Com S 417
Software Testing

Fall 2017 – Week 1, Lecture 2

Terminology: Bugs

Talking about Bugs

Generic

“Bug” is a generic term: it doesn’t distinguish between the mistake in the code and the resulting unexpected behavior.

More precise (from O&A)

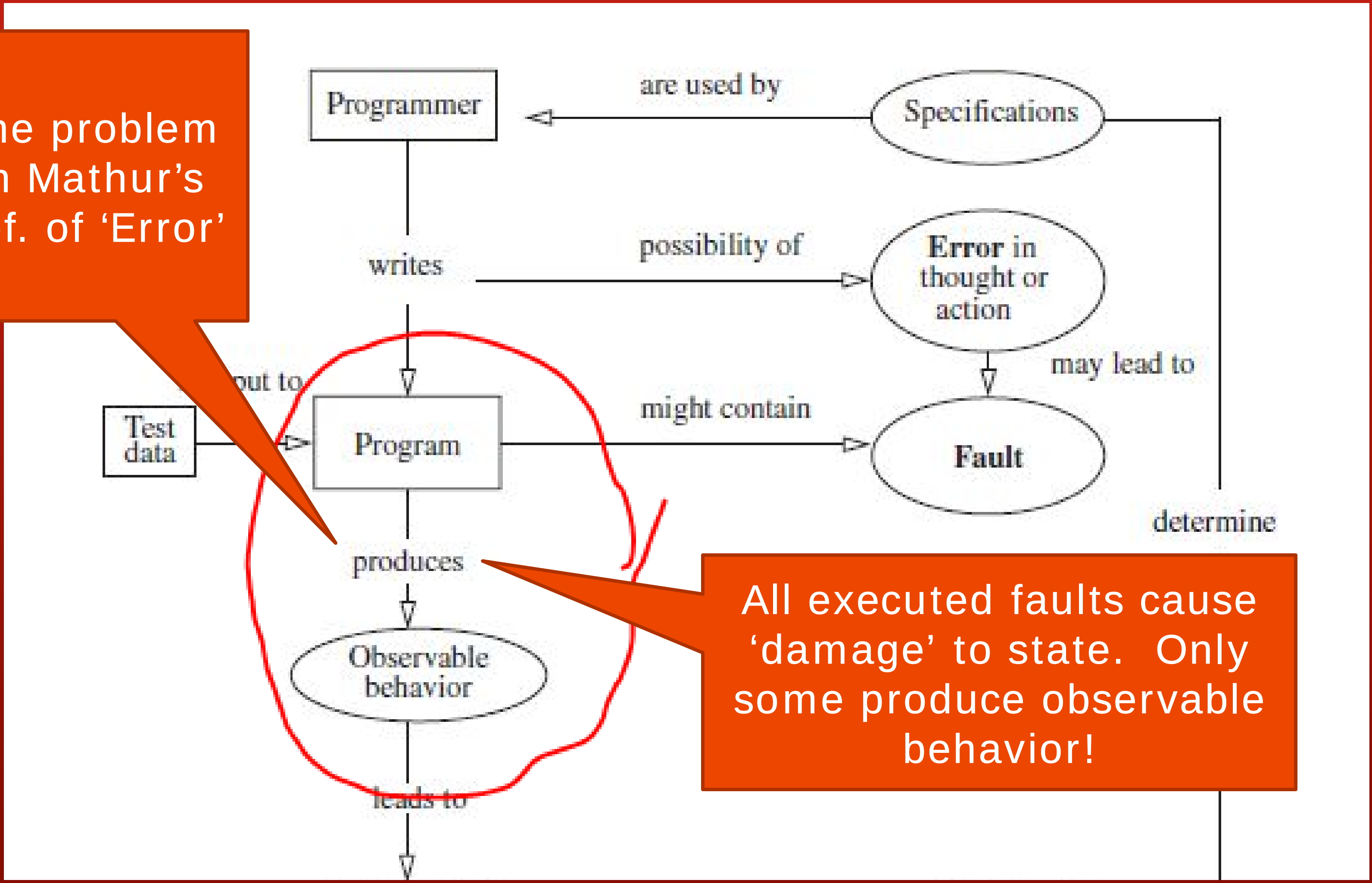
Fault: A static defect in the software. The fault exists as soon as the programmer writes the offending code.

