

# SOFTWARE TESTING

## JUNIT, TDD, AND TEST COVERAGE

**Hakam Alomari**

Miami University Software Technology & Analysis Group (MUSTANG)  
Computer Science & Software Engineering  
Miami University, Oxford, Ohio, USA

# AGENDA

- Project's Due Dates
- Quiz 5 Review
- Principles of Testing
- Black-Box Techniques
  - Equivalence partitioning
  - Boundary value analysis
- White-Box Techniques
  - Code Coverage
- Unit testing

# PROJECT'S DUE DATES ?

- ~~April 2<sup>nd</sup> → 5<sup>th</sup> week deliverables~~
- ~~April 6<sup>th</sup> → Iteration 1 presentation → weeks 3 + 4 → 90 mints~~
- April 16<sup>th</sup> → 6<sup>th</sup> week deliverables
- April 20<sup>th</sup> → Iteration 2 presentation → weeks 5 + 6 → 90 mints
- April 23<sup>rd</sup> → 7<sup>th</sup> week deliverables
- April 30<sup>th</sup> → 8<sup>th</sup> week deliverables
- May 4<sup>th</sup> → Iteration 3 (final) presentation → weeks 7 + 8 → 90 mints

# QUIZ 5 REVIEW

# Q1

- What is the purpose of an automatic build system? In your description of its purpose, include at least one example of an automatic build system.

# Q1: SAMPLE SOLUTION

- Ant Build is an example of an automatic build system. It is used to ease the repetitive process of building complex pieces of software. It can be used for example to automate compilation and testing processes.

## Q2

- A fundamental principle of software construction is minimizing complexity. Why is it important to minimize complexity? How can software engineers minimize complexity?

## Q2: SAMPLE SOLUTION

- Future software engineers will have to maintain your code. If they do not understand it, they will likely rewrite it or break it.
- Don't write clever code. Write elegant and clear code. Use DRY and SRP.
  - DRY: Avoid duplicating code. Logical elements should be represented in only one place and then reused.
  - SRP: Every class should have one responsibility. Classes with more than one responsibility should be decoupled.



# REASONS NOT TO TEST (SARCASM)

- 5) I want to get this done fast – testing is going to slow me down.
- 4) I started programming when I was 2. Don't insult me by testing my perfect code!
- 3) Testing is for incompetent programmers who cannot hack.
- 2) We're not Harvard students – our code actually works!
- 1) “Most of the functions in Graph.java, as implemented, are one or two line functions that rely solely upon functions in HashMap or HashSet. I am assuming that these functions work perfectly, and thus there is really no need to test them.” – from a student's email

# THE FACTS

- 35MLOC
- 63K known bugs at the time of release
- 2 bugs per KLOC



- Only 32% of software projects are considered successful
  - (Full featured, on time, on budget)
- Software failures cost the US economy \$59.5 billion dollars every year [[NIST 2002 Report](#)]
  - 1 - 10 defects/KLOC: typical industry software.
  - 0.1 - 1 defects/KLOC: high-quality validation.
    - The Java libraries might achieve this level of correctness.
  - 0.01 - 0.1 defects/KLOC: the very best, safety-critical validation.
    - NASA for example can achieve this level.

# VALIDATION

- Testing is an example of a more general process called *validation*. The purpose of validation is to uncover problems in a program and thereby increase your confidence in the program's correctness.
- Validation includes:
  - **Formal reasoning** about a program, usually called *verification*.
  - **Code review**. Having somebody else carefully read your code, and reason informally about it, can be a good way to uncover bugs.
  - **Testing**. Running the program on carefully selected inputs and checking the results.

# PRINCIPLES OF TESTING

- **Be systematic**

- Haphazard testing is less likely to find bugs
- Exhaustive testing is generally impossible

- **Test early and often**

- Test-first programming: write the tests before writing the code

- **Automate testing**

- A test driver should not be an interactive program that prompts for inputs, etc.
- JUnit – a test driver that automates testing

# WHY IS TESTING SOFTWARE DIFFICULT?

- We want to
  - Know when product is stable enough to launch
  - Deliver product with known failure rate (preferably low)
- However
  - It's very hard to measure or ensure quality in software
    - Residual defect rate after shipping:
      - 1 - 10 defects/kloc (typical)
      - 0.1 - 1 defect kloc (high quality: Java libraries?)
      - 0.01 - 0.1 defects/kloc (very best: Praxis, NASA)
- Exhaustive testing is infeasible
  - Space is generally too big to cover exhaustively
    - Imagine exhaustively testing a 32-bit floating-point multiply operation ( $a * b$ )
      - There are  $2^{64}$  test cases

# CONCEPTS THAT ARE OFTEN CONFUSED

- Problem of finding bugs in defective code
- Problem of showing the absence of bugs in good code
- Approaches:
  - **Testing** – helpful for finding bugs
    - “Program testing can show the presence of bugs, but not their absence” **Dijkstra**
  - **Reasoning** – can be helpful for both finding bugs and showing the absence of bugs

# WHAT IS A GOOD TEST CASE?

- One that give correct answers?
  - **NO!**
- A good test case is one that finds an error!
  - Think about the psychology of testing.
    - Test cases are developed to exercise your program with the goal of uncovering errors.
    - As a tester, your goal is to **make it fail**.
  - How many faults does it find?
  - Coverage

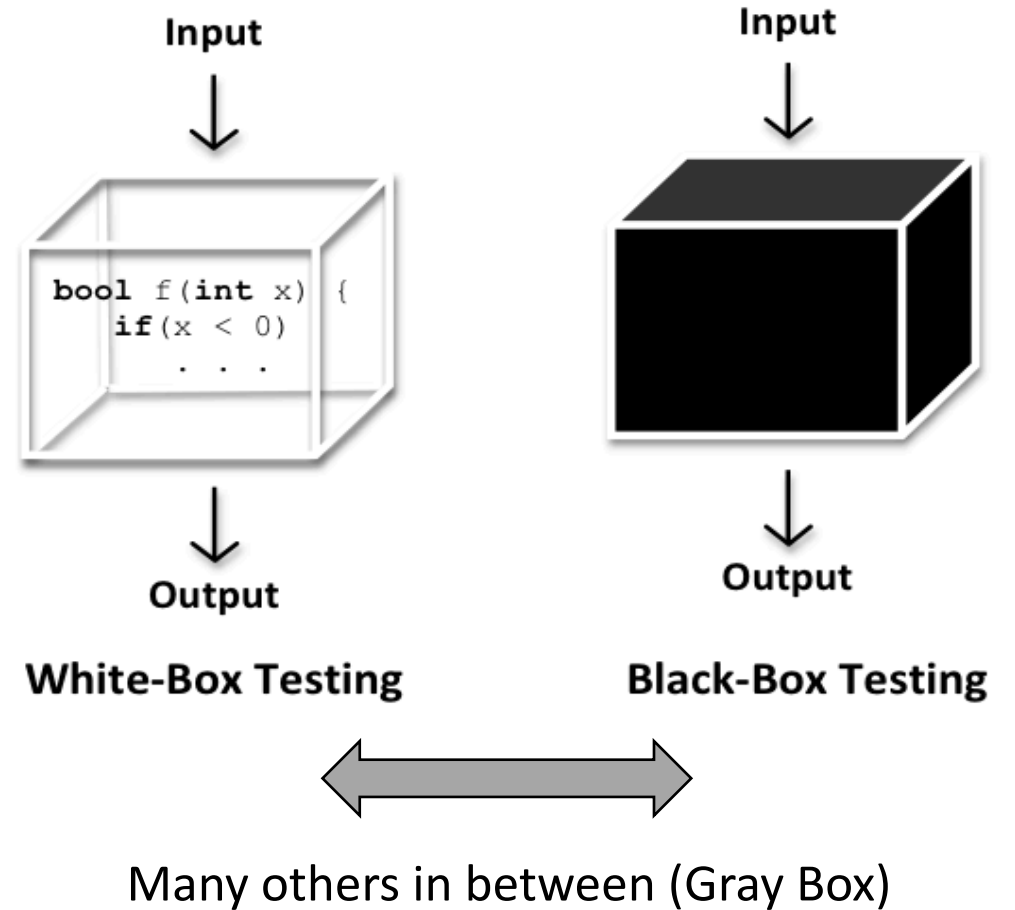
# INFORMATION NEEDED

- A method/function is defined by an input/output specifications
  - I/O spec.
  - The *pre-* and *post-conditions* describe the I/O spec
  - Test to specifications
- A method/function is also defined by its implementation details,
  - For-loop vs while loop vs recursive
  - Test to code



# TESTING STRATEGIES

- Two very broad testing strategies are:
  - White-Box
    - Test cases are derived from knowledge of the implementation.
    - Tests are evaluated in terms of coverage of the source code
  - Black-Box
    - Test cases are derived from external software specifications.
    - Input/output driven
    - Source code is ignored



# WHITE & BLACK BOX TESTING ARE COMPLEMENTARY

- What's the danger (types of defects missed) if you do only white box testing?
  - You risk testing what was written not what was suppose to be written.
- What's the danger (types of defects missed) if you do only black box testing?
  - If you don't look at the code, many control flow paths are likely to go untested.

# BLACK-BOX TECHNIQUES FOR SELECTING TEST CASES

## ■ Equivalence partitioning

- Tests are divided into groups according to the criteria that two test cases are in the same group if both test cases are likely to find the same error.
- Classes can be formed based on inputs or outputs.

