

Jin L.C. Guo

M4 – Unit Testing

Image Source: https://upload.wikimedia.org/wikipedia/commons/2/2b/Cepaea_nemoralis_active_pair_on_tree_trunk.jpg

Covered by Aaron Sossin and Eric Liu



Aaron Sossin



Eric Liu

Slides provided by 2019 Winter TAs

Material from:

Introduction to Software Design with Java, by Martin Robillard, lecture notes for COMP 303. (LN)

(Module 4)

The Pragmatic Programmer by Andrew Hunt and David Thomas, Addison-Wesley, 2000. (PP)

(34, 43)

Clean Code by Robert C. Martin, Prentice Hall, 2008

(Chapter 9)

Objectives

- Be able to explain the foundational concepts of testing using the proper terminology;
- Understand type annotations and program reflection and be able to use them effectively;
- Be able to write unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection or mock objects;
- Be able to understand the output of a test coverage tool such as EcEmma;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Objectives – This Class (Feb 12.)

- Be able to explain the foundational concepts of testing using the proper terminology;
- Understand type annotations and program reflection and be able to use them effectively;
- Be able to write unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection or mock objects;
- Be able to understand the output of a test coverage tool such as EcEmma;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Objectives – Next Class (Feb 14.)

- Be able to explain the foundational concepts of testing using the proper terminology;
- Understand type annotations and program reflection and be able to use them effectively;
- Be able to write unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection or mock objects;
- Be able to understand the output of a test coverage tool such as EcEmma;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Objective 1: Foundational Concepts

Me: I don't need to test this functionality...

Functionality: *breaks immediately*

Me:



What is Testing? 🤔

Based on the idea that: software issues arise when code does not do what we expect.

- A fairly intuitive concept.
- You might have done some testing before!

“All software you write *will* be tested—if not by you and your team, then by the eventual users—so you might as well plan on testing it thoroughly ... ”

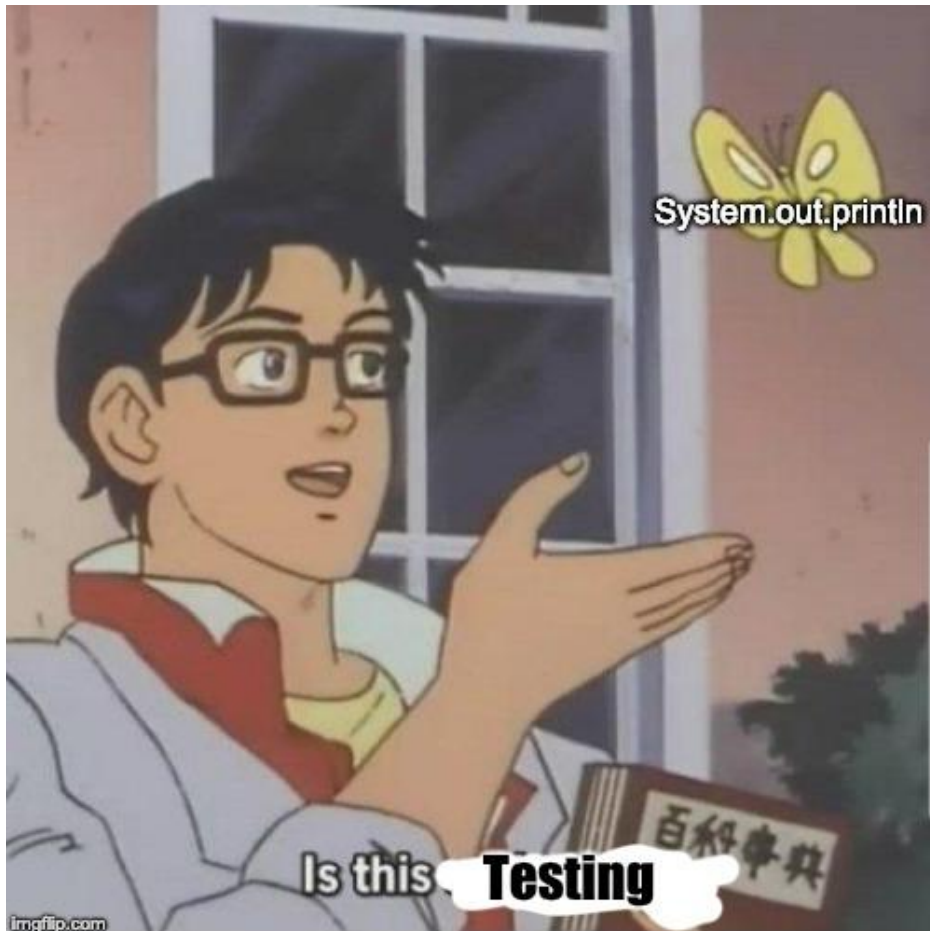
What is Testing? 🤔

A process of automatically running code to verify expectations about it.

Activity: Why automatically?

Automatic: Easy to (re-)run

- Code is evolutionary: when we change code, are we (re-)introducing bugs?
- Code should run in multiple environments, with multiple configurations.
 - Consider web applications, phone apps



(Automatic) Testing is important!

How do we get started?
What should we be testing?

What should we test?

Recall: Design by Contract

Design by Contract

- Documenting rights and responsibilities of software modules to ensure program correctness

Contracts, a Perspective:

Contracts:

- Help us clarify (and verify) expectations
- Help us write code that is easy to test

What should we test?

Recall: Design by Contract

Problem: no built-in support...

We will see how to address this.

What specifically can we test?

Recall: Separation of concerns.

We have a design that minimizes (eliminates) changes in one place affecting other places.

Let's test small, self-contained units of functionality.

Unit Testing

Validate that each unit of the software performs as designed

An idea: We can test each unit of functionality in isolation – gives us confidence that when we put them together they won't break.

Breaking things down into units (of functionality)

Activity:

What are some units of functionality in:

- A Car (Driving)
- Cards (playing Solitaire)
- ...

Testing Code

In practice, you should test functions you write as individual units

Try to test every case (combination of inputs)

Code coverage to be seen later

Objective 3: Unit Testing in Java

Unit Testing in Java

Using: **J**Unit⁽¹⁾

- A Unit Testing framework
- Part of the xUnit family of frameworks (PHPUnit, PyUnit)

It's a framework in that it provides structure for writing your tests.

(1) - <https://junit.org/junit4/>

Testing Structure

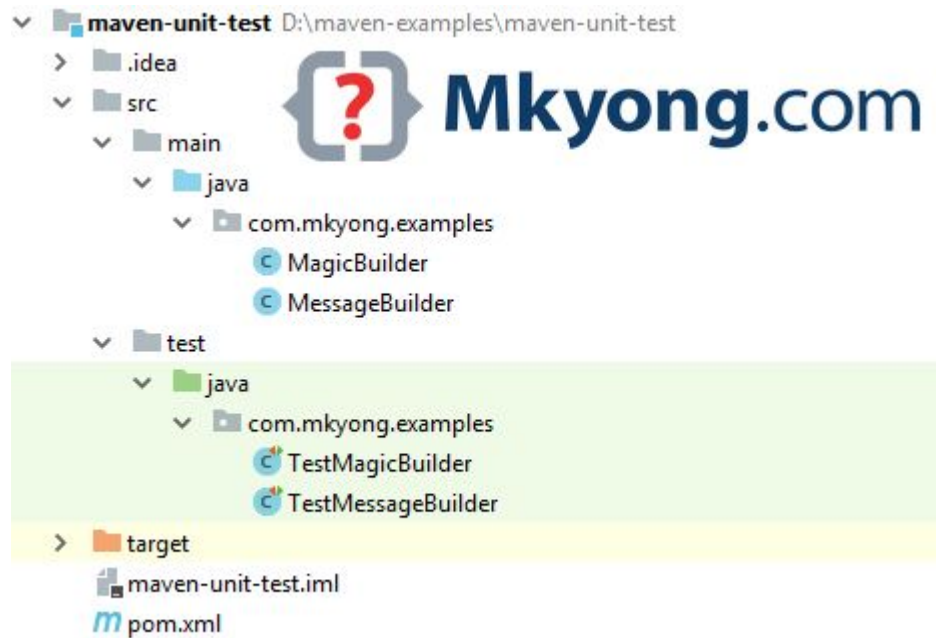
Generally, one “test” class per application class

In each class, test every contained function

Want to make sure no function has potentially undefined behaviour

Testing Structure

Mirror the document structure



Let's start with an example:

What does a unit test in JUnit look like? 🤔

A minimal test:

```
1  package testpackage;
2
3  import static org.junit.Assert.*;
4
5
6
7  public class TestCase {
8
9      @Test
10     public void test() {
11         assertTrue(true);
12     }
13
14 }
```

Note this is a fully functional unit test.