Error: An incorrect internal state that is he manifestation of some fault.

Failure: External, incorrect behavior with respect to the requirements of other description of the expected behavior.

NOTE:
This definition of ‘Error’ differs from the textbook. We will use *this* definition.

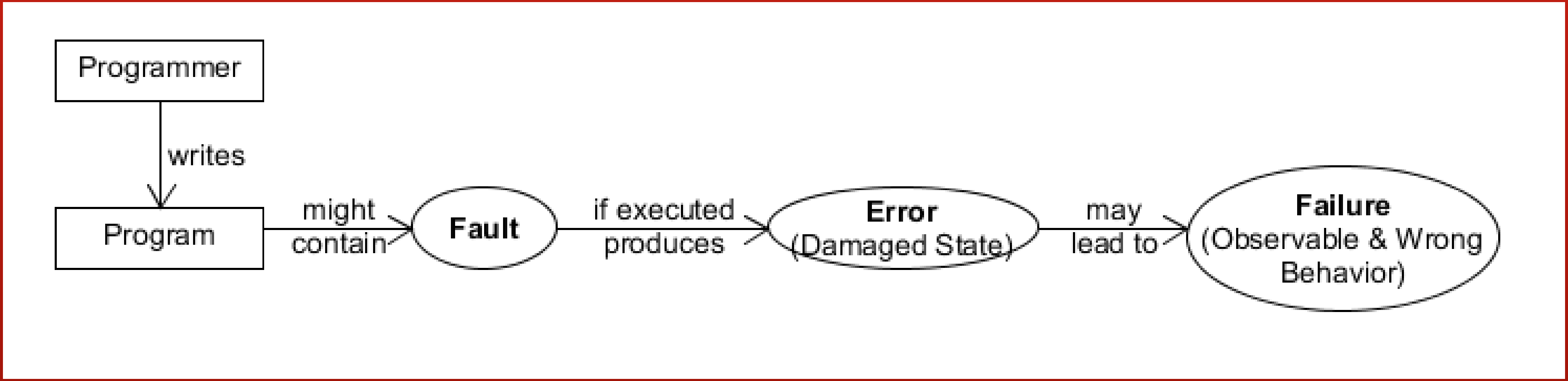
Thinking Error vs. runtime state Error



The problem in Mathur's def. of 'Error'

All executed faults cause 'damage' to state. Only some produce observable behavior!

The Fault-Error-Failure Cycle

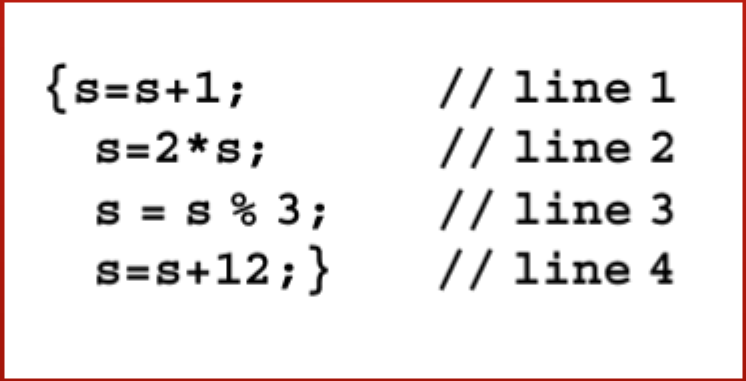


- Damage to state could include setting the program counter such that it would execute inappropriate code. (e.g., taking a wrong branch in an if)