## ■ Boundary value analysis

- Create test cases with values that are on the edge of equivalence partitions

# BLACK-BOX TECHNIQUES FOR WRITING TEST CASES

- **Test-Driven Development (TDD)**
  - Write tests before you write the code.
  - Why would you do this?
    - Code will not bias the tests
    - Help assure that we understand the specifications
  - Steps:
    1. Write a specification for the function.
    2. Write tests that exercise the specification.
    3. Write the actual code.
    4. Run the code against the test cases.
    5. Fix any faults (debugging and implementation).
    6. Go to 4 (or if serious problems 1).
    7. Once your code passes the tests you wrote, you're done.

# SYSTEMATIC TESTING: EQUIVALENCE CLASSES

- Partition the input space
  - Identify sets of inputs with the same behavior
  - Try on input from each block
- **Goals:** for each partition (block), all elements from that block will behave the same way as every other element with regard to whether it reveals a fault or not.

# EXAMPLE: BigInteger.Multiply()

- Let's look at an example. [BigInteger](#) is a class built into the Java library that can represent integers of any size, unlike the primitive types **int** and **long** that have only limited ranges.
- BigInteger has a method multiply that multiplies two BigInteger values together:

```
/**  
 * @param val  another BigInteger  
 * @return a BigInteger whose value is (this * val).  
 */  
public BigInteger multiply(BigInteger val)
```

- For example, here's how it might be used:

```
BigInteger a = ...;  
BigInteger b = ...;  
BigInteger ab = a.multiply(b);
```

# EXAMPLE: `BIGINTEGER.MULTIPLY()`

- **`multiply : BigInteger × BigInteger → BigInteger`**
- We have a two-dimensional input space, consisting of all the pairs of integers (a, b).
- Now let's partition it. Thinking about how multiplication works, we might start with these partitions:
  - a and b are both positive
  - a and b are both negative
  - a is positive, b is negative
  - a is negative, b is positive
- There are also some special cases for multiplication that we should check: 0, 1, and -1.
  - a or b is 0, 1, or -1

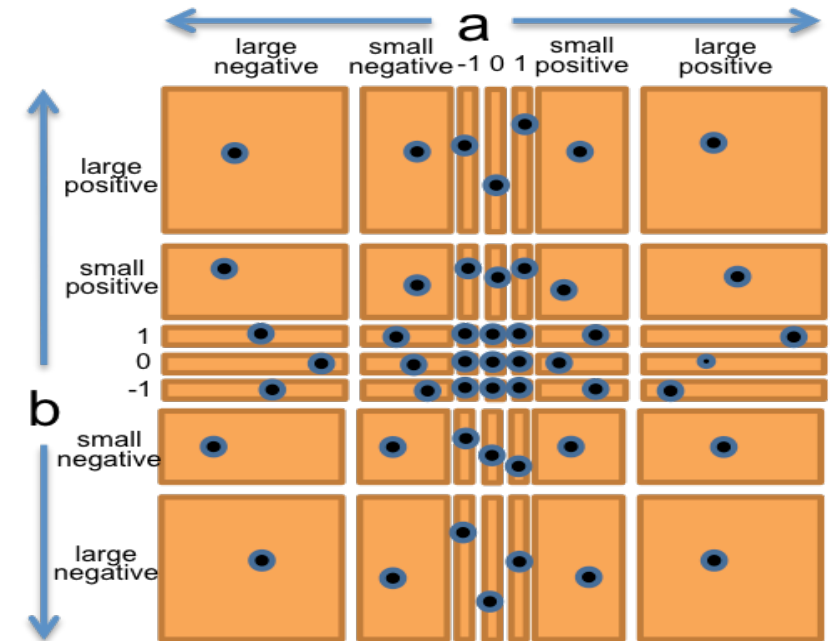
# EXAMPLE: `BIGINTEGER.MULTIPLY()`

- **`multiply : BigInteger × BigInteger → BigInteger`**
- Finally, as a suspicious tester trying to find bugs, we might suspect that the implementor of `BigInteger` might try to make it faster by using `int` or `long` internally when possible, and only fall back to an expensive general representation (like a list of digits) when the value is too big.
- So we should definitely also try integers that are very big, bigger than the biggest long.
  - a or b is small
  - the absolute value of a or b is bigger than `Long.MAX_VALUE`, the biggest possible primitive integer in Java, which is roughly  $2^{63}$ .



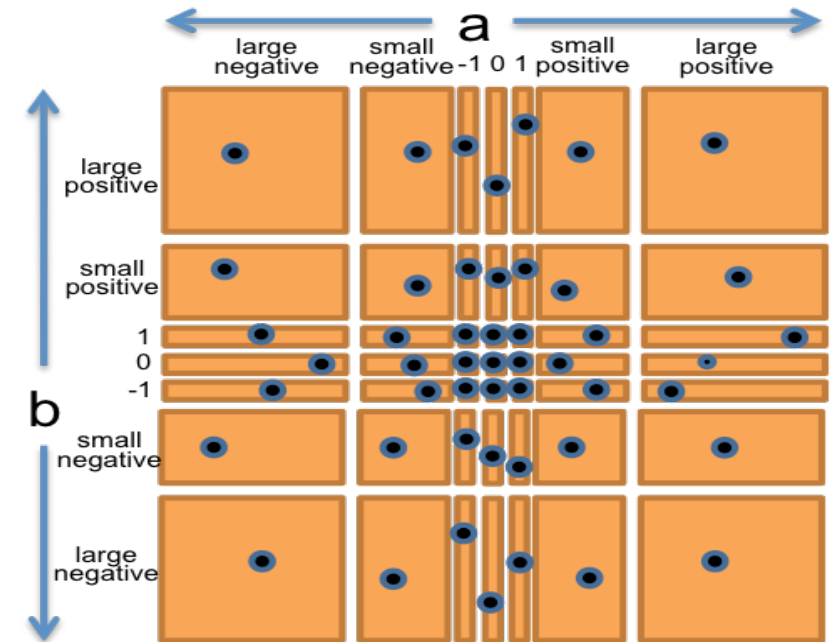
# EXAMPLE: BigInteger.Multiply()

- **multiply : BigInteger × BigInteger → BigInteger**
- Let's bring all these observations together into a straightforward partition of the whole (a, b) space. We'll choose a and b independently from:
  - 0
  - 1
  - -1
  - small positive integer
  - small negative integer
  - huge positive integer
  - huge negative integer
- So this will produce  $7 \times 7 = 49$  partitions that completely cover the space of pairs of integers.



# EXAMPLE: BigInteger.Multiply()

- To produce the test suite, we would pick an arbitrary pair (a,b) from each square of the grid, for example:
  - (a,b) = (-3, 25) to cover (small negative, small positive)
  - (a,b) = (0, 30) to cover (0, small positive)
  - (a,b) = ( $2^{100}$ , 1) to cover (large positive, 1)
  - etc.



# INPUT PARTITION EXAMPLES

- **intersect : Set  $\times$  Set  $\rightarrow$  Set**
- Partition Set into:
  - $\emptyset$ , singleton, many
- Partition whole input space into
  - this = that, this  $\subseteq$  that, that  $\subseteq$  this, this  $\cap$  that  $\neq \emptyset$ , this  $\cap$  that =  $\emptyset$
- Pick values that cover both partitions
  - $\{\}, \{\}$        $\{\}, \{2\}$        $\{\}, \{2,3,4\}$
  - $\{5\}, \{\}$        $\{5\}, \{2\}$        $\{4\}, \{2,3,4\}$
  - $\{2,3\}, \{\}$        $\{2,3\}, \{2\}$        $\{1,2\}, \{2,3\}$

# INPUT PARTITION EXAMPLES

- Consider an example from the Java library: the integer max() function, found in the Math class.

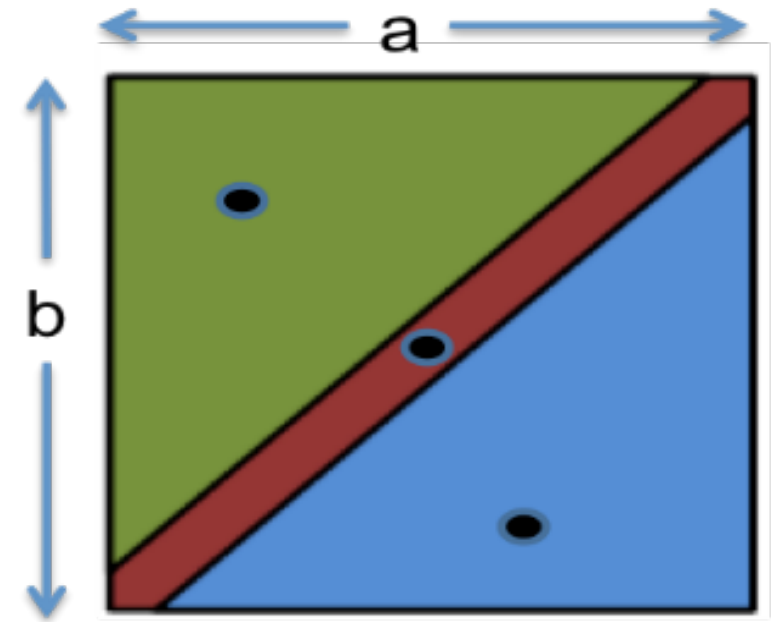
```
/**  
 * @param a  an argument  
 * @param b  another argument  
 * @return the larger of a and b.  
 */  
public static int max(int a, int b)
```