Things to Notice:

Still a regular class:

```
1  package testpackage;
2
3  + import static org.junit.Assert.*;
6
7  public class TestCase {
8
9  -    @Test
10     public void test() {
11         assertTrue(true);
12     }
13
14 }
```

(line 9) This is new ->

No main method?

@Test – An Annotation

Annotations:

- Are metadata, i.e. data about the program.

Recall: @Override - Interfaces, Subclasses

- Many more annotations (seen shortly)

Annotations – Cont'd

The `@Test` annotation tells JUnit that the following method can be run as a test case.

Annotations can take (optional) arguments:

```
@Test(expected=...)
```

```
@Test(timeout=...)
```

Annotations – Cont'd

Tutorial on the Oracle documentation website:

<https://docs.oracle.com/javase/tutorial/java/annotations/index.html>

How does JUnit run tests without a main method?

JUnit runner class

Anatomy of a Unit Test

Idea: We compare the result of executing a unit under test with an oracle.

Result: Obtained after running some specific code

Unit Under Test: an isolated, usually small part of code

Oracle: The expected result

Anatomy of a Unit Test

Example: testing the power function in the Math class

Unit Under Test: `Math.pow(2, 3)`

Result: the output of this function run on inputs 2 and 3

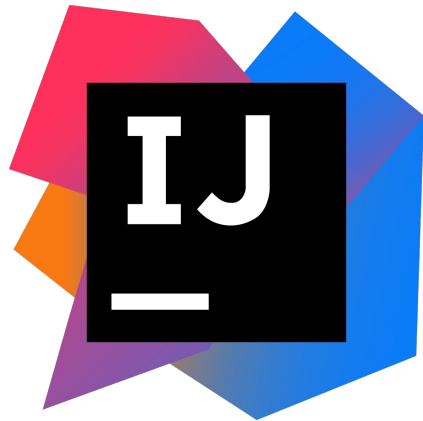
Oracle: the actual mathematical result of $2^3 = 8$

Anatomy of a Unit Test

Check that the result matches the oracle

JUnit assert statements

- assertEquals
- assertTrue
- assertEquals
- assertNull
- etc...



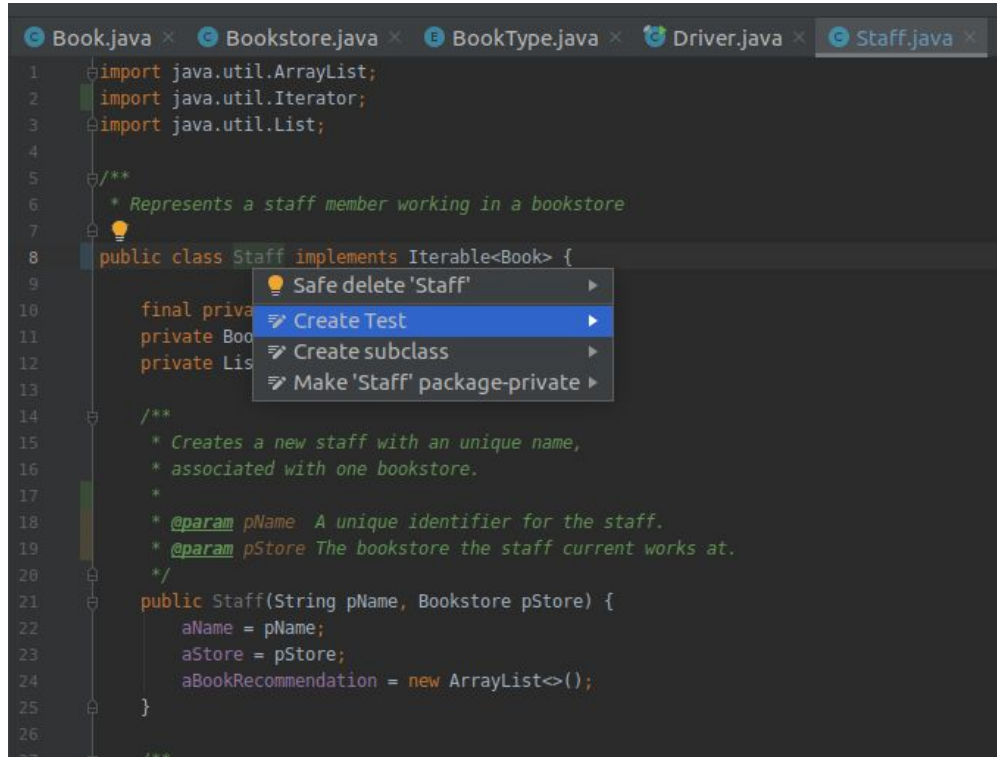
Using JUnit in IntelliJ

JUnit in IntelliJ

How do we create test suites (collection of unit tests) in IntelliJ to test our code?

Built-in and easy integration for JUnit5

JUnit in IntelliJ



The screenshot shows the IntelliJ IDEA IDE with a project containing several Java files: Book.java, Bookstore.java, BookType.java, Driver.java, and Staff.java. The Staff.java file is open, and the line `public class Staff implements Iterable<Book> {` is highlighted. A context menu is displayed over this line, offering several actions: "Safe delete 'Staff'", "Create Test" (which is highlighted in blue), "Create subclass", and "Make 'Staff' package-private". The code in the background includes imports for `java.util.ArrayList`, `java.util.Iterator`, and `java.util.List`, a Javadoc comment describing the Staff class, and the start of the Staff constructor.

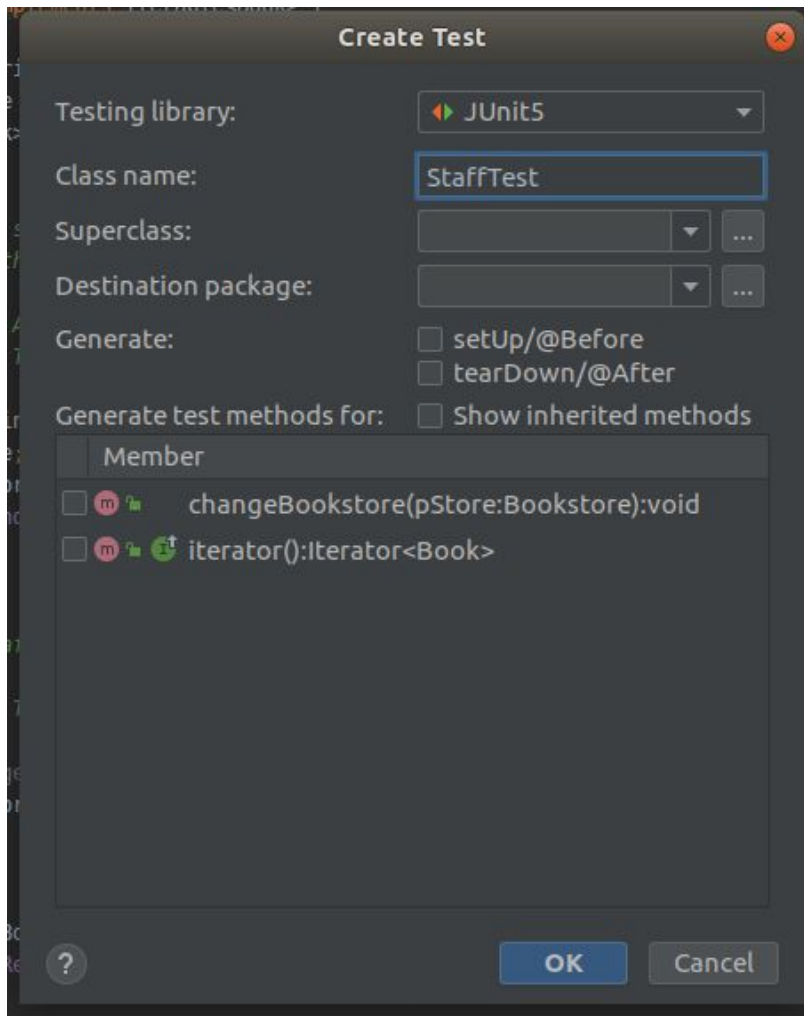
```
1 import java.util.ArrayList;
2 import java.util.Iterator;
3 import java.util.List;
4
5 /**
6  * Represents a staff member working in a bookstore
7  */
8 public class Staff implements Iterable<Book> {
9
10     final private
11     private Book
12     private List
13
14     /**
15      * Creates a new staff with a unique name,
16      * associated with one bookstore.
17      *
18      * @param pName A unique identifier for the staff.
19      * @param pStore The bookstore the staff current works at.
20      */
21     public Staff(String pName, Bookstore pStore) {
22         aName = pName;
23         aStore = pStore;
24         aBookRecommendation = new ArrayList<>();
25     }
26
27     /**
```

