Australian National University

COMP2100/6442
Software Design Methodologies / Software Construction

Software Testing

Bernardo Pereira Nunes

Image source: Wikipedia.org

Outline

> Motivation and Intro to Software Testing

> Testing approaches

  >> Black box

  >> White box

> Testing levels

  >> Unit testing

  >> Integration testing, System testing

> Coverage testing

  >> Statement, branch, …, path completion

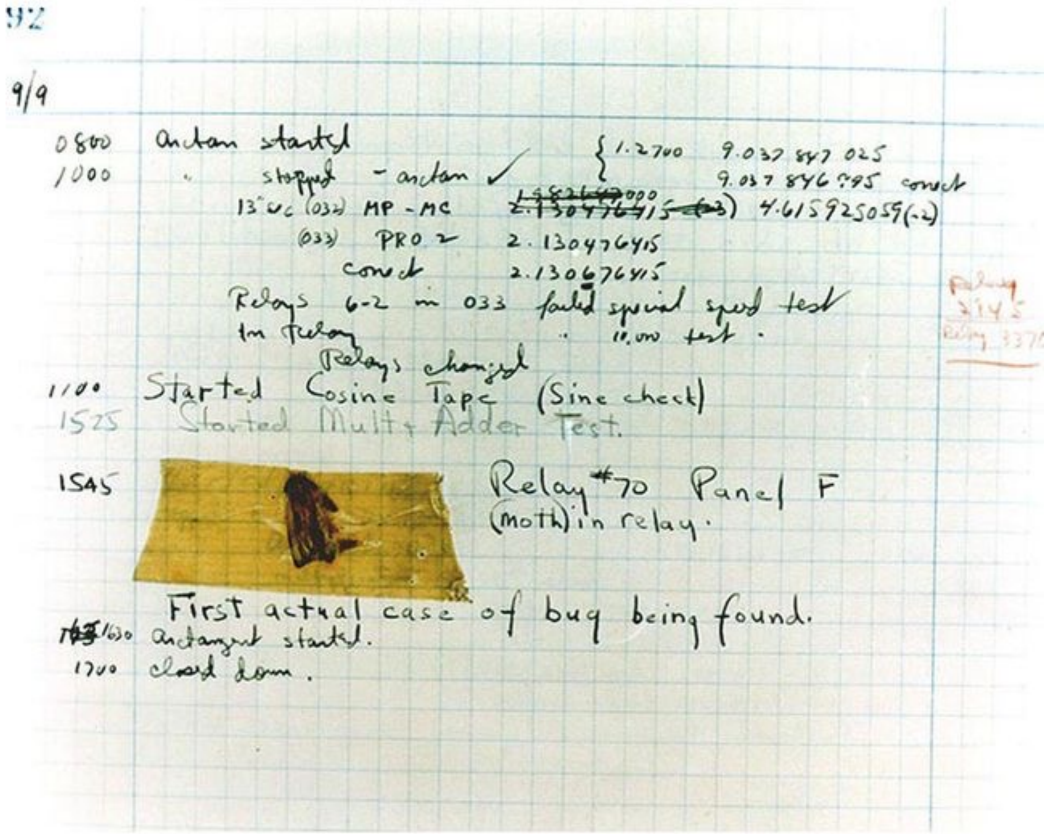
Curious Fact: World’s first computer bug

A team from Harvard University found that one of their computers was showing consistent errors.

They opened the computer hardware and found a moth. The insect disrupted the computer's electronics causing the errors.

As a myth, Grace Hopper is often credited with reporting the first bug.

Photo # NH 96566-KN (Color) First Computer “Bug”, 1947



92

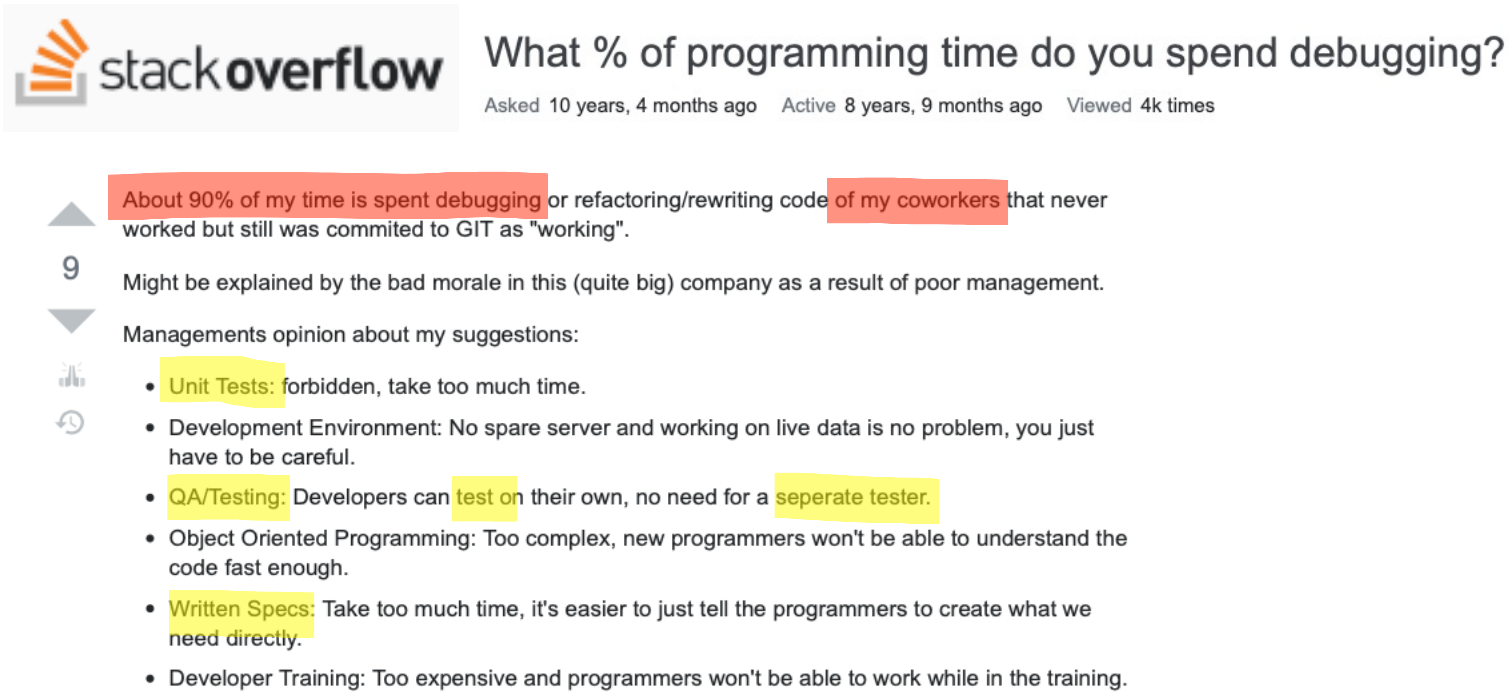
9/9

0800 antan started
1000 " stopped - antan ✓ {1.2700 9.037 847 025
9.037 846 995 conect
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415 (-3)~~ 4.615925059(-2)
(033) PRO 2 2.130476415
conect 2.130676415
Relays 6-2 in 033 failed special speed test
In telay " 11,000 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
~~1545~~ 1630 antangent started.
1700 closed down.

Relay 2145
Relay 3370

Software Testing

Why is Software Testing important?



What % of programming time do you spend debugging?

Asked 10 years, 4 months ago Active 8 years, 9 months ago Viewed 4k times

About 90% of my time is spent debugging or refactoring/rewriting code of my coworkers that never worked but still was commited to GIT as "working".

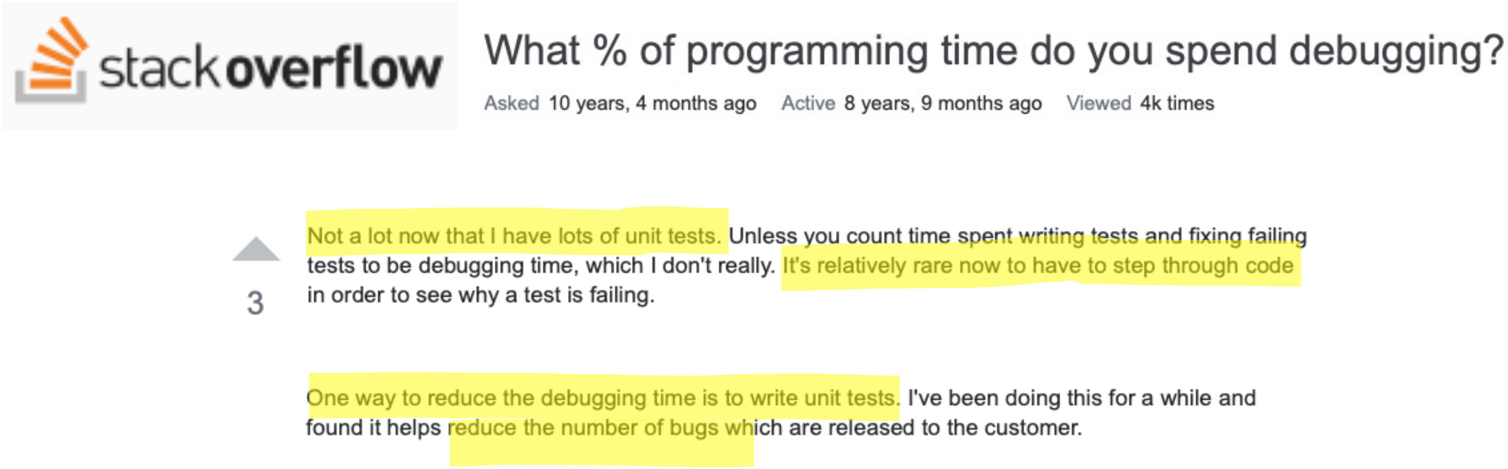
Might be explained by the bad morale in this (quite big) company as a result of poor management.

Managements opinion about my suggestions:

- Unit Tests: forbidden, take too much time.
- Development Environment: No spare server and working on live data is no problem, you just have to be careful.
- QA/Testing: Developers can test on their own, no need for a seperate tester.
- Object Oriented Programming: Too complex, new programmers won't be able to understand the code fast enough.
- Written Specs: Take too much time, it's easier to just tell the programmers to create what we need directly.
- Developer Training: Too expensive and programmers won't be able to work while in the training.

Software Testing

Why is Software Testing important?



stackoverflow

What % of programming time do you spend debugging?

Asked 10 years, 4 months ago Active 8 years, 9 months ago Viewed 4k times

3

Not a lot now that I have lots of unit tests. Unless you count time spent writing tests and fixing failing tests to be debugging time, which I don't really. It's relatively rare now to have to step through code in order to see why a test is failing.

One way to reduce the debugging time is to write unit tests. I've been doing this for a while and found it helps reduce the number of bugs which are released to the customer.

Every human being makes mistakes…

…not testing is one of them!

Software Testing

Software Testing encompasses a wide range of activities:

- To **verify** that the **software is behaving as expected**
- Range from compiling your code to system testing that runs software under realistic loads

Testing will *not prove* code to be correct!

> "*Program testing can be used to show the presence of bugs, but never to show their absence*" [1]
>
> E. W. Dijkstra

It does provide confidence and it will often uncover problems within the code

[1] E. W. Dijkstra, "Notes on Structured Programming," Structured Programming, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Editors, Academic Press, London (1972), pp. 1-82.

Benefits of Testing

> Testing checks the code is **behaving as expected**

> It allows to **find and isolate bugs/defects early** in the development cycle

> To **demonstrate** that the software developed **meets its requirements**

> It increases the confidence of both the developer and the customer in the software

> Help **reflect** on the correctness of the **design and implementation**

>> Useful in *marking programming tasks*

Software Testing

What makes developers/companies not test?

"Belief in the malleability of software - that whatever is produced can be repaired or enhanced later" [2]

"Ever-changing requirements (which, in many cases, are never written down)" [2]

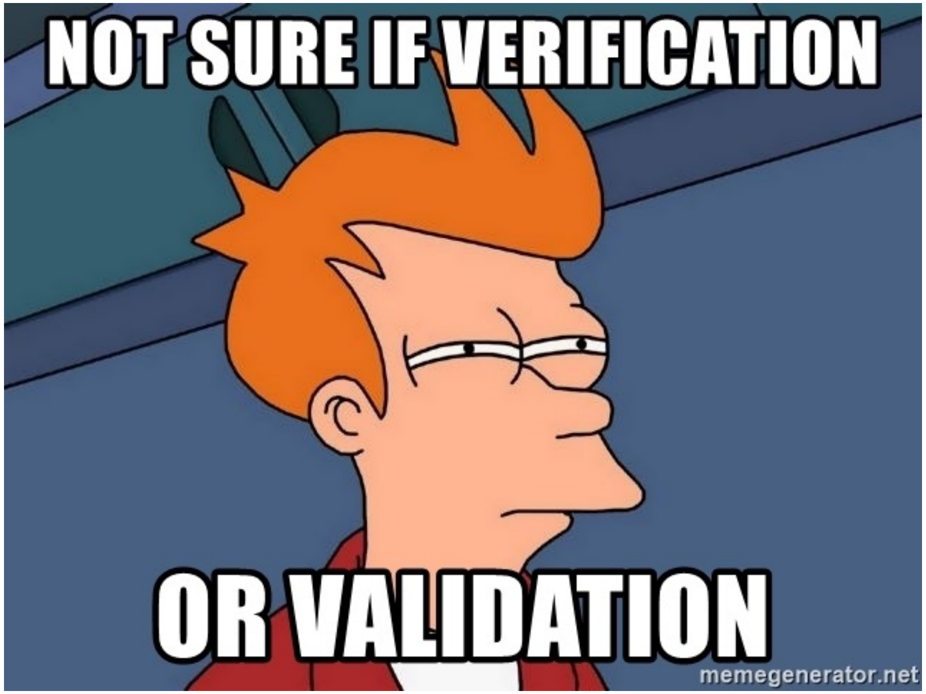
[2] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification", IBM Systems Journal, 41(1):4--12, 2002.