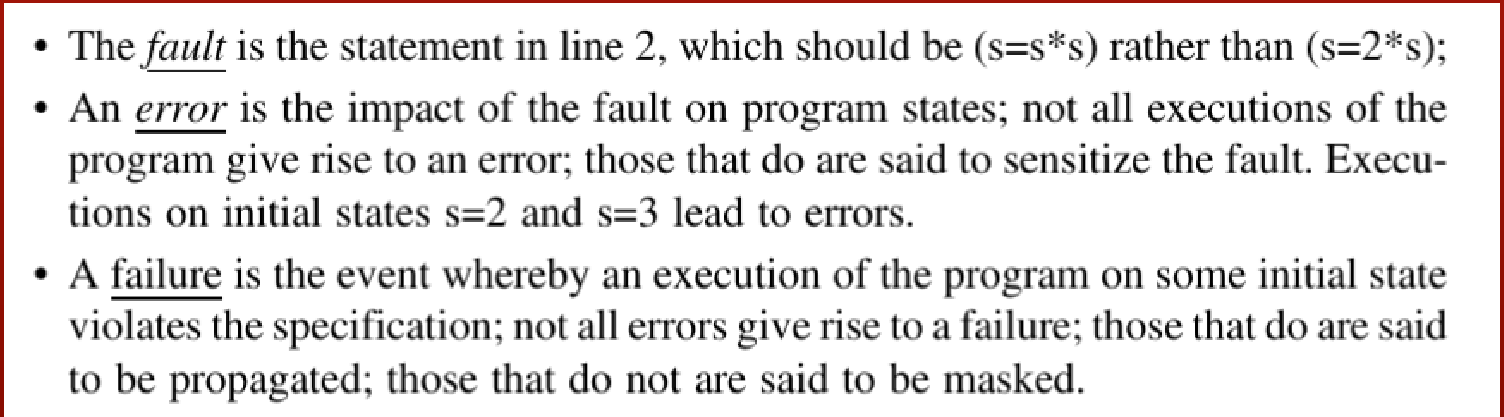
From: *Software Testing : Concepts and Operations,* by Ali Mili , and Fairouz Tchier , 2015.

An example

- Example to produce (s+1)^2 mod 3



```
{s=s+1;         // line 1
  s=2*s;        // line 2
  s = s % 3;    // line 3
  s=s+12;}      // line 4
```



- The *fault* is the statement in line 2, which should be (s=s*s) rather than (s=2*s);
- An *error* is the impact of the fault on program states; not all executions of the program give rise to an error; those that do are said to sensitize the fault. Executions on initial states s=2 and s=3 lead to errors.
- A failure is the event whereby an execution of the program on some initial state violates the specification; not all errors give rise to a failure; those that do are said to be propagated; those that do not are said to be masked.

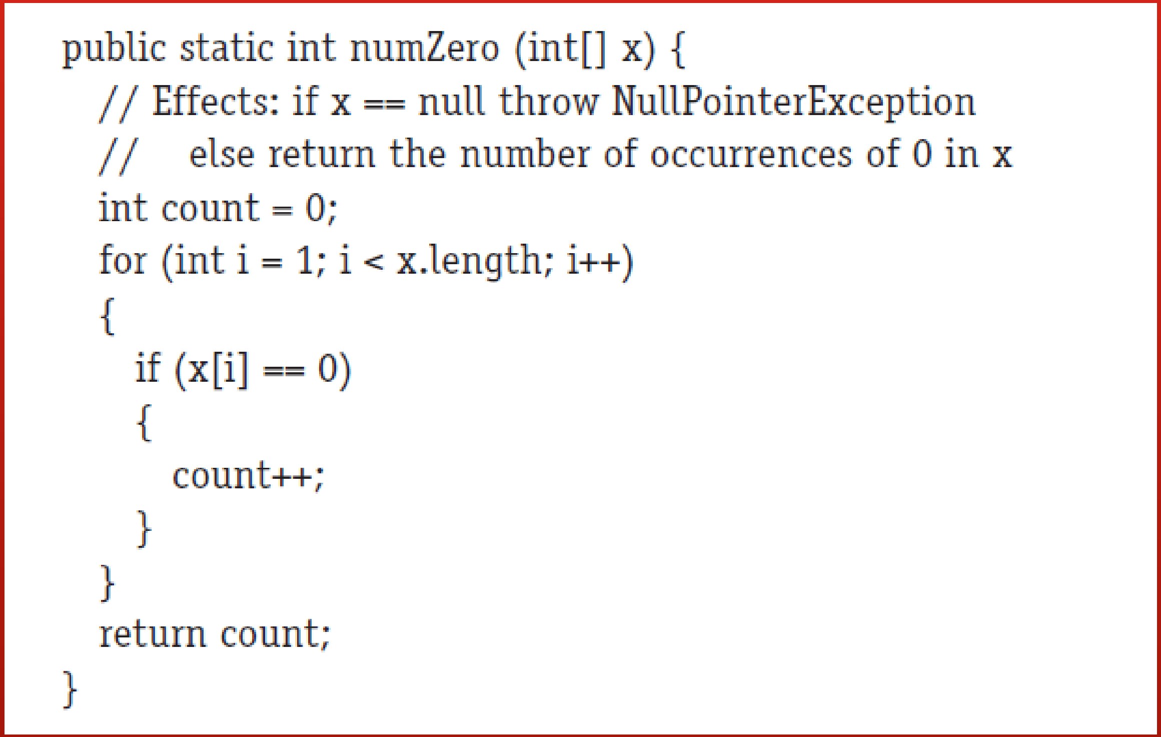
From: *Software Testing : Concepts and Operations*, by Ali Mili , and Fairouz Tchier , 2015.

