

# COMP 587 Lecture 6

Kyle Dewey

# Outline

- Project next steps
- Continuation: tests as specifications
- Note on code linters
- Testability: dependency injection and mocking
- Measuring test quality: coverage

# Project Next Steps

- Join <https://github.com/orgs/CSUN-COMP587-F18>
- Make new / fork repository in organization
  - Caveat: non-public code
- Start working: submit first pull request by Sunday
- Fill out list of known languages

# Continuation: Tests as Specifications

- I'll give another 10 minutes to let you finish implementation / spec (if needed)
- Cover spec and code
- Give tests to another group. Swap with each other.
- Other group: implement from tests. Cannot ask any questions. Note any ambiguities. Give 20 minutes.
- Groups get together. Discuss any sort of ambiguities. Discuss why things were implemented or tested a certain way. 4 minutes.

# Code Linters

# Motivation

-We saw code like this that was super wide...

# Motivation

```
public static ImmutableMap<UserDefinedTypeName, TypeDef> makeTDefs(final Program p) {
    final ImmutableList<Pair<UserDefinedTypeName, TypeDef>> pairs =
        p.tdefs.map(new Function1<UserDefinedTypeDef, Pair<UserDefinedTypeName, TypeDef>>() {
            public Pair<UserDefinedTypeName, TypeDef> apply(final UserDefinedTypeDef input) {
                final ImmutableList<Pair<ConstructorName, Type>> constructorPairs =
                    input.cdefs.map(new Function1<ConstructorDefinition, Pair<ConstructorName, Type>>() {
                        public Pair<ConstructorName, Type> apply(final ConstructorDefinition cdef) {
                            return new Pair<ConstructorName, Type>(cdef.cn, cdef.tau);
                        }
                    });
                final TypeDef typeDef = new TypeDef(input.ts,
                    new ImmutableMap<ConstructorName, Type>(constructorPairs));
                return new Pair<UserDefinedTypeName, TypeDef>(input.un, typeDef);
            }
        });
    return new ImmutableMap<UserDefinedTypeName, TypeDef>(pairs);
}
```

-We saw code like this that was super wide...





# Mc on

```
public static ImmutableMap<UserDefinedTypeName, TypeDef>
    final ImmutableList<Pair<UserDefinedTypeName, TypeDef>
    p.tdefs.map(new Function1<UserDefinedType, Pair<UserDefinedType, TypeDef>>() {
        public Pair<UserDefinedType, TypeDef> call(UserDefinedType t) {
            final ImmutableList<Pair<UserDefinedType, TypeDef>
            input.cdefs.map(new Function1<UserDefinedType, Pair<UserDefinedType, TypeDef>>() {
                public Pair<UserDefinedType, TypeDef> call(UserDefinedType c) {
                    return new Pair<UserDefinedType, TypeDef>(c, new TypeDef());
                }
            });
            final TypeDef typeDef = new TypeDef();
            return new Pair<UserDefinedType, TypeDef>(t, typeDef);
        }
    });
return new ImmutableMap<UserDefinedTypeName, TypeDef>(p.tdefs, typeDef);
}
```

```
final Program p) {
    final ImmutableMap<UserDefinedTypeName, TypeDef>() {
        final ImmutableList<Pair<UserDefinedType, TypeDef>
        constructorPairs =
        initialization, Pair<ConstructorName, Type>>() {
            final ConstructorDefinition cdef) {
                return cdef;
            }
        };
        final Map<ConstructorName, Type>(constructorPairs);
        return un.un, typeDef);
    }
}
```

- It'd be nice to throw these sort of obviously-wrong things out ahead of time
- These are exaggerated cases, but it's easy to slip in more minor problems as part of a larger work. These are things which are mundane, and we shouldn't waste human time finding these sorts of simple things.



- Line length
- Variable names (`this_thing` vs. `thisThing`)
- Indentation and use of whitespace
- All `imports` are used
- Many, many more things according to PEP8

-I'm going to use Pylint as a case study in linters  
-Pylint is used to help enforce the Python style guide (PEP8)

# Linters in Practice

- Standard part of many code check-ins
- Language specific, and many linters exist
- Biggest consistent thing: enforce style guide
  - May help find bugs and bug-like things
  - **Will** help with readability

# Testability, Dependency Injection, and Mocking

```
@Test
public void testCreditCardCharge() {
    CreditCard c = new CreditCard(
        "1234 5678 9012 3456", 5, 2008);
    c.charge(100);
}
```