- Mathematically, this method is a function of the following type:

$\text{max} : \text{int} \times \text{int} \rightarrow \text{int}$

# INPUT PARTITION EXAMPLES

- **max : int  $\times$  int  $\rightarrow$  int**
- Partition into:
  - $a < b$ ,  $a = b$ ,  $a > b$
- Pick value from each class
  - (1, 2), (1, 1), (2, 1)
- Our test suite might then be:
  - $(a, b) = (1, 2)$  to cover  $a < b$
  - $(a, b) = (9, 9)$  to cover  $a = b$
  - $(a, b) = (-5, -6)$  to cover  $a > b$



# ANOTHER METHOD: BOUNDARY VALUE TESTING

- Include classes at **boundaries** of the input space
  - 0 is a boundary between positive numbers and negative numbers
  - The maximum and minimum values of numeric types, like **int** and **double**
  - Emptiness (the empty string, empty list, empty array) for collection types
  - The first and last element of a collection
- Why? because bugs often occur at boundaries
  - off-by-one errors
    - like writing `<=` instead of `<`, or initializing a counter to **0** instead of **1**
  - Some boundaries may need to be handled as special cases in the code.
  - Another is that boundaries may be places of discontinuity in the code's behavior.
    - When an **int** variable grows beyond its maximum positive value, for example, it abruptly becomes a negative number

# LET'S REDO $\text{MAX} : \text{INT} \times \text{INT} \rightarrow \text{INT}$ .

- Partition into:

- relationship between a and b

- $a < b$
    - $a = b$
    - $a > b$

- values of a

- $a = 0$
  - $a < 0$
  - $a > 0$
  - a = minimum integer
  - a = maximum integer

- ▶ values of b

- ▶  $b = 0$
  - ▶  $b < 0$
  - ▶  $b > 0$
  - ▶ b = minimum integer
  - ▶ b = maximum integer

- ▶ Now let's pick test values that cover all these classes:

- ▶ (1, 2) covers  $a < b$ ,  $a > 0$ ,  $b > 0$
  - ▶ (-1, -3) covers  $a > b$ ,  $a < 0$ ,  $b < 0$
  - ▶ (0, 0) covers  $a = b$ ,  $a = 0$ ,  $b = 0$
  - ▶ (Integer.MIN\_VALUE, Integer.MAX\_VALUE) covers  $a < b$ ,  $a = \text{minint}$ ,  $b = \text{maxint}$
  - ▶ (Integer.MAX\_VALUE, Integer.MIN\_VALUE) covers  $a > b$ ,  $a = \text{maxint}$ ,  $b = \text{minint}$

# HOW EXHAUSTIVE?

- After partitioning the input space, we can choose how exhaustive we want the test suite to be:
- Full Cartesian Product.
  - Every legal combination of the partition dimensions is covered by one test case.
    - This is what we did for the **multiply** example, and it gave us  $7 \times 7 = 49$  test cases.
  - However, in practice not all of these combinations are possible.
    - For example, there's no way to cover the combination  $a < b$ ,  $a = 0$ ,  $b = 0$ , in the **max** example because **a** can't be simultaneously less than zero and equal to zero.
- Cover Each Part.
  - Every part of each dimension is covered by at least one test case, but not necessarily every combination.

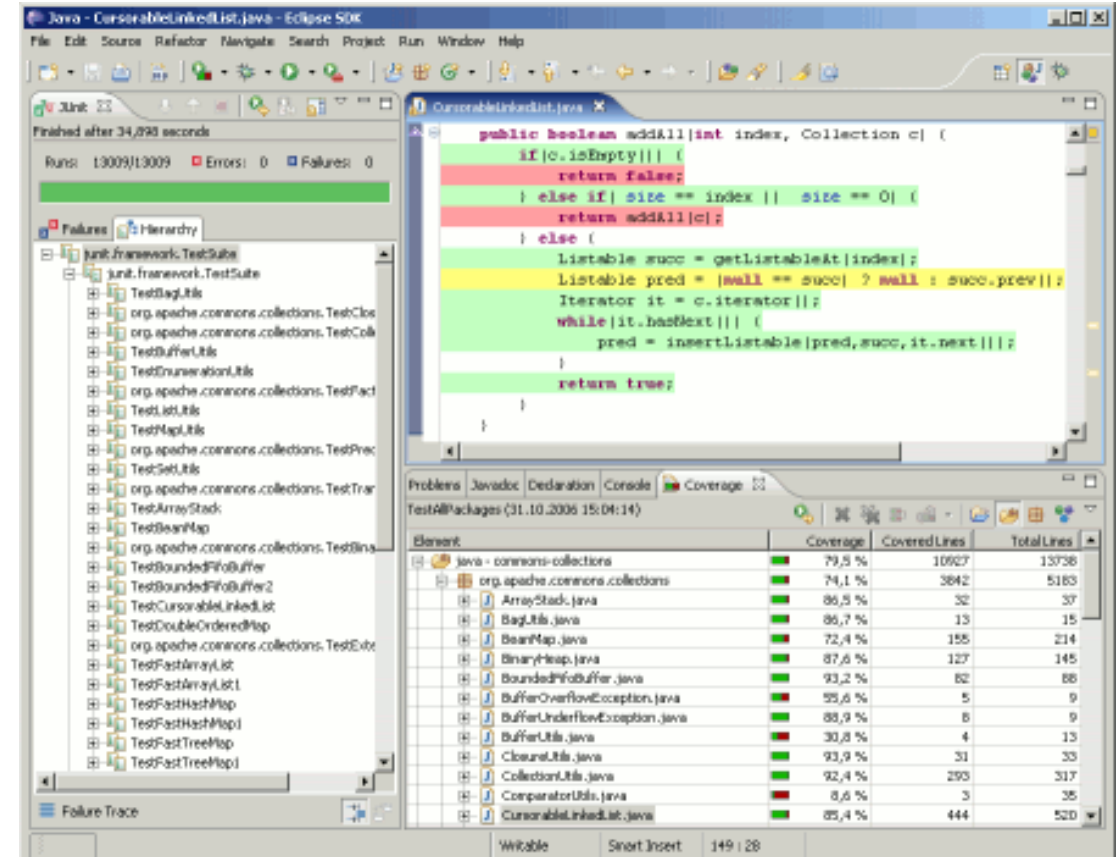


# LEVELS OF WHITE-BOX CODE COVERAGE

- A measurement to evaluate the percentage of code tested
- Options are:
  - **Statement coverage:**
    - Has each statement in the program been executed?
  - **Decision coverage** (aka branch coverage)
    - Has each branch of each control structure (such as in if and case statements) been executed?  
Another way of saying this is, has every edge in the CFG been executed
  - **Condition coverage** (or predicate coverage)
    - Has each Boolean sub-expression evaluated both to true and false?
  - **Path coverage**
    - Is every possible combination of branches – every path through the program – taken by some test case?

# ECLEmma IS A FREE JAVA CODE COVERAGE TOOL FOR ECLIPSE

<http://www.eclemma.org>



The screenshot shows the Eclipse IDE with the following components:

- JUnit Runner:** Shows a successful run of 13009 tests in 34.893 seconds with 0 errors and 0 failures.
- Package Explorer:** Displays a hierarchy of test packages under 'org.apache.commons.collections', including 'TestCursorableLinkedList'.
- Code Editor:** Displays the source code for 'CursorableLinkedList.java', showing a method 'addAll' with conditional logic and list manipulation.
- Coverage View:** A table showing coverage data for various classes in the 'org.apache.commons.collections' package.
 

Element	Coverage	Covered Lines	Total Lines
org.apache.commons.collections	79,5 %	10927	13738
org.apache.commons.collections	74,1 %	3842	5183
ArrayStack.java	86,5 %	32	37
BagUtils.java	86,7 %	13	15
BeanMap.java	72,4 %	155	214
BinaryHeap.java	87,6 %	127	145
BoundedFifoBuffer.java	93,2 %	62	66
BufferOverflowException.java	55,6 %	5	9
BufferUnderflowException.java	88,9 %	8	9
BufferUtils.java	30,8 %	4	13
CloneUtils.java	93,9 %	31	33
CollectionUtils.java	92,4 %	293	317
ComparatorUtils.java	8,6 %	3	35
CursorableLinkedList.java	85,4 %	444	520

# INDUSTRY PRACTICE

- All-statements is common goal, rarely achieved (due to unreachable code)
- All branches if possible:
  - Safety critical industry has more arduous criteria (e.g., “MCDC”, modified condition/decision coverage)
- All-paths is infeasible

# STATEMENT COVERAGE

- Each line of code is executed.

if (a)

stmt1;

if (b)

stmt2;

- $a = t; b = t$  gives statement coverage
- $a = t; b = f$  doesn't give statement coverage

# DECISION COVERAGE

- Decision coverage is also known as branch coverage
- The Boolean condition at every branch point (if, while, etc.) has been evaluated to both T and F.

if (a and b)

stmt1;

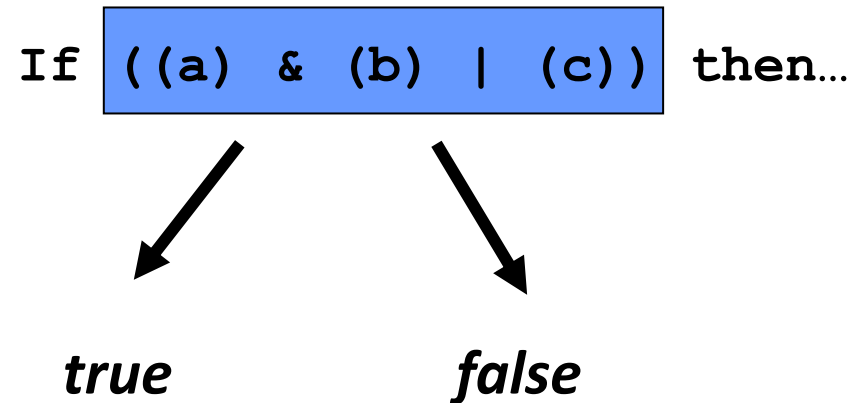
if (c)

stmt2;

- a = t; b = t; c = t and a = f; b = ?; c = f gives decision coverage

# EXAMPLE

- The decision has taken all possible outcomes at least once.



