similar results

Therefore a good test suite will cover all equivalence categories and focus on the boundaries between categories

White-Box Testing

White-box testing focuses on the internal structure of the code

The goal is to ensure that every **path** through the code is tested

Paths through the code are governed by any conditional or looping statements in a program

A good testing effort will include both black-box and white-box tests

Limits of testing

Let's imagine you could, in theory, test your code on every possible combination of input and output

Would your code be guaranteed to be correct?

No! Perhaps you wrote correct code but it doesn't do what the customer asked

- Probably because the customer wasn't clear, or they didn't know what they wanted
- This happens all the time

→ defining what correctness means is a tough problem!

JUnit

JUnit: Unit Testing for Java

Unit Testing: Providing individual test cases that target specific **units** of code, usually a specific method or a specific branch of code.

JUnit: package that helps you write unit tests and execute them.

Offers a form of regression testing: make a few changes, re-run all JUnit tests to check if we broke previously-tested code

JUnit: Unit Testing for Java

We write test methods that call methods such as:

```
assertTrue(boolExpr)
```

```
assertEquals(Object expected, Object actual)
```

```
assertNotNull(Object o)
```

→ Integrates very well with IDEs (also works on command-line)

JUnit Example

```
public class Triangle {

    public int side1, side2, side3;

    public Triangle(int side1, int side2, int side3) {
        this.side1 = side1;
        this.side2 = side2;
        this.side3 = side3;
    }

    //is this area formula correct? We should test it...
    public double calculateArea () {
        //Heron's Formula for area of a triangle
        double s = (side1 + side2 + side3)/2;
        return Math.sqrt(s*(side1-s)*(side2-s)*(side3-s));
    }
}
```

JUnit Example – in DrJava

- Create test case → assert something → run the test

The screenshot shows the DrJava IDE interface. The top window displays the source code for `TriangleTest.java`:

```
1 import org.junit.*; // Test (used as @Test)
2 import static org.junit.Assert.*; // assertTrue
3
4 public class TriangleTest {
5
6     @Test
7     public void testMyTriangle() {
8         Triangle t = new Triangle(3,4,5);
9         assertTrue(6.0==t.calculateArea());
10    }
11 }
```

The bottom window shows the **Test Output** pane with the following message:

```
1 test failed:
  TriangleTest
    testMyTriangle
File: /Users/muddywaters/211/examples/shared/juniting/TriangleTest.java [line: 9]
Failure: java.lang.AssertionError:
```

On the right side of the Test Output pane, there is a **Test Progress** indicator with a red bar, and a **Show Stack Trace** button. The **Highlight source** checkbox is checked.

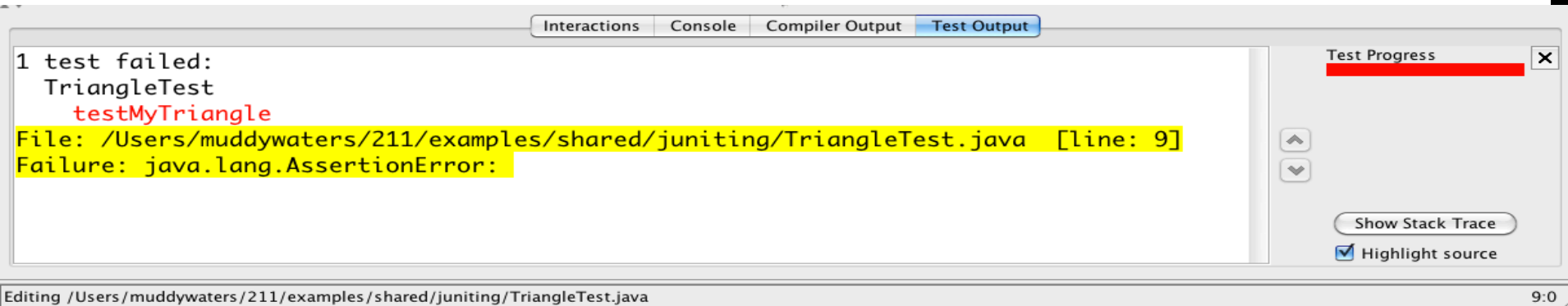
The status bar at the bottom indicates: `Editing /Users/muddywaters/211/examples/shared/juniting/TriangleTest.java` and the time `9:0`.

JUnit: Running Tests (in DrJava)

We can run our JUnit tests – just click the "Test" button with your testing class.

→ get results (green=good, red=bad), and see a trace of where the failing test failed.

Here, our assertion of equality being true failed. Time to go double check Heron's Formula as we've implemented it!