RIPR Model
For Fault Activation and Detection

RIPR Fault Activation Model

Explains “degrees of activation” related to “triggering a failure” a bug.

- Reachability: the line containing the fault must be reached (executed).
- Infection: executing the fault must cause the state of the program to become incorrect.
- Propagation: the incorrect state must result in some incorrect and observable behavior.
- Reveal: failure was observed or detected.



```
public static int numZero (int[] x) {
  // Effects: if x == null throw NullPointerException
  //    else return the number of occurrences of 0 in x
  int count = 0;
  for (int i = 1; i < x.length; i++)
  {
    if (x[i] == 0)
    {
      count++;
    }
  }
  return count;
}
```

Where is the fault?
What input will reach the fault?
How should we represent state?
With what input would the error not propagate?
What input will cause a failure?
How could a propagated error not be revealed?
How would you observe the failure?
How could you automate a test for reachability?

Code from A&O, p 12.

Instrumentation

- Code/functionality added to the software primarily to improve observability.
- Not all instrumentation can be included in production code.
  - Java assert keyword can be helpful (along with AOP techniques).
  - Custom log/trace appenders can log to a stream during test and be evaluated for expected content.
  - *ABSOLUTELY DO NOT* keep instrumented and non-instrumented variants of the source!!

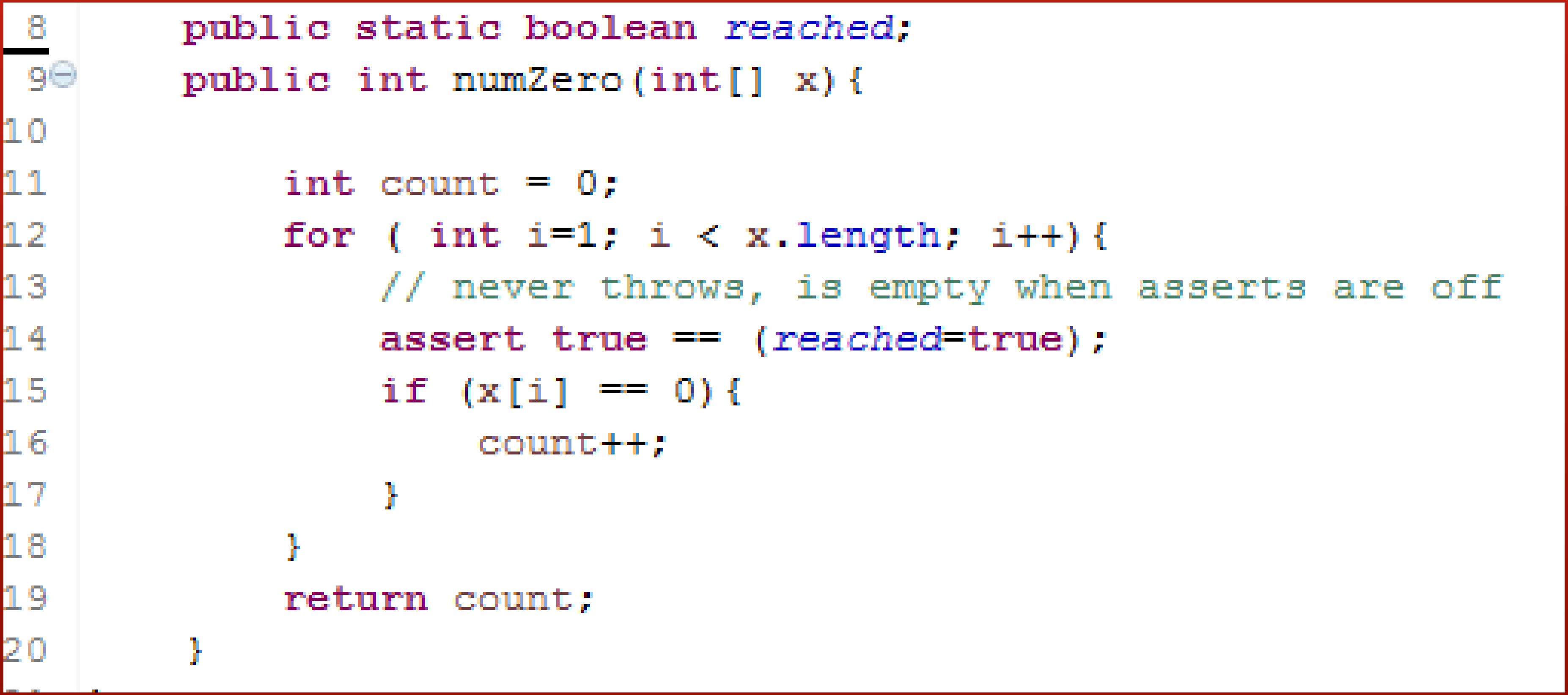
About *assert*

- assert has nothing to do with junit; it is a java keyword.
- it is not a built-in method! assert starts a statement.
- Syntax

```
assert Expression ; // Expression must be boolean
//or
assert Expression1 : Expression2;
// Expression1 is boolean,
// Expression2 must have a value
// (for exception message);
```

- Normally asserts compile to empty statements.

assert Example



```
    public static boolean reached;
    public int numZero(int[] x){

        int count = 0;
        for ( int i=1; i < x.length; i++){
            // never throws, is empty when asserts are off
            assert true == (reached=true);
            if (x[i] == 0){
                count++;
            }
        }
        return count;
    }
```

About Junit

- Junit is a tookit for constructing test harnesses.
- What are the tradeoffs between creating a Junit test and using a main() to drive your code?