-Code example from here: [xywang.100871.net/CS4723\\_lec3.ppt](http://xywang.100871.net/CS4723_lec3.ppt)

-From Prof. Xiaoyin Wang (<http://xywang.100871.net/>)

```
@Test
public void testCreditCardCharge() {
    CreditCard c = new CreditCard(
        "1234 5678 9012 3456", 5, 2008);
    c.charge(100);
}
```

**Run with rest of tests: success (and charged \$100)**

**Run alone: fails**

- Code example from here: [xywang.100871.net/CS4723\\_lec3.ppt](http://xywang.100871.net/CS4723_lec3.ppt)
- From Prof. Xiaoyin Wang (<http://xywang.100871.net/>)
- Individual tests should operate independently from each other.
- Even if they aren't independent, we should know why

```
@Test
public void testCreditCardCharge() {
    CreditCard c = new CreditCard(
        "1234 5678 9012 3456", 5, 2008);
    CreditCardProcessor.init();
    c.charge(100);
}
```

- We dive into the code of CreditCard, and we see that the charge method depends on this CreditCardProcessor class
- We need to initialize that class first

```
@Test
public void testCreditCardCharge() {
    CreditCard c = new CreditCard(
        "1234 5678 9012 3456", 5, 2008);
    CreditCardProcessor.init();
    c.charge(100);
}
```

**Run with rest of tests: success (and charged \$100)**

**Run alone: fails**

-Still missing something



```
@Test
public void testCreditCardCharge() {
    CreditCard c = new CreditCard(
        "1234 5678 9012 3456", 5, 2008);
    CreditCardProcessor.init();
    TaskQueue.init();
    c.charge(100);
}
```

- We dive into the code more, and we see that CreditCardProcessor internally uses a TaskQueue.
- Ok, so we need to initialize TaskQueue

```
@Test
public void testCreditCardCharge() {
    CreditCard c = new CreditCard(
        "1234 5678 9012 3456", 5, 2008);
    CreditCardProcessor.init();
    TaskQueue.init();
    c.charge(100);
}
```

**Run with rest of tests: success (and charged \$100)**

**Run alone: fails**

- We dive into the code more, and we see that CreditCardProcessor internally uses a TaskQueue.
- Ok, so we need to initialize TaskQueue

```
@Test
public void testCreditCardCharge() {
    CreditCard c = new CreditCard(
        "1234 5678 9012 3456", 5, 2008);
    CreditCardProcessor.init();
    TaskQueue.init();
    Database.init();
    c.charge(100);
}
```

- Ok, ok, so now we dive into the source code of TaskQueue
- Turns out TaskQueue depends on a Database, so let's initialize that, too

```
@Test
public void testCreditCardCharge() {
    CreditCard c = new CreditCard(
        "1234 5678 9012 3456", 5, 2008);
    CreditCardProcessor.init();
    TaskQueue.init();
    Database.init();
    c.charge(100);
}
```

**Run with rest of tests: success (and charged \$100)**

**Run alone: success (and charged \$100)**

- Ok, ok, so now we dive into the source code of TaskQueue
- Turns out TaskQueue depends on a Database, so let's initialize that, too

```
@Test
public void testCreditCardCharge() {
    CreditCard c = new CreditCard(
        "1234 5678 9012 3456", 5, 2008);
    c.charge(100);
}
```

```
@Test
public void testCreditCardCharge() {
    CreditCard c = new CreditCard(
        "1234 5678 9012 3456", 5, 2008);
    CreditCardProcessor.init();
    TaskQueue.init();
    Database.init();
    c.charge(100);
}
```

- We started with the code on the top, and ended up with the code on the bottom
- This reveals that the code isn't very testable. Why?

```
@Test
public void testCreditCardCharge() {
    CreditCard c = new CreditCard(
        "1234 5678 9012 3456", 5, 2008);
    c.charge(100);
}
```

```
@Test      Not very testable. Why?
public void testCreditCardCharge() {
    CreditCard c = new CreditCard(
        "1234 5678 9012 3456", 5, 2008);
    CreditCardProcessor.init();
    TaskQueue.init();
    Database.init();
    c.charge(100);
}
```

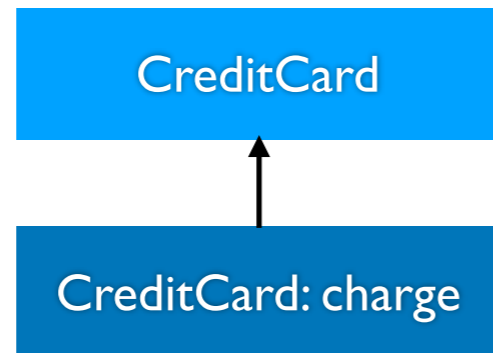
- We started with the code on the top, and ended up with the code on the bottom
- This reveals that the code isn't very testable. Why?

