Unit Testing Code

Testing a unit of code

```
int findMax(int a, int b, int c) {
            if (a > b) {
                  if (a > c) return a;
                  else return c;
            }
            else {
                  if (b > c) return b;
                  else return a;          // should be c
            }
```

Testing a unit of code

```
int findMax(int a, int b, int c) {
            if (a > b) {
                  if (a > c) return a;
                  else return c;
            }
            else {
                  if (b > c) return b;
                  else return a;          // should be c
            }
```

Identify:

1. **INPUT,** possibly including any state variables
2. Generate, manually or through means OUTSIDE of your code an **EXPECTED OUTPUT**
3. Executed code to get an **ACTUAL OUTPUT**

Test Case

- An Input
- An EXPECTED output
- And an ACTUAL output.

- If an expected output doesn’t match the actual output, one of the two is wrong
  - Usually, but not necessarily, the actual output is wrong

Testing a unit of code

```
int findMax(int a, int b, int c) {
            if (a > b) {
                  if (a > c) return a;
                  else return c;
            }
            else {
                  if (b > c) return b;
                  else return a;          // should be c
            }
```

Test Case #1: Input = {3,2,1}; Expected output = 3; Actual output = 3

PASS!!!

Test Case #2: Input = {1,2,3}; Expected output = 3; Actual output = 1

FAIL!!!

Testing is like potato chips

- They both contribute to my overall poor health
- Additionally, you can’t have just one
  - One test passing may have no bearing on another test passing

Why does Test 1 Pass and Not Test 2

- Test 1 does not cover/execute the underlying ***FAULT*** in the code.
- A fault is a static defect in the code, or “bug”

Test Case #1: Input = {3,2,1}; Expected output = 3; Actual output = 3

PASS!!!

Test Case #2: Input = {1,2,3}; Expected output = 3; Actual output = 1

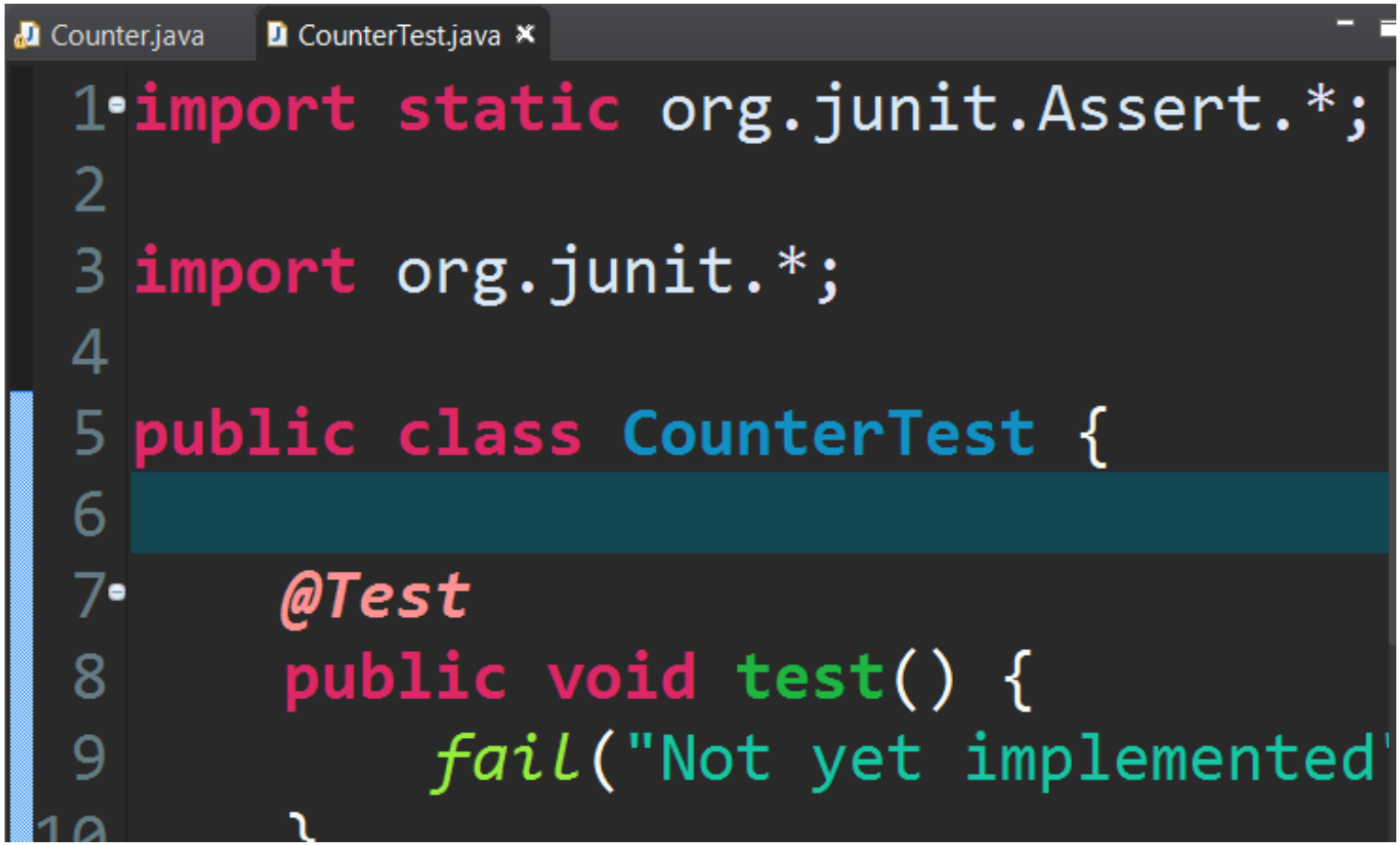
FAIL!!!