- We could also say we cover both the true and the false branch (Branch Testing).

# DECISION VS STATEMENT

- Does Decision Coverage Guarantee Statement Coverage?
  - Yes assuming the program has at least one decision, there is only one entry point, and you ignore exception handling code.
- Does Statement Coverage Guarantee Decision Coverage?
  - if (a)  
    stmt1;
  - If no, give an example of input that gives statement coverage but not decision coverage.
  - No.  $a = t$ . If this is the only test case, you don't have decision coverage because you haven't tested the decision where the if statement evaluates to false.

# CONDITION COVERAGE

- Each Boolean sub-expression at a branch point has been evaluated to true and false.

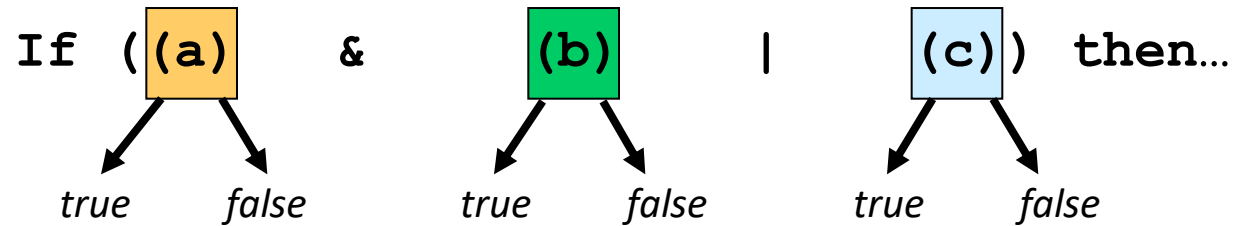
```
if (a and b)  
    stmt1;
```

- $a = t, b = t$  and  $a = f; b = f$  gives condition coverage



# EXAMPLE

- Every condition in the decision has taken all possible outcomes at least once.



# DECISION VS CONDITION

- Does condition coverage guarantee decision coverage?

```
if (a and b)  
    stmt1;
```

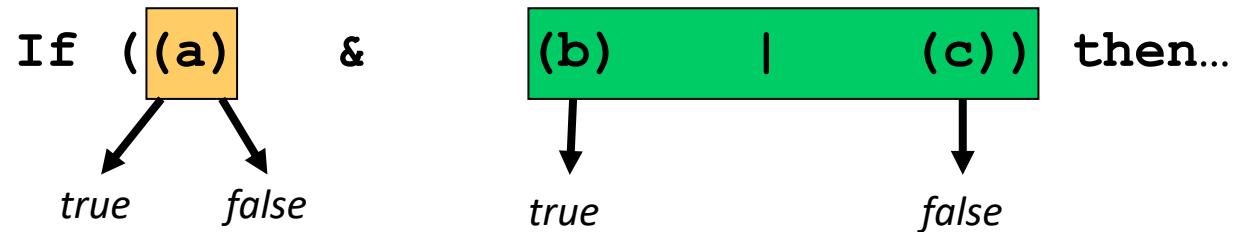
- If no, give example input that gives condition coverage but not decision coverage.
- No.  $a = t, b = f$  and  $a = f, b = t$ . This gives condition coverage but doesn't test the case where the branch is taken.

# CONDITION/DECISION COVERAGE

- Consider the following code:
  - if (**a** or **b**) and **c** then
- The condition/decision criteria will be satisfied by the following set of tests:
  - a=true, b=true, c=true
  - a=false, b=false, c=false

# MODIFIED CONDITION/DECISION COVERAGE (MC/DC)

- Every condition in the decision independently affects the decision's outcome.



Change the value of each condition individually while keeping all other conditions constant.

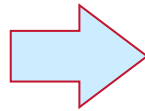
- This criterion extends condition/decision criteria with requirements that each condition should affect the decision outcome independently.

# MODIFIED CONDITION/DECISION COVERAGE (MC/DC)

**If (A and B) then...**

- (1) Create truth table for conditions.
- (2) Extend truth table so that it indicated which test cases can be used to show the independence of each condition.

A B	result
T T	T
T F	F
F T	F
F F	F



number	A B	result	A	B
1	T T	T	3	2
2	T F	F		1
3	F T	F	1	
4	F F	F		

# CREATING TEST CASES CONT'D

number	A B	result	A	B
1	T T	T	3	2
2	T F	F		1
3	F T	F	1	
4	F F	F		

- Show independence of **A**:
  - Take 1 + 3
- Show independence of **B**:
  - Take 1 + 2
- Resulting test cases are
  - 1 + 2 + 3
  - (T , T) + (T , F) + (F , T)

- (T,T) test is required as it is the only one that returns true
- (F,T) test is required as it is the only test that changes the value of only *A* and also changes the decision's outcome, thereby establishing the independence of *A* In similar fashion
- (T,T) and (T,F) tests are required to show the independence of *B*.
- Test set {(T,T), (T,F), (F,T)} satisfies MC/DC for the expression *A* and *B*.

# MORE ADVANCED EXAMPLE

**If (A and (B or C)) then...**

number	ABC	result	A	B	C
1	TTT	T	5		
2	TTF	T	6	4	
3	TFT	T	7		4
4	TFF	F		2	3
5	FTT	F	1		
6	FTF	F	2		
7	FFT	F	3		
8	FFF	F			

$T_a = \{(1, 5) (2, 6) (3, 7)\}$

$T_b = \{(2, 4)\}$

$T_c = \{(4, 3)\}$

T T F

T F T

T F F

One of {6 or 7} to cover A

# ANOTHER MC/DC EXAMPLE

- Consider the following code:
  - if (a or b) and c then
- The condition/decision criteria will be satisfied by the following set of tests:
  - a=true, b=true, c=true
  - a=false, b=false, c=false
- However, the above tests set will not satisfy modified condition/decision coverage, since in the first test, the value of 'b' and in the second test the value of 'c' would not influence the output.
- So, the following test set is needed to satisfy MC/DC:

■ a = false, b = false, c = true	F
■ a = true, b = false, c = true	T
■ a = false, b = true, c = true	T
■ a = false, b = true, c = false	F



# PATH COVERAGE

- In order to achieve path coverage you need a set of test cases that executes every possible route through a unit of code.
  - Sometimes, path coverage is impractical for all
- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once.
  - The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control

# PATH COVERAGE

- How many paths are there in the following unit of code?

if (a)

    stmt1;

if (b)

    stmt2;

if (c)

    stmt3;

Answer:  $2^3$  or 8

a=f,b=f,c=f

a=f,b=f,c=t

a=f,b=t,c=f

a=f,b=t,c=t

a=t,b=f,c=f

a=t,b=f,c=t

a=t,b=t,c=f

a=t,b=t,c=t

- If there is a loop in your flow chart and you can't "unroll" the loop (calculate an upper bound on the number of times through the loop) its impossible to achieve path coverage.

# PATH COVERAGE

- What inputs (test cases) are needed to achieve path coverage on the following code fragment?

```
procedure AddTwoNumbers()
```

```
    top: print "Enter two numbers";
```

```
        read a;
```

```
        read b;
```

```
        print a+b;
```

```
        if (a != -1) goto top;
```

- There is no finite number of test cases that will achieve path coverage.

# CONTROL FLOW TESTING

- Control flow testing uses the **control structure** of a program to develop the test cases for the program.
- The test cases are developed to sufficiently **cover** the whole control structure of the program.
- The control structure of a program can be represented by the **control flow graph** of the program.
  - A **test case** is a **complete path** from the entry node to the exit node of a control flow graph.

