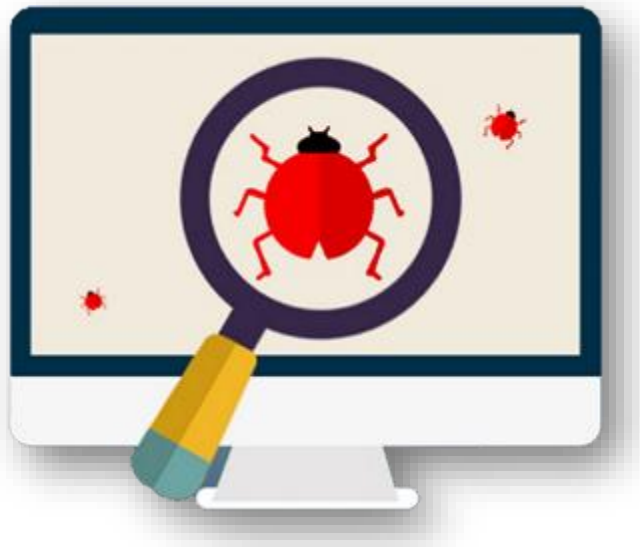

SOFTWARE
TESTING
Sid Chi-Kin Chau
[Lecture 1]


I FOUND A CRITICAL BUG IN OUR SOFTWARE THAT COULD MAKE OUR PRODUCT WORTHLESS IN A WEEK.
DILBERT.COM @SCOTTADAMSSAYS
IF YOU GIVE ME A HUGE RAISE, I WON'T TELL ANYONE ABOUT THE PROBLEM UNTIL YOU SELL ALL OF YOUR COMPANY STOCK.
DEAL!
2-7-18 ©2018 Scott Adams, Inc./Dist. by Andrews McMeel
TWO WEEKS LATER
WHY HAVEN'T I HEARD ABOUT THE BUG YET?
YOU DIDN'T ASK ME IF I KNEW HOW TO FIX IT.

Software Testing

- Testing is an example of validation
  - To uncover problems (i.e., bugs) in a program
  - Increase confidence in the program's correctness
- Validation includes:
  - Verification
    - Construct a formal proof that a program is correct
  - Code review
    - Ask a third-party to carefully read your code (i.e., informal proofread)
  - Testing
    - Running the testing program on carefully selected inputs and checking the outputs for consistency



stackoverflow

What % of programming time do you spend debugging?

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

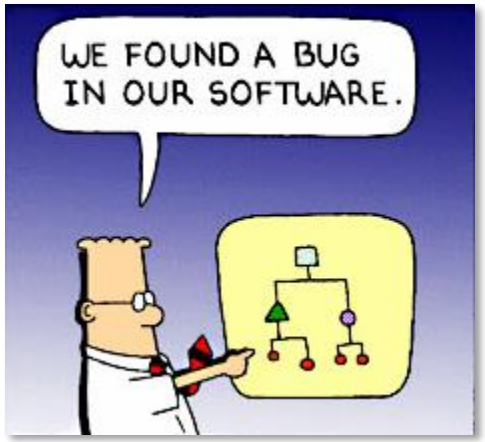
9

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.



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.

Why Software Testing is Hard

- Exhaustive testing is infeasible
  - The space of possible test cases is generally too big to cover exhaustively
- Haphazard (unsystematic) testing is ineffective
  - Less likely to find bugs, unless the program is so buggy
  - Not necessary to increase our confidence in program correctness
- Random or statistical testing provides low confidence
  - Random deflects may be present in physical systems
  - Software systems may mot have random bugs
- Need for efficient and systematic testing

Goals of This Lecture

- Testing approaches
  - Black box testing
  - White box testing
- Coverage testing
  - Statement complete
  - Branch complete
  - Path complete
- Testing levels
  - Unit testing
  - Integration testing

Software Testing

- “Program testing can be used to show the presence of bugs, but never to show their absence.”

  E. W. Dijkstra

- Testing will not prove code to be correct!
  - But it does provide confidence and it will often uncover problems within the code

- 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

Benefits of Testing

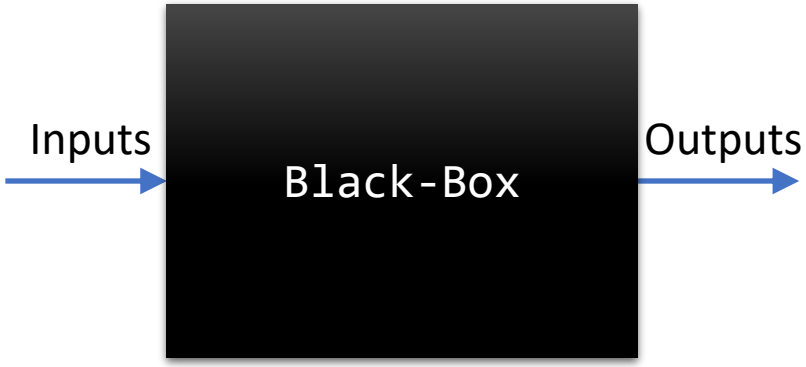
- Check the code is behaving as expected
- Find and isolate bugs/defects early in the development cycle
- Demonstrate that the software developed meets its requirements
- Increase the confidence of both the developer and the customer in the software system
- Help reflect on the correctness of the design and implementation
- Mark programming assignments

Validation and Verification

- Validation
  - Building the right thing
    - Certify that the system meets customers needs
  - Check whether the developer is building the correct product
  - 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
  - Check the system against the design
  - Certify the quality of the system

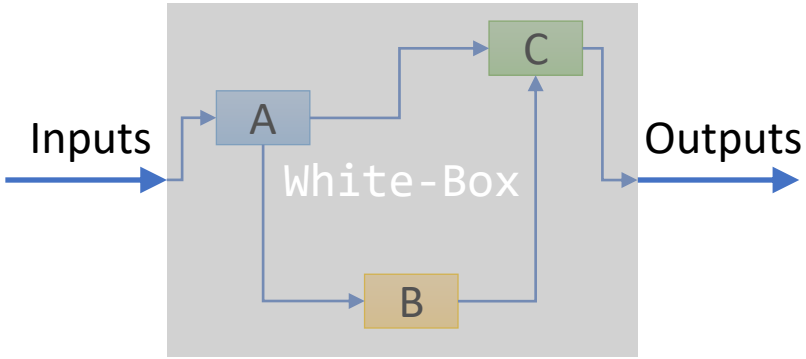
Black-Box Testing

- Tests are based purely on the functional specifications of the code
- There are three types of possible cases:
  - Normal function
  - Boundary cases in the specifications
  - Testing cases outside requirements and robustness