JUnit

- An automatic testing tool that allows you to write tests once and continue to use them again and again
- In this way, if you change something later that breaks code that worked previously, you will immediately know because your tests fail
- Technically not built into Java

Import Statements

Start all Test files with the two important statements below.



Writing a test

```
@Test  //This must be before every test function
public void testFindMax0() { //Notice – no static keyword
        //inputs
        int a = 3;
        int b = 2;
        int c = 1;
        //expected – generated manually
        int expected = 3;
        //actual – Execute the code with the above input
        int actual = max(a, b, c);
        //Assertion – if the two things below aren't equal, the
        //              test fails. Always put expected argument first.
        assertEquals(expected, actual).
}
```

Writi

This is not optional!

```
@Test  //This must be before every test function
public void testFindMax0() { //Notice – no static keyword
        //inputs
        int a = 3;
        int b = 2;
        int c = 1;
        //expected – generated manually
        int expected = 3;
        //actual – Execute the code with the above input
        int actual = max(a, b, c);
        //Assertion – if the two things below aren't equal, the
        //              test fails. Always put expected argument first.
        assertEquals(expected, actual).
}
```

Write

This is not optional!

```
@Test
public void testFindMax0() {
        //have a error message if test fails
        String message = “ERROR: findMax(3,2,1) returned an
                                        incorrect result”;
        int expected = 3; //you manually find and enter this
        int actual = findMax(3,2,1); //generated by your code
        assertEquals(message, expected, actual); //the test
}
```

What a test failing means

- A test failing doesn't always mean the code has a bug
  - The test could be written wrong (that is, the test writer came up with the wrong expected output)
- A test passing doesn't mean there is no bug
  - The test code not execute a buggy statement
  - The test could execute a buggy statement in a way that a failure doesn't manifest

Consider these test cases

```
int findMax(int a, int b, int c) {
            if (a > b) {
                  if (a > c) return a;
                  else return c;
            }
            else {
                  if (b > c) return b;
                  else return a;          // should be c
            }
```

Test Case #3: Input = {1,1,1}; Expected output = 1; Actual output = 1

PASS!!!

Test Case #4: Input = {4,5,6}; **Expected output = 4**; Actual output = 4

PASS!!!

Consider these test cases

- Covering the fault doesn't mean your test will fail.
- Your test could be erroneous!

Test Case #3: Input = {1,1,1}; Expected output = 1; Actual output = 1

PASS!!!

Test Case #4: Input = {4,5,6}; **Expected output = 4**; Actual output = 4

PASS!!!

False positive

- If your test is erroneous, you could get a false positive.
- This test DOESN”T cover the fault, but still fails, due to erroneous testing

Test Case #4: Input = {9,8,7}; **Expected output = 7**; Actual output = 9

FAIL!!!

Testing Strategies

- Exhaustive Testing
  - Attempt a test with every possible input
  - Not even remotely feasible in most cases

- Random Testing
  - Select random inputs
  - Likely to miss narrow inputs that are special cases (example, dividing by zero)

Testing Strategies

- Black-box Testing
  - Select inputs based on the specification space
  - “Assume the code can’t be seen”
  - *We focus on this one*

- White-box Testing
  - Select inputs based on the code itself
  - Have every line of code covered by at least one test

The need for automatic testing

- Automatic testing (such as JUnit) allows for testing rapidly after each update
- If an update breaks a test, a commit can be rejected
- Ensure you don’t break something that already worked
  - Not fool proof

Black-Box Testing Exercise

- Write tests based on the specification.
- Identify “spaces” of solutions that should behave similarly
  - Equivalence partitioning (spaces that “behave” the same)
- Identify “edge cases”

Example, Power Function

- What are the spaces of inputs?
- What are the edge cases?
- What tests should we write?

Additional In-Class Exercise

```
public double calculateBill (
        int credits, double overdue,
        boolean exempt)
```

- total is:
  - 8000/credit if less than 3 credits
  - 6000/credit if 3-6 credits
  - 5500/credit if more than 6 credits
- increase total by 10% if overdue is more than 2000
- increase overdue by 10% if exempt is false
- return sum of total and overdue

Black-Box Testing

- If we have a test for 4 credits, do we also need to test 5?
- If we have a test for 8 credits, do we also need to test 10?
- If we have a test for overdue = 2500, do we need one for 3000?

Equivalence Partitioning

- Assumption: “Similar” inputs, relative to the spec, behave similarly.
- Therefore, divide the space of inputs into similar groups and pick a representative example