# CONTROL FLOW GRAPH

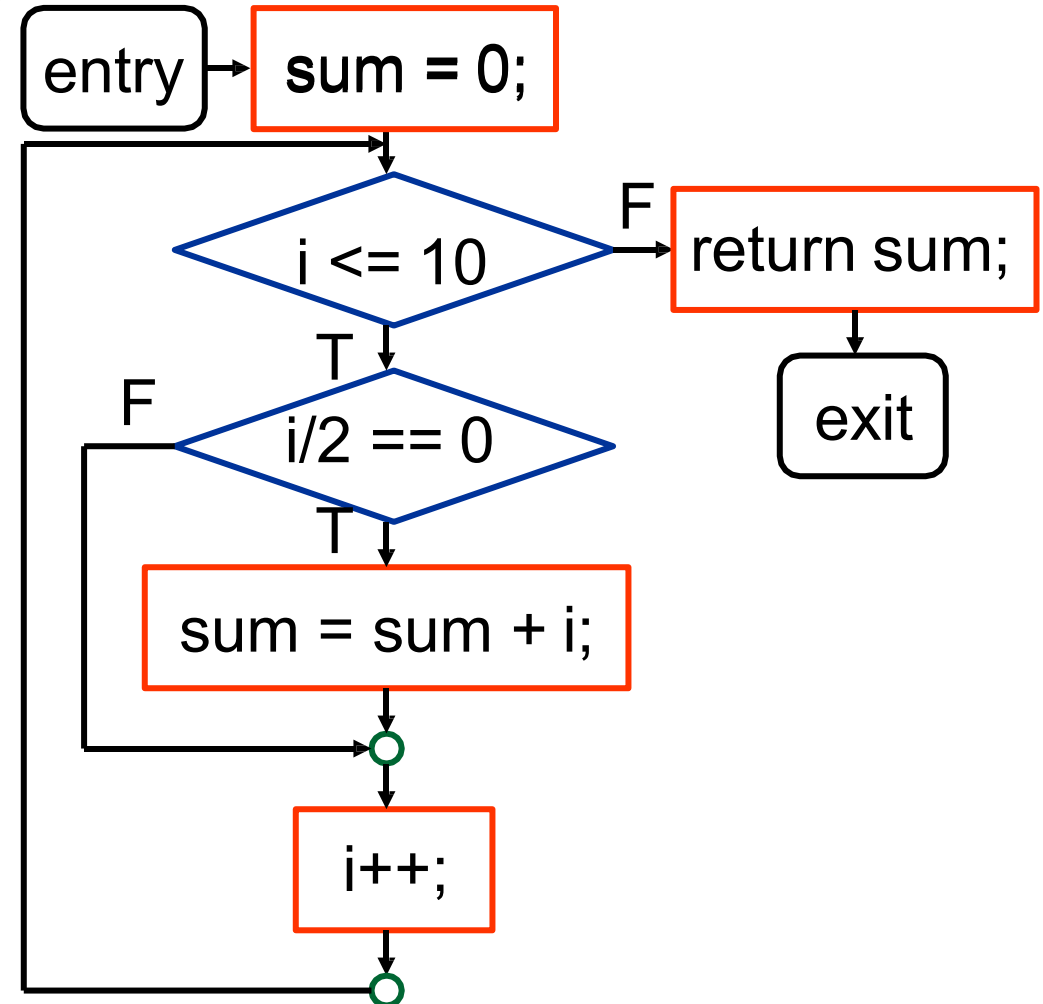
- The control flow graph  $G = (N, E)$  of a program consists of a set of **nodes**  $N$  and a set of **edges**  $E$ .
  - Each **node** represents a set of program statements. There are **five** types of nodes.
  - There is an **edge** from node  $n_1$  to node  $n_2$  if the control may flow from the last statement in  $n_1$  to the first statement in  $n_2$ .

# CONTROL FLOW GRAPH: NODES

- There is a unique **entry** node and a unique **exit** node.
- A **decision** node contains a conditional statement that creates 2 or more control branches (e.g. if or switch statements).
- A **merge** node usually does not contain any statement and is used to represent a program point where multiple control branches merge.
- A **statement** node contains a sequence of statements. The control must **enter** from the **first** statement and **exit** from the **last** statement.

# CONTROL FLOW GRAPH: AN EXAMPLE

```
int evensum(int i) {  
    int sum = 0;  
    while (i <= 10) {  
        if (i/2 == 0)  
            sum = sum + i;  
        i++;  
    }  
    return sum;  
}
```



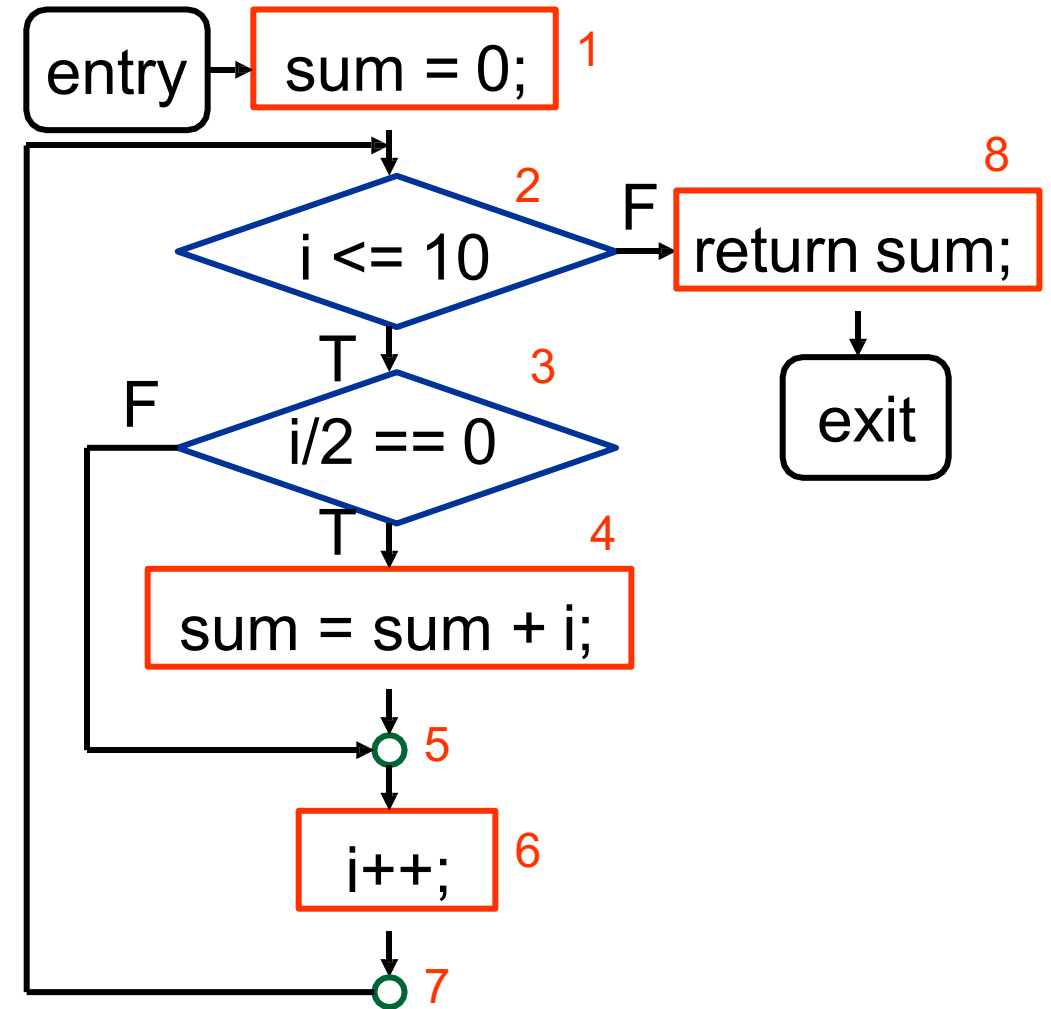
# COVERAGES

## ► Statement Coverage

- Every statement in the program has been executed at least once.
- 1 → 2 → 3 → 4 → 5 → 6 → 7 → 2 → 8

## ► Decision Coverage

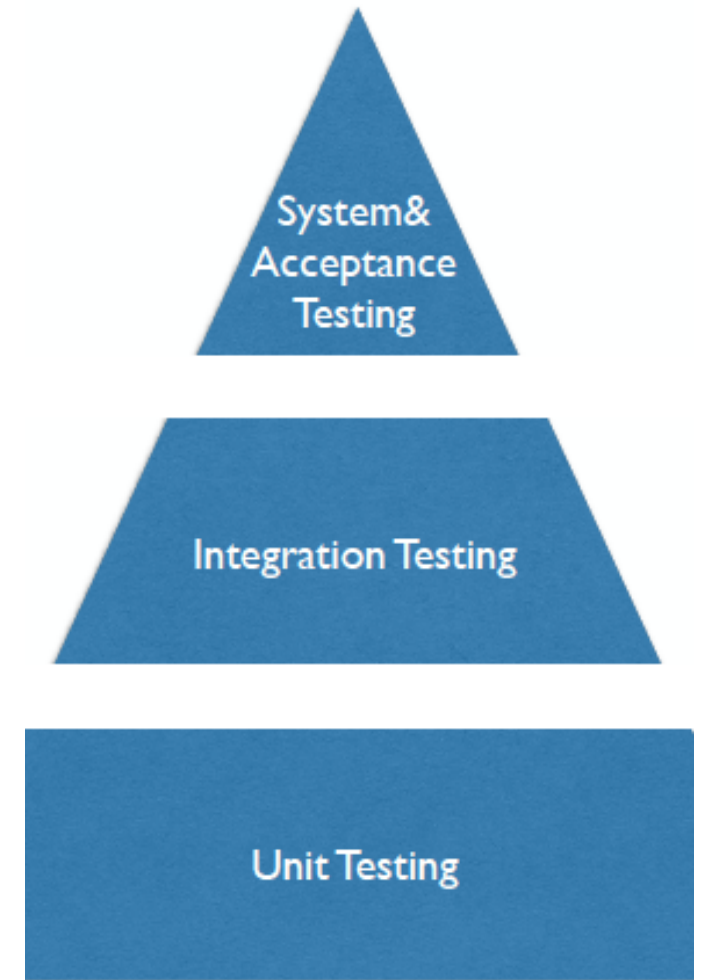
- Every statement in the program has been executed at least once, and every decision in the program has taken all possible outcomes at least once.
- 1 → 2 → 3 → 5 → 6 → 7 → 2 → 3 → 4 → 5 → 6 → 7 → 2 → 8