White-Box Testing

- A set of tests can be constructed based on the code
  - For example, if the code selects different algorithms depending on the input, then you should partition tests according to those domains of input
  - This enables the tester to target more specific test cases
- White-box testing would normally involve generating test cases that have good "code coverage"
  - Test cases can be constructed based on the boundary cases in the code



Code Coverage of Testing

- *Big question:* How to find test cases?
  - Based on how to measure what percentage of code has been executed by test case suite
- Basic strategies:
  - Statement complete
  - Branch complete
  - Path complete
- Test coverage
  - Path $\supseteq$ Branch $\supseteq$ Statement

Statement Complete

- Statement Completeness
  - Check whether all statements are executed at least once in the test code
  - Statements are not lines of code!
- How to measure statement coverage
  - Statement-Coverage = $\frac{\text{number of executed statements}}{\text{total number of statements}}$
- Goal is to find the **minimal set of test cases** to ensure statement completeness

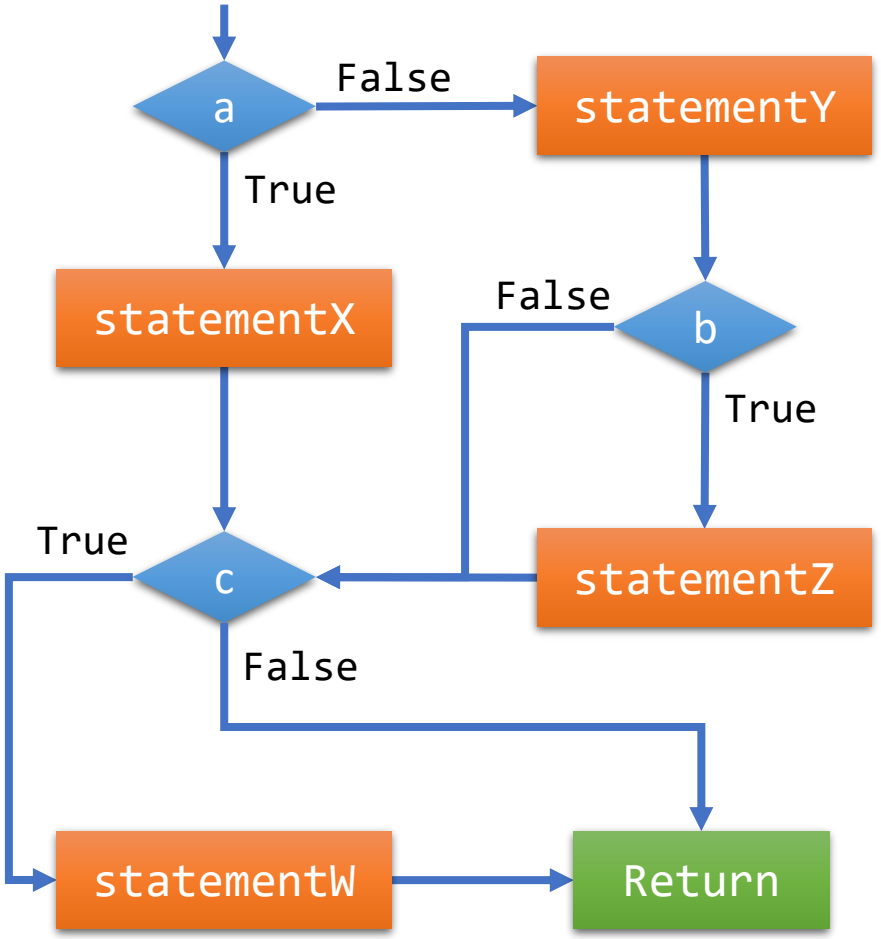
Statement Complete Example

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

- 7 statements:
  - `if(a), if(b), if(c), statementX, statementY, statementZ, statementW`
- What would be the **minimal set of test cases** to ensure statement completeness?
  - 2 test cases. Why?

Statement Complete Example

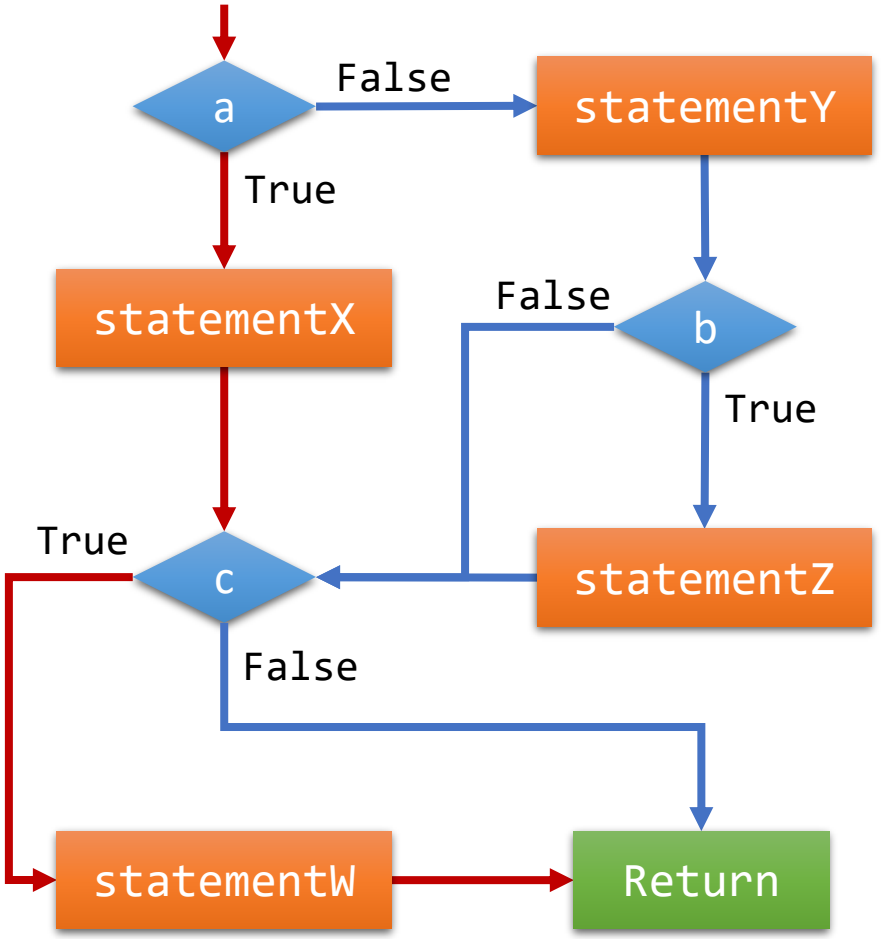
```
someMethod(boolean a, boolean b, boolean c){
        if(a){
                statementX;
        }else{
                statementY;
                if(b) statementZ;
        }
        if(c) statementW;
}
```