alt-enter when
highlighting a line with
a class declaration

JUnit in IntelliJ

Very integrated and
intuitive setup

Choose which methods to
generate test declarations
for



JUnit in IntelliJ

Basic structure of a
test suite in IntelliJ

More annotations

@BeforeEach,
@AfterEach,
@BeforeAll

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class StaffTest {

    @BeforeEach
    void setUp() {
        /*
         Initialize variables which are used in every test
         */
    }

    @AfterEach
    void tearDown() {
        /*
         Whatever necessary teardown you may have
         */
    }

    @Test
    void changeBookstore() {
        /*
         Test functionality of changeBookstore() with all possible input combinations
         */
    }

    @Test
    void iterator() {
    }
}
```

Flash Light Example

Let's model a (simple) Flash Light 

It has one button that toggles it on and off:

CODE DEMO

Transition



Activity

1. Identify UUTs
2. Test the UUT against some oracle (The term *oracle* designates the correct or expected result of the execution of a UUT) ie determine the expected behavior of each UUT
3. Write the tests

```
1 package custompackage;  
2  
3 public class FlashLight {  
4  
5     private boolean isOn = false;  
6  
7     public void pressButton() {  
8         this.isOn = ! this.isOn;  
9     }  
10  
11     public boolean isOn() {  
12         return this.isOn;  
13     }  
14 }
```

Step 1: Identify UUTs

- `pressButton`
- The constructor (Technically?) – We'd just like to test the default state of the Flash Light.

```
1  package custompackage;
2
3  public class FlashLight {
4
5      private boolean isOn = false;
6
7      public void pressButton() {
8          this.isOn = ! this.isOn;
9      }
10
11     public boolean isOn() {
12         return this.isOn;
13     }
14 }
```

Step 2: Determine expected behavior of UUTs

```
2  
3 public class FlashLight {  
4  
5     private boolean isOn = false;  
6
```

Expect it to be off by default

```
7  
8     public void pressButton() {  
9         this.isOn = ! this.isOn;  
10    }
```

Expect: it to be on after a press
it to be off after two presses
It to be on after three presses

...

Step 3: Let's write this in JUnit (Demo)

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class FlashlightTest {
    @Test
    public void testOffByDefault() {
        Flashlight f = new Flashlight();
        assertFalse(f.isOn());
    }
    @Test
    public void testOnePress() {
        Flashlight f = new Flashlight();
        f.pressButton();
        assertTrue(f.isOn());
    }
    @Test
    public void testTwoPresses() {
        Flashlight f = new Flashlight();
        f.pressButton();
        f.pressButton();
        assertFalse(f.isOn());
    }
}
```

FYI:

```
@Test
public void g() {
    //This passes!
}
```

Demo similar to what
may be on exam

What did we observe?

- Independence of Tests

- JUnit provides no ordering guarantee of any kind for the execution of unit tests
- Tests need to be independent from one another
- Example: tests on OnePress and TwoPress need to be two independent and complete tests

- Repeat Initialization

- `FlashLight f = new FlashLight();`
- We can improve this...

FlashLightTest Version 2.0

```
class FlashlightTest {  
    private Flashlight f;  
    @BeforeEach  
    public void initialize() {  
        f = new Flashlight();  
    }  
    @Test  
    public void testOffByDefault() {  
        assertFalse(f.isOn());  
    }  
    @Test  
    public void testOnePress() {  
        f.pressButton();  
        assertTrue(f.isOn());  
    }  
    @Test  
    public void testTwoPresses() {  
        f.pressButton();  
        f.pressButton();  
        assertFalse(f.isOn());  
    }  
}
```

Keypoints in JUnit

- **Independency of Tests**
 - JUnit provides no ordering guarantee of any kind for the execution of unit tests
 - Tests need to be independent from one another
 - Example: tests on onepress and twopress need to be two independent and complete tests
- **@BeforeEach runs before every @Test**
 - Other useful annotations includes @BeforeClass, @After, @AfterClass

Tests and Exceptional Conditions

Next, we modify the code for FlashLight Class a little...

To add brightness level (0-5), where 0 means off and 5 means brightest

```
private int brightness = 0;

public void setBrightness(int pBrightness) {
    if (pBrightness > 5 || pBrightness < 0) {
        throw new IllegalArgumentException("Incorrect Brightness Value");
    }
    this.brightness = pBrightness;
}

public int getBrightness() {
    return this.brightness;
}
```

Tests and Exceptional Conditions

@Test //Method1

```
public void checkIncorrectBrightness() {  
    try {  
        f.setBrightness(420);  
        fail(); ←New function, automatically  
                fails a test  
    }  
    catch (IllegalArgumentException e){  
  
    }  
}
```

@Test //Method2

```
public void checkIncorrectBrightness2() {  
    assertThrows(IllegalArgumentException.class, () -> f.setBrightness(420));  
}
```

May be asked to
write on exams

Keypoints in JUnit

- Independence of Tests

- JUnit provides no ordering guarantee of any kind for the execution of unit tests
- Tests need to be independent from one another
- Example: tests on onepress and twopress need to be two independent and complete tests

- @Before runs before every @Test

- Other useful annotations includes @BeforeClass, @After, @AfterClass

- 2 ways to test exceptions

- assertThrows
- Try and catch + fail()

Encapsulation and Unit Testing

- Tests are in separate classes than the code you are testing (no access to private attributes)
- What if we want to test something private?

RULE OF THUMB: create helper methods in test class opposed to changing code in tested class.

Encapsulation and Unit Testing (cont'd)

Example: You wish to test the size of Deck class after the execution of push() and pop() methods. But, there is no non-private attribute denoting size of deck.

Solution: Create a 'helper' method in DeckTester.java that returns the size of the deck.

```
public int sizeOfDeckHelper(Deck d) {  
    List<Card> temp = new ArrayList<>();  
    int size = 0;  
    //Remove cards from deck while counting  
    while (!d.isEmpty()) {  
        size++;  
        temp.add(d.pop());  
    }  
    //Add cards back to deck  
    while (!temp.isEmpty()) {  
        d.push(temp.remove(index: temp.size() - 1));  
    }  
    return size;  
}
```

Parameterized Tests

- JUnit allows you to use parameters in a tests class
- Helps developers save time when executing the same tests which differ only in their inputs and expected results

DEMO

```
@RunWith(Parameterized.class) //NEW ANNOTATION
public class FlashlightTestParamterized {
    private int press_count;
    private boolean expected_state;
    private Flashlight f;

    @BeforeEach
    public void initialize() {
        f = new Flashlight();
    }

    //The constructor takes the parameters as input
    public FlashlightTestParamterized(int ppress_count, boolean pexpected_state) {
        this.press_count = ppress_count;
        this.expected_state = pexpected_state;
    }

    @Parameterized.Parameters //NEW ANNOTATION
    public static Collection params() {
        return Arrays.asList(new Object[][] {
            {67, false},
            {68, true},
            {69, true},
            {70, true}
        });
    }
}
```

```
//Since there are 4 parameters, the Test will be run 4 times
@Test
public void testButton() {
    assertEquals(expected_state, pressButtonXTimes(press_count));
}

//Helper method
private boolean pressButtonXTimes(int x) {
    for (int y = 0; y < x; y++) f.pressButton();
    return f.isOn();
}
```