# GRANULARITY OF TESTING

- **Unit testing**
  - Test for each single module
- **Integration testing**
  - Test the interaction between modules
- **System testing**
  - Test the system as a whole, by developers
- **Acceptance testing**
  - Validate the system against user requirements by customers with formal test cases



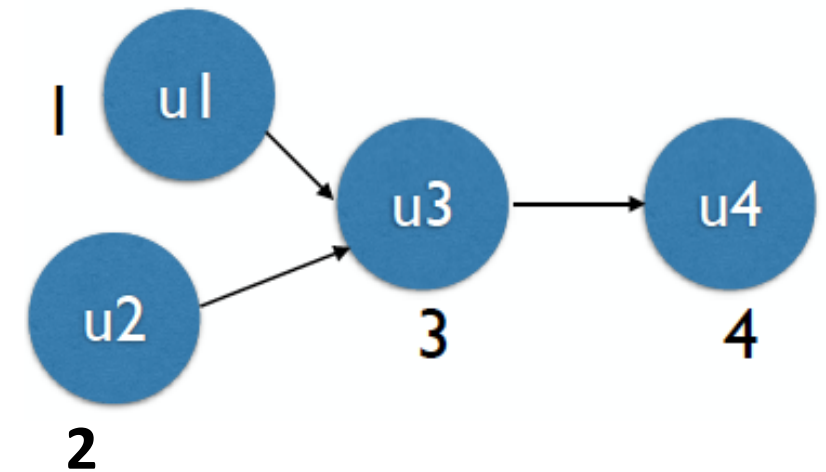
# UNIT TESTING

- Testing of basic module of the software
  - A function, a class, etc.
- Typical problems revealed
  - Local data structures
  - Algorithms
  - Boundary conditions
  - Error handling



# WHY & HOW

- Why Unit testing
  - Divide-and-conquer approach
    - Split system into units
    - Debug unit individually
    - Narrow down places where bugs can be
- How to do Unit testing
  - Build systems in layers
    - Starts with classes that don't depend on others.
    - Continue testing building on already tested classes.



# UNIT TEST FRAMEWORK

- xUnit
  - Created by Kent Beck in 1989
  - This is the same guy who invented TDD
  - The first one was sUnit (for smalltalk)
  
- JUnit
  - The most popular xUnit framework
  - There are about 70 xUnit frameworks for corresponding languages

# PROGRAM TO TEST

```
public class lMath {  
  
    /**  
     * Returns an integer to the square root of x (discarding the fractional parts)  
     */  
    public int isqrt(int x) {  
        int guess = 1;  
        while (guess * guess < x) {  
            guess++;  
        }  
        return guess;  
    }  
}
```

# CONVENTIONAL TESTING

```
/** A class to test the class IMath. */  
public class IMathTestNoJUnit {  
    /** Runs the tests. */  
    public static void main(String[] args) {  
        printTestResult(0);  
        printTestResult(1);  
        printTestResult(2);  
        printTestResult(3);  
        printTestResult(100);  
    }  
    private static void printTestResult(int arg) {  
        IMath tester=new IMath();  
        System.out.print("isqrt(" + arg + ") ==> ");  
        System.out.println(tester.isqrt(arg));  
    }  
}
```

# CONVENTIONAL TEST OUTPUT

- What does this say about the code? Is it right?
- What's the problem with this kind of test output?

```
lsqrt(0) ==> 1  
lsqrt(1) ==> 1  
lsqrt(2) ==> 2  
lsqrt(3) ==> 2  
lsqrt(100) ==> 10
```

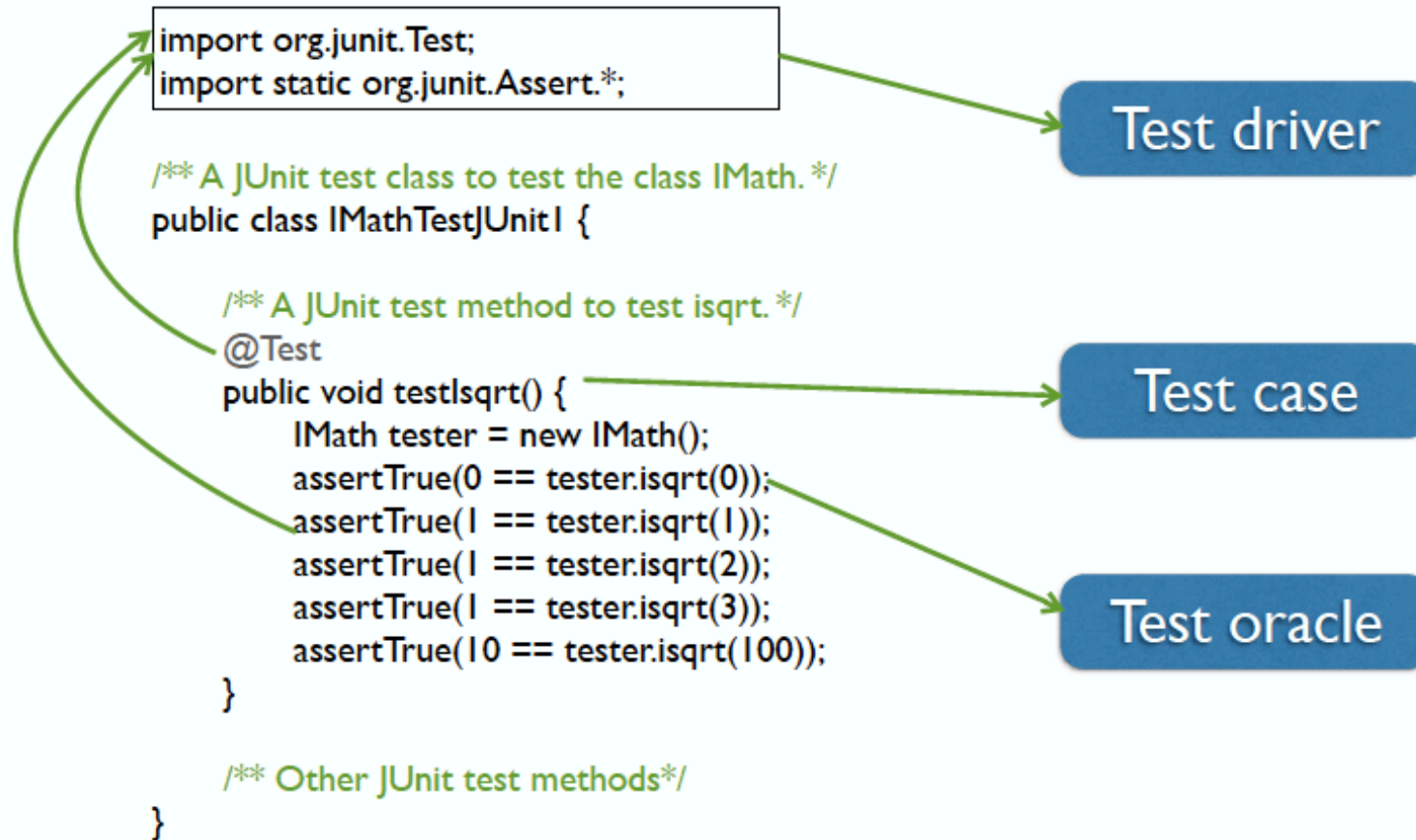
# SOLUTION?

- Automatic verification by testing program
  - Can write such a test program by yourself, or
  - Use testing tool supports, such as JUnit
- JUnit
  - A simple, flexible, easy-to-use, open-source, and practical unit testing framework for Java.
  - Can deal with a large and extensive set of test cases.
  - Refer to [www.junit.org](http://www.junit.org)

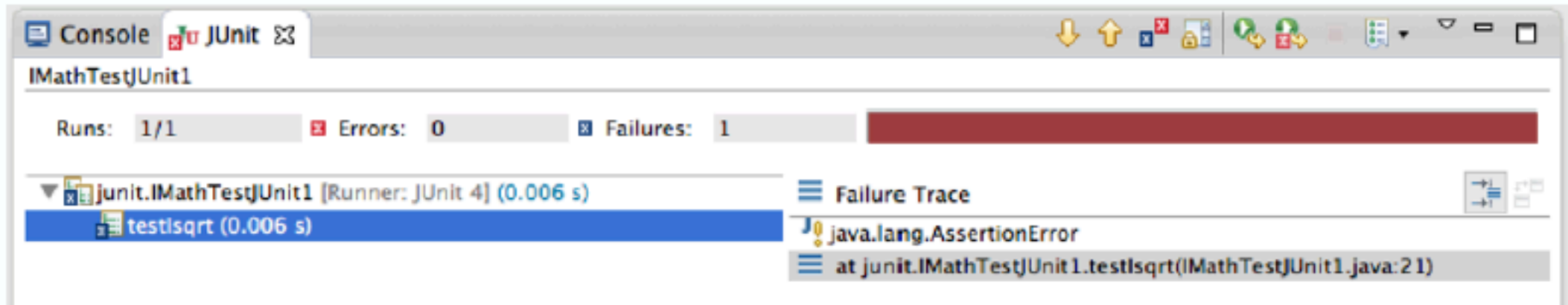
The JUnit logo, with 'J' in green and 'Unit' in red.