Statement Complete Example

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

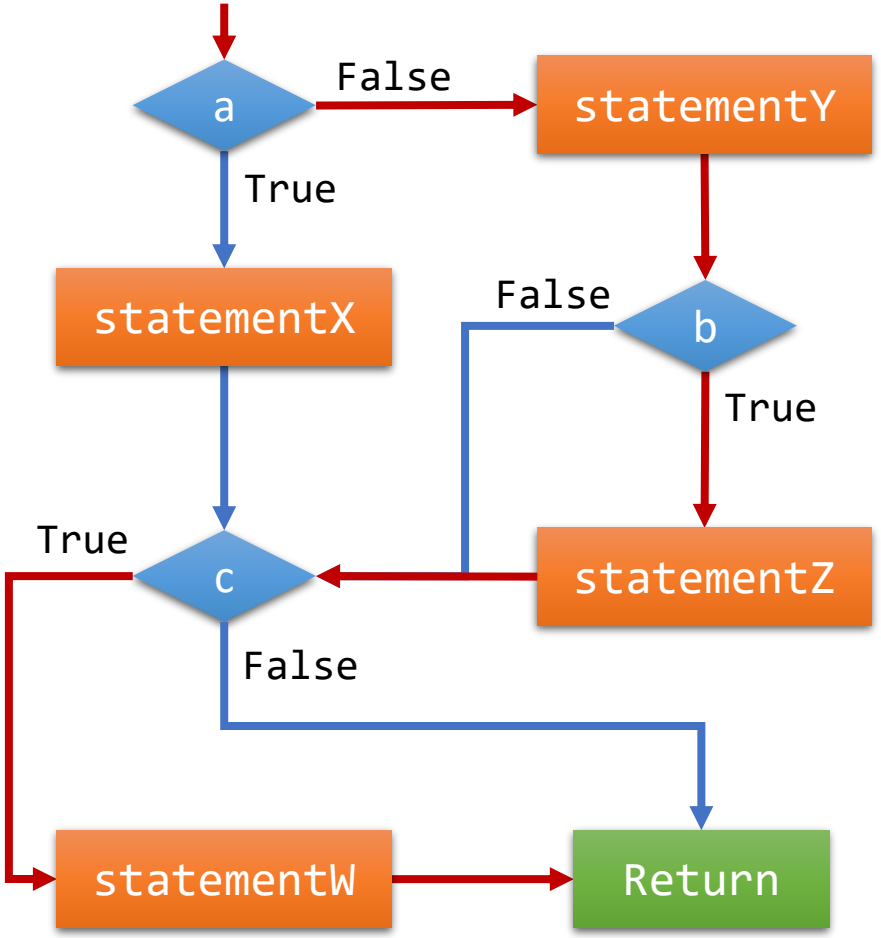
- Test Case 1: someMethod(true, true, true);
- Statement-Coverage = 4/7 = 57.1%



Statement Complete Example

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

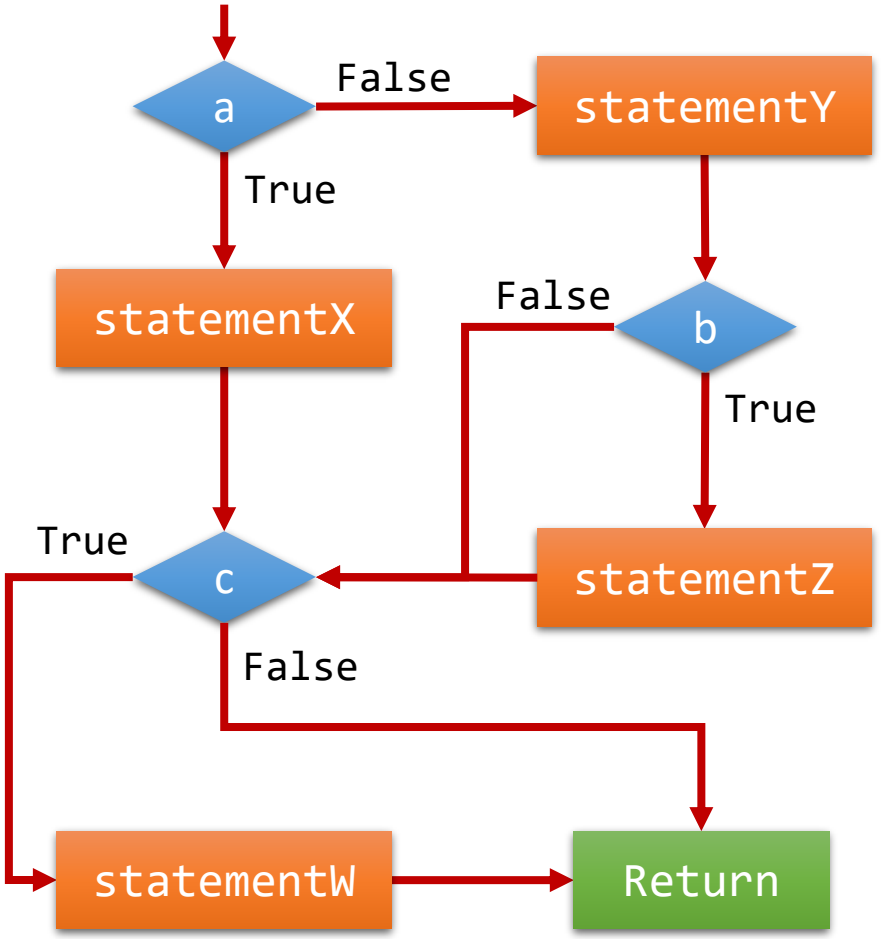
- Test Case 2: someMethod(false, true, true);
- Statement-Coverage = 6/7 = 85.7%