Key points of Parameterized Tests

- Constructor that takes as input a single parameter including the input and expected result
- `@RunWith(Parameterized.class)` header
- `@Parameterized.Parameters` header denoting the static parameter factory class (that returns a 'Collection')
- For every parameter, the 'runner' in IntelliJ calls the constructor, runs the `@Before` and then runs the `@Tests`

Keypoints in JUnit

- Independence of Tests

- JUnit provides no ordering guarantee of any kind for the execution of unit tests
- Tests need to be independent from one another
- Example: tests on onepress and twopress need to be two independent and complete tests

- @Before runs before every @Test

- Other useful annotations includes @BeforeClass, @After, @AfterClass

- 2 ways to test exceptions

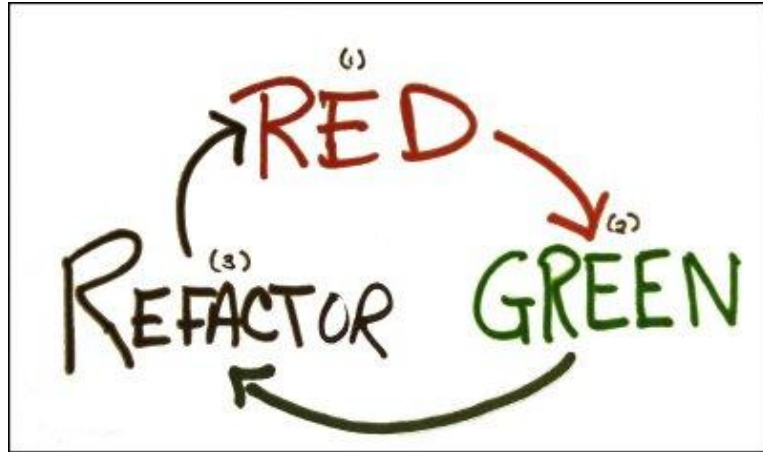
- Test annotation parametrs
- Try and catch + fail()

- Parameterized Tests

- Annotations
- Constructor

An aside: Test-Driven Development (TDD)

- Writing tests before you write the implementation.
- Only write new code when a test fails!



An aside: Test-Driven Development (TDD)

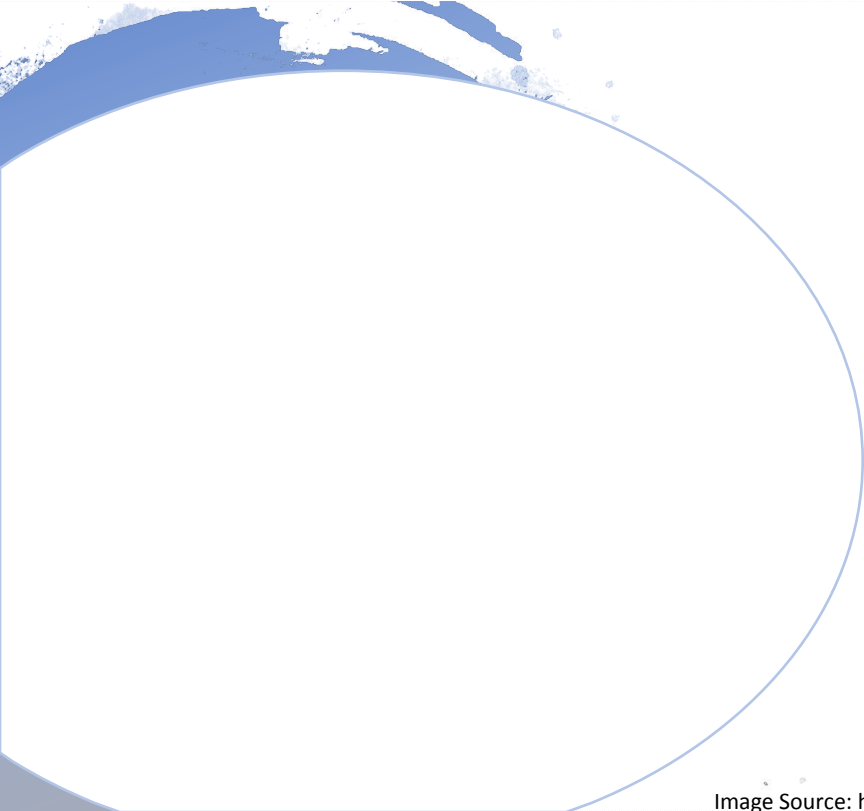
Writing a failing unit test: **RED**

Write the minimum code that makes the test pass:
GREEN

Rewrite the code so it follows standards, doesn't have code smells, etc: REFACTOR



END OF CLASS



Jin L.C. Guo

M4 – Unit Testing

February 14

Image Source: https://upload.wikimedia.org/wikipedia/commons/2/2b/Cepaea_nemoralis_active_pair_on_tree_trunk.jpg

Recap of Last Class

- Learned the foundational concepts of testing using the proper terminology;
- Understood type annotations and program reflection and be able to use them effectively;
- Looked at unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection;
- Be able to understand the output of a test coverage tool;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Overview of This Class

- Be able to explain the foundational concepts of testing using the proper terminology;
- Understand type annotations and program reflection and be able to use them effectively;
- Be able to write unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection;
- Be able to understand the output of a test coverage tool;
- Be able to understand basic test suite adequacy criteria and the relations between them;

BrightnessFlashlight code demo

- Proper unit test creation
- Running the unit test
- Coverage of test (seen later)

Encapsulation and Unit Testing

- How can we test private methods? (Two Views)
 - Private methods are not units
 - Private access modifier is a tool to help us structure the project code

Encapsulation and Unit Testing

- How can we test private methods? (Two Views)
 - Private methods are not units
 - Private access modifier is a tool to help us structure the project code
- Metaprogramming (Reflection)
 - Should not be used indiscriminately

Reflection ✨

- Reflection (metaprogramming) lets you examine/modify runtime behavior of your code.
- The following class exists: **java.lang.Class**
- Instances represent classes (and interfaces, etc.), not instances of a class

Reflection ✨

- Ways to instantiate a Class object
 - `Class c = Flashlight.class;`
 - `Class c = f.getClass();`
 - `Class c = Class.forName("Flashlight");`

Reflection ✨

- `Flashlight f = new Flashlight();`
 - `f` is an instance of the `Flashlight` class
- `Class c = f.getClass();`
 - `c` is an instance of the `Class` class, representing the `Flashlight` class (not the object `f`)

Reflection ✨

- Very powerful concept
- Through reflection we can invoke methods at runtime *regardless of the access specifier* used with them
- Tutorial: <https://docs.oracle.com/javase/tutorial/reflect/>

Uses of Reflection

- Extensibility Features

- Use external, user-defined classes by creating instances of extensibility objects using their fully-qualified names

- Class Browsers and Visual Development Environments

- Use available information for visual development environments

- Debuggers and Test Tools

- Examine private members of classes with debuggers and test runners

Drawbacks of Reflection

- Performance Overhead
 - Involves types that are dynamically resolved
- Security Restrictions
 - Requires a runtime permission which may not be present when running under a security manager
- Exposure of Internals
 - Might result in unexpected side-effects
 - Breaks abstractions

Example

- Reflection provides an API for accessing fields, methods, and constructors.
- There are distinct methods for accessing members declared directly on the class
- Let's define a private function in BrightnessFlashlight

Example (Contd.)