Validation (vs Verification)

- Building the right thing
    - Certify that the **system meets customers needs**
- Check whether the **developer is building the correct product**
- Testing can focus on validation, so the test cases go all the way back to requirement specifications

Verification

- Building the thing right
  - Verification checks whether **each function within the implementation is working correctly**
  - Whether each function complies with the specification
- It checks the system against the design
- Verification certifies the quality of the system
- The focus of testing in this course is on (automated) verification



- Building the right thing
  (validation)

- Building the thing right
  (verification)

Tip to never forget:

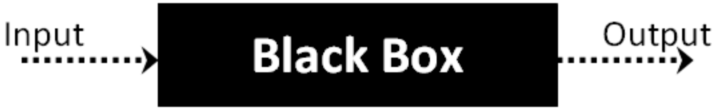
List "Validation" and "Verification" in the alphabetical order
Unsurprisingly: Validation comes before Verification

"Right" comes before "thing" in validation and after "thing" in verification

Software Testing

> Testing approaches
    > Black box
    > White box
> Testing levels
    > Unit testing
    > Integration testing
    > System testing
> Coverage of testing

Black-Box Testing



> Tests can be created based purely on the functional requirements of the code

> There are three types of possible cases:

> Normal function

> Boundary cases

> Testing cases outside requirements & robustness

Image source: guru99.com

Black-Box Testing

Example:

> UBER Eats offers you “Free Delivery” if you spend $40 or more.

> Requirements (consider only the **red** items for simplification)

> “Free Delivery” if you spend $40 or more in products

> Unlimited until 30 Jun 2020

> Promotion cannot be used in conjunction with any other Uber Eats promo

> Valid only in Australia

> This offer is valid only to specific users

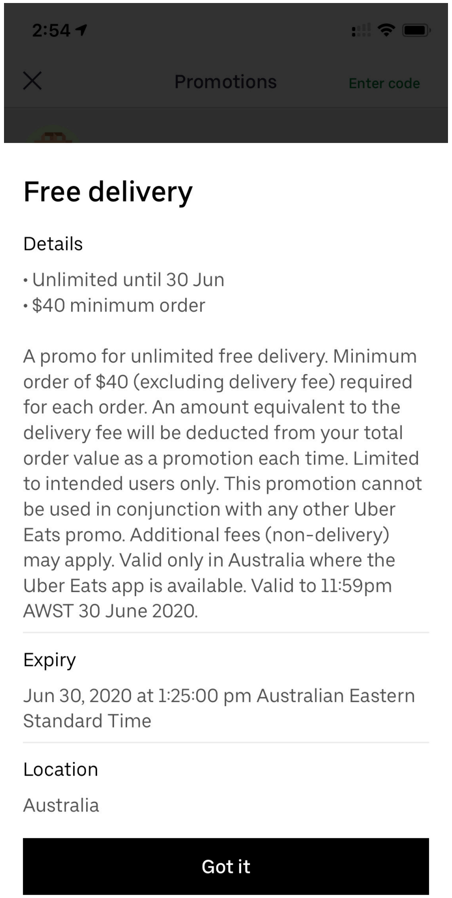
> Expected Outcomes (based on the red requirements only)

(1) Total order value + delivery fee - delivery fee, if (total_order >= 40 && (validUser==true))

(2) Total order value + delivery fee, if (total_order < 40 || (validUser==false))

> Input ?

Real Uber Eats Offer for Bernardo



Free delivery

Details

- Unlimited until 30 Jun
- $40 minimum order

A promo for unlimited free delivery. Minimum order of $40 (excluding delivery fee) required for each order. An amount equivalent to the delivery fee will be deducted from your total order value as a promotion each time. Limited to intended users only. This promotion cannot be used in conjunction with any other Uber Eats promo. Additional fees (non-delivery) may apply. Valid only in Australia where the Uber Eats app is available. Valid to 11:59pm AWST 30 June 2020.

Expiry

Jun 30, 2020 at 1:25:00 pm Australian Eastern Standard Time

Location

Australia

Black-Box Testing

Expected Outcomes (based on the previous red requirements only)

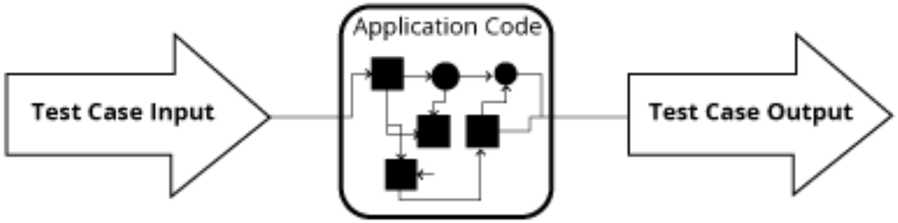
> (1) Total order value + delivery fee - delivery fee, if (total_order >= 40 && (validUser==true))

> (2) Total order value + delivery fee, if (total_order < 40 | | (validUser==false))

Input			Output	
Total Order	User	validUser?	Program's Output	Expected
1.00	Bernardo	True	(2)	(2)
1.00	Henry	False	(2)	(2)
39.99	Bernardo	True	(2)	(2)
39.99	Henry	False	(2)	(2)
40.00	Bernardo	True	(1)	(1)
40.00	Henry	False	(1)	(2)
40.01	Bernardo	True	(1)	(1)
40.01	Henry	False	(2)	(2)
…				

!= (error found)

White-Box Testing



> A set of **tests** can be constructed **based on the code**

> This enables the tester to **target the test cases**

> White-box testing would **normally** involve generating test cases that have good "**code coverage**"

> Test cases can also be constructed based on **boundary cases based on the code**

Image source: invensis.net

Unit Testing

JUnit

> Unit: The smallest testable part of any software

  > often method in OOP

  > Has some inputs and single output

> Unit testing: Looking for errors in a subsystem in isolation

  > The Java library JUnit helps us to perform automated unit testing

  > Often performed by using white-box method

JUnit 4

> Automated unit testing framework developed for Java

> Requires Java 8 (or higher)

> What about JUnit 5? I know you will ask this! ☺

  > (JUnit 5 IS **NOT USED** IN THIS COURSE)

  > Why not? Simplification, more dependency files

> Getting started: https://github.com/junit-team/junit4/wiki/Getting-started

Basic Framework

> For a given class **Foo**, create a test class **FooTest**, containing various “test case” methods to run

> Each method looks for particular results and passes/fails

> JUnit provides “**assert**” commands to help us write tests

>> The idea: Put assertion calls in your test methods to check things you expect to be true

A JUnit test class

```
import org.junit.Test;

public class SomethingTest {
    ...
    // a test case method
    @Test
    public void testSomething() {
        ...
    }
}
```

- A method with `@Test` is flagged as a JUnit test case
    - All `@Test` methods run when JUnit runs your test class

JUnit assertion methods

> Each method can also be passed a string to display if it fails:

- e.g. assertEquals(java.lang.String **message**, **expected**, **actual**);

assertTrue(**test**)	fails if the boolean test is false
assertFalse(**test**)	fails if the boolean test is true
assertEquals(**expected**, **actual**)	fails if the values are not equal
assertSame(**expected**, **actual**)	fails if the values are not the same (by ==)
assertNotSame(**expected**, **actual**)	fails if the values *are* the same (by ==)
assertNull(**value**)	fails if the given value is *not* null
assertNotNull(**value**)	fails if the given value is null
fail()	causes current test to immediately fail

JUnit assertion methods

`assertEquals(java.lang.Object expected, java.lang.Object actual)`
>> asserts that two objects are equal

!

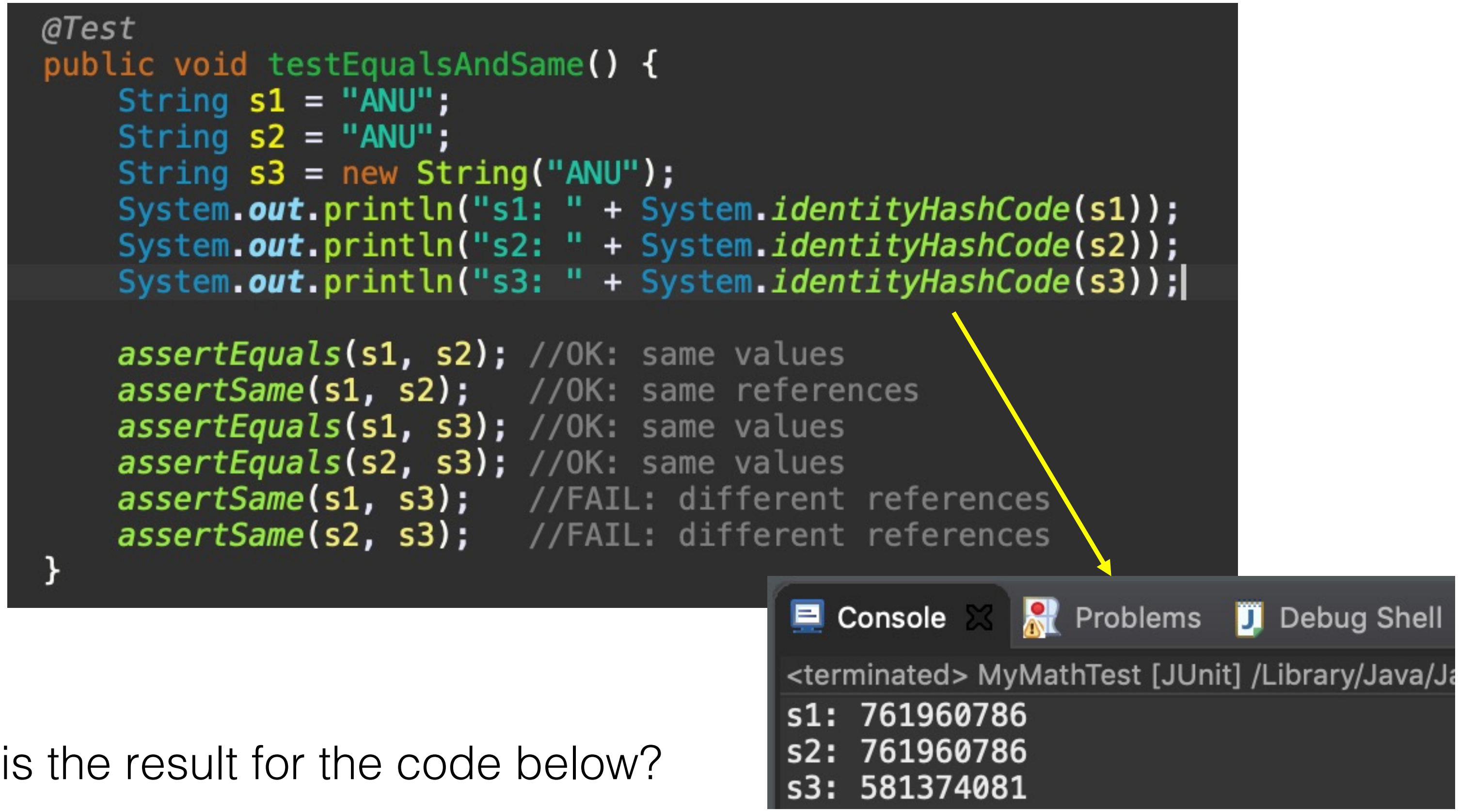
`assertSame(java.lang.Object expected, java.lang.Object actual)`
>> asserts that two objects refer to the same object

Note that **assertEquals** uses the equals() method to assert that two objects are equal, (if the equals() method is not overridden, it asserts that two objects refer to the same object)

```
@Test
public void testEqualsAndSame() {
    String s1 = “ANU”;
    String s2 = “ANU”;
    String s3 = new String(“ANU”);

    assertEquals(s1, s2); //OK: same values
    assertSame(s1, s2);   //OK: same references
    assertEquals(s1, s3); //OK: same values
    assertEquals(s2, s3); //OK: same values
    assertSame(s1, s3);   //FAIL: different references
    assertSame(s2, s3);   //FAIL: different references
}
```

*all test cases in the same method for teaching purposes only (not recommended)

JUnit assertion methods



```
@Test
public void testEqualsAndSame() {
    String s1 = "ANU";
    String s2 = "ANU";
    String s3 = new String("ANU");
    System.out.println("s1: " + System.identityHashCode(s1));
    System.out.println("s2: " + System.identityHashCode(s2));
    System.out.println("s3: " + System.identityHashCode(s3));

    assertEquals(s1, s2); //OK: same values
    assertSame(s1, s2);   //OK: same references
    assertEquals(s1, s3); //OK: same values
    assertEquals(s2, s3); //OK: same values
    assertSame(s1, s3);   //FAIL: different references
    assertSame(s2, s3);   //FAIL: different references
}
```

```
<terminated> MyMathTest [JUnit] /Library/Java/Ja
s1: 761960786
s2: 761960786
s3: 581374081
```