Statement Complete Example

```
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);
- Test Case 2: someMethod(false, true, true);
- Total Statement-Coverage = 7/7 = 100%



Statement Complete

- Not easy to achieve 100% of statement coverage (usually 80%)
  - Some code may be designed to be executed in rare conditions
- May not test what it really should (just covering the statements)
  - You still need to analyze what is being covered!
- Difficult to be validated by code coverage tools
  - Java Code tool for Eclipse 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.)
- Line of 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 lines (70%)

Branch Complete

- Branch Completeness
  - Check all possible branches in branch condition statements have been included in test cases
  - Branch condition statements
    - if-else, case, while, repeat
- How to measure branch coverage
  - Branch-Coverage = $\frac{\text{number of executed branches}}{\text{total number of branches}}$
- Goal is to find the **minimal set of test cases** to ensure branch completeness

Branch Complete Example

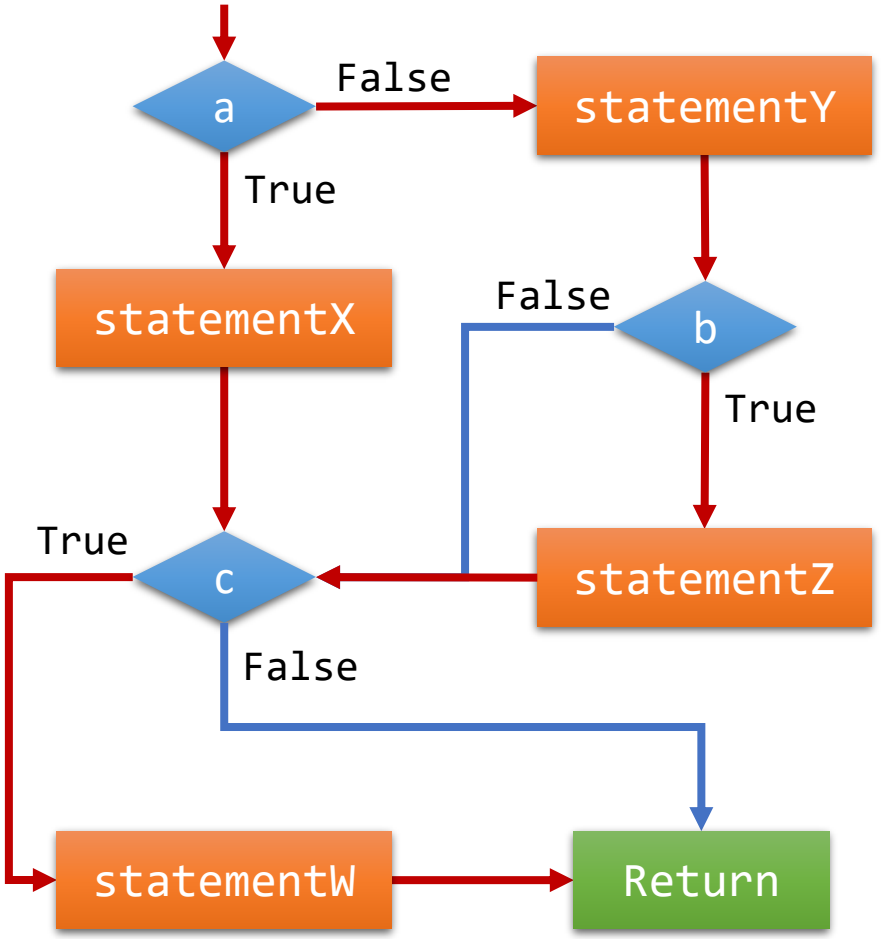
```
someMethod(boolean a, boolean b, boolean c){
        if(a){
                statementX;
        }else{
                statementY;
                if(b) statementZ;
        }
        if(c) statementW;
}
```

- 6 branches
  - a = True or False, b = True or False, c = True or False
- What would be the **minimal set of test cases** to ensure branch completeness?
  - 3 test cases. Why?

Branch Complete Example

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

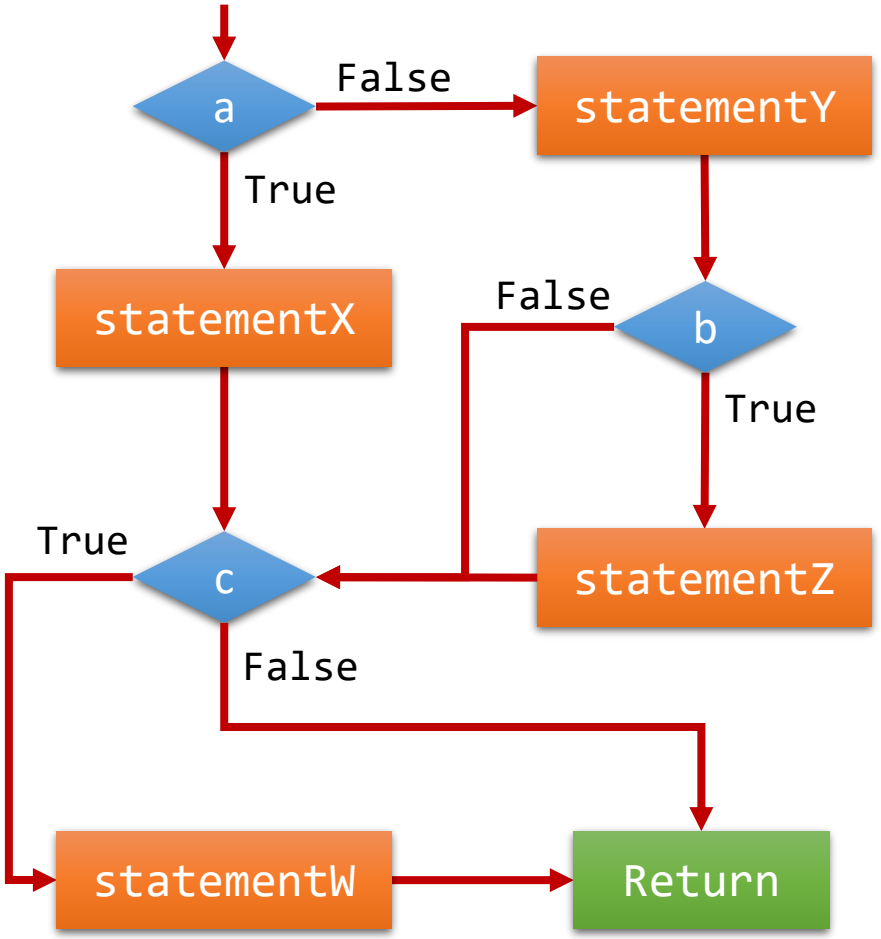
- Test Case 1: someMethod(true, true, true);
- Test Case 2: someMethod(false, true, true);
- Total Branch-Coverage = 4/6 = 66.6%



Branch Complete Example

```
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);
- Test Case 2: someMethod(false, true, true);
- Test Case 3: someMethod(false, false, false);
- Total Branch-Coverage = 6/6 = 100%



Branch Complete

- Usually strictly more test cases to achieve 100% Branch Coverage than it is needed to achieve 100% Statement Coverage
  - 100% Branch Coverage implies 100% Statement Coverage
- Requires more test cases, and more overhead
- 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-complete test cases