# Explicit Dependencies

```
CreditCard c = new CreditCard(  
    "1234 5678 9012 3456", 5, 2008);  
CreditCardProcessor.init();  
TaskQueue.init();  
Database.init();  
c.charge(100);
```

# Explicit Dependencies

```
CreditCard c = new CreditCard(  
    "1234 5678 9012 3456", 5, 2008);  
CreditCardProcessor.init();  
TaskQueue.init();  
Database.init();  
c.charge(100);
```

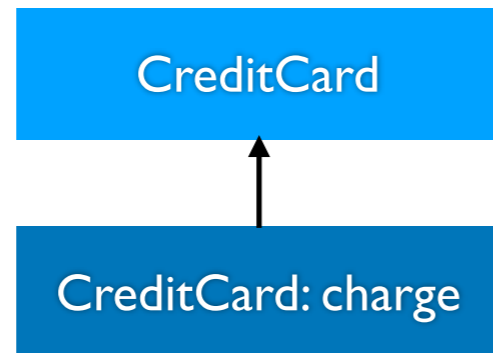


- This is the only explicit dependency
- Looking at only this code, there is nothing which explicitly requires CreditCardProcessor, TaskQueue, or Database



# With Implicit Dependencies

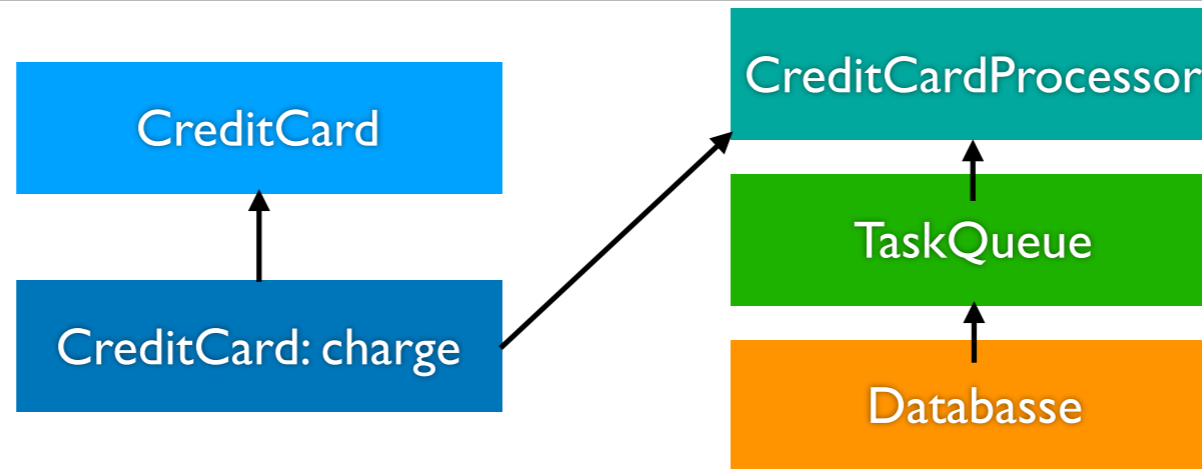
```
CreditCard c = new CreditCard(  
    "1234 5678 9012 3456", 5, 2008);  
CreditCardProcessor.init();  
TaskQueue.init();  
Database.init();  
c.charge(100);
```



- This is the only explicit dependency
- Looking at only this code, there is nothing which explicitly requires CreditCardProcessor, TaskQueue, or Database

# With Implicit Dependencies

```
CreditCard c = new CreditCard(  
    "1234 5678 9012 3456", 5, 2008);  
CreditCardProcessor.init();  
TaskQueue.init();  
Database.init();  
c.charge(100);
```