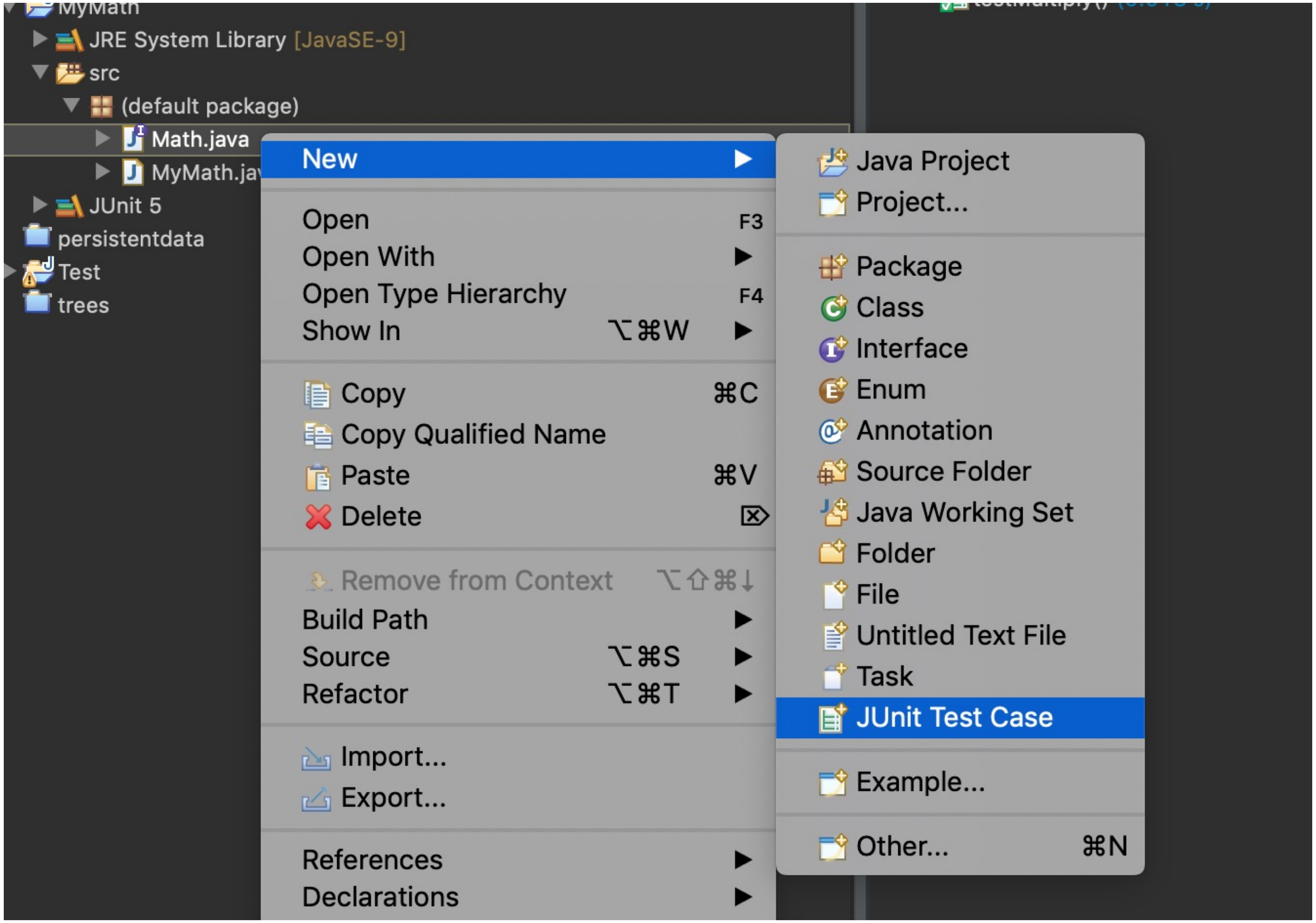
What is the result for the code below?

```
assertTrue(s1.equals(s3));
```

Starting JUnit with Eclipse

- JUnit does not come with JDK
  - Need to include JUnit into classpath
- To add JUnit to an Eclipse project, click on:
  - Project → Properties → Build Path → Libraries → Add Library → JUnit → JUnit 4 → Finish
- To create a test case:
  - right-click a file and choose New → Test Case
  - or click on File → New → JUnit Test Case
  - Eclipse can create stubs of method tests for you
- You can download JUnit 4 from our repo

Create Test Case



Floating point assertion

> What is 2/3?

0.66666666666666666…

(It cannot be represented in floating point expression.

We need to allow a small error)

```
assertEquals(x, y, delta)
```

Float or Double

Asserts that two doubles or floats are equal to within a positive delta.

```
//Example
assertEquals(0.3333333, 1.0/3.0, 0.0) -> fail
assertEquals(0.3333333, 1.0/3.0, 0.0000001) -> pass
```

Demo with simple Math class

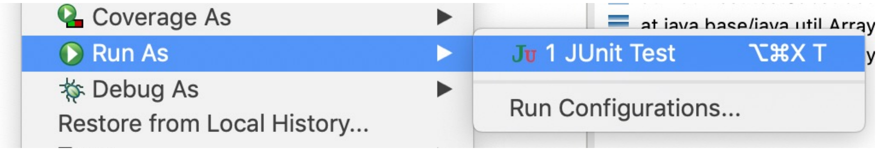
```
public class MyMath {
     int add(int a, int b) {
          return (a+b);
     }

}
```

Demo with simple Math class

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

class MyMathTest {
    private MyMath math;

    @Test
    public void testAdd() {
        math = new MyMath();
        assertEquals(4, math.add(2, 2));
    }

}
```

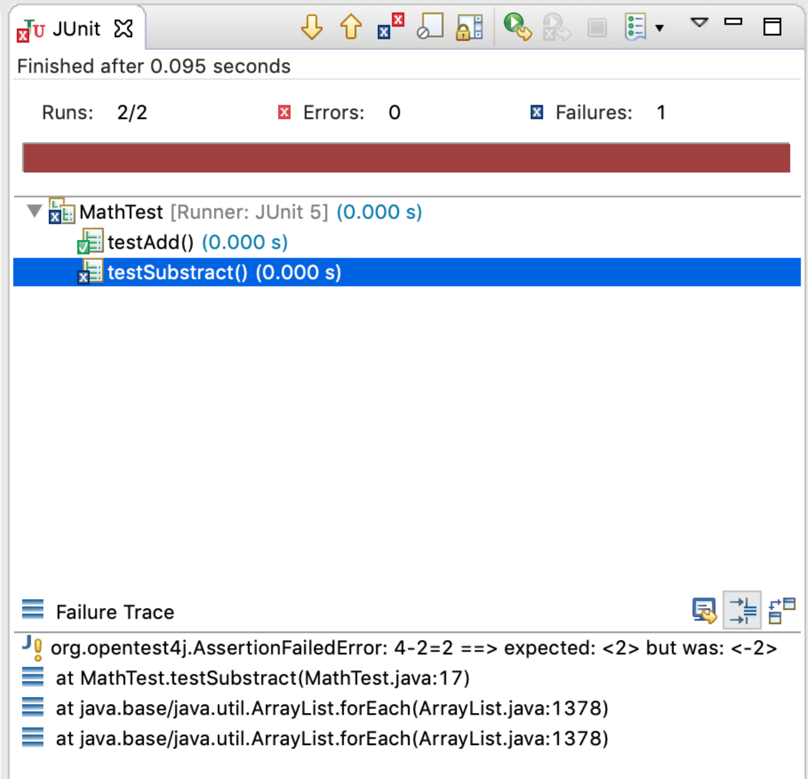
Running a Test

Right click on the Eclipse Package Explorer (usually on the left side), and choose: Run As → JUnit Test



The **JUnit bar** will show a tick if all tests pass or a cross if any fail

The **Failure Trace** shows which tests failed, if any, and why



Finished after 0.095 seconds

Runs: 2/2 Errors: 0 Failures: 1

MathTest [Runner: JUnit 5] (0.000 s)
testAdd() (0.000 s)
testSubstract() (0.000 s)

Failure Trace

```
org.opentest4j.AssertionFailedError: 4-2=2 ==> expected: <2> but was: <-2>
at MathTest.testSubstract(MathTest.java:17)
at java.base/java.util.ArrayList.forEach(ArrayList.java:1378)
at java.base/java.util.ArrayList.forEach(ArrayList.java:1378)
```

Resource Reuse/Release

`@Before / @After`

> Denotes that the annotated method should be executed before/after each @Test

i.e., methods run before/after each test case method is called

`@Before usage:`

e.g. initialisation code: class initialisation, array initialisation, …

`@After usage:`

e.g. cleanup code: freeing resources (set objects to null or call garbage collector), …

Dummy Example: @Before and @After

```
class MyMathTest {
        private MyMath math;

        @Before
        public void setUp() {
            System.out.println(“>Before.setUp”);
            math = new MyMath();
        }

        @After
        public void tearDown() {
            System.out.println(“>After.tearDown”);
            math = null;
        }

        @Test
        public void testAdd() {
            System.out.println(“>>Test.testAdd”);
            assertEquals(4, math.add(2, 2));
        }

        @Test
        public void testDivide() {
            System.out.println(“>>Test.testDivide”);
            assertEquals(4, math.divide(4, 1));
        }
}
```

Console - Output

```
>Before.setUp
>>Test.testAdd
>After.tearDown
>Before.setUp
>>Test.testDivide
>After.tearDown
```

Resource Reuse/Release

`@BeforeClass` | **`@AfterClass`**

> Denotes that the annotated method should be executed before/after all **`@Test`** in a class

i.e. methods to run once before/after the entire test class runs

> Must be static method

> example: database connection

`@BeforeClass` `usage:`

e.g. initialisation code, establish a connection (database), …

`@AfterClass` `usage:`

e.g. cleanup code: freeing resources, close a connection…

Dummy Example: @BeforeClass and @AfterClass

```
class MyMathTest {
        private MyMath math;

        @BeforeClass
        public static void runBeforeClass() {
             System.out.println(“RunBeforeClass”);
             //e.g. open database connection
        }

        @AfterClass
        public static void runAfterClass() {
             System.out.println(“RunAfterClass”);
             //e.g. close database connection
        }

        @Before
        public void setUp() {
             System.out.println(“>Before.setUp”);
             math = new MyMath();
        }

        @After
        public void tearDown() {
             System.out.println(“>After.tearDown”);
             math = null;
        } ...
}
```

Console - Output

```
RunBeforeClass
>Before.setUp
>>Test.testAdd
>After.tearDown
>Before.setUp
>>Test.testDivide
>After.tearDown
RunAfterClass
```

Testing for Exceptions

```
@Test(expected = ArithmeticException.class)
public void exceptionTesting() {
    //assertEquals(0, math.divide(1, 0), 1e-10);
    math.divide(1, 0);
}
```

This test will pass if it throws the given exception

> If the exception is not thrown, the test fails

> Use this to test for expected errors

Tests with Timeout

```
@Test(timeout = 1000)
public void timeoutNotExceeded() {
  assertEquals(5, math.add(2, 3));
}
```

> If it does not finish within 1 second, the test will fail

> Can be used to:

>> Guarantee performance

>> Prevent infinite loop

Parameterized Test

> If you need to use the same inputs across different test methods (e.g., add/sub/div)

>> Define test class with: `@RunWith(Parameterized.class)`

>> Define `static` method returns list of parameter arrays as `@Parameters`

>> Each parameter can be defined as a class member with `@Parameter(#)`

Parameterized Test

```
@RunWith(Parameterized.class)
public class MyMathParameterizedTest {
    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {{0.7f, 0.3f, 1}, {0.4f, 0.2f, 0}});
    }
    //first entry of each array
    @Parameter(0)
    public float a;
    //second entry of each array
    @Parameter(1)
    public float b;
    //third entry of each array
    @Parameter(2)
    public int expected;
    @Test
    public void test() {
         MyMath math = new MyMath();
         assertEquals(expected, math.sumAndFloor(a, b));
}}
```

Test Suite

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses(
  { TestClass1.class,
    TestClass2.class,
    … })
public class FeatureTestSuite {
// the class remains empty
}
```

Test suite: One class that runs multiple JUnit test classes

Tips for Testing

> You **cannot test every possible input**, parameter value, etc.

>> So you must think of a limited set of **tests likely to find bugs**

> For example, think about the **boundary cases**:

>> positive; zero; negative numbers

>> Very large values

> Think about **empty cases** and **error cases**

>> 0, -1, null; an empty list or array

> **Test behaviour in combination**

> maybe add usually works, but fails after you call subtract

> make multiple calls; maybe size fails the second time only

Unit Test Guidelines

> **Test one thing at a time per test method**

>> 10 small tests are much better than 1 test 10x as large

> Each test method should have **few (likely 1) assert statements**

>> If you assert many things, the **first that fails stops the test**

>> You won't know whether a later assertion would have failed

> **Tests should avoid logic**

>> minimise if/else, loops, switch, etc.

>> avoid try/catch

> **Torture** (stress) tests are okay, but only **in addition to simple tests**

Test-driven Software Development

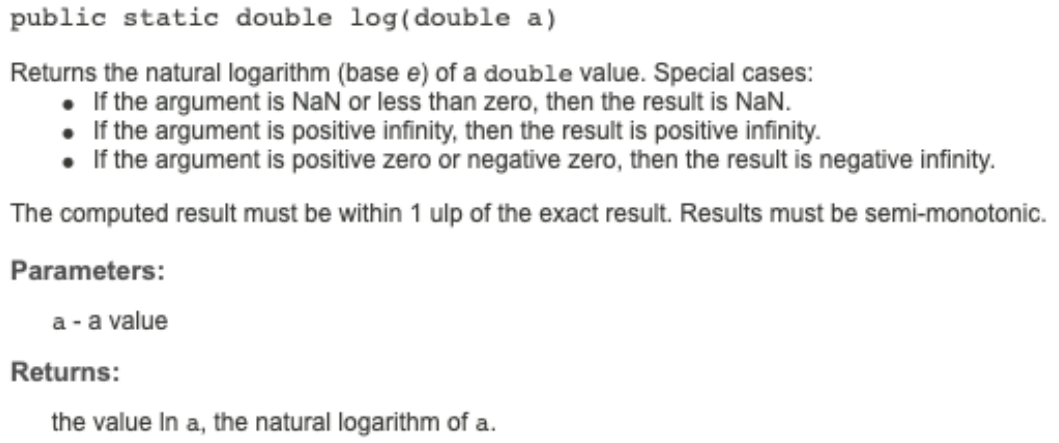
> Unit tests can be written after, during, or even before coding

> Test-Driven Development:

write tests, then write code to pass them

> Imagine that we'd like to add a "log" method to our MyMath class:

>> Write code to test this method before it has been written. Once we do implement the method, we'll know if it works.



```
public static double log(double a)
```

Returns the natural logarithm (base *e*) of a `double` value. Special cases:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is positive zero or negative zero, then the result is negative infinity.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:

`a` - a value

Returns:

the value ln `a`, the natural logarithm of `a`.