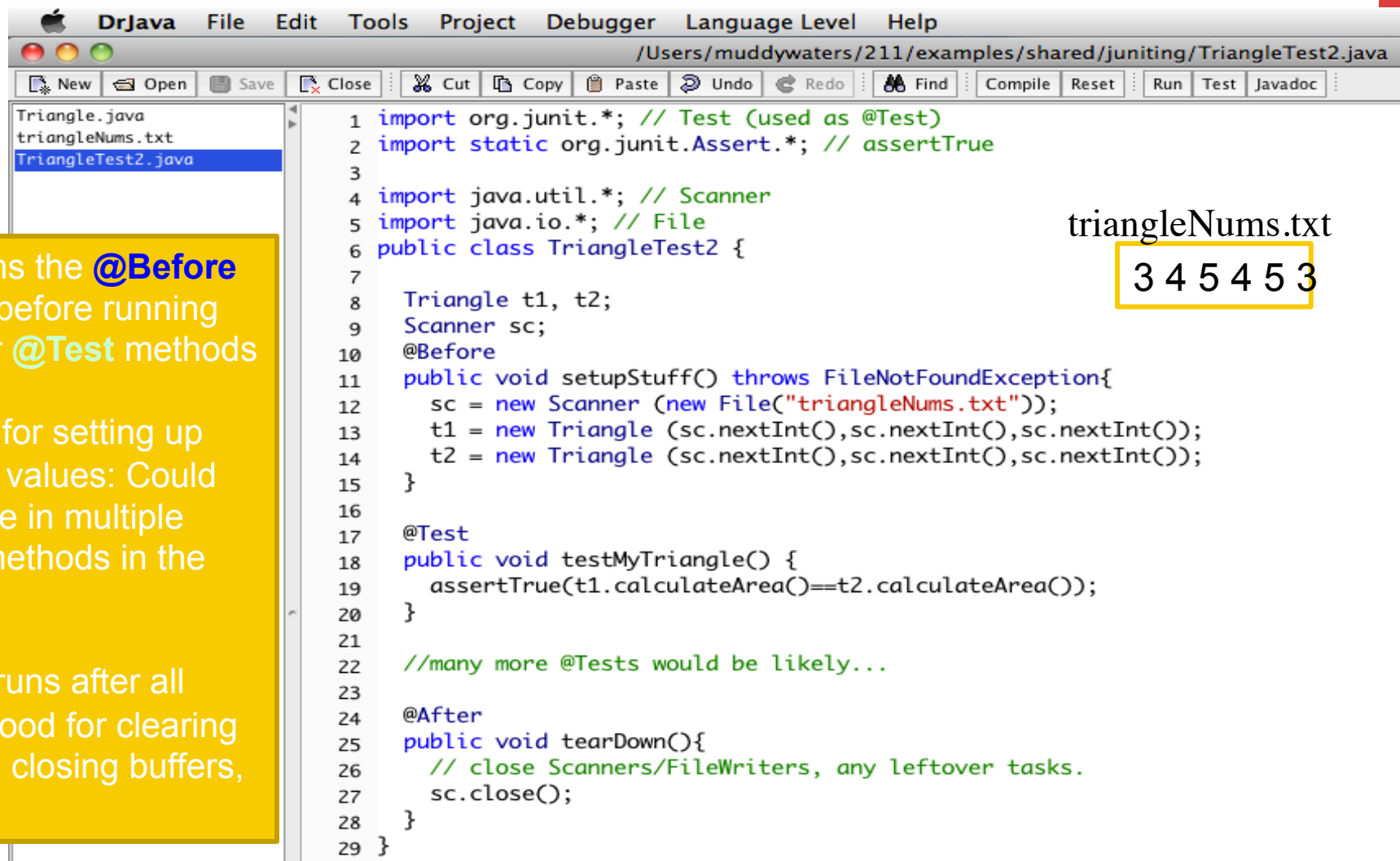
The screenshot shows the DrJava IDE interface. At the top, there are tabs for "Interactions", "Console", "Compiler Output", and "Test Output". The "Test Output" tab is active, displaying the following text:

```
1 test failed:  
TriangleTest  
  testMyTriangle  
File: /Users/muddywaters/211/examples/shared/juniting/TriangleTest.java [line: 9]  
Failure: java.lang.AssertionError:
```

On the right side of the window, there is a "Test Progress" section with a red progress bar and a close button. Below it are up and down arrow buttons, a "Show Stack Trace" button, and a checked checkbox for "Highlight source".

At the bottom of the IDE, the status bar shows "Editing /Users/muddywaters/211/examples/shared/juniting/TriangleTest.java" and the time "9:0".