Path Complete

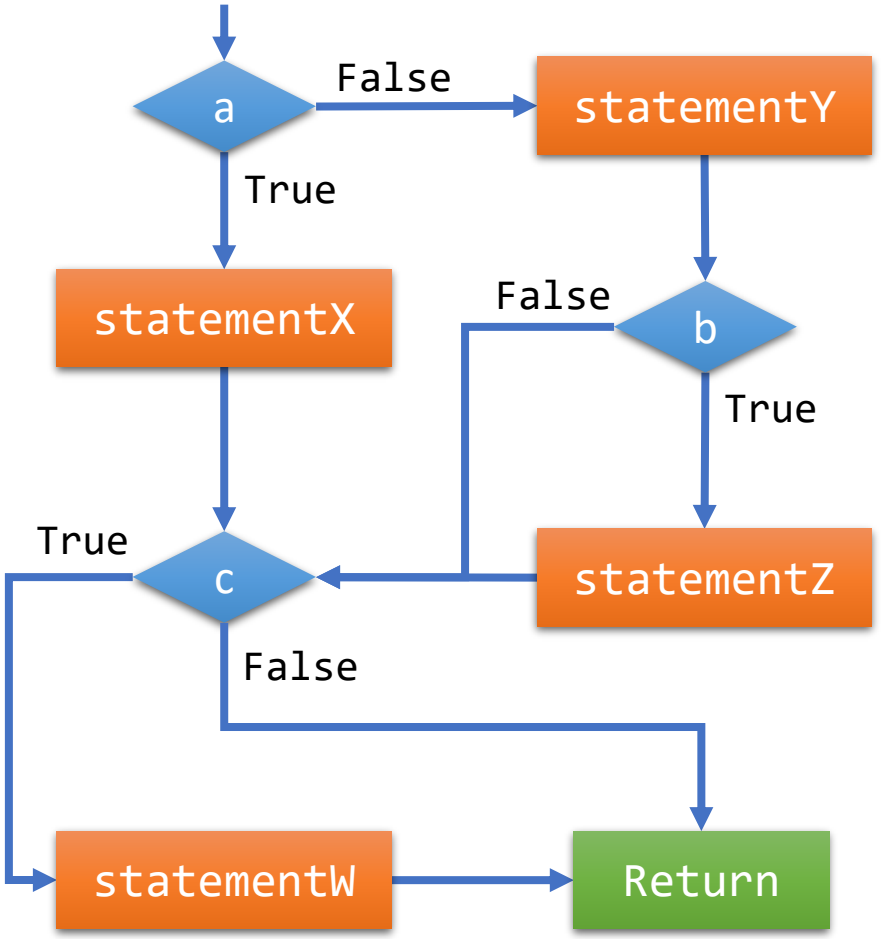
- Path Completeness
  - Check all possible execution paths have been covered by test cases
  - Execution paths in the flowchart
- How to measure path coverage
  - Path-Coverage = $\frac{\text{number of executed paths}}{\text{total number of paths}}$
- Goal is to find the **minimal set of test cases** to ensure path completeness

Path Complete Example

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

- 6 paths in flowchart

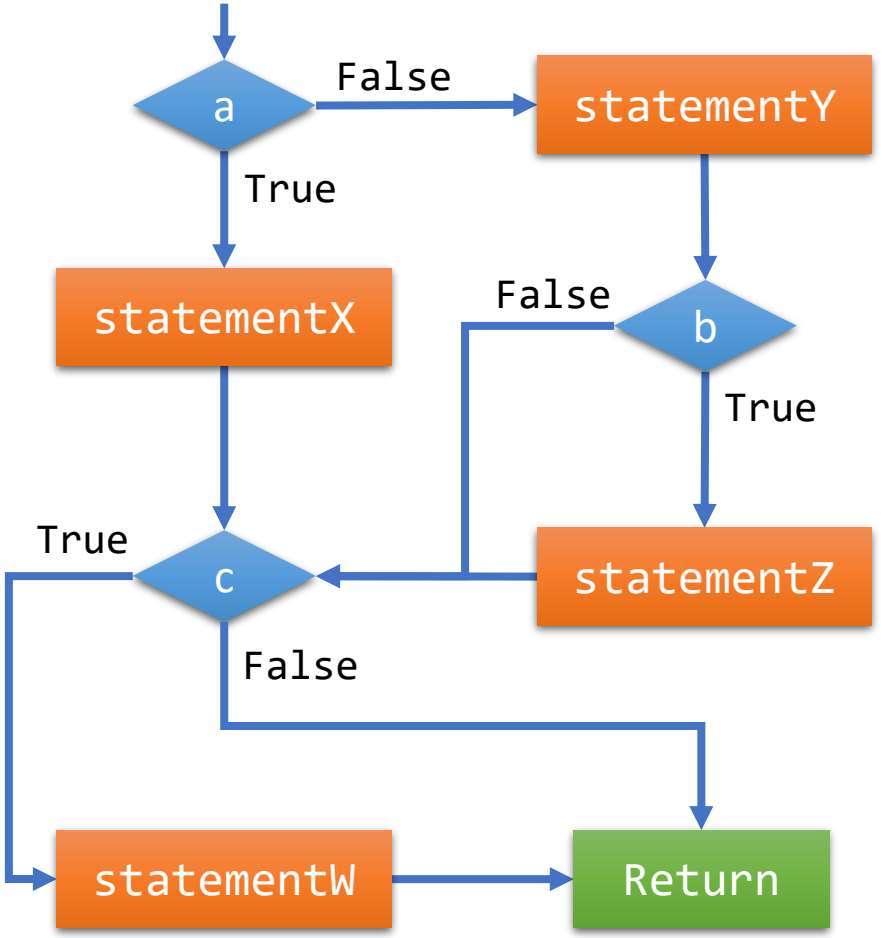
a	b	c
True	-	True
False	True	True
False	False	True
True	-	False
False	True	False
False	False	False