Source: https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html#log(double)

Test-driven Software Development

```
//Case a < 0 or a is NaN, returns Double.NaN
assertEquals(Double.NaN, math.log(-1), 0.000001);
assertEquals(Double.NaN, math.log(Double.NaN), 0.000001);

//Case a is +infinity, returns +infinity
assertEquals(Double.POSITIVE_INFINITY,
math.log(Double.POSITIVE_INFINITY), 0.000001);

//Case a = 0.0 or a = -0.0, returns –infinity
assertEquals(Double.NEGATIVE_INFINITY, math.log(0.0), 0.000001);
assertEquals(Double.NEGATIVE_INFINITY, math.log(-0.0), 0.000001);

//Otherwise returns the natural logarithm (base e) of a
assertEquals(1, math.log(Math.E), 0.000001);
```

JUnit Summary

> Tests need failure ***atomicity*** (ability to know exactly what failed)

>> Each test should have a clear, long, descriptive name

>> Assertions should always have clear messages to know what failed

>> Write many small tests, not one big test

> Test for **expected errors / exceptions**

> Add ***timeout*** **to every test** (e.g., @Test(timeout=1000))

> Choose a **descriptive assert method**, not always assertTrue

> Choose representative test cases from equivalent input classes

> Avoid complex logic in test methods if possible

Reference: https://github.com/junit-team/junit4/wiki/

Lack of Integration Testing

2 UNIT TESTS, 0 INTEGRATION TESTS
via reddit.com/r/programmerhumor

Image source: reddit/programmerhumor

Integration Testing

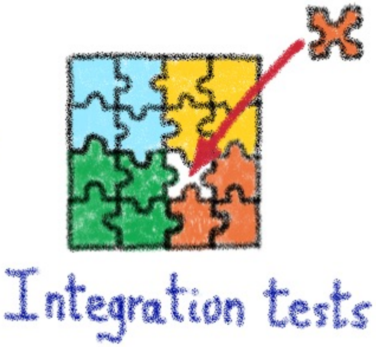
> Integration Testing can be performed **after unit testing**

> Individual units are combined and **tested as a group**

> Both black-box and white-box testing can be used

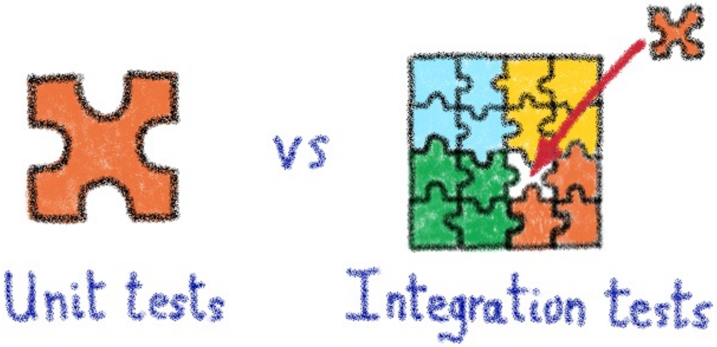
> Interactions between each unit **are clearly defined**

> Unit testing framework can be used

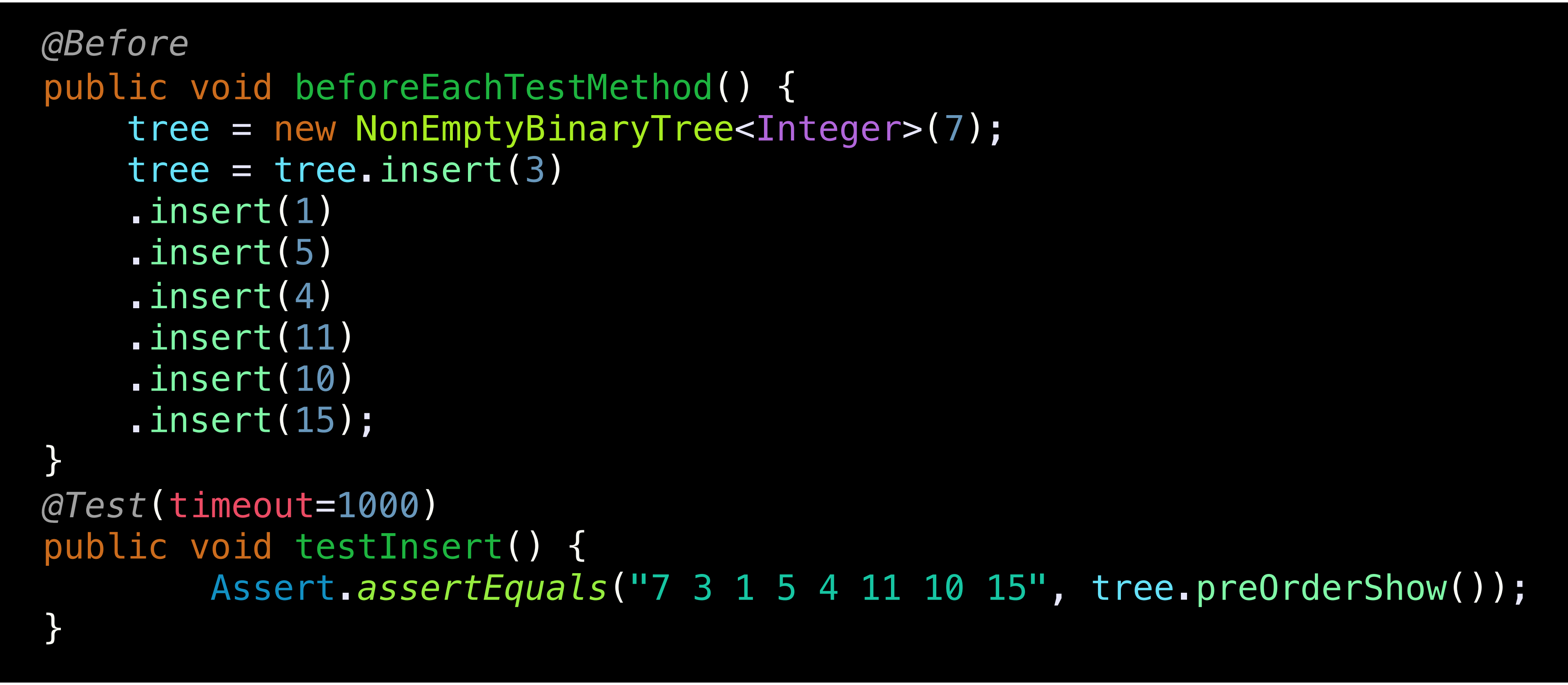


Integration Testing

Why should we not perform unit and integration tests together?

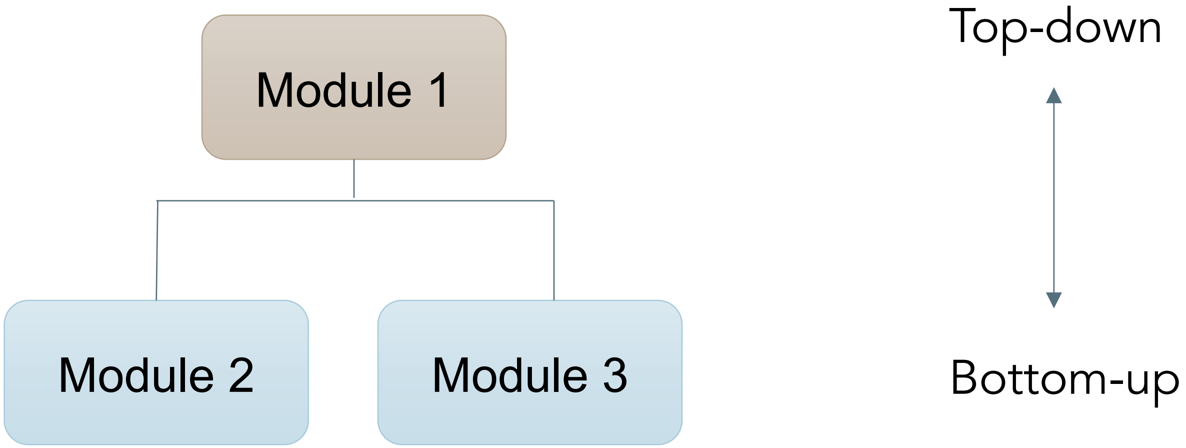


Example of Unit Testing



```java
@Before
public void beforeEachTestMethod() {
    tree = new NonEmptyBinaryTree<Integer>(7);
    tree = tree.insert(3)
    .insert(1)
    .insert(5)
    .insert(4)
    .insert(11)
    .insert(10)
    .insert(15);
}
@Test(timeout=1000)
public void testInsert() {
        Assert.assertEquals("7 3 1 5 4 11 10 15", tree.preOrderShow());
}
```

Example of Integration Testing



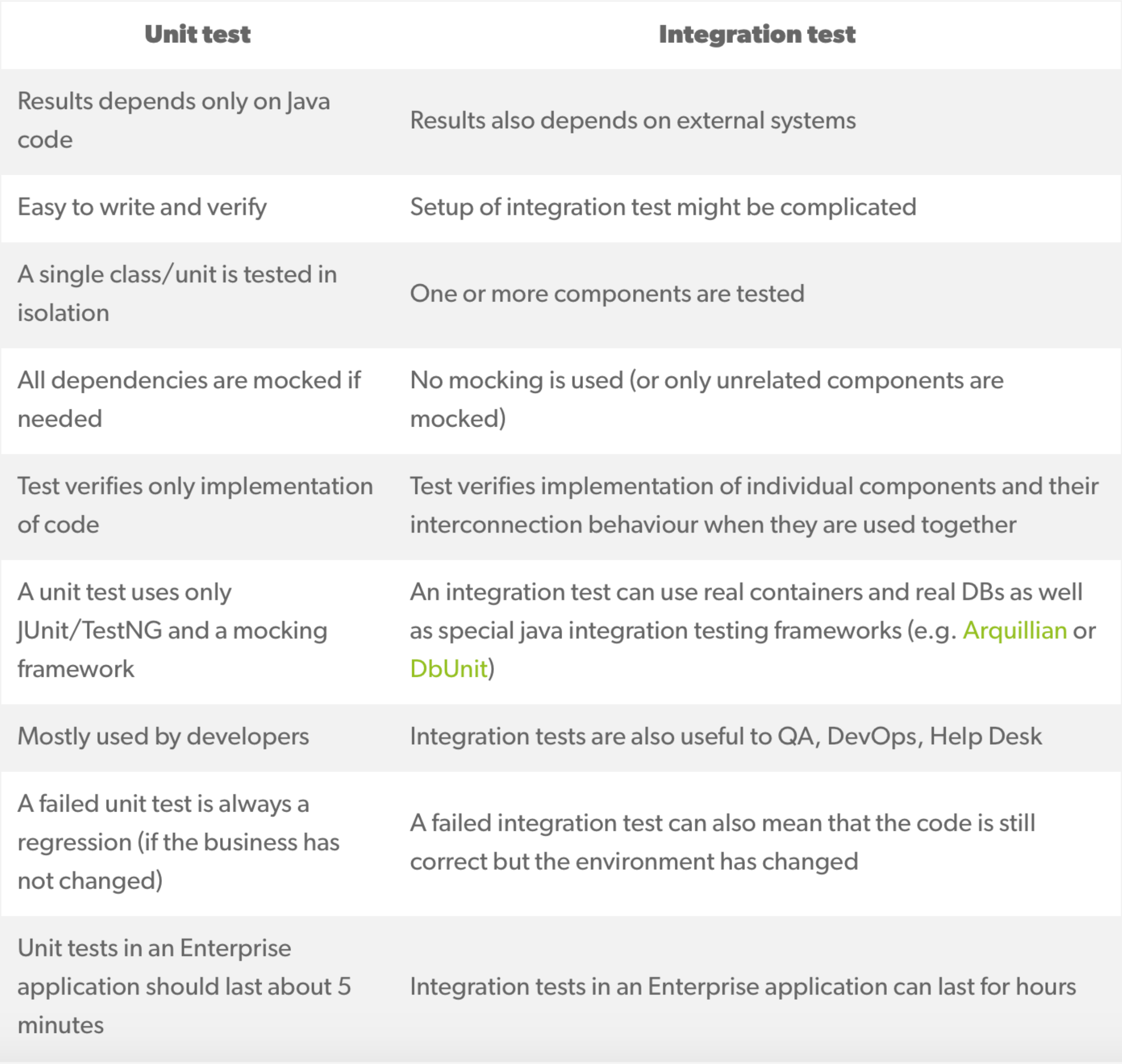
The **lowest level components are tested first in bottom-up testing**. They are then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested. All the bottom or low-level modules, procedures or functions are integrated and then tested. **After the integration testing of lower level integrated modules, the next level of modules will be formed and can be used for integration testing**. This approach is helpful only when all or most of the modules of the same development level are ready. This method also helps to determine the levels of software developed and makes it easier to report testing progress in the form of a percentage.