Overall Idea

- A test case is a single method – and performs some check that can pass or fail.
- Several test methods (or test cases) are in a single test class.
- A TestSuite will run tests in several test classes.
- Typically, there are Multiple test suites based on intent.

From: a similar presentation by Dr. Mitre

On Eclipse - Test Directory

- Create a **source folder** called *test* in the project. This should be in the same level as *src* and *bin* folders.

- Create test classes in packages mirroring original classes.
  - For example:
    - Suppose you have a class com.sample.Account
    - Create packages com and com.sample in test and create test class in test/com.sample package.
- Advantages
  - source and test code separate (for development/deployment)
  - similar package level access
  - easier to run entire suite of tests

Test Class

- Name should indicate INTENT of testing.
- To start, one or more test class per source class.
  - Examples: AccountBasicTest, AccountOverdraftTest etc.
- Later, one or more test classes per group of classes working together (again – name by intent).
- Each test class will have many test methods.

Running Tests

- Make sure junit is on build path.
  - right click on project->properties-> Java Build Path, click on libraries tab, click on Add Library, click on JUnit, click next, select JUNIT 4 and Finish.

- Right click on individual file and “run as” junit 4 tests
- Or, right click on “test" directory or package and “run as” open run Dialog and customize.

Assertion Types

Types of Evaluation

- check a condition
- check if object references null?
- check if objects are identical?
- check if objects are equal?
- check if arrays are equal?
- fail

Check a condition

- assertTrue(condition)
- assertTrue(error_msg, condition)

- assertFalse(condition)
- assertFalse(error_msg, condition)

Check object reference null?

- assertNull(obj)
- assertNull(error_msg, obj)

- assertNotNull(obj)
- assertNotNull(error_msg, obj)

Check object reference identical?

- assertSame(expected_obj, actual_obj)
- assertSame(error_msg, expected_obj, actual_obj)

- assertNotSame(expected_obj, actual_obj)
- assertNotSame(error_msg, expected_obj, actual_obj)

- Here we mean that both expected and actual are the SAME object.

From: a similar presentation by Dr. Mitre

Check object reference equal?