JUnit: @Before many @Tests (in DrJava)



```
1 import org.junit.*; // Test (used as @Test)
2 import static org.junit.Assert.*; // assertTrue
3
4 import java.util.*; // Scanner
5 import java.io.*; // File
6 public class TriangleTest2 {
7
8     Triangle t1, t2;
9     Scanner sc;
10    @Before
11    public void setupStuff() throws FileNotFoundException{
12        sc = new Scanner (new File("triangleNums.txt"));
13        t1 = new Triangle (sc.nextInt(),sc.nextInt(),sc.nextInt());
14        t2 = new Triangle (sc.nextInt(),sc.nextInt(),sc.nextInt());
15    }
16
17    @Test
18    public void testMyTriangle() {
19        assertTrue(t1.calculateArea()==t2.calculateArea());
20    }
21
22    //many more @Tests would be likely...
23
24    @After
25    public void tearDown(){
26        // close Scanners/FileWriters, any leftover tasks.
27        sc.close();
28    }
29 }
```

triangleNums.txt
3 4 5 4 5 3

JUnit runs the **@Before** method before running all of our **@Test** methods

→ great for setting up our data values: Could use value in multiple **@Test** methods in the class

@After runs after all tests. Good for clearing memory, closing buffers, etc.

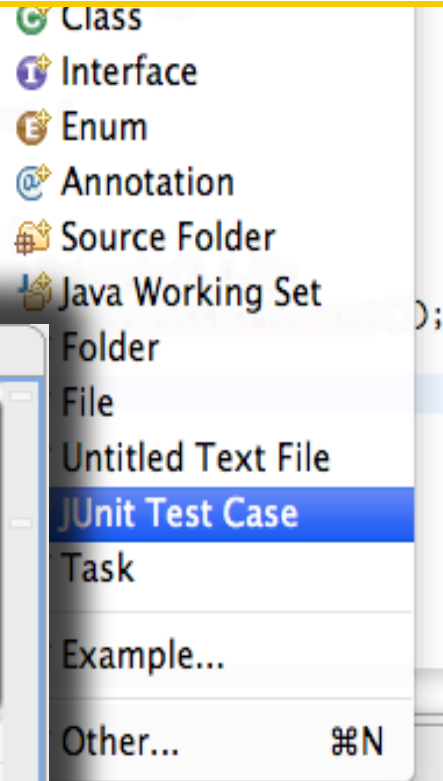
JUnit Example – In Eclipse

- Create test case
- assert something
- run the test

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

public class TriangleTest {

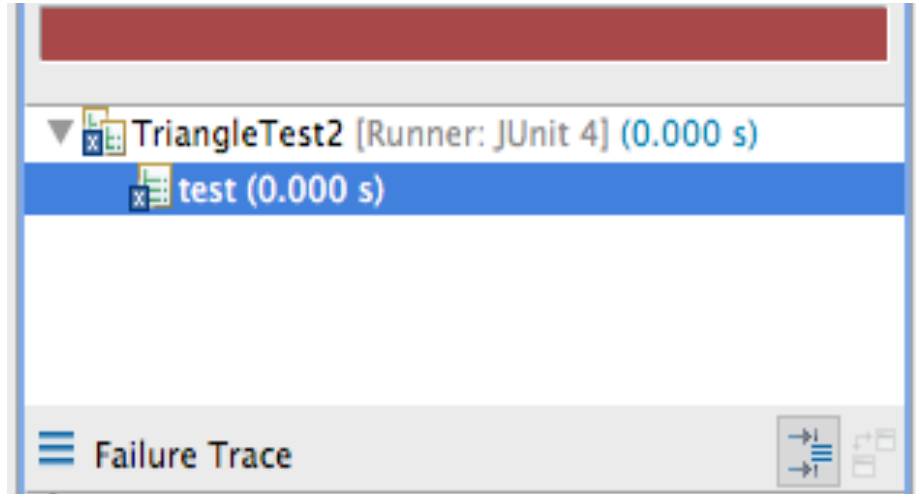
    @Test
    public void test() {
        Triangle t = new Triangle(3,4,5);
        assertTrue(6.0==t.calculateArea());
    }
}
```



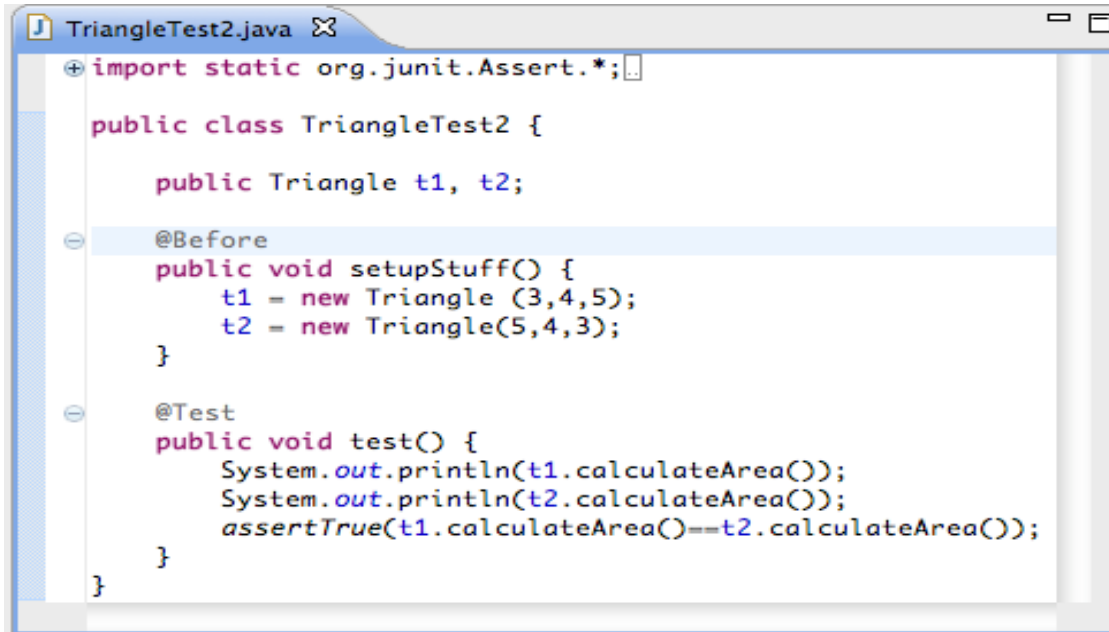
JUnit: Running Tests (in Eclipse)

We can run our JUnit tests, get results (green=good, red=bad), and even look at a trace of where the failing test failed.

Here, our assertion of equality being true failed. Time to go double check Heron's Formula as we've implemented it!



JUnit: @Before many @Tests (in Eclipse)