# TESTING WITH JUnit (1)



# JUnit EXECUTION (1)



Not so good, why? 

# TESTING WITH JUnit (2)

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

```
/** A JUnit test class to test the class IMath. */
public class IMathTestJUnit2 {
```

```
/** A JUnit test method to test isqrt. */
@Test
public void testIsqrt() {
```

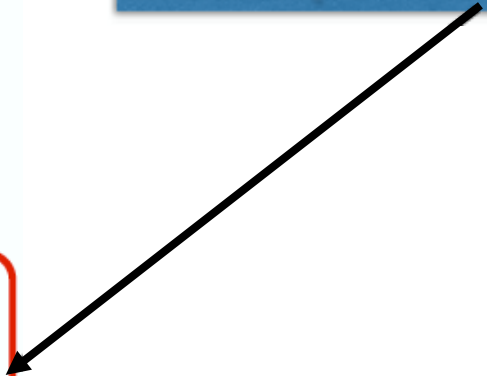
```
    IMath tester = new IMath();
    assertEquals(0, tester.isqrt(0));
    assertEquals(1, tester.isqrt(1));
    assertEquals(1, tester.isqrt(2));
    assertEquals(1, tester.isqrt(3));
    assertEquals(10, tester.isqrt(100));
```

```
}
```

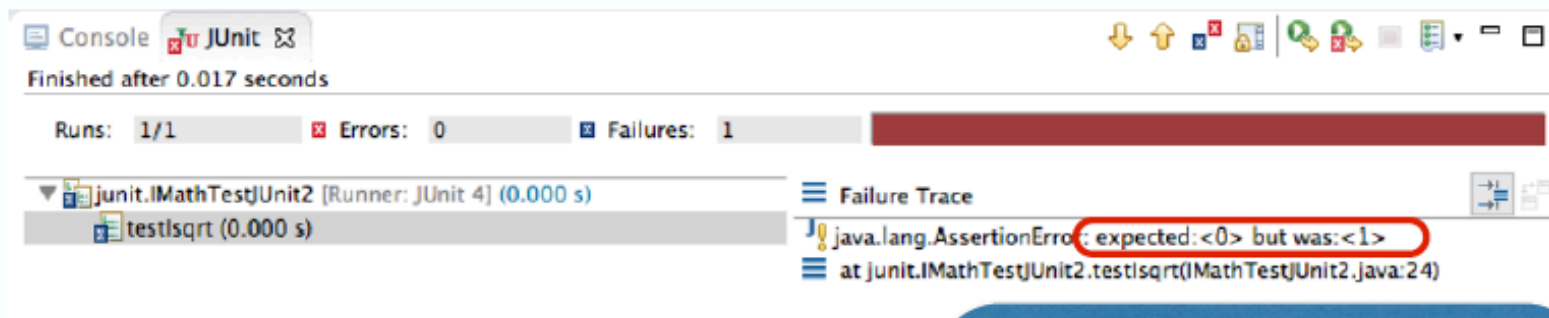
```
/** Other JUnit test methods*/
```

```
}
```

```
assertTrue(0 == tester.isqrt(0));
assertTrue(1 == tester.isqrt(1));
assertTrue(1 == tester.isqrt(2));
assertTrue(1 == tester.isqrt(3));
assertTrue(10 == tester.isqrt(100));
```



# JUnit EXECUTION (2)

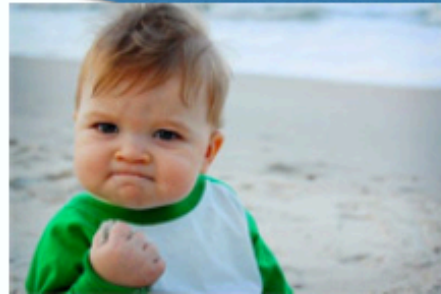


- Why now better error info?
- `assertTrue(0==tester.isqrt(0))`
- `assertEquals(0, tester.isqrt(0))`

detailed result is abstracted into boolean before passed to JUnit

the detailed result is passed to JUnit

Can we make it better?



# TESTING WITH JUnit (3)

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

```
/** A JUnit test class to test the class IMath. */
public class IMathTestJUnit3 {
```

```
    /** A JUnit test method to test isqrt. */
```

```
    @Test
```

```
    public void testIsqrt() {
```

```
        IMath tester = new IMath();
```

```
        assertEquals("square root for 0 ", 0, tester.isqrt(0));
```

```
        assertEquals("square root for 1 ", 1, tester.isqrt(1));
```

```
        assertEquals("square root for 2 ", 1, tester.isqrt(2));
```

```
        assertEquals("square root for 3 ", 1, tester.isqrt(3));
```

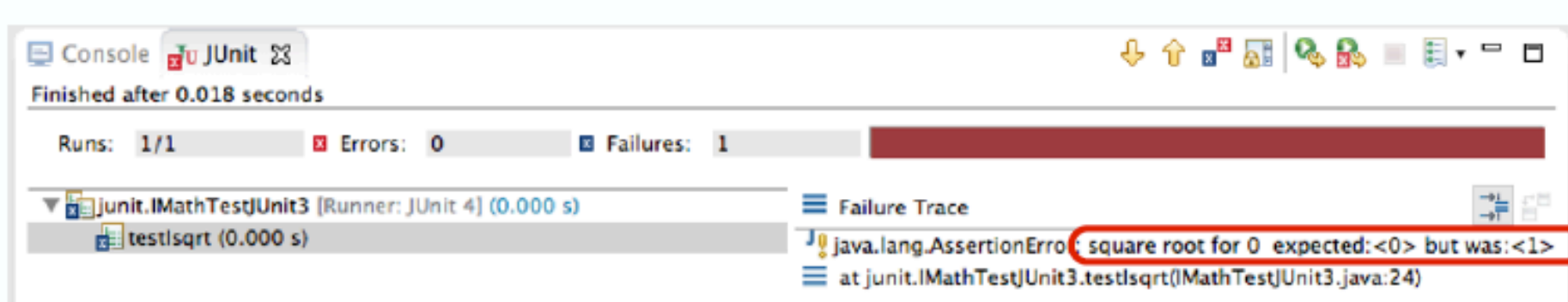
```
        assertEquals("square root for 100 ", 10, tester.isqrt(100));
```

```
    }
```

```
    /** Other JUnit test methods*/
```

```
}
```

# JUnit EXECUTION (3)




Still have problems, why?

We only see the error info for the  
first input...



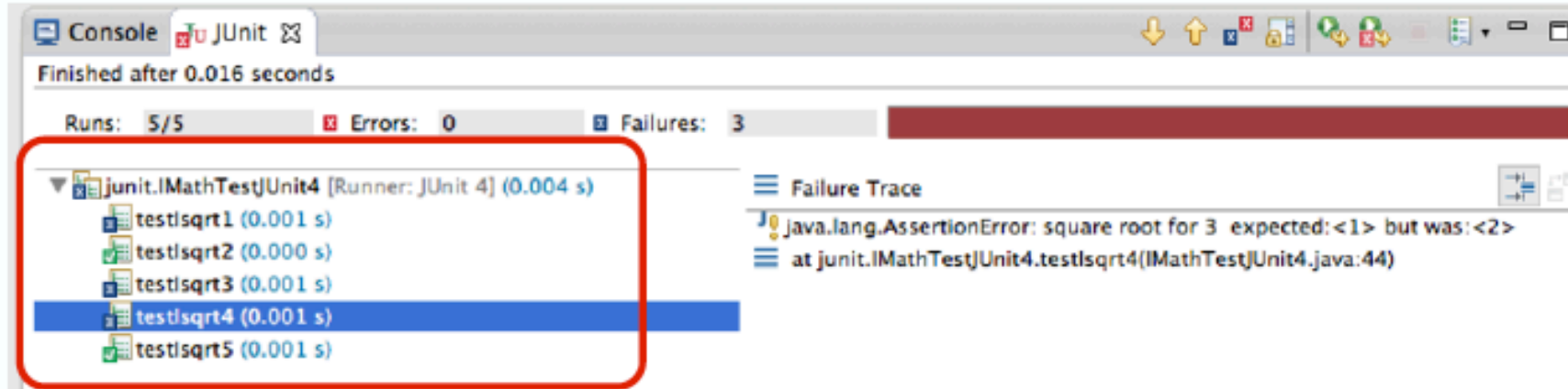
# TESTING WITH JUnit (4)

```
public class IMathTestJUnit4 {  
    private IMath tester;  
  
    @Before /** Setup method executed before each test */  
    public void setup(){  
        tester=new IMath();  
    }  
  
    @Test /** JUnit test methods to test isqrt. */  
    public void testIsqrt1() {  
        assertEquals("square root for 0 ", 0, tester.isqrt(0));  
    }  
    @Test  
    public void testIsqrt2() {  
        assertEquals("square root for 1 ", 1, tester.isqrt(1));  
    }  
    @Test  
    public void testIsqrt3() {  
        assertEquals("square root for 2 ", 1, tester.isqrt(2));  
    }  
    ...  
}
```