- assertEquals(expected_obj, actual_obj)
- assertEquals(error_msg, expected_obj, actual_obj)
- Here we mean that both expected and actual are equal to each other – they will use the equals() operator if available – else it means they will use Object’s equal operator which checks for “same”-ness.
- For primitive types, they get converted to objects first and are then checked.
- for real numbers - assertEquals(expected, actual, delta)
  - check if sufficiently close to each other!

Check arrays equal?

- assertArrayEquals(expected_obj, actual_obj)
- assertArrayEquals(error_msg, expected_obj, actual_obj)

- Both arrays must be same size.
- Elements must be equal.
- Elements can themselves be arrays!

fail

- always fail the test if you reach this statement!

- fail()
- fail(error_msg)

Annotation Types

List of Items

- @Test and @Ignore
- @Before @After
- @BeforeClass @AfterClass
- handling exceptions
- handling timeouts

@Test and @Ignore

- @Test annotation to method indicates that the method is a test case.
  - (Note: earlier versions of JUNIT required using specific naming conventions AND extending TestCase class etc).
  - Method must be **public** and have **no parameters**.
- @Ignore annotation means ignore this test when running.
- @Ignore(message) – also can give a reason for ignoring.

From: a similar presentation by Dr. Mitre

@Before and @After

- When you want some initializations to be done BEFORE every test in the test class and some cleanups to be done AFTER every test in the test class (regardless of result or exceptions).
  - Method must be **public** and have **no parameters**.
  - Be aware of sideeffects of each test.
- These are known as **test-fixtures**.
- One guideline - Create as many test classes as test-fixtures needed.

@BeforeClass and @AfterClass

- When you want something done ONCE initially before any test cases run in the test class and then a cleanup once all tests are done.

- The before class and after class method must be **static** (in addition to being public and with no parameters).

handling exceptions

- add **expected** attribute to @Test

- @Test(expected=ArithmeticException.class)
    - Error IF no exception thrown OR
    - wrong type of exception thrown.

handling exceptions

- can catch the thrown exception and process it also – before failing or passing.

```
public void testException()
{
   try
   {
      exceptionCausingMethod();

      // If this point is reached, the expected
      // exception was not thrown.

      fail("Exception should have occurred");
   }
   catch ( ExceptedTypeOfException exc )
   {
      String expected = "A suitable error message";
      String actual = exc.getMessage();
      Assert.assertEquals( expected, actual );
   }
}
```

timeouts

- add **timeout** attribute to @Test
- @Test(timeout=2)
    - if execution time exceeds 2 milliseconds – the test will fail.

Test Suites

Steps

1. Create a new class
2. @RunWith(Suite.class)
3. @Suite.SuiteClasses({ CalculatorTest.class, SquareTest.class })
4. public class AllCalculatorTests { } //empty class

Eclipse will create a dynamic “suite” if you select a set of test classes or test packages and then “Run As Junit Test”.

Lab 1

- for some piece of code with one or more faults: create five tests (with different inputs):
  - one passes whenever the fault is not reached. (!Rx)
  - one passes only if the fault is reached, but does not infect. (R)
  - one passes only if the fault is reached, infects, but does not propagate. (RI^P)
  - one passes only if the fault is reached, infects, and propagates – i.e., you detect a failure in the output. (RIPR)
  - one passes only if the fault is not reached and the result is correct.(!R no other failure)
- The code you use should be unarguably original and unique, but does not need to be large or complex.

Lab 1

- Due date: Thursday, Sep 5, midnight.
- Submission:
    - submit a zip with your project (probably just two files) through blackboard.
    - include separately, a screen capture of the SUT, with the fault circled.
    - write good clean code. Follow the test guidelines here, and generally accepted practice.
    - name each test clearly so there is no question which condition it is meant to identify.

Lab 1

- Notes:
  - While we will attempt to give feedback on your project, we will not try to correct your code during grading. If you have problems, visit a TA or me so you can get them fixed.
  - DO NOT submit class files.
  - Name your project and your submitted zip lastName_firstName_lab1
  - Use Junit4.
  - While we are using Junit, we are *not* using junit to test software in the normal meaning of test. You must already know where and what the fault is to complete this project.

Lab 1

- Additional Resources:
  - Unit Tests: Bob Martin, *Clean Code*, Chapter 9.
  - RIPR: Ammann & Offutt, p12-14
  - Junit javadoc