The top integrated modules are tested first in top-down testing and the branch of the module is tested step by step until the end of the related module. (Wikipedia)

Source: https://en.wikipedia.org/wiki/Integration_testing

Example of Integration Testing (Design Pattern – State)

```
@Before
public void before() {
    UserActivityDao.getInstance().deleteAll();
}

@Test(timeout=1000)
public void testUserStateLoginActionFail() {
    UserSession userSession = new UserSession();
    boolean login = userSession.login("admin", "1233");
    assertFalse(login);
    assertNull(userSession.username);
    UserActivity userActivity = userSession.createPost("abc");
    boolean likePost = userSession.likePost(1);
    List<ActionCount> actionCountList = userSession.generateActionCountReport();
    List<UserActivity> allPosts = userSession.findAllPosts();
    boolean logout = userSession.logout();
    assertNull(userActivity);
    assertFalse(likePost);
    assertFalse(logout);
    assertNull(actionCountList);
    assertNull(allPosts);

}
```



Unit test	Integration test
Results depends only on Java code	Results also depends on external systems
Easy to write and verify	Setup of integration test might be complicated
A single class/unit is tested in isolation	One or more components are tested
All dependencies are mocked if needed	No mocking is used (or only unrelated components are mocked)
Test verifies only implementation of code	Test verifies implementation of individual components and their interconnection behaviour when they are used together
A unit test uses only JUnit/TestNG and a mocking framework	An integration test can use real containers and real DBs as well as special java integration testing frameworks (e.g. Arquillian or DbUnit)
Mostly used by developers	Integration tests are also useful to QA, DevOps, Help Desk
A failed unit test is always a regression (if the business has not changed)	A failed integration test can also mean that the code is still correct but the environment has changed
Unit tests in an Enterprise application should last about 5 minutes	Integration tests in an Enterprise application can last for hours

Source: https://www.jrebel.com/blog/how-to-use-java-integration-testing & https://www.jrebel.com/blog/mock-unit-testing-with-mockito

System Testing

> System testing checks that **the entire system works within the environment that it will be placed in**

> It checks that all the hardware and software parts work with other external components in the context of normal loads

> Ideally this would involve a duplication of the hardware/software resources. However in some cases this is not feasible so system testing is performed "live"

Code Coverage of Testing

> How to find test cases?

> It should measure what percentage of code has been executed by test suite

Basic strategies [1]:

>> *"test whatever you test; let the users test the rest"*

not recommended, but very common

>> Statement coverage

"testing less than 100% stmt complete is unconscionable and should be criminalized", Boris Beizer

>> Decision/Branch coverage

>> Condition coverage

>> Multiple Condition coverage

>> Path coverage

[1] Lee Copeland. 2003. A Practitioner's Guide to Software Test Design. Artech House, Inc., USA.

Statement Complete

> To check whether all statements are executed at least once in the test code

```
someMethod(boolean a, boolean b, boolean c){
    if(a){
        statementX;
    }else{
        statementY;
        if(b) statementZ;
    }
    if(c) statementW;
}
```

```
someMethod(boolean a, boolean b, boolean c){
      if(a){
              statementX;
      }else{
                statementY;
                if(b) statementZ;
      }
      if(c) statementW;
}
```

Q: What would be the **minimal set of test cases** that is statement complete?

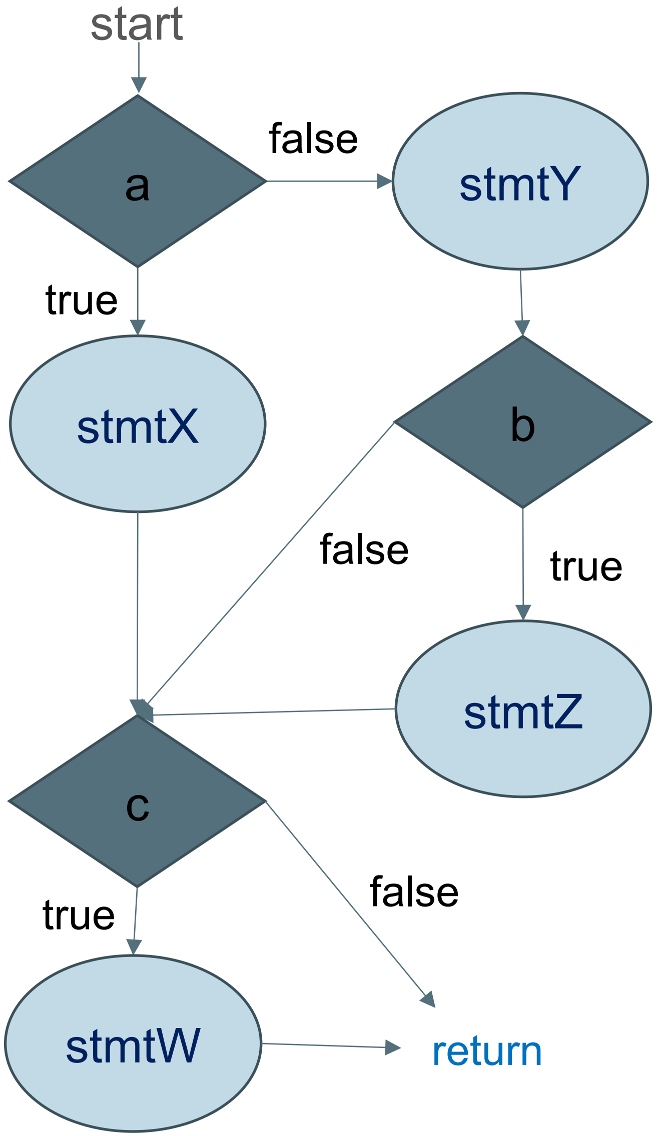
A: 2 test cases, why?

Q: How to calculate the statement coverage?

A: $$\text{Statement}_{\text{Coverage}} = \mathbf{100\ x}\ \frac{\text{number of } \textbf{exercised} \text{ statements}}{\text{total number of statements}}$$

Control Flow Graph

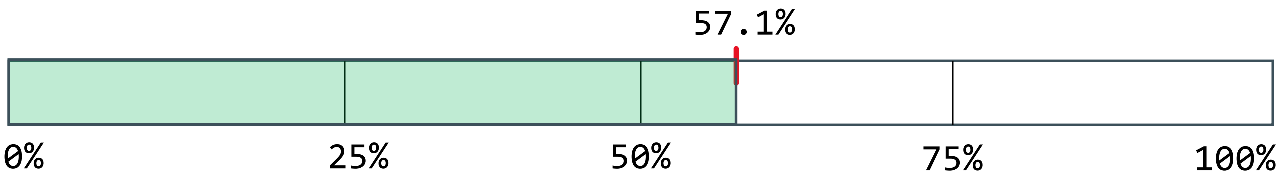
```
 someMethod(boolean a, boolean b, boolean c){
          if(a){
              statementX;
          }else{
              statementY;
              if(b) statementZ;
          }
          if(c) statementW;
}
```



Test Case 1: someMethod(true, true, true);

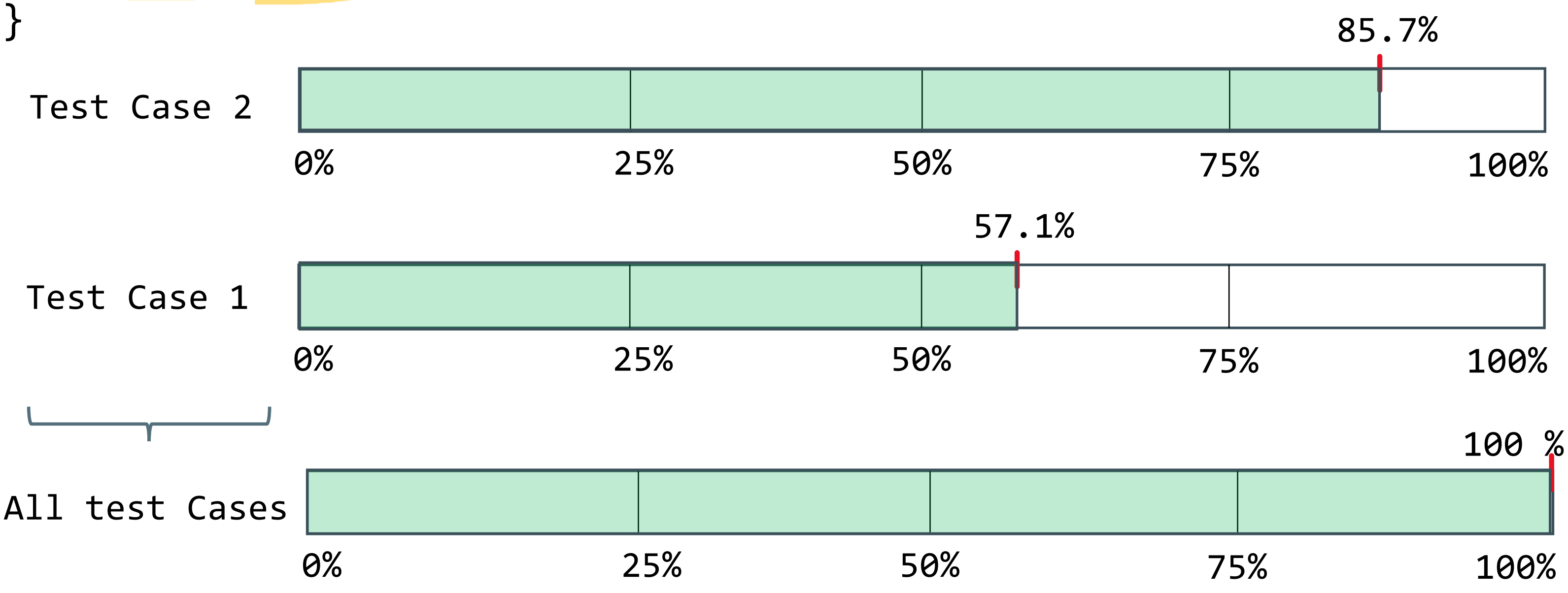
```
someMethod(boolean a, boolean b, boolean c){
      if(a){
            statementX;
      }else{
            statementY;
            if(b) statementZ;
      }
      if(c) statementW;
}
```

4 out of 7 statements are covered



```
Test Case 2: someMethod(false, true, true);

   someMethod(boolean a, boolean b, boolean c){
        if(a){
              statementX;
        }else{
              statementY;
              if(b) statementZ;
        }
        if(c) statementW;
}
```

6 out of 7 statements are covered



Statement Complete - Summary

> Statement Coverage is a **simple form to find errors** (there are better forms, but it is a good start)

> A statement can be considered covered if it is exercised

> It is **not easy to achieve 100%** of statement **coverage** (usually 80%) -> some code may be designed to be executed in rare conditions

> It **may not test what it really should** (just interested in covering the statements). **You still need to analyse what is being covered!**

> Difficult to be performed by code coverage tools: the Java Code tool for **Eclipse (JaCoCo) presents the results based on the number of lines covered** as it has no access to the source files (only to the byte code, which has no clear information about statements, types, etc.)

Statement Complete - Summary

Line code coverage != Statement coverage

> Line code coverage can lead developers to increase the number of lines in the code to present better statistical reports: 3/6 lines (50%), but if you expand your code to 10 lines and cover 7 => 70%

Tip: do not try this on your project!

Branch Complete

> To check all possible branch condition statements have been included in test cases

Q: What would be the **minimal set of test cases** that is branch complete?

A: 3 test cases, why?

Q: How to calculate the branch coverage?

A: $\text{Branch}_{\text{Coverage}} = \mathbf{100 \times} \frac{\text{number of } \textbf{executed branches}}{\text{total number of branches}}$

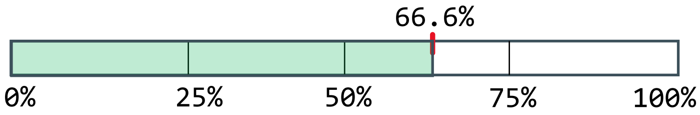
Control Flow Graph

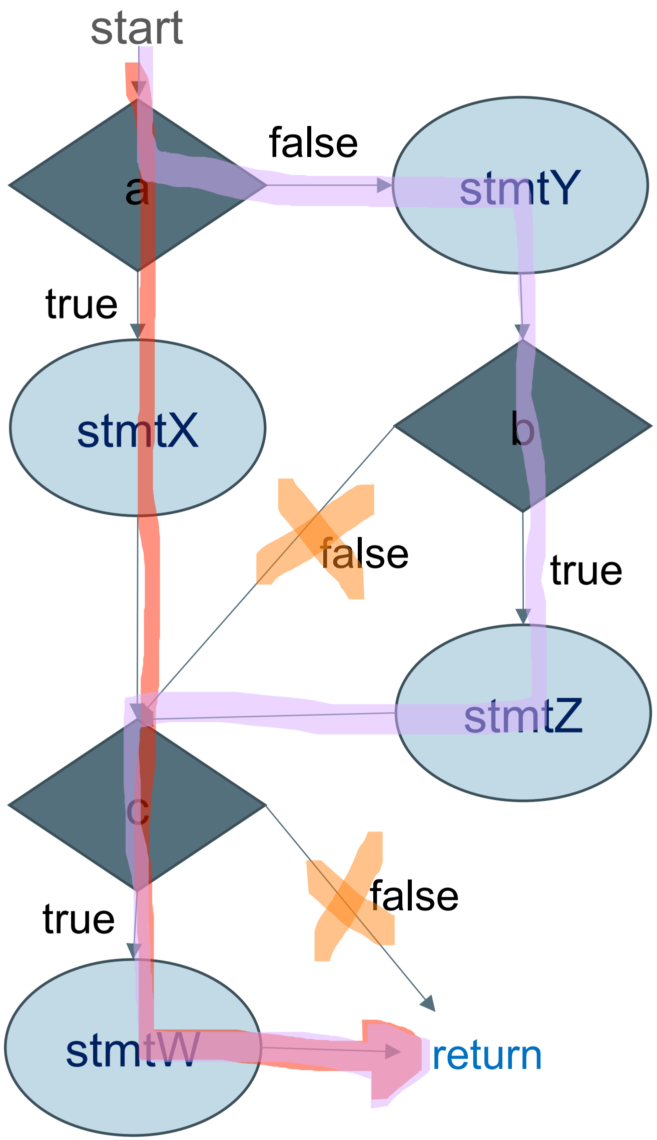
```
someMethod(boolean a, boolean b, boolean c){
        if(a){
            statementX;
        }else{
            statementY;
            if(b) statementZ;
        }
        if(c) statementW;
 }
```

Let's try first using the statement complete test cases:

```
someMethod(true, true, true);

someMethod(false, true, true);
```

It covers 4 out of 6 branches!