```
TriangleTest2.java
+ import static org.junit.Assert.*;

public class TriangleTest2 {

    public Triangle t1, t2;

    @Before
    public void setupStuff() {
        t1 = new Triangle (3,4,5);
        t2 = new Triangle(5,4,3);
    }

    @Test
    public void test() {
        System.out.println(t1.calculateArea());
        System.out.println(t2.calculateArea());
        assertTrue(t1.calculateArea()==t2.calculateArea());
    }
}
```

JUnit runs the @Before method before running all of our @Test methods
→ great for setting up our data values: Could have multiple @Test methods in the class

@After runs after all tests. Good for clearing memory, closing buffers, etc.

Debuggers

What is a Debugger?

Debugger: an interactive tool that allows us to slowly **trace through a running program**, inspecting values, modifying values, and generally observing the execution of the program.

→ doesn't modify the source code

→ directly shows what memory is in variables.

It is a ***valuable alternative to inserting print statements***, re-running the code, and seeing what printed out.

→ we can't insert extra print statements once it begins running

→ the print statements change the meaning of the code (bad)

Usage: Stepping

Breakpoint: a location in source code where the debugger must pause when reached.

→ (Then stepping is allowed).

Step: evaluate the next statement; pause.

Step into: take a step. if a method is called, continue stepping through its code

Step over: take a step. if a method is called, completely call/return from it as a single action.

Step out: run code until a return statement, and pause at that returned-to site.

→ if other breakpoints reached, pause no matter what

DrJava and Debugging

Debugging is built in.

To set a breakpoint:

- put cursor on line, press Ctrl-B (⌘-B)
- debugger pauses **before** running the statement.

To run in debugging mode:

- enable Debugger → Debug Mode (⌘-Shift-D)
- run as normal; breakpoints will trigger.
- choose to step in various ways, or resume.

Watches / Watchlist

A variable of interest (one we want to watch) may be added to a watch list.

- each time the debugger pauses, each watched value is re-inspected and displayed.
- use for locals, fields, etc.

An interactive interpreter may allow typing in other expressions to see their values/effect.

DrJava and WatchLists

With Debug Mode enabled, the Debugging Panel is visible.

→ Watches area (very useful!)

→ Stacks/Threads (we won't be looking here)

type in variables that you'll want to track.

- click in upper-leftmost cell under "Name" to type new entry.
- might only be in scope during part of execution; that's ok.

Modifying Values Mid-execution

Debuggers often allow you to directly modify values.

→ explore "what-if" scenarios

→ quickly test different parts of code

Can't modify source code while it runs (at least, won't see those changes until the next run/debugging).

effort of modifying values isn't remembered in later runs/
debugging.

DrJava and Mid-execution Modifications

While debugging, and paused:

- type assignment statements into the Interactions Panel.
- watch-list might not show change until next step is taken.

Other IDEs

Most heavy-weight IDEs will have debuggers of various design.

→ Eclipse, jGrasp, IntelliJ IDEA, JSwat, etc.

Many languages have debuggers available.

→ you'll experience the gdb (GNU debugger) for C at some point.

Not all debuggers are built into an IDE

→ we have command-line debuggers, too!

No matter what, the same principles apply: set breakpoints, watch values, inspect/change values as it runs, step into/over/out, etc.