Test fixture

# JUnit EXECUTION (4)



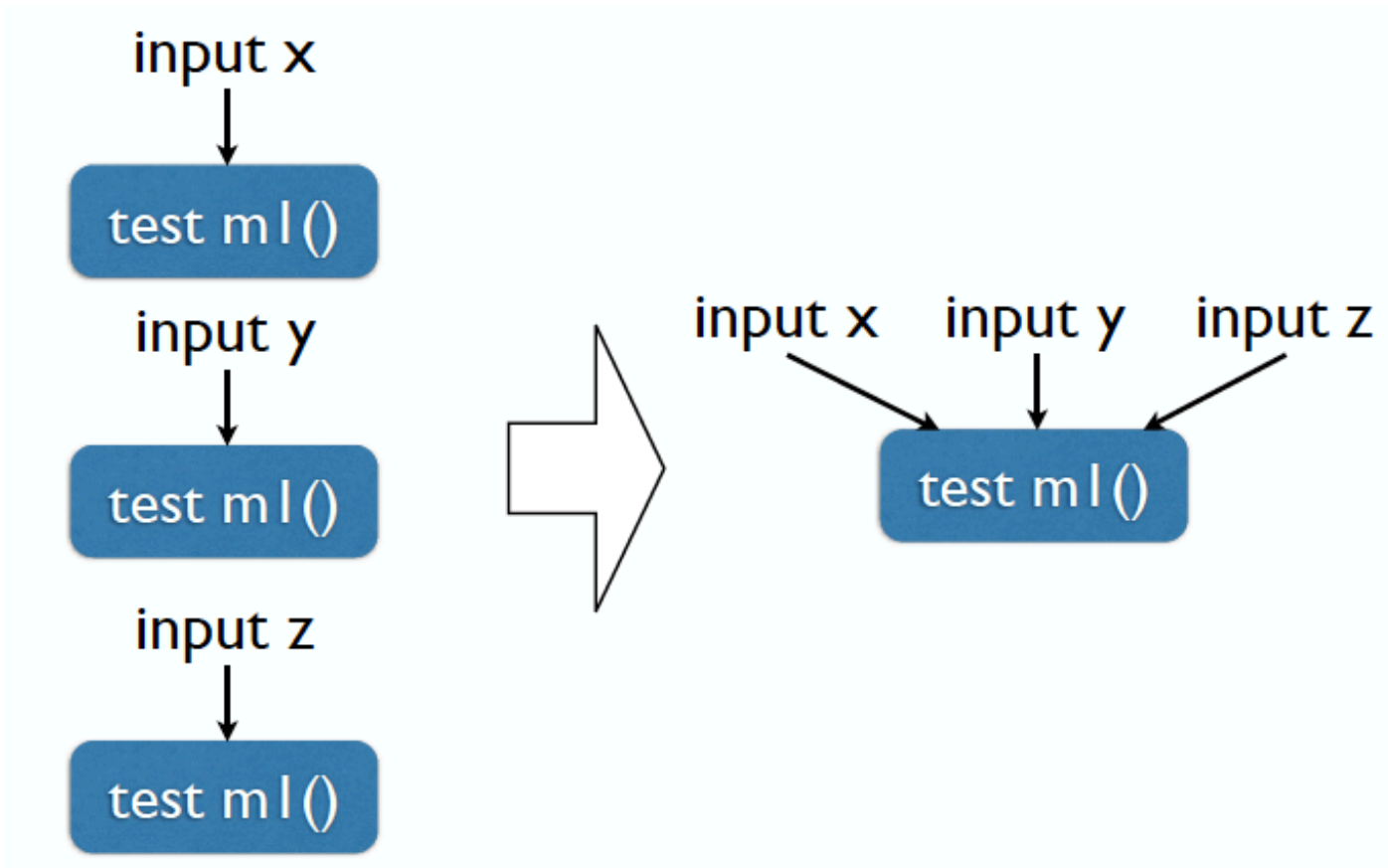
Still may have trouble, why?

We need to write so many similar  
test methods...





# PARAMETERIZED TESTS: ILLUSTRATION



# TESTING WITH JUNIT: PARAMETERIZED TESTS

`@RunWith(Parameterized.class)`

Indicate this is a  
parameterized test class

```
public class IMathTestJUnitParameterized {  
    private IMath tester;  
    private int input;  
    private int expectedOutput;
```

To store input-output pairs

*/\*\* Constructor method to accept each input-output pair \*/*

```
public IMathTestJUnitParameterized(int input, int expectedOutput) {  
    this.input = input;  
    this.expectedOutput = expectedOutput;  
}
```

*@Before /\*\* Set up method to create the test fixture \*/*

```
public void initialize() {tester = new IMath();}
```

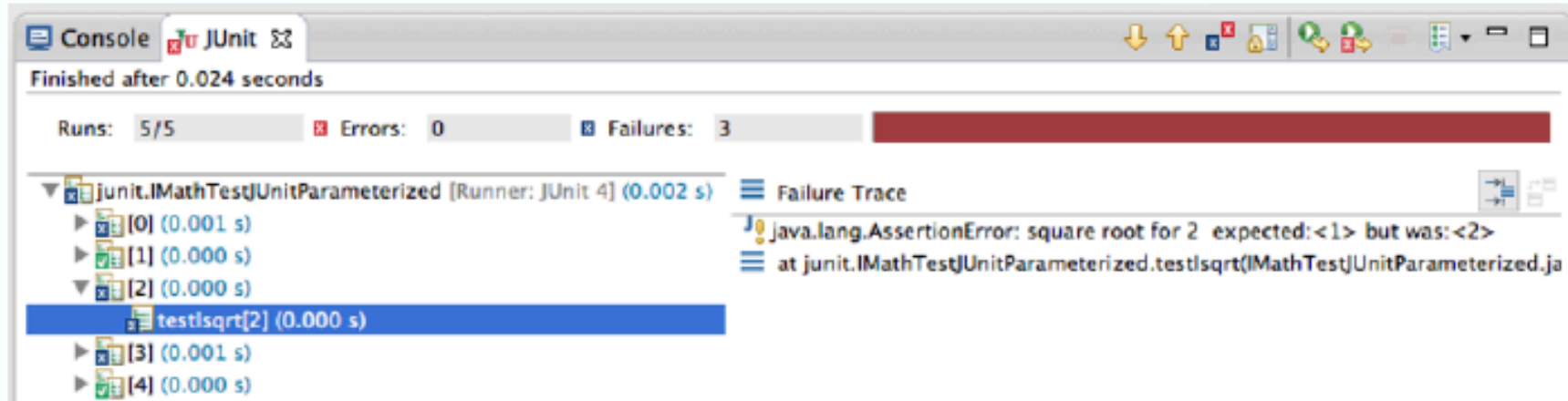
*@Parameterized.Parameters /\*\* Store input-output pairs, i.e., the test data \*/*

```
public static Collection<Object[]> valuePairs() {  
    return Arrays.asList(new Object[][] { { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 1 }, { 100, 10 } });  
}
```

*@Test /\*\* Parameterized JUnit test method \*/*

```
public void testIsqrt() {  
    assertEquals("square root for " + input + " ", expectedOutput, tester.isqrt(input));  
}
```

# JUNIT EXECUTION: PARAMETERIZED TESTS



Note that not all tests can be abstract into parameterized tests

# A COUNTER EXAMPLE

```
public class ArrayList {  
    ...  
    /** Return the size of current list */  
    public int size() {  
        ...  
    }  
    /** Add an element to the list */  
    public void add(Object o) {  
        ...  
    }  
    /** Remove an element from the list */  
    public void remove(int i) {  
        ...  
    }  
}
```

```
public class ListTestJUnit {  
    List list;  
    @Before /** Set up method to create the test fixture */  
    public void initialize() {  
        list = new ArrayList();  
    }  
    /** JUnit test methods */  
    @Test  
    public void test1() {  
        list.add(1);  
        list.remove(0);  
        assertEquals(0, list.size());  
    }  
    @Test  
    public void test2() {
```

These tests cannot be abstract  
into parameterized tests, because  
the tests contains different  
method invocations

# JUNIT ANNOTATIONS

Annotation	Description
@Test	Identify test methods
@Test (timeout=100)	Fail if the test takes more than 100ms
@Before	Execute before each test method
@After	Execute after each test method
@BeforeClass	Execute before each test class
@AfterClass	Execute after each test class
@Ignore	Ignore the test method

# JUNIT ASSERTIONS

Assertion	Description
<code>fail([msg])</code>	Let the test method fail, optional msg
<code>assertTrue([msg], bool)</code>	Check that the boolean condition is true
<code>assertFalse([msg], bool)</code>	Check that the boolean condition is false
<code>assertEquals([msg], expected, actual)</code>	Check that the two values are equal
<code>assertNull([msg], obj)</code>	Check that the object is null
<code>assertNotNull([msg], obj)</code>	Check that the object is not null
<code>assertSame([msg], expected, actual)</code>	Check that both variables refer to the same object
<code>assertNotSame([msg], expected, actual)</code>	Check that variables refer to different objects

# MORE ON JUNIT?

- Homepage:
  - [www.junit.org](http://www.junit.org)
  
- Tutorials
  - <http://www.vogella.com/tutorials/JUnit/article.html>
  - <http://www.tutorialspoint.com/junit/>
  - <https://courses.cs.washington.edu/courses/cse143/11wi/eclipse-tutorial/junit.shtml>

# MANTRA

- Develop test cases before you code!
- Test as you go!
- Test always and often!