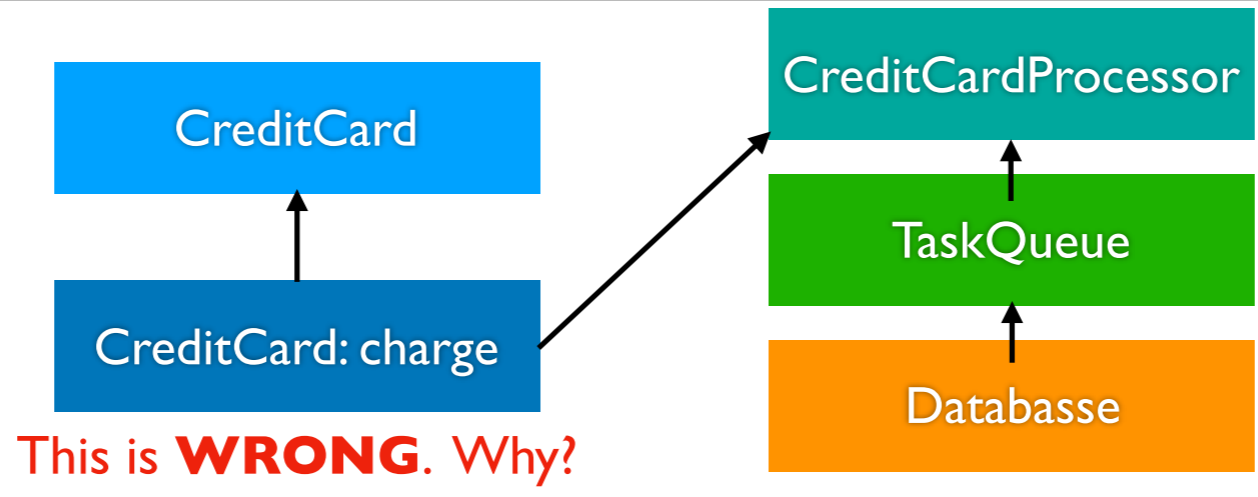


-This is the only explicit dependency

-Looking at only this code, there is nothing which explicitly requires CreditCardProcessor, TaskQueue, or Database

# With Implicit Dependencies

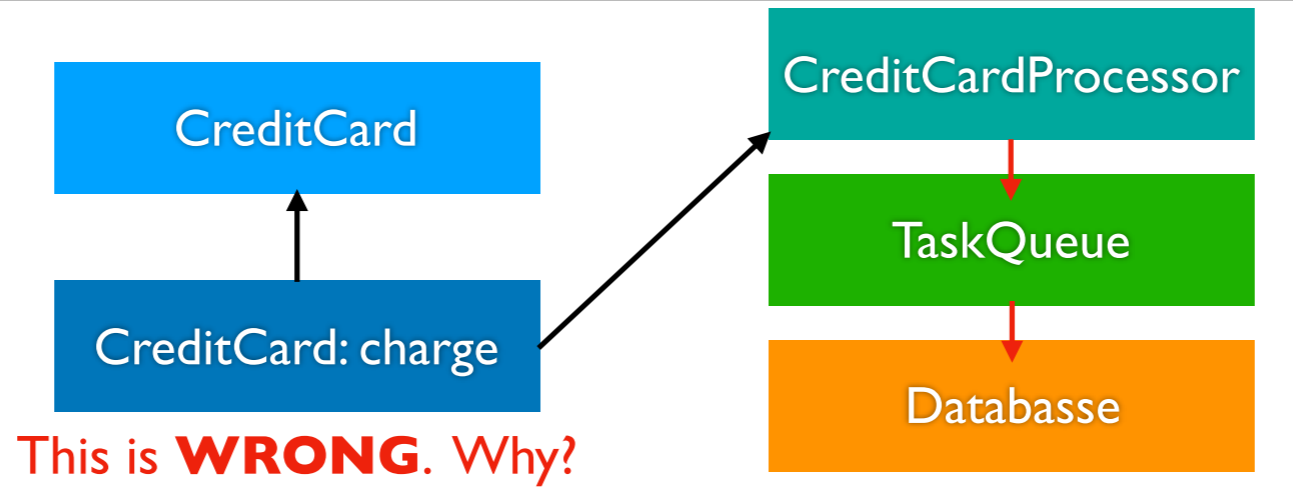
```
CreditCard c = new CreditCard(  
    "1234 5678 9012 3456", 5, 2008);  
CreditCardProcessor.init();  
TaskQueue.init();  
Database.init();  
c.charge(100);
```



- This is the only explicit dependency
- Looking at only this code, there is nothing which explicitly requires CreditCardProcessor, TaskQueue, or Database

# With Implicit Dependencies

```
CreditCard c = new CreditCard(  
    "1234 5678 9012 3456", 5, 2008);  
CreditCardProcessor.init();  
TaskQueue.init();  
Database.init();  
c.charge(100);
```



- We said that charge depended on CreditCardProcessor, which in turn depended on TaskQueue, which in turn depended on Database
- We need to flip these arrows to show this.
- Probably should update the code to reverse the initialization order, too

# Fundamental Problem

Few actual code dependencies are explicit.  
Nothing enforces we get implicit dependencies correct.

```
CreditCard c = new CreditCard(  
    "1234 5678 9012 3456", 5, 2008);  
CreditCardProcessor.init();  
TaskQueue.init();  
Database.init();  
c.charge(100);
```

- "Nothing enforces" is another way of saying "nothing stops us from introducing bugs here"
- Implicit dependencies can only be handled with informal documentation and sufficient system knowledge. Generally, we'll never know how much system knowledge is "sufficient".