Control Flow Graph

```
someMethod(boolean a, boolean b, boolean c){
        if(a){
            statementX;
        }else{
            statementY;
            if(b) statementZ;
        }
        if(c) statementW;
}
```

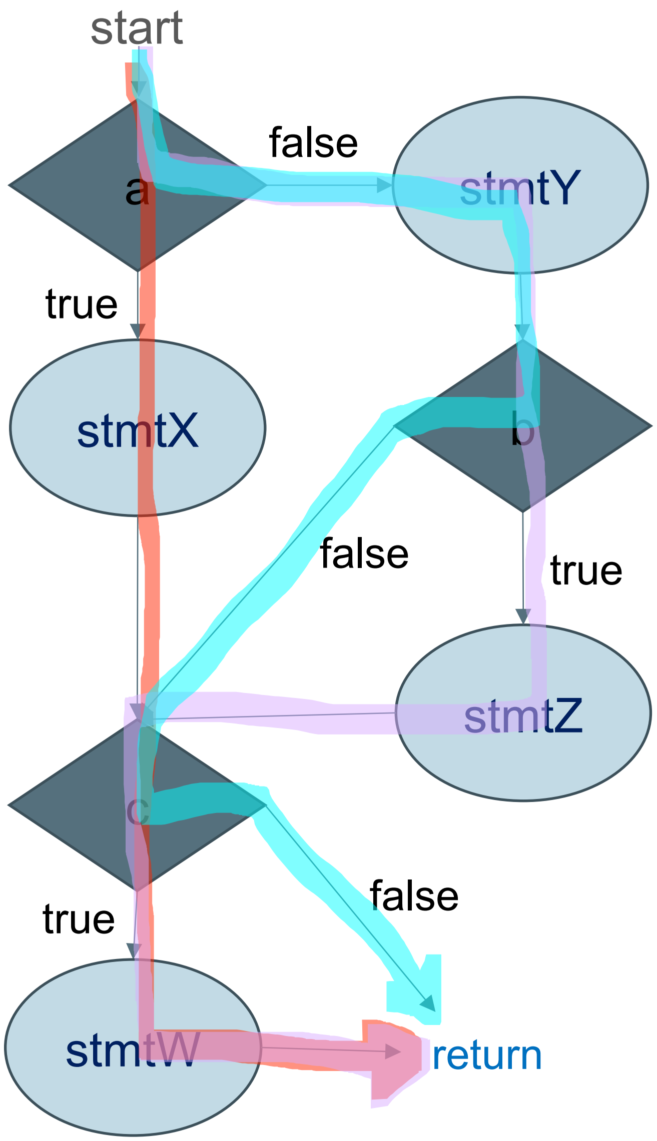
```
someMethod(true, true, true);

someMethod(false, true, true);
```

Do we need another test case? Which one?

```
someMethod(false, false, false);
```

Are these the only test cases possible to achieve Branch Complete?

No! There are other cases that would also achieve 100% branch coverage! Find it!



Branch Complete - Summary

> It usually **requires more test cases to achieve 100% Branch Coverage than** it is needed to achieve **100% Statement Coverage**

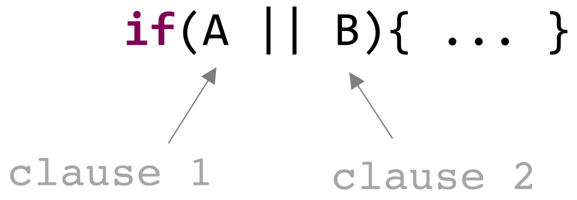
> Note that if you achieve 100% Branch Coverage, you will also achieve 100% Statement Coverage: **Branch Coverage subsumes Statement Coverage**

> It requires more test cases -> **more expensive**

> It also **does not guarantee** that your code is **bug free!** But, as it covers all branches (and, consequently, all statements), you may encounter code defects that could not be discovered with the statement strategy

Condition Complete

> When the conditions on the branches have more than one clause, we can test each clause separately.



```
if(A || B){ ... }
```

> Condition Complete: check if all conditions within decision expressions have been ***evaluated*** to both true and false.

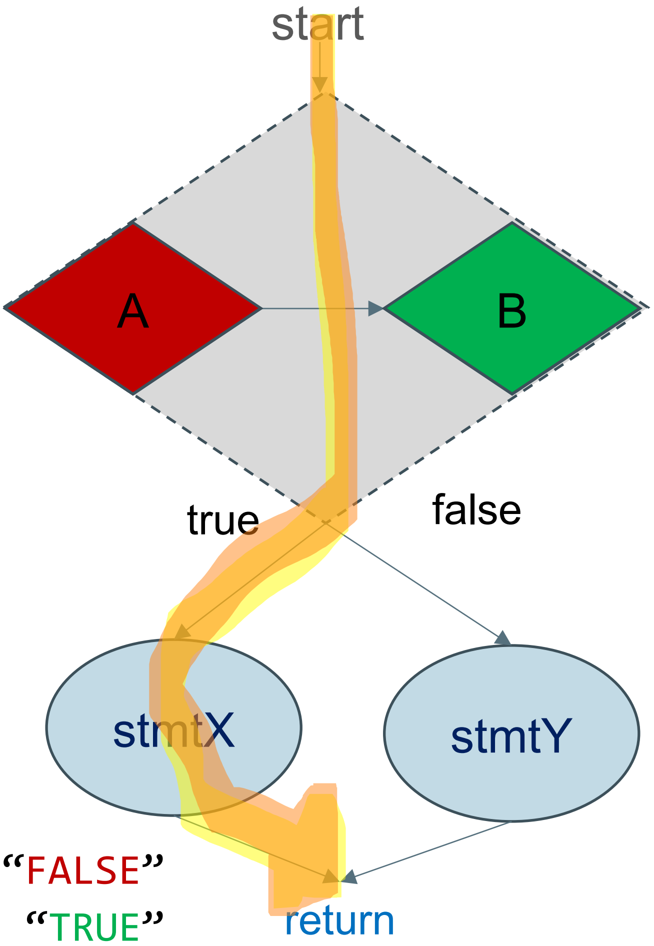
```
Q: How to calculate the condition coverage?
```

A: $$\text{Condition}_{\text{Coverage}} = \mathbf{100 \times} \frac{\text{number of } \textbf{clauses evaluated to both [F,T]}}{\text{total number of clauses}}$$

**Note that in most programming languages non-decisive clauses will not be evaluated* —

Condition Complete

```
someMethod(...){
        if(A || B){
                statementX;
        }else{
                statementY;
        }
}
```



Q: How to achieve 100% condition coverage?

1) A is evaluated to "TRUE" and B is evaluated to "FALSE"

2) A is evaluated to "FALSE" and B is evaluated to "TRUE"

Does Condition Coverage imply Branch Coverage?

No, "else" is not exercised. Only 50% of the branches are covered.

Condition Complete - Summary

> if you achieve 100% Condition Coverage, you may not achieve 100% Branch Coverage:

Condition Coverage does not subsume Branch Coverage

> You must write test cases so that each condition that has a *true* or *false* outcome that makes up a decision is evaluated at least once

> **Condition coverage** is usually considered to be **better than branch coverage** as it tests at least once each individual condition. **Branch coverage may be achieved without testing every condition.**

Multiple Condition Complete

> For completeness, consider how the compiler evaluates the multiple conditions in a decision/branch.

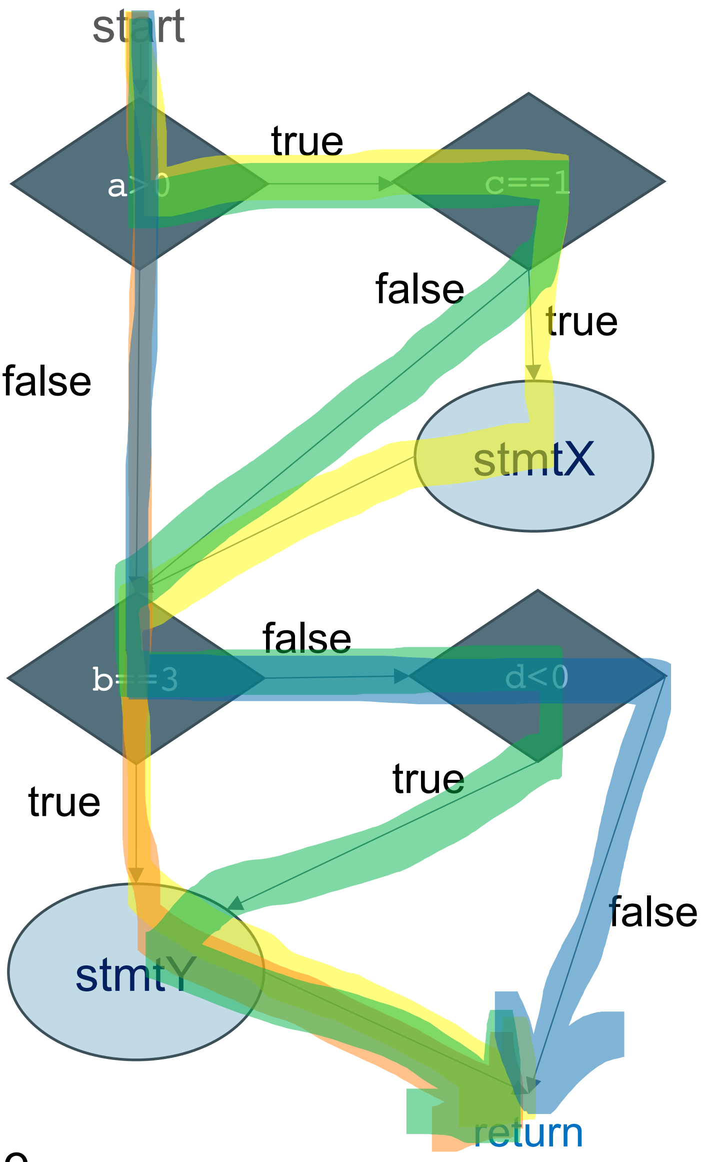
```
someMethod(int a, int b, int c, int d){
        if(a > 0 && c == 1){
                statementX;
        }
        if (b == 3 || d < 0){
                statementY;
        }
}
```

Q: How to calculate the multiple condition coverage?

A: $$\text{MultCondition}_{\text{Coverage}} = 100 \times \frac{\text{number of conditions and branches evaluated}}{\text{total number of conditions and branches}}$$

Multiple Condition Complete

```
someMethod(int a, int b, int c, int d){
        if(a > 0 && c == 1){
                statementX;
        }
        if (b == 3 || d < 0){
                statementY;
        }
}
```



How to achieve 100% multiple condition coverage?

```
someMethod(a>0, b=3, c=1, d<0);
someMethod(a<=0, b=3, c=1, d>=0);
someMethod(a>0, b!=3, c!=1, d<0);
someMethod(a<=0, b!=3, c!=1, d>=0);
```

Test cases for 100% coverage: Yellow, Green and Blue

Multiple Condition Complete - Summary

> if you achieve **100% Multiple Condition Coverage**, you will **achieve 100% decision and 100% condition coverage**.

> Multiple Condition coverage is more thorough

> Multiple Condition Coverage does not guarantee Path Coverage

> **Branch and Multiple Condition are terms that are always used interchangeably but have different meanings.** Branch Coverage in Eclipse (JaCoCo), for example, is actually reporting Multiple Condition Coverage.

Path Complete

> To check all possible paths have been covered by test cases

Q: What would be the **minimal set of test cases** that is path complete?

A: 6 test cases, why?

Q: How to calculate the path coverage?

A: $\text{Path}_{\text{Coverage}} = \mathbf{100 \times} \dfrac{\text{number of executed paths}}{\text{total number of paths}}$

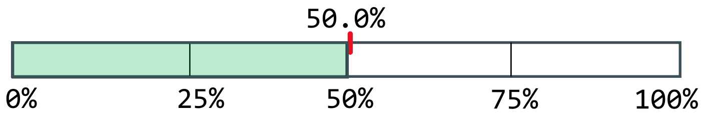
Control Flow Graph

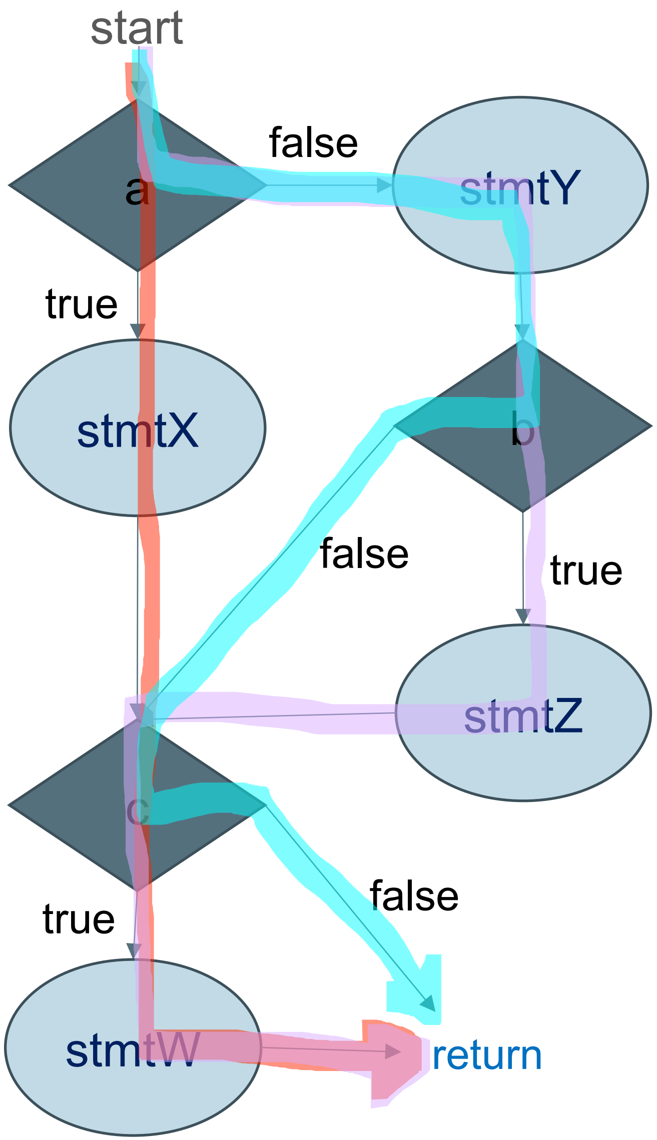
```
someMethod(boolean a, boolean b, boolean c){
        if(a){
            statementX;
        }else{
            statementY;
            if(b) statementZ;
        }
        if(c) statementW;
}
```