- Now we need to update our test suite to test this new function
- Use java.lang's Class and Method classes

Other Use Cases of Reflection

- Obtaining fields and methods types and modifiers
- Obtaining names of method parameters
- Invoking methods
- Finding constructors
- Retrieving and Parsing Constructor Modifiers
- Creating New Class Instances

Testing with Stubs/Mock Object

- The key to unit testing is to test small parts of the program *in isolation*
- Classes with extremely sophisticated behaviour can be difficult to test (Strategy design pattern)
- An object stub or a mock object is a greatly simplified version of an object that mimics its behavior sufficiently to support the testing of a UUT that uses this object

Testing with Stubs/Mock Object

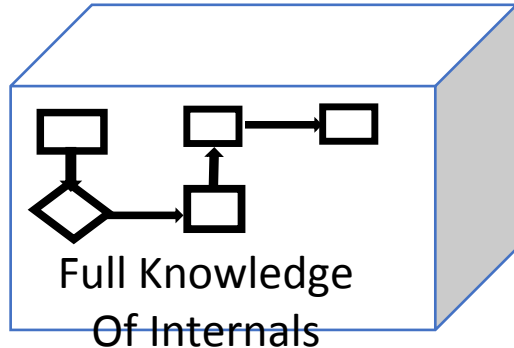
- The use of mock objects in unit testing can get extremely sophisticated, and frameworks exist to support this task (e.g., jMock)

Wrote Code ✓

Wrote our tests ✓

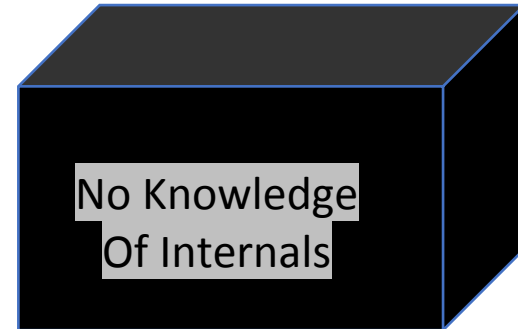
Structural Testing

- White-box Testing
- Checks the *implemented behaviour*
- Based on *how it works*



Functional Testing

- Black-box Testing
- Checks the *specified behaviour*
- Based on *what it should do*



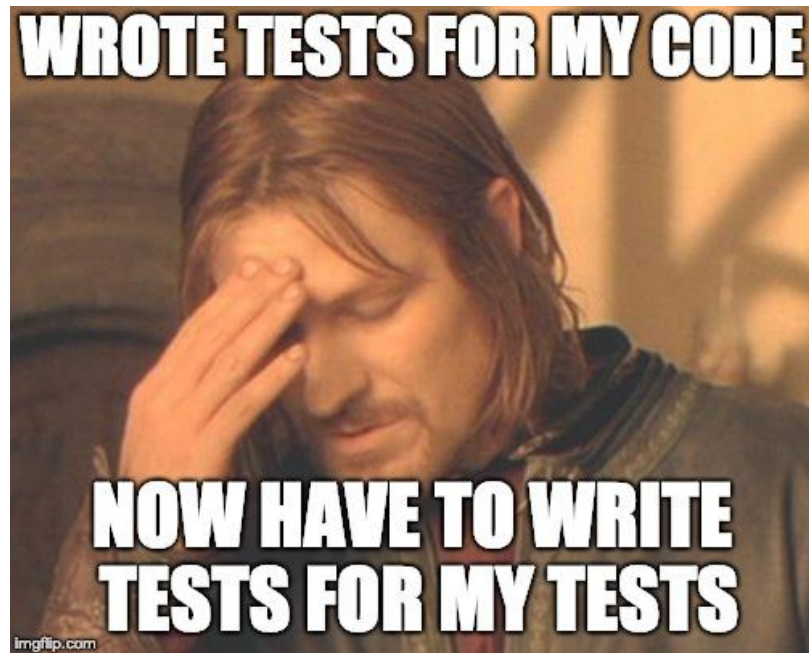
Test Suite Adequacy Criteria

- Measures the *thoroughness* of the Test Suite
- If a test suite covers ALL possible obligations of the software, it is adequate.
- Generally only applicable in white-box testing

Test Adequacy

Are our tests sufficient?

How do we check?



Code Coverage

- Measures the *extent of* thoroughness of a test suite.
- What percentage of components does our test suite execute in the program?

The development and testing team get to decide when a test suite is “sufficient” to test their software.

"If our test suite covers N scenarios in our program, we're good to go!"

"If our test suite satisfies $X\%$ of the rules or obligations, it's doing well!"

100 % Test Suite Adequacy??

But sometimes there are scenarios that CAN'T be checked or it doesn't make sense to check.

Defensive programming checks edge cases that CAN NOT happen.

The toggle in yesterday's flashlight example – there's no end to testing the number of consecutive clicks!



npm v4.41.6

node >=6.11.5 dependencies up to date build passing build passing Azure Pipelines succeeded coverage 95% license scan failing

PRs welcome

semver stability of 4.41.6 97% downloads 41M/month install size 17.5 MB backers 1066 sponsors 416 contributors 414 chat on gitter

webpack

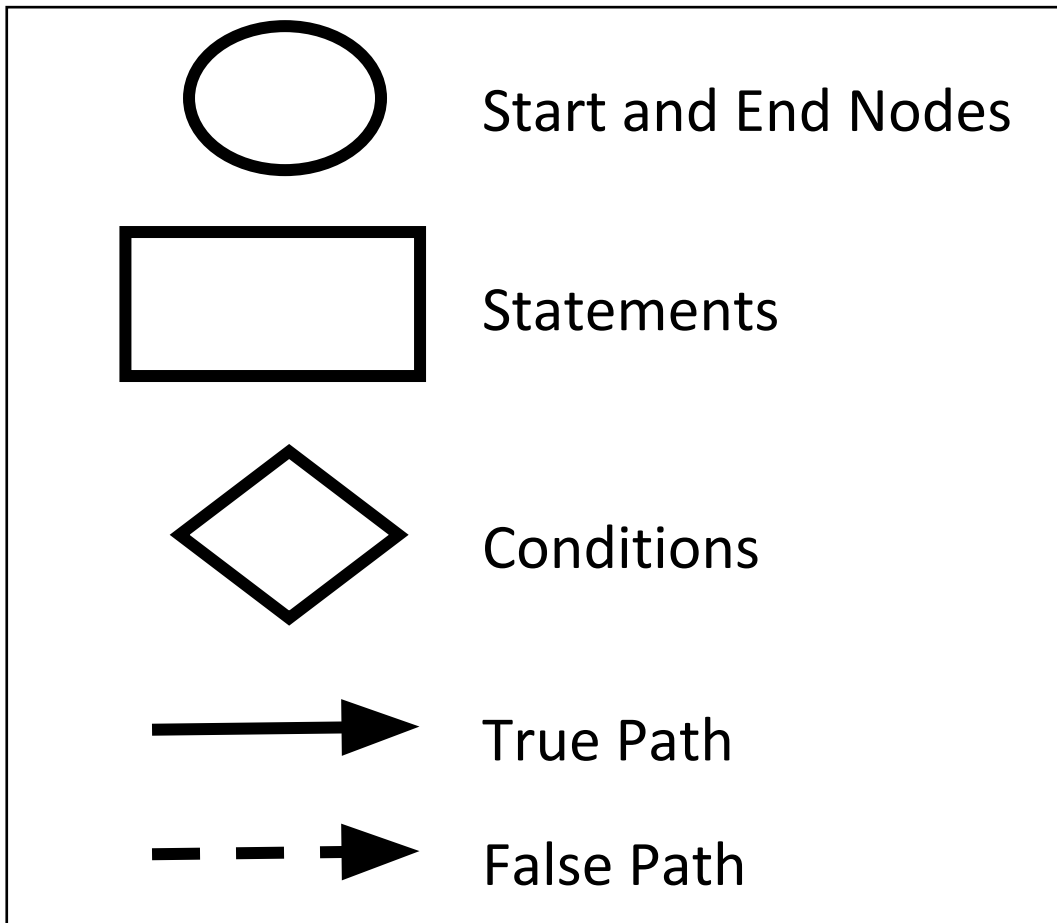
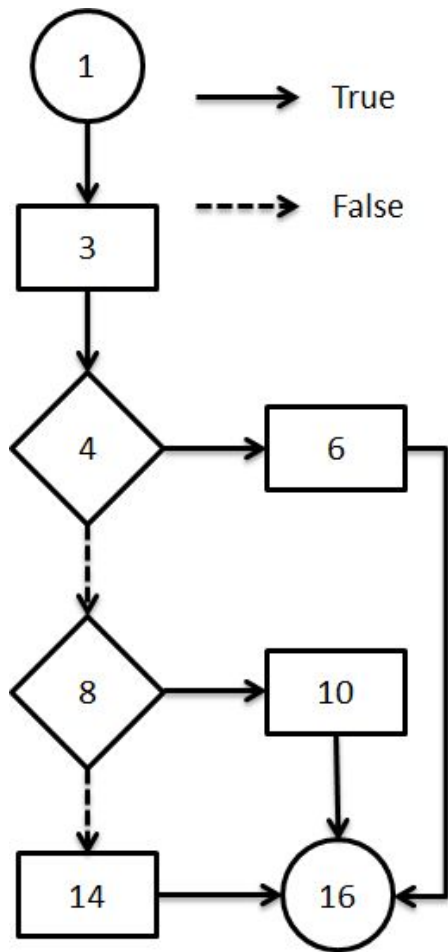
Why is percentage (sometimes) okay?