Path Complete Example

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

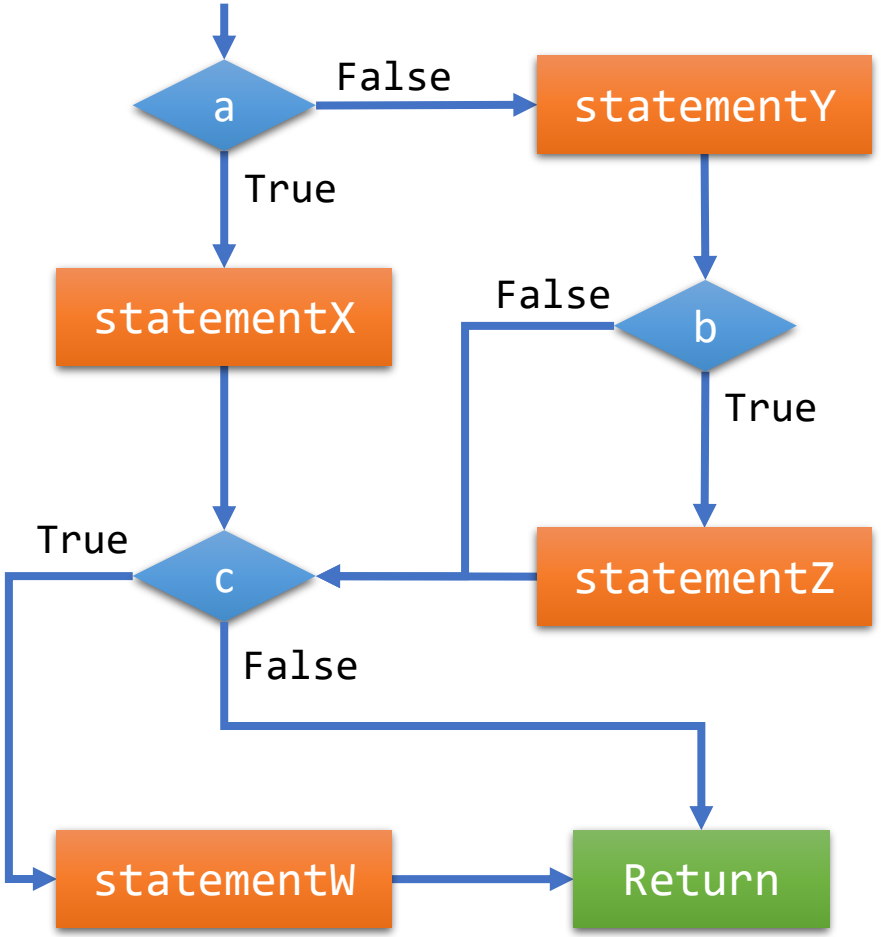
- 6 paths in flowchart
- What would be the **minimal set of test cases** to ensure path completeness?
  - 6 test cases. Why?



Path Complete Example

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

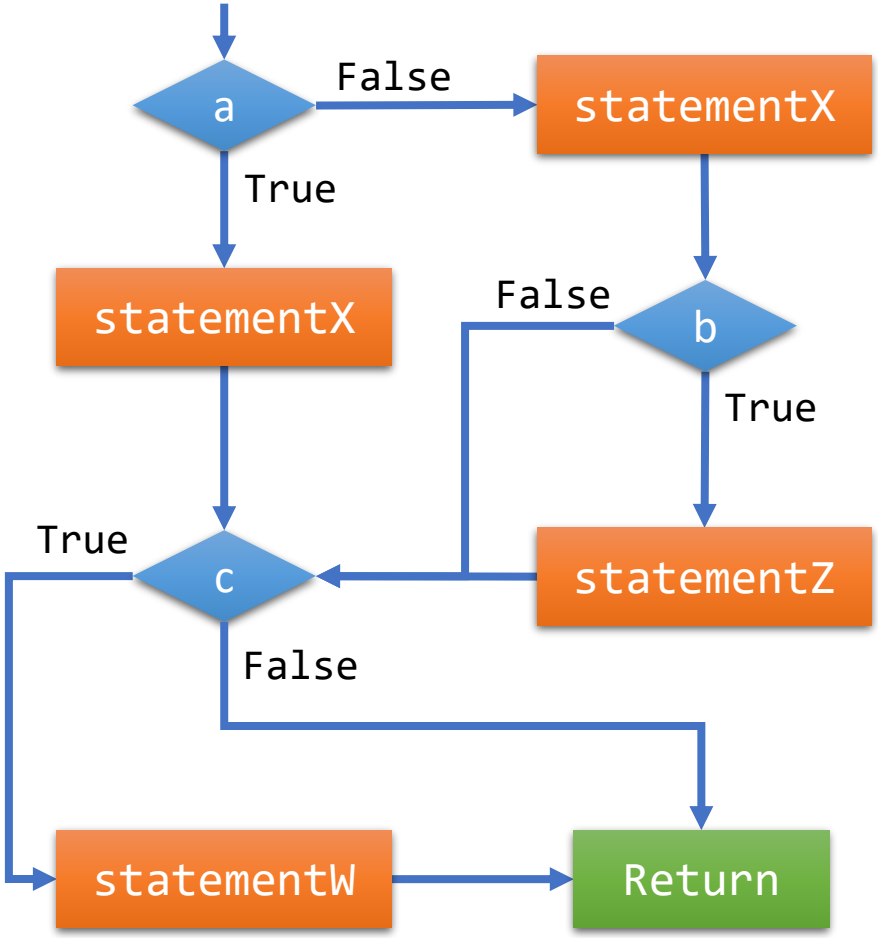
- Test Case 1: someMethod(true, true, true);
- Test Case 2: someMethod(false, true, true);
- Test Case 3: someMethod(false, false, false);
- Total Path-Coverage = 3/6 = 50%



Path Complete Example

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

a	b	c
True	-	True
False	True	True
False	False	True
True	-	False
False	True	False
False	False	False



- Test Case 1: someMethod(true, true, true);
- Test Case 2: someMethod(false, true, true);
- Test Case 3: someMethod(false, false, false);
- Test Case 4: someMethod(true, true, false);
- Test Case 5: someMethod(false, true, false);
- Test Case 6: someMethod(false, false, true);

Path Complete

- Usually require many more test cases to achieve 100% Path Coverage than it is needed to achieve 100% Branch Coverage
  - 100% Path Coverage implies 100% Branch Coverage and 100% Statement 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)
  - Cover all paths might not be feasible
    - For a medium size software, the number of paths it may have, and how to compute all of them (impractical and too expensive)

Exercise

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

- What would be the **minimal set of test cases** to ensure
  - Statement complete?
  - Branch complete?
  - Path complete?

Code Coverage in Eclipse

- Eclipse supports
  - Line Coverage
  - Branch Coverage
  - And more

```java
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	

Unit Testing

- 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
  - 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
  - Basic idea: Put certain assertion calls in your test methods to check things you expect to be true

JUnit Test Class

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

```
import org.junit.Test;