# Dependency Injection

- Fundamentally: make dependencies explicit
- Frequently done by parameter passing

```
CreditCard c = new CreditCard(  
    "1234 5678 9012 3456", 5, 2008);  
CreditCardProcessor.init();  
TaskQueue.init();  
Database.init();  
c.charge(100);
```

---

-This is the code we started with...

```
CreditCard c = new CreditCard(
    "1234 5678 9012 3456", 5, 2008);
CreditCardProcessor.init();
TaskQueue.init();
Database.init();
c.charge(100);
```

```
Database db = new Database();
TaskQueue tq = new TaskQueue(db);
CreditCardProcessor ccp =
    new CreditCardProcessor(tq);
CreditCard c = new CreditCard(
    "1234 5678 9012 3456", 5, 2008);
c.charge(ccp, 100);
```

- ...and the code on the bottom is much more explicit
- We cannot make a TaskQueue without a Database. Explicit, enforced dependency.
- We cannot make a CreditCardProcessor without a TaskQueue. Explicit, enforced dependency.
- We cannot call charge without a CreditCardProcessor. Explicit, enforced dependency.



# Relationship to Testing

When dependencies must be explicitly provided, we can easily substitute them with special test inputs.

# Relationship to Testing

When dependencies must be explicitly provided, we can easily substitute them with special test inputs.

```
Database db = new Database();
TaskQueue tq = new TaskQueue(db);
CreditCardProcessor ccp =
    new CreditCardProcessor(tq);
CreditCard c = new CreditCard(
    "1234 5678 9012 3456", 5, 2008);
c.charge(ccp, 100);
```

-Here we have the code with dependency injection

# Relationship to Testing

When dependencies must be explicitly provided, we can easily substitute them with special test inputs.

```
Database db = new Database();
TaskQueue tq = new TaskQueue(db);
CreditCardProcessor ccp =
    new CreditCardProcessor(tq);
CreditCard c = new CreditCard(
    "1234 5678 9012 3456", 5, 2008);
c.charge(ccp, 100);
```

-This code still makes a real \$100 charge each time we run it

# Relationship to Testing

When dependencies must be explicitly provided, we can easily substitute them with special test inputs.

```
CreditCardProcessor ccp =  
    new NoSendCreditCardProcessor();  
CreditCard c = new CreditCard(  
    "1234 5678 9012 3456", 5, 2008);  
c.charge(ccp, 100);
```

- However, we can easily swap out the CreditCardProcessor with a testing variant. CreditCardProcessor is now an interface instead of a concrete class.
- NoSendCreditCardProcessor conforms to the CreditCardProcessor interface, but it won't send the charge.
- Since it doesn't actually do anything, we don't need the Database and the TaskQueue anymore
- We can still have this test, but we won't get charged \$100

# Mocking

- The name of this substitution for testing is *mocking*
  - As in, a mock-up
- Libraries exist to assist with mocking
  - Easier to define custom test inputs
  - Easy to adapt to different testing scenarios

# Example: Mockito (Java)

```
CreditCardProcessor ccp =  
    new NoSendCreditCardProcessor();  
CreditCard c = new CreditCard(  
    "1234 5678 9012 3456", 5, 2008);  
c.charge(ccp, 100);
```

-<https://site.mockito.org/>

-We had the code on the top originally

# Example: Mockito (Java)

```
CreditCardProcessor ccp =  
    new NoSendCreditCardProcessor();  
CreditCard c = new CreditCard(  
    "1234 5678 9012 3456", 5, 2008);  
c.charge(ccp, 100);
```

```
CreditCardProcessor ccp =  
    mock(CreditCardProcessor.class);  
when(ccp.chargeAccepted(100))  
    .thenReturn(true);  
CreditCard c = new CreditCard(  
    "1234 5678 9012 3456", 5, 2008);  
c.charge(ccp, 100);
```

-<https://site.mockito.org/>

-We can replace this code with the code on the bottom

-Biggest difference: we don't need to define an explicit NoSendCreditCardProcessor anymore

-While