- Say you're looking to buy a laptop.
- A used laptop is for sale.
- Though you *expect* all parts to be functioning, it turns out the external memory card reader doesn't work.
- But you're getting it at a great price!
- What do you do?
- Requirement vs Cost Trade-Off
 - As long as the *important* parts work, and in general, everything is fine, we'll take it!

Transition

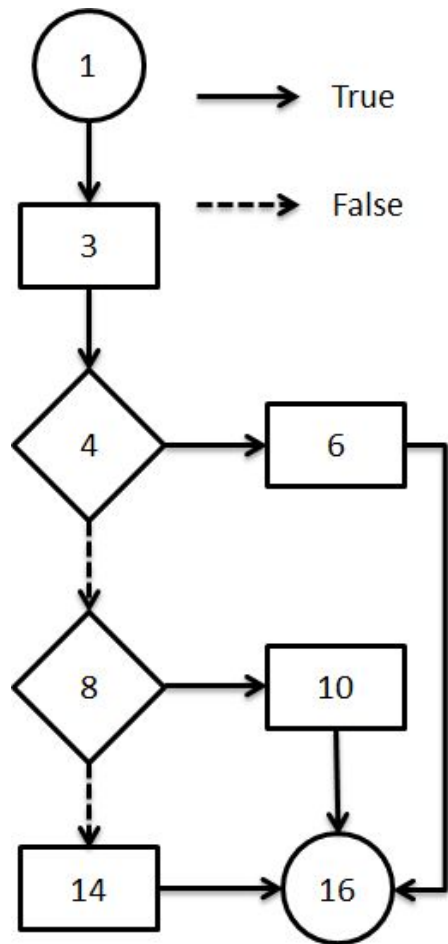


Control Flow Graph (CFG)



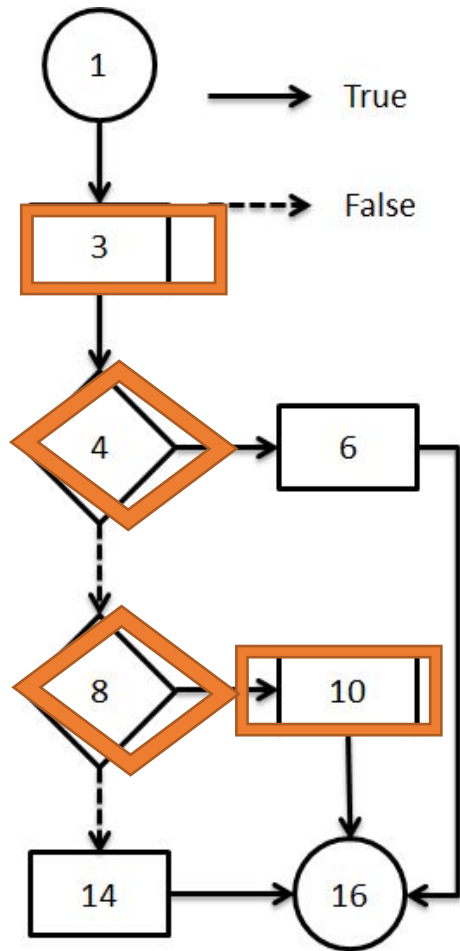
Measuring Coverage

The Control Flow Graph (CFG)



```
1 public static double[] roots(double a, double b, double c)
2 {
3     double q = b*b - 4*a*c;
4     if( q > 0 && a != 0 ) // Two roots
5     {
6         return new double[] { (-b+q)/2*a, (-b-q)/2*a };
7     }
8     else if( q == 0 ) // One root
9     {
10        return new double[] { -b/2*a };
11    }
12    else // No root
13    {
14        return new double[0];
15    }
16 }
```

<- Numbers represent line numbers



Statement Coverage

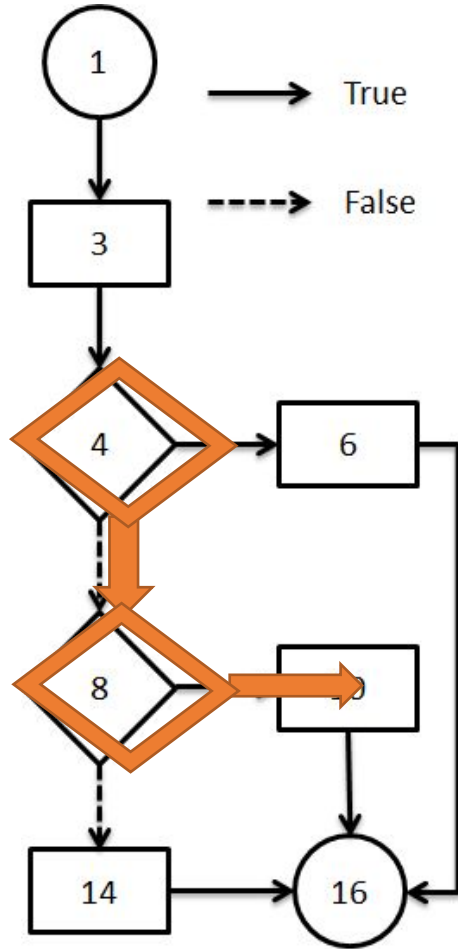
• Input: (1,2,1)

= (Number of statements executed / total number of statements)

= $4/6 = 67\%$

(start and end nodes are ignored)

```
1 public static double[] roots(double a, double b, double c)
2 {
3     double q = b*b - 4*a*c;
4     if( q > 0 && a != 0 ) // Two roots
5     {
6         return new double[] { (-b+q)/2*a, (-b-q)/2*a };
7     }
8     else if( q == 0 ) // One root
9     {
10        return new double[] { -b/2*a };
11    }
12    else // No root
13    {
14        return new double[0];
15    }
16 }
```



Branch Coverage

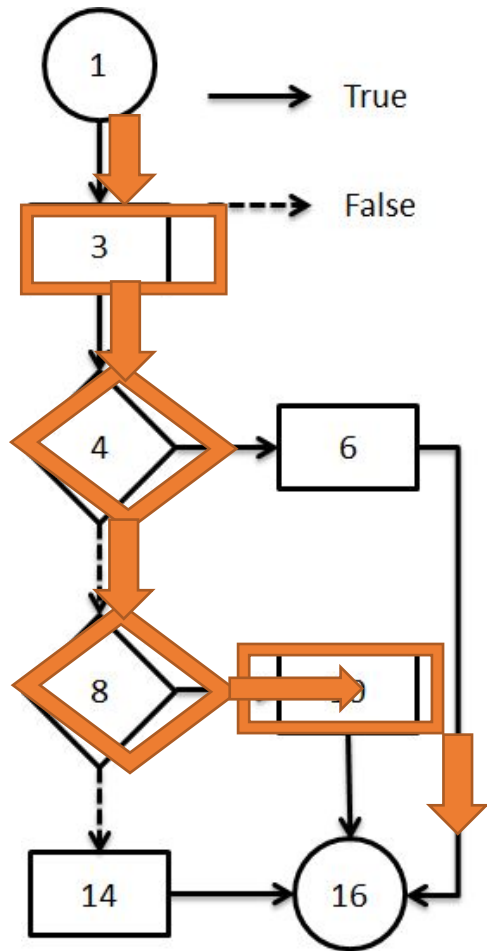
• Input: (1,2,1)