Branch Coverage test cases:

```
someMethod(true, true, true);
someMethod(false, true, true);
someMethod(false, false, false);
```

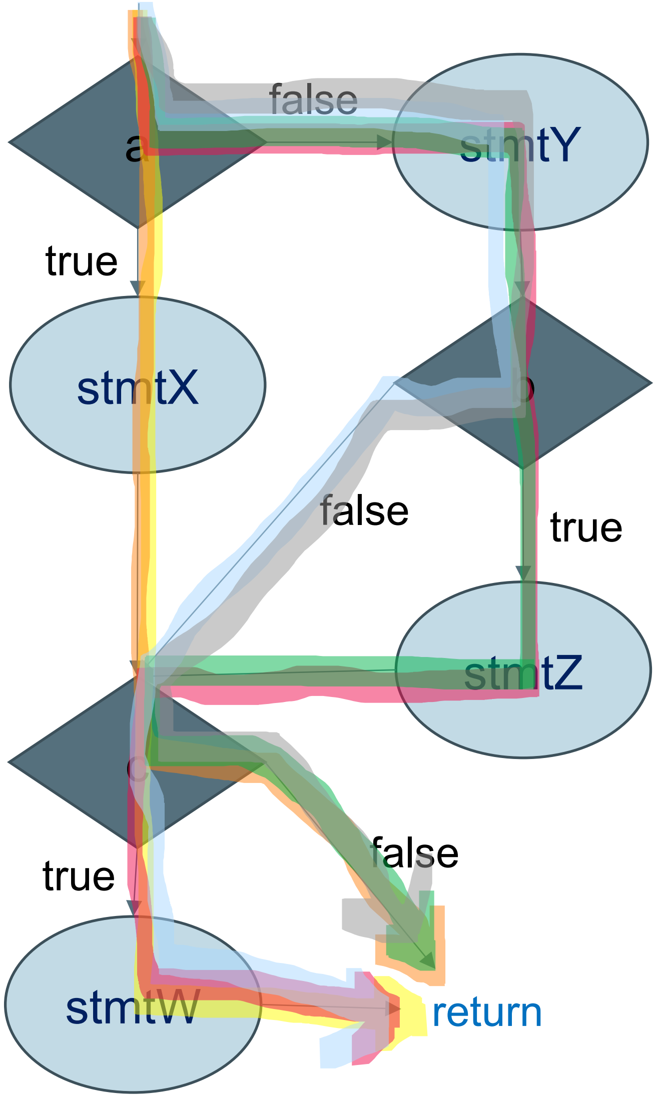
Does 100% branch coverage necessarily imply Path Complete?





Control Flow Graph

```
someMethod(true, true, true);
someMethod(true, true, false);
someMethod(false, true, true);
someMethod(false, true, false);
someMethod(false, false, true);
someMethod(false, false, false);
```



Path Complete - Summary

> It usually **requires more test cases to achieve 100% Path Coverage than** it is needed to achieve **100% Branch Coverage**

> Note that if you achieve 100% Path Coverage, you will also achieve 100% Branch Coverage: Path Coverage subsumes Branch Coverage

> Every possible combination is considered a **path (even when the else condition is not present)**

> **Loops can generate unlimited paths** (hard to cover all)

> It requires even more test cases -> **more expensive**

> **Cover all paths might not be feasible** (think about a medium size software, the number of paths it may have, and how to compute all of them (impractical and too expensive)).

Code that does not get covered!

> Usually, code executed under exceptional circumstances, such as low memory, full disk, lost connections, bad registry data, etc.

> How to solve this problem and increase coverage?

>> **Fault Injections tools**

>> It introduces faults during compile-time and runtime.

Examples:

(i) Mutation: change lines of the code (`i=i+1` becomes `i=i-1`)
(ii) Code insertion: add code to produce faults
(iii) Memory Corruption: RAM, I/O map
(iv) Network Fault: loss or reordering of network packets

Instrumentation

> How do we know your solution achieves 100% coverage?

> Code Instrumentation

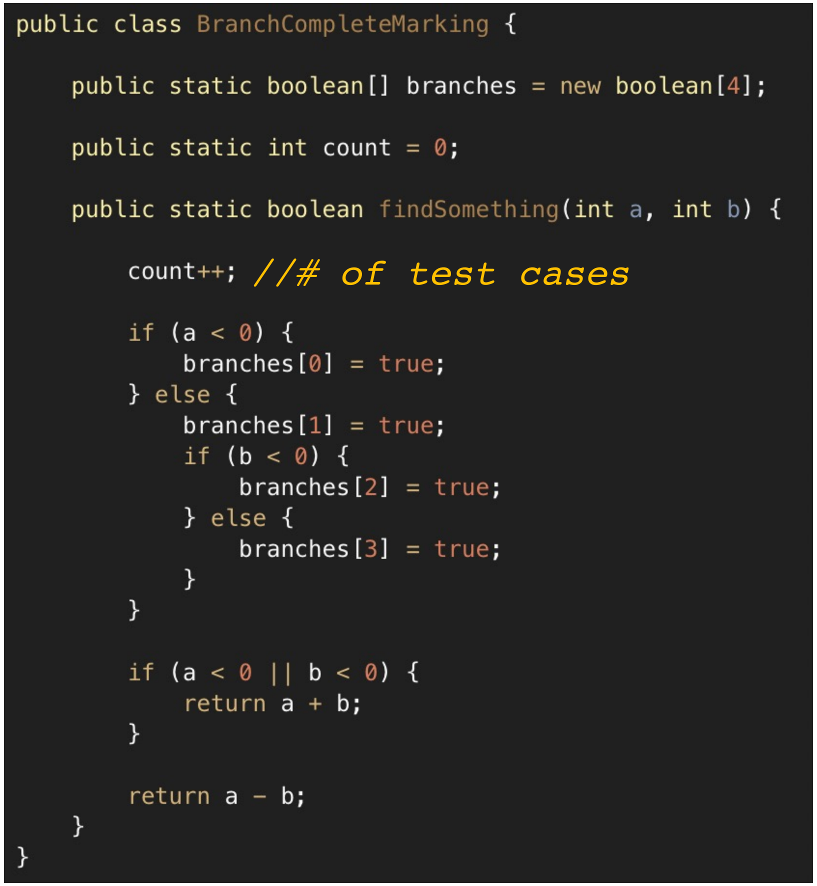
extra code is added to the software to monitor information about the software execution

> It is often used to observe functional aspects of the software as for example **function calls** and returned values; and data aspects such as variable value changes and memory allocation

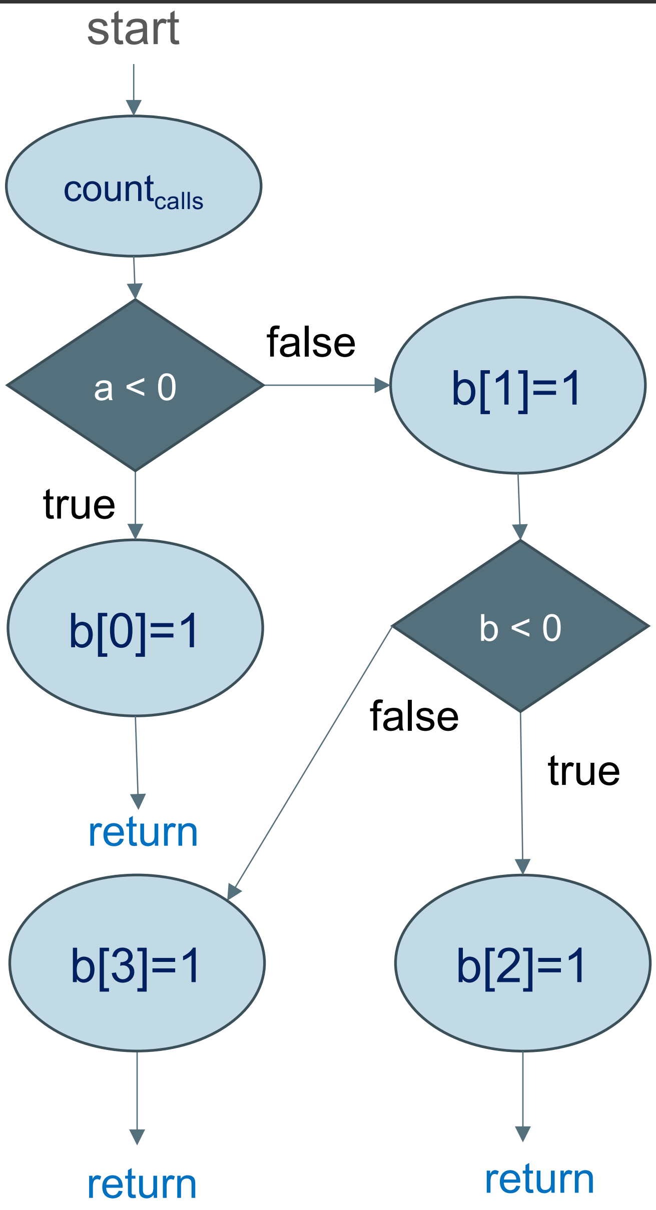
> The information extracted can be analysed and used to measure test coverage

Example - Instrumentation

> How do we know your solution achieves 100% multiple condition coverage?



```
public class BranchCompleteMarking {

    public static boolean[] branches = new boolean[4];

    public static int count = 0;

    public static boolean findSomething(int a, int b) {

        count++; //# of test cases

        if (a < 0) {
            branches[0] = true;
        } else {
            branches[1] = true;
            if (b < 0) {
                branches[2] = true;
            } else {
                branches[3] = true;
            }
        }

        if (a < 0 || b < 0) {
            return a + b;
        }

        return a - b;
    }
}
```

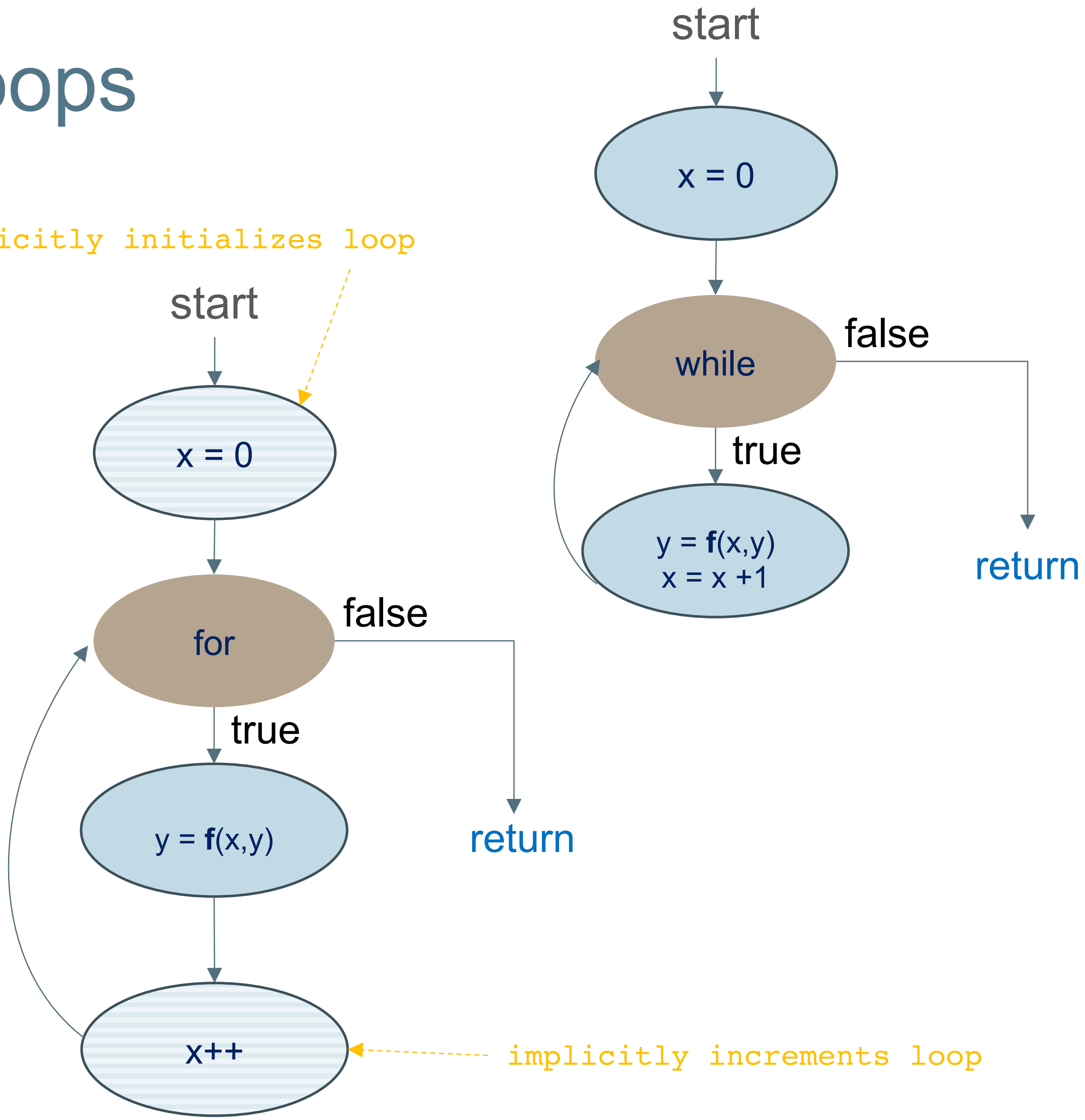


Simplified example and illustration

While and *For* loops

```
int x = 0;
while(x < y){
  y = f(x,y);
  x = x + 1;
 }
return x;
```

```
for(int x = 0; x < y; x++){
  y = f(x,y);
}
return x;
```

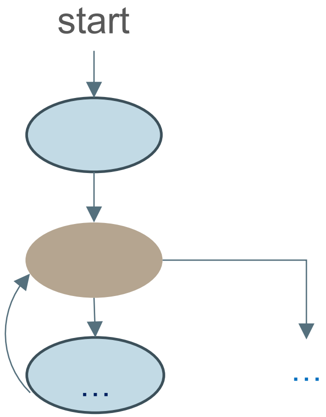


implicitly initializes loop

implicitly increments loop

Source: Introduction to Software Testing, Edition 2, Ammann & Offutt

While and *For* loops



> How do we cover/test loops?