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


Simple Math Class Example

Class to be Tested

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

}
```

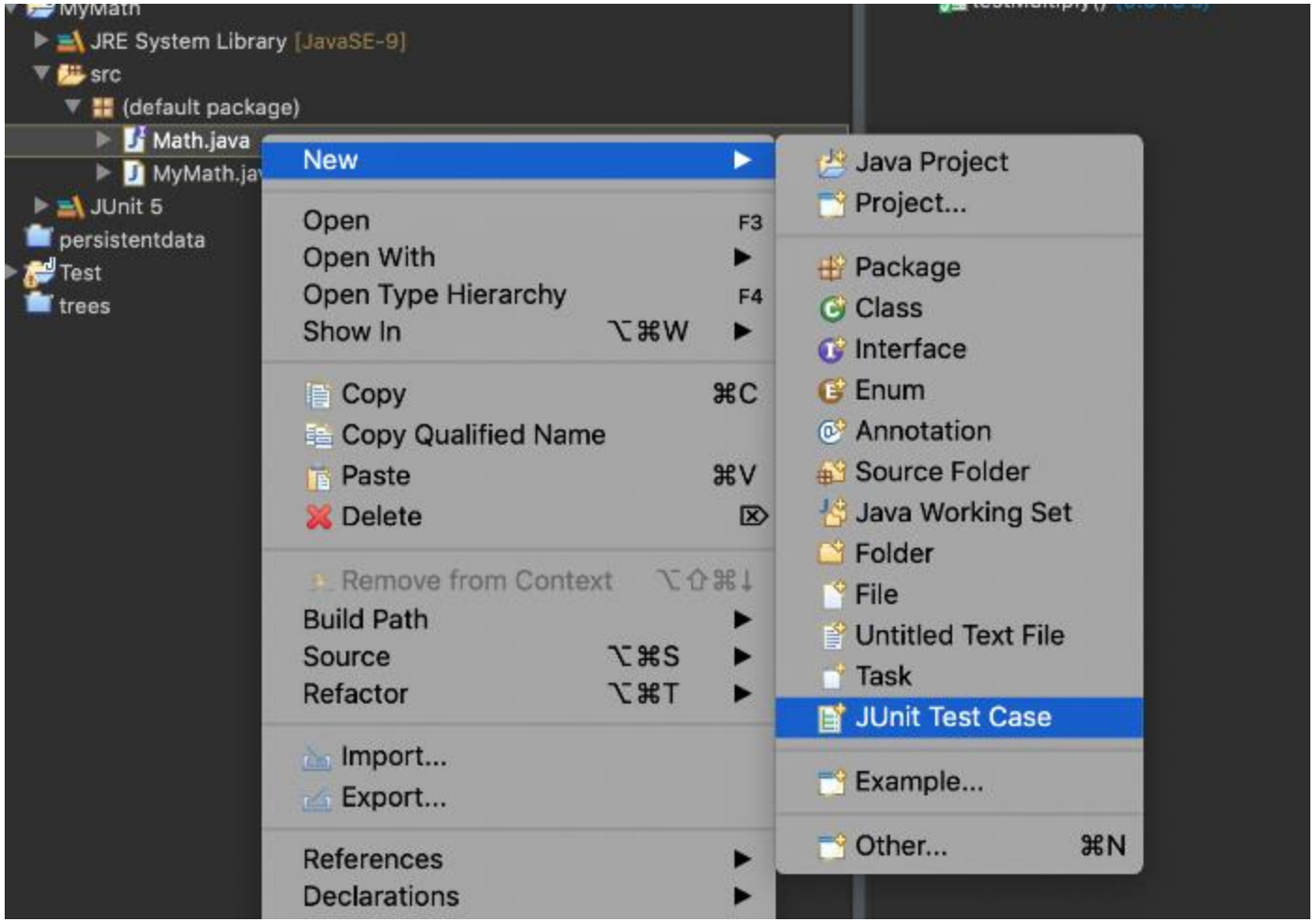
Test Case

```
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));
    }

}
```


Starting JUnit with Eclipse

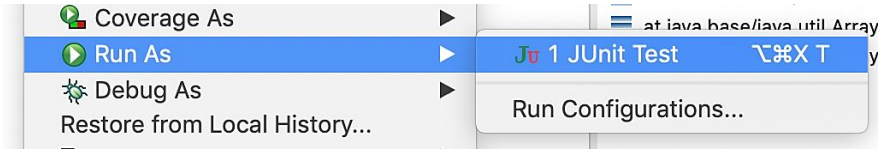
- JUnit does not come with JDK
  - Need to include JUnit into classpath
- To add JUnit to an Eclipse project, click on:
  - Project→Properties→Build Path→Libraries→Add Library→JUnit→JUnit 4→Finish
- To create a test case:
  - right-click a file and choose New→Test Case
  - or click on File→New→JUnit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

Create Test Case



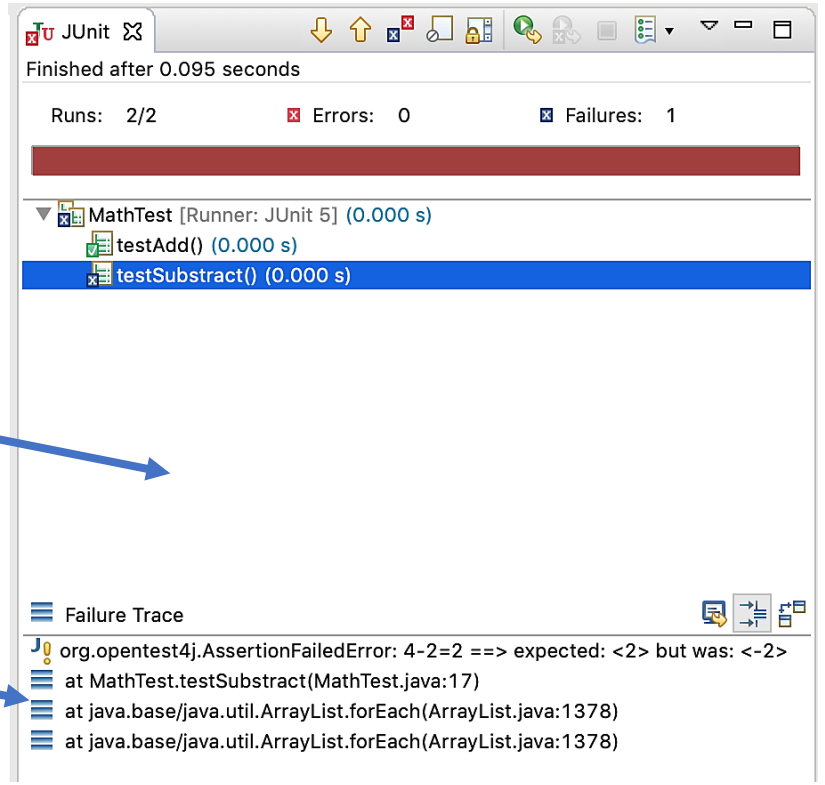
Running JUnit Test

- 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



JUnit Assertion

- Each method can also be passed a string to display if it fails:
  - e.g., `assertEquals(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

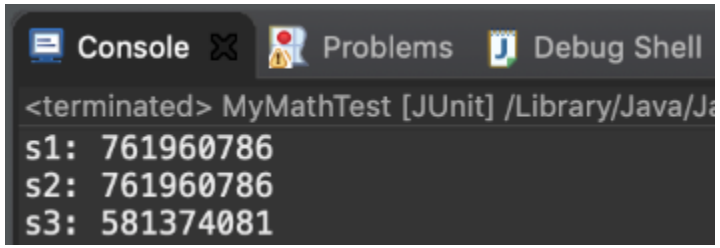
- `assertEquals(java.lang.Object expected, java.lang.Object actual)`
  - Assert that two objects are equal
- `assertSame(java.lang.Object expected, java.lang.Object actual)`
  - Assert that two objects refer to the same object
- Note that assertEquals uses the equals() method to assert that two objects 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
}
```