= (Number of branches executed / total number of branches)

= $2/4 = 50\%$

(start and end nodes are ignored)

```
1 public static double[] roots(double a, double b, double c)
2 {
3     double q = b*b - 4*a*c;
4     if( q > 0 && a != 0 ) // Two roots
5     {
6         return new double[] { (-b+q)/2*a, (-b-q)/2*a };
7     }
8     else if( q == 0 ) // One root
9     {
10        return new double[] { -b/2*a };
11    }
12    else // No root
13    {
14        return new double[0];
15    }
16 }
```



Path Coverage

• Input: (1,2,1)

= (Number of paths executed / total number of paths)

= $1/3 = 33\%$

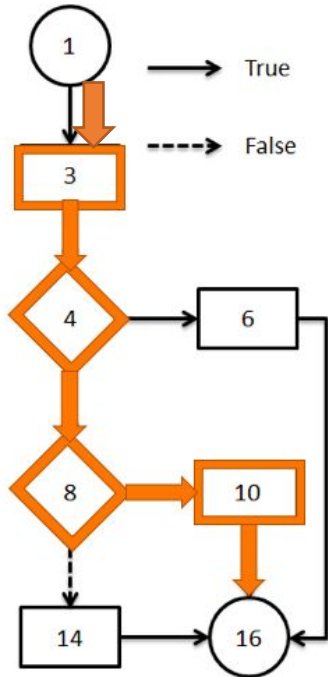
“Theoretical” because how would you calculate it if there were loops?!

```

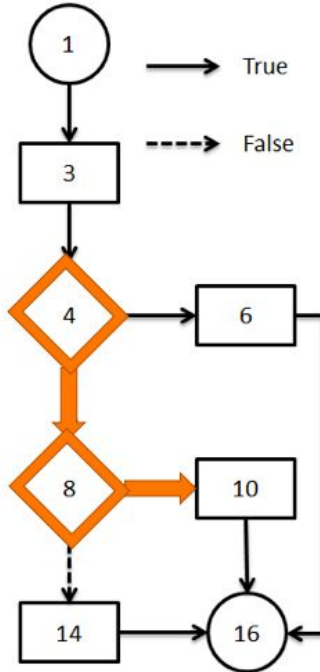
1 public static double[] roots(double a, double b, double c)
2 {
3     double q = b*b - 4*a*c;
4     if( q > 0 && a != 0 ) // Two roots
5     {
6         return new double[] { (-b+q)/2*a, (-b-q)/2*a; }
7     }
8     else if( q == 0 ) // One root
9     {
10        return new double[] { -b/2*a; }
11    }
12    else // No root
13    {
14        return new double[0];
15    }
16 }
  
```

The Subsumes Relationship

Path Coverage



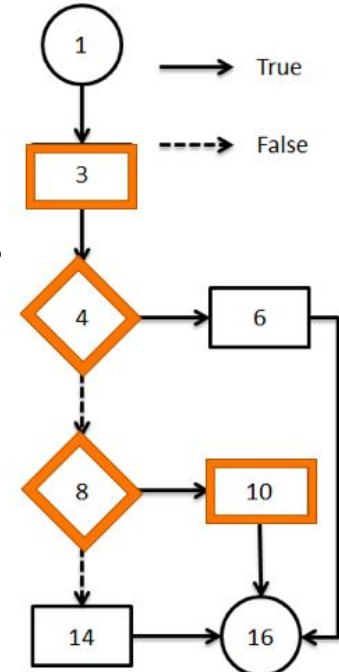
Branch Coverage



Subsumes

Subsumes

Statement Coverage



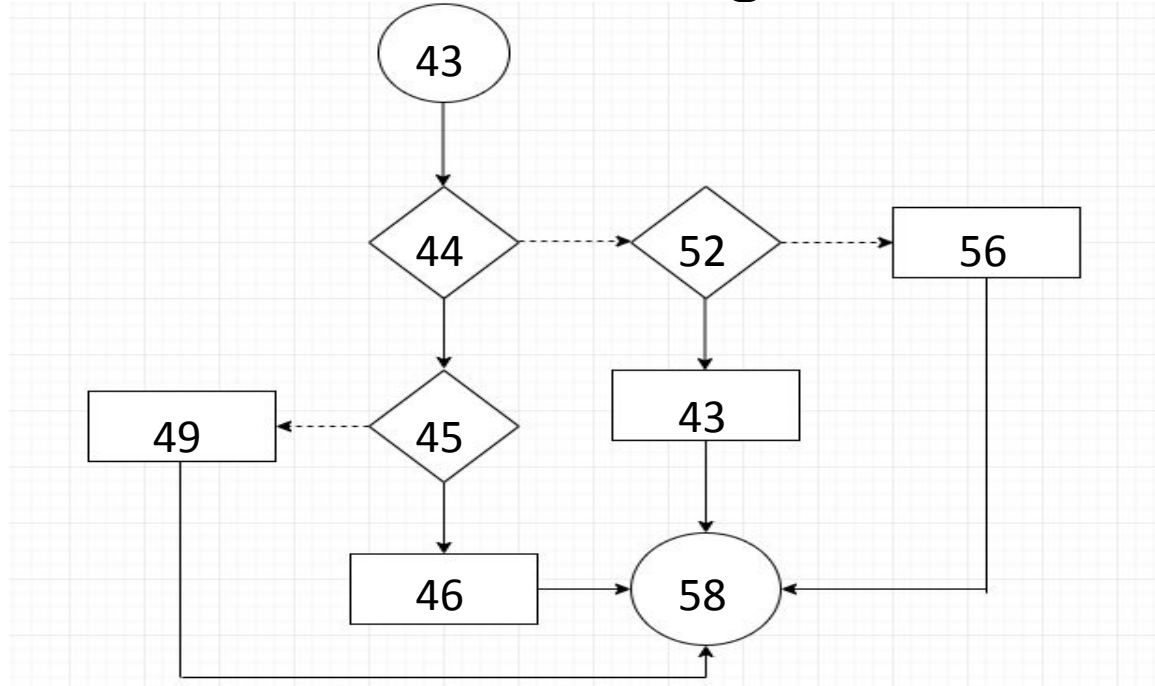
Activity:

1. Draw the CFG
2. What are the minimum set of test cases that will execute all statements? List them down.

```
36 /*  
37  * Say we have a function to recommend brightness level based on different parameters  
38  * The function returns 1 or 5 if the light is not already on but is needed,  
39  *     depending on the darkness.  
40  * It returns 0 if light is not needed but the light is currently on.  
41  * It returns -1 in all other scenarios.  
42  */  
43 public int recommend_brightness(boolean dark, boolean need_light, boolean is_on) {  
44     if (need_light == true && is_on == false) {  
45         if(dark == true) {  
46             return 5;  
47         }  
48         else {  
49             return 1;  
50         }  
51     }  
52     else if (need_light == false && is_on == true){  
53         return 0;  
54     }  
55     else{  
56         return -1;  
57     }  
58 }
```

Activity:

Does your CFG look something like this?



Activity:

What are the minimum set of test cases that will execute all statements?

- (True, True, False) - Statements 44, 45, 46
- (False, True, False) - Statements 44, 45, 49
- (True, False, True) - Statements 44, 52, 53
- (True, True True) - Statements 44, 52, 56

This test suite with 4 test cases has 100% Statement Coverage!

Basic Conditions Coverage

- Every condition can have either a True or False outcome
- Total number of *distinct* condition-outcome pairs =
number_of_conditions * 2

if (need_light == true && is_on == false)

What is the percentage of condition-outcome pairs executed?

Basic Conditions Coverage

- In this case how many condition-outcome pairs exist?
- How many are covered with the input (True, False,

```
36  /*
37  * Say we have a function to recommend brightness level based on different parameters
38  * The function returns 1 or 5 if the light is not already on but is needed,
39  * depending on the darkness.
40  * It returns 0 if light is not needed but the light is currently on.
41  * It returns -1 in all other scenarios.
42  */
43  public int recommend_brightness(boolean dark, boolean need_light, boolean is_on) {
44      if (need_light == true && is_on == false) {
45          if (dark == true) {
46              return 5;
47          }
48          else {
49              return 1;
50          }
51      }
52      else if (need_light == false && is_on == true){
53          return 0;
54      }
55      else{
56          return -1;
57      }
```

Basic Conditions Coverage

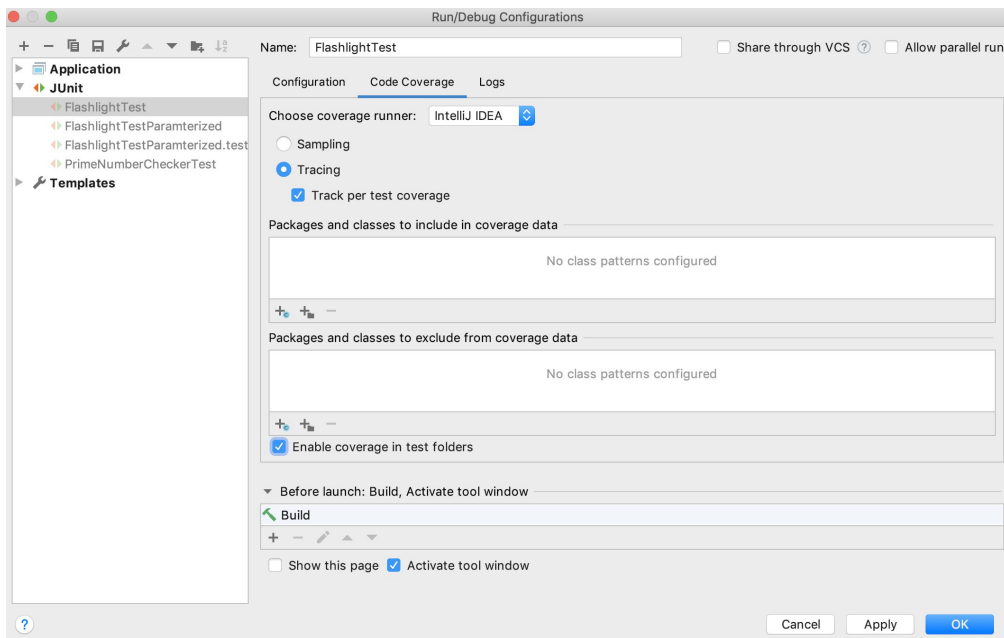
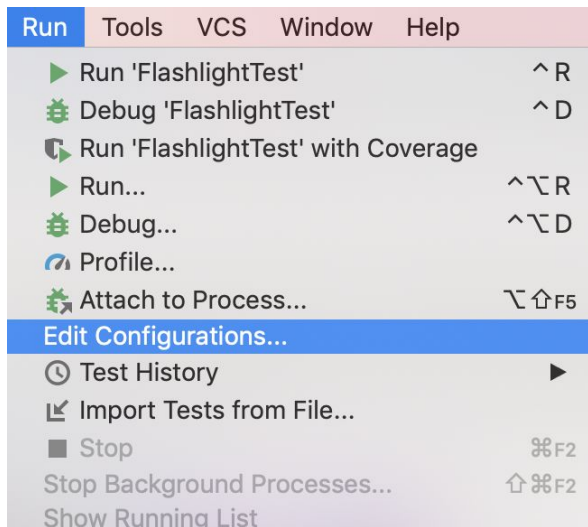
- In this case how many condition-outcome pairs exist?

Ans. 10

- How many are covered with the input (True, False, True)? Ans. 3/10

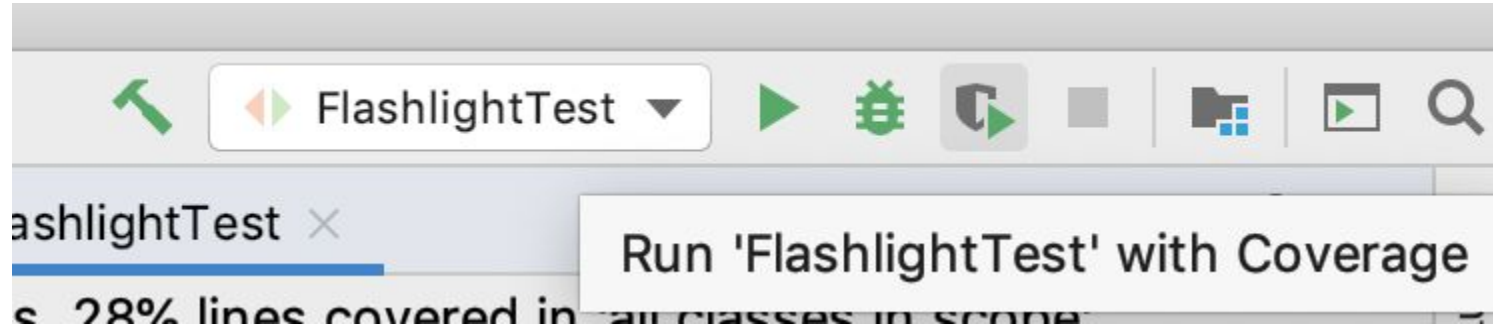
```
36 /*  
37  * Say we have a function to recommend brightness level based on different parameters  
38  * The function returns 1 or 5 if the light is not already on but is needed,  
39  *     depending on the darkness.  
40  * It returns 0 if light is not needed but the light is currently on.  
41  * It returns -1 in all other scenarios.  
42  */  
43 public int recommend_brightness(boolean dark, boolean need_light, boolean is_on) {  
44     if (need_light == true && is_on == false) {  
45         if(dark == true) {  
46             return 5;  
47         }  
48         else {  
49             return 1;  
50         }  
51     }  
52     else if (need_light == false && is_on == true){  
53         return 0;  
54     }  
55     else{  
56         return -1;  
57     }
```

Coverage Tool: Built-in to IntelliJ



Important points: Select 'Tracing' and 'Enable coverage in test folders'

After Configuring, Run your test with coverage



Coverage View

Coverage: FlashlightTest ×

20% classes, 31% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %	Branch, %
META-INF				
netscape				
oracle				
org				
resources				
sun				
toolbarButto...				
Card	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
Deck	0% (0/1)	0% (0/3)	0% (0/3)	100% (0/0)
DeckTest	0% (0/1)	0% (0/1)	0% (0/8)	0% (0/4)
Flashlight	100% (1/1)	75% (3/4)	72% (8/11)	75% (3/4)
FlashlightTest	100% (1/1)	100% (6/6)	83% (15/...	100% (0/0)
FlashlightTes...	0% (0/1)	0% (0/5)	0% (0/15)	0% (0/2)
Main	0% (0/1)	0% (0/1)	0% (0/2)	100% (0/0)
MainTest	0% (0/1)	100% (0/0)	100% (0/0)	100% (0/0)
PrimeNumber...	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
PrimeNumber...	0% (0/1)	0% (0/4)	0% (0/16)	100% (0/0)

Source Code Annotation

- Green: Fully Covered
- Yellow: Partially Covered
- Red: Not Covered



```

1 public class Flashlight {
2     private boolean isOn = false;
3     public void pressButton() { this.isOn = !this.isOn; }
4
5     public boolean isOn() {
6         return this.isOn;
7     }
8
9     private int brightness = 0;
10
11    public void setBrightness(int pBrightness) {
12        if (pBrightness > 5 || pBrightness < 0) {
13            throw new IllegalArgumentException("Incorrect Brightness");
14        }
15        this.brightness = pBrightness;
16    }
17    public int getBrightness() {
18        return this.brightness;
19    }
20 }

```

Types of Coverages in IntelliJ

- *branch coverage* provides information about the amount of branches executed for if/switch statement
- *Line coverage* provides information about the amount of lines that has been executed or missed. A source line is considered executed when at least one instruction that is assigned to this line has been executed.

Counter Mode

- *Method coverage* provides information about the amount of methods that has been executed or missed. A method is considered as executed when at least one instruction has been executed.

WHEN CLASS IS OVER

