DA-IICT

# IT314: Software Engineering

*JUnit Testing Framework*

Saurabh Tiwari

1

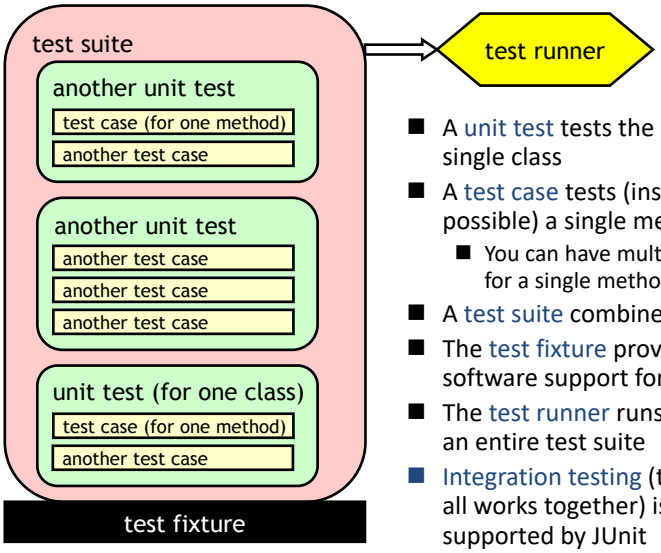
If you don't unit test then you are not a software engineer, you are a typist who understands a programming language.

--Moses Jones

## Example: Old way vs. New way

- ```
  int max(int a, int b) {
      if (a > b) {
          return a;
      } else {
          return b;
      }
  }
  ```
- ```
  void testMax() {
      int x = max(3, 7);
      if (x != 7) {
          System.out.println("max(3, 7) gives " + x);
      }
      x = max(3, -7);
      if (x != 3) {
          System.out.println("max(3, -7) gives " + x);
      }
  }
  ```
- ```
  public static void main(String[] args) {
      new MyClass().testMax();
  }
  ```
- ```
  @Test
  void testMax() {
      assertEquals(7, max(3, 7));
      assertEquals(3, max(3, -7));
  }
  ```

## Junit, In PICTURES




- A unit test tests the methods in a single class
- A test case tests (insofar as possible) a single method
  - You can have multiple test cases for a single method
- A test suite combines unit tests
- The test fixture provides software support for all this
- The test runner runs unit tests or an entire test suite
- Integration testing (testing that it all works together) is not well supported by JUnit

## Writing a JUnit test class, I

- Start by importing these JUnit 4 classes:

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

- Declare your test class in the usual way

```
public class MyProgramTest {
```

- Declare an instance of the class being tested
- You can declare other variables, but *don't* give them initial values here

```
public class MyProgramTest {
    MyProgram program;
    int someVariable;
```

## Writing a JUnit test class, II

- Define a method (or several methods) to be executed *before each test*
- Initialize your variables in this method, so that each test starts with a fresh set of values

```
@Before
public void setUp() {
    program = new MyProgram();
    someVariable = 1000;
}
```

- You can define one or more methods to be executed after each test
- Typically such methods release resources, such as files
- Usually there is no need to bother with this method

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

## A Simple Example

- Suppose you have a class Arithmetic with methods int multiply(int x, int y), and boolean isPositive(int x)
- import org.junit.*;
  import static org.junit.Assert.*;
- public class ArithmeticTest {

```
    @Test
    public void testMultiply() {
        assertEquals(4, Arithmetic.multiply(2, 2));
        assertEquals(-15, Arithmetic.multiply(3, -5));
    }

    @Test
    public void testIsPositive() {
        assertTrue(Arithmetic.isPositive(5));
        assertFalse(Arithmetic.isPositive(-5));
        assertFalse(Arithmetic.isPositive(0));
    }

}
```

## Example: Counter Class

- For the sake of example, we will create and test a trivial "counter" class
  - The constructor will create a counter and set it to zero
  - The increment method will add one to the counter and return the new value
  - The decrement method will subtract one from the counter and return the new value
- We write the test methods before we write the code
  - This has the advantages described earlier
  - However, we usually write the method **stubs** first, and let the IDE generate the test method stubs
- Don't be alarmed if, in this simple example, the JUnit tests are more code than the class itself

## JUnit tests for Counter

```
public class CounterTest {
   Counter counter1; // declare a Counter here

   @Before
   void setUp() {
      counter1 = new Counter(); // initialize the Counter here
   }

   @Test
   public void testIncrement() {
      assertTrue(counter1.increment() == 1);
      assertTrue(counter1.increment() == 2);
    }

   @Test
   public void testDecrement() {
      assertTrue(counter1.decrement() == -1);
   }
}
```

- Note that each test begins with a *brand new* counter
- This means you don't have to worry about the order in which the tests are run

## Writing a JUnit test class, III

- **This page is really only for expensive setup, such as when you need to connect to a database to do your testing**
  - If you wish, you can declare *one* method to be executed *just once*, when the class is first loaded
- 
```
@BeforeClass
public static void setUpClass() throws Exception {
      // one-time initialization code
}
```
  - If you wish, you can declare *one* method to be executed *just once*, to do cleanup after all the tests have been completed
- 
```
@AfterClass
public static void tearDownClass() throws Exception {
      // one-time cleanup code
}
```

## Special features of @Test

- You can limit how long a method is allowed to take
- This is good protection against infinite loops
- The time limit is specified in milliseconds
- The test fails if the method takes too long

```
@Test (timeout=10)
 public void greatBig() {
    assertTrue(program.ackerman(5, 5) > 10e12);
 }
```

- Some method calls should throw an exception
- You can specify that a particular exception is expected
- The test will pass if the expected exception is thrown, and fail otherwise

```
@Test (expected=IllegalArgumentException.class)
public void factorial() {
    program.factorial(-5);
}
```

## Test-Driven Development (TDD)


TDD

public method in your class
↳ at least

public → Test Driven Development

Stubs → Compile ✓
↳ runtime values ✗

1) Write test cases first → all test cases FAIL ✗
2) Implement public methods → run test cases PASS ✓

## Stubs

- In order to run our tests, the methods we are testing have to exist, but they don't have to be right
- Instead of starting with "real" code, we start with stubs—minimal methods that always return the same values
  - A stub that returns void can be written with an empty body
  - A stub that returns a number can return 0 or -1 or 666, or whatever number is most likely to be *wrong*
  - A stub that returns a boolean value should usually return false
  - A stub that returns an object of any kind (including a String or an array) should return null
- When we run our test methods with these stubs, we want the test methods to *fail!*
  - This helps "test the tests"—to help make sure that an incorrect method doesn't pass the tests

## Ignoring a test

  - The @Ignore annotation says to not run a test
- 
```
@Ignore("I don't want Dave to know this doesn't work")
@Test
public void add() {
    assertEquals(4, program.sum(2, 2));
}
```
  - You shouldn't use @Ignore without a very good reason!

## Test Suite

- You can define a suite of tests

```
@RunWith(value=Suite.class)
@SuiteClasses(value={
                MyProgramTest.class,
                AnotherTest.class,
                YetAnotherTest.class
            })
public class AllTests { }
```

## Recommended Approach

- Write a test for some method you intend to write
  - If the method is fairly complex, test only the simplest case
- Write a stub for the method
- Run the test and make sure it fails
- Replace the stub with code
  - Write just enough code to pass the tests
- Run the test
  - If it fails, debug the method (or maybe debug the test); repeat until the test passes
- If the method needs to do more, or handle more complex situations, add the tests for these first, and go back to step 3