JUnit Assertion

- What is the result for the code below?
  - `assertTrue(s1.equals(s3));`

```java
@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
}
```



Floating Point Assertion

- What is 2/3?
  - 0.6666666666666666…
  - It cannot be represented in floating point expression.
  - Hence, we need to allow a small error
- Asserts that two doubles or floats are equal to within a positive delta
  - `assertEquals(x, y, delta)`
- Example
  - `assertEquals(0.3333333, 1.0/3.0, 0.0) // fail`
  - `assertEquals(0.3333333, 1.0/3.0, 0.0000001) // pass`

Resource Reuse/Release

- `@Before / @After`
  - Denotes that the annotated method should be executed before/after each `@Test`
  - Methods run before/after each test case method is called
- `@Before`
  - Initialization code: class initialization, array initialization, …
- `@After`
  - Cleanup code: freeing resources (set objects to null or call garbage collector), …

Example: @Before, @After

```
public 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
  - Methods to run once before/after the entire test class runs
  - Must be static method
  - e.g., Database connection
- `@BeforeClass`
  - Initialization code, establish a connection (database), …
- `@AfterClass`
  - Cleanup code: freeing resources, close a connection…

Example: @BeforeClass, @AfterClass

```
public 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
```


More Testing

- Testing for Exceptions
  - Use “`expected = xxxxxxxException.class`”

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

- Tests with Timeout
  - Use “`timeout = nnnn`”

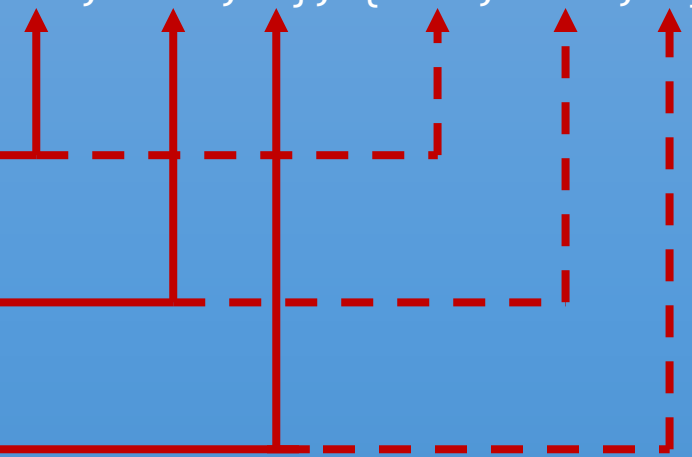
```
@Test(timeout = 1000)
public void timeoutNotExceeded() {
        assertEquals(5, math.add(2, 3));
}
```


Parameterized Test

- If you need to use the same inputs across different test methods
  - For example, methods 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

- Use one class that runs multiple JUnit test classes

```
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

}
```


Tips for Testing

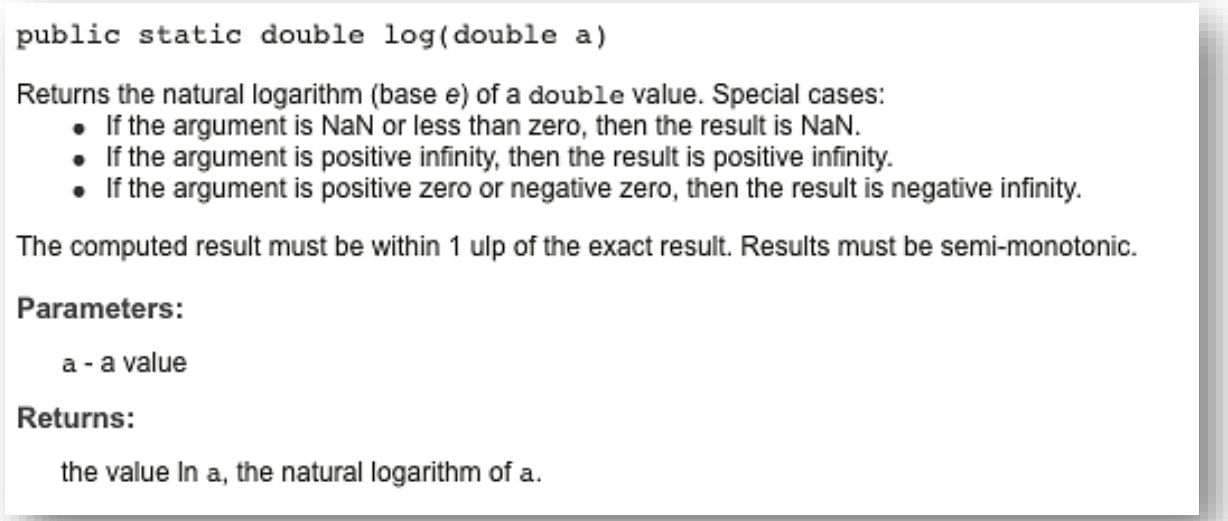
- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to find bugs
- For example, think about the boundary cases:
  - Positive; zero; negative numbers
  - Very large values
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- Test behavior 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

- Test-driven development:
  - Unit tests can be written after, during, or even before coding
  - Write tests, then write code to pass them
- Imagine that we would like to add a “log” method to 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)`
- Tips
  - 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

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

System Testing

- System testing checks that the entire system works within the environment that it will be placed in
- Check that all the hardware and software parts work with other external components in the context of normal loads
- Ideally, this would involve a replication of the hardware/software resources
  - However in some cases this is not feasible so system testing is performed "live"

Reference

- JUnit 4
  - https://junit.org/junit4/
- Useful website
  - Software Testing Fundamentals: http://softwaretestingfundamentals.com/
- Books:
  - “Software Engineering”, Ian Sommerville
  - “The Mythical Man-Month”, Fred Brooks


IF WE BUILD OUR SOFTWARE WITH NO BUGS, WE CAN MAKE A 10% RETURN ON OUR INVESTMENT.
10%
©
Dilbert.com DilbertCartoonist@gmail.com
BUT IF WE DO A POOR JOB, WE CAN MAKE A 40% RETURN BY SELLING UPGRADES AND SERVICE.
2-3-11 ©2011 Scott Adams, Inc./Dist. by UFS, Inc.
BUT DON'T WORRY. WE ONLY HAVE THE BUDGET FOR A POOR JOB.
I CAN'T REMEMBER IF WE'RE CHEAP OR SMART.
PHEW!

Acknowledgement

- Parts of the lecture slides are partially based on those from the previous instructors
  - Bernardo Nunes
  - Dongwoo Kim
  - Eric McCreath

- Some lecture contents may be based on public information on the Internet