>> Run the loop **zero times**

>> Run the loop exactly **once**

>> Run the loop ***n* times** where *n* is a small and **typical loop value**

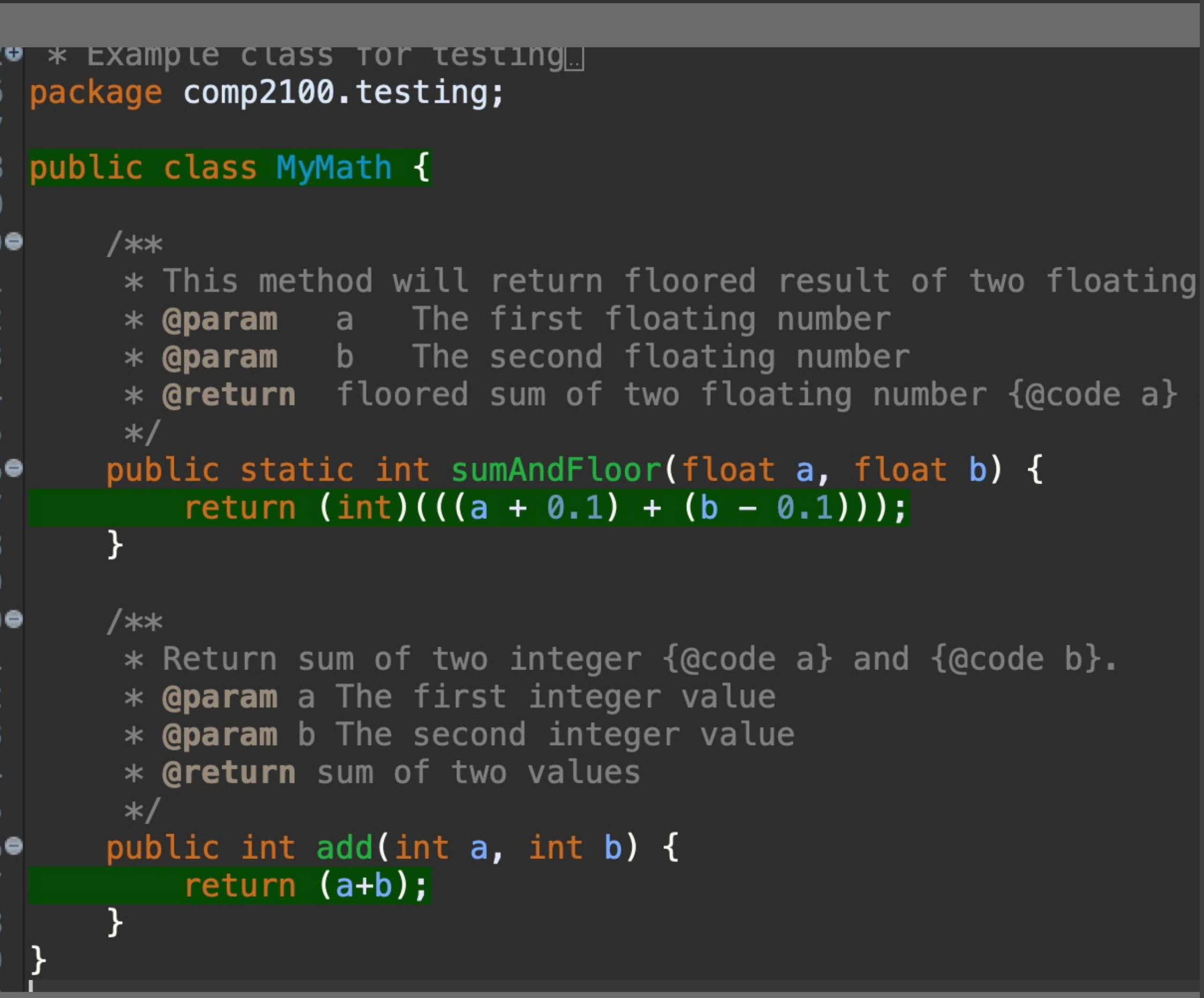
>> Run the loop ***m* times** where *m* is the **maximum** number of times the loop can be executed

>> Run the loop ***m+1*** and ***m-1 times*** *[optional]*

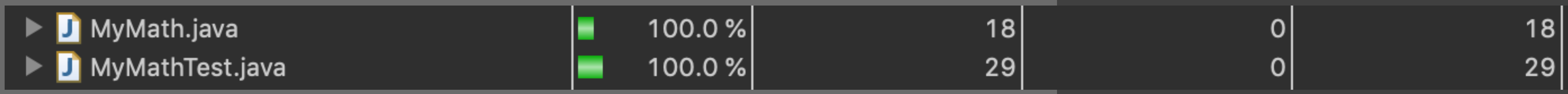
Source: A practitioner's guide to software test design by Copeland

Example: in Eclipse

- Eclipse supports
- Line Coverage
- Branch Coverage
- …



```
 * Example class for testing
package comp2100.testing;

public class MyMath {

    /**
     * This method will return floored result of two floating
     * @param    a    The first floating number
     * @param    b    The second floating number
     * @return   floored sum of two floating number {@code a}
     */
    public static int sumAndFloor(float a, float b) {
        return (int)(((a + 0.1) + (b - 0.1)));
    }

    /**
     * Return sum of two integer {@code a} and {@code b}.
     * @param a The first integer value
     * @param b The second integer value
     * @return sum of two values
     */
    public int add(int a, int b) {
        return (a+b);
    }
}
```



MyMath.java	100.0 %	18	0	18
MyMathTest.java	100.0 %	29	0	29

Exercise

Fill in the blank in the following statement: _____ certifies that the system meets the requirements (Building the right thing).

Select one:

1. Authentication
2. Confirmation
3. Verification
4. Validation

Exercise

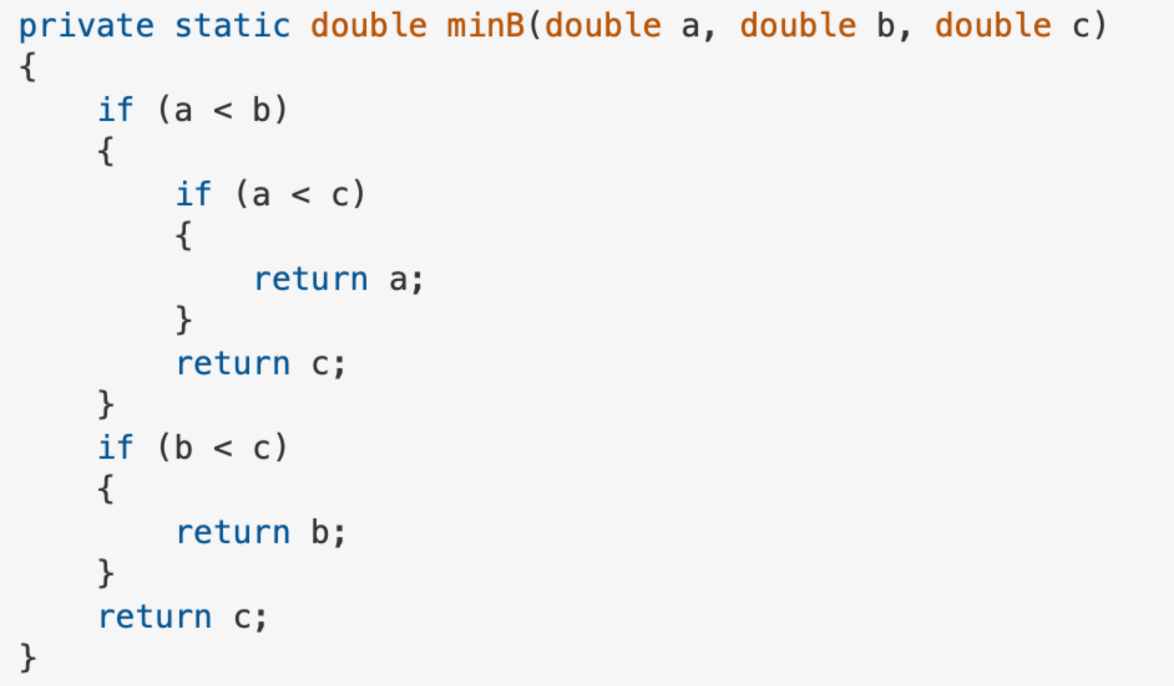
In JUnit version 4, if you want to execute the method A after each test, you have to ______ .

Select one:

1. annotate the method A with @After
2. annotate the method A with @Test
3. annotate the method A with @AfterClass
4. annotate the method A with @Before

Exercise

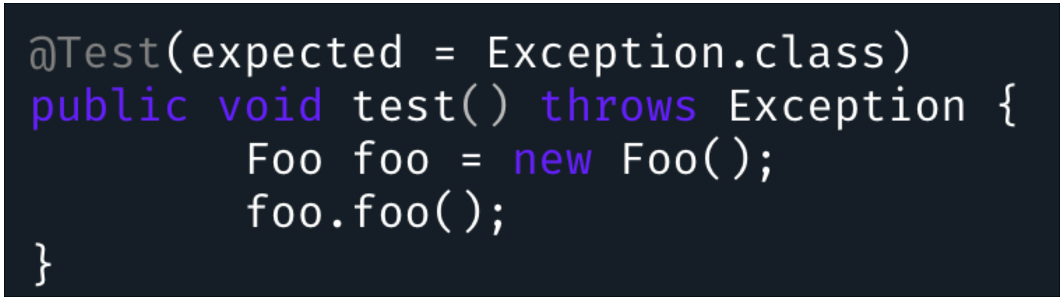
Draw the control flow graph for the method in the figure above. What is the total number of branches?



```
private static double minB(double a, double b, double c)
{
    if (a < b)
    {
        if (a < c)
        {
            return a;
        }
        return c;
    }
    if (b < c)
    {
        return b;
    }
    return c;
}
```

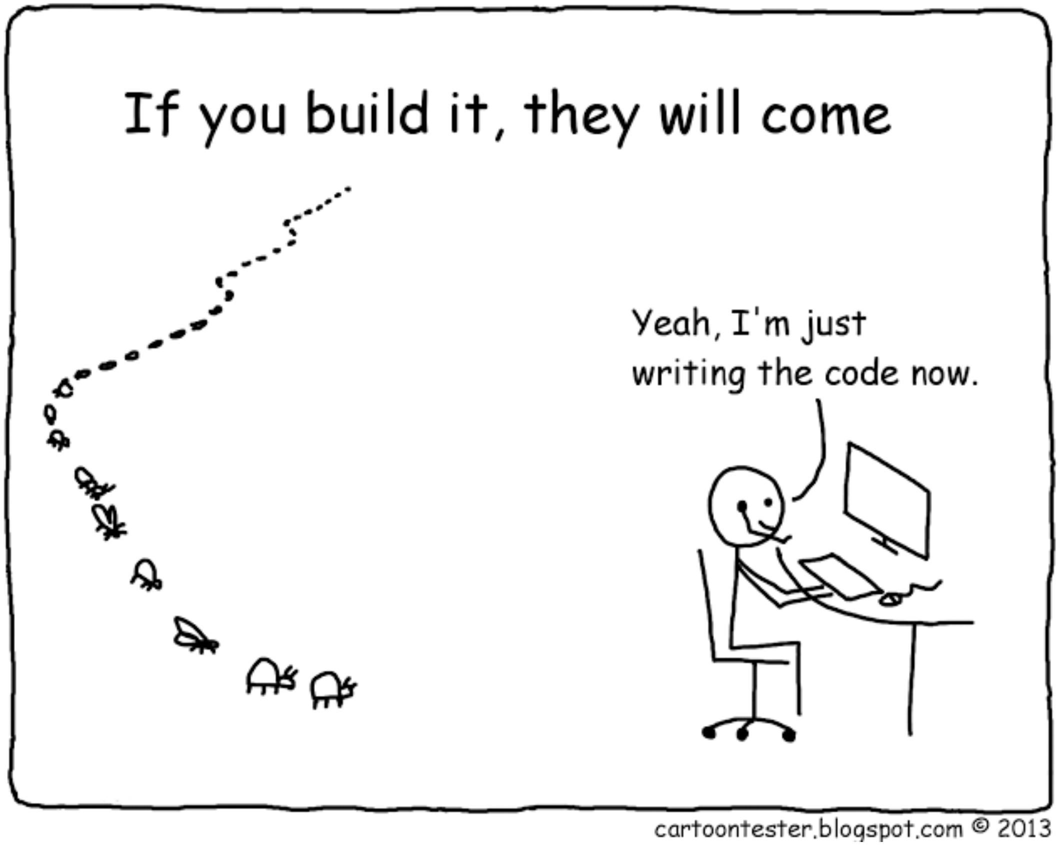
Exercise

In the following test case, will the test pass or fail if the exception is thrown? Why?



```
@Test(expected = Exception.class)
public void test() throws Exception {
        Foo foo = new Foo();
        foo.foo();
}
```

Meme for today's lecture! Keep practicing!


If you build it, they will come
Yeah, I'm just writing the code now.
cartoontester.blogspot.com © 2013

I still prefer a bug you generated than one generated by someone else!

References

JUnit 4

- https://junit.org/junit4/

Useful site:

- **Software Testing Fundamentals:** http://softwaretestingfundamentals.com/

Books:

- Software Engineering by Ian Sommerville
- The Mythical Man-Month by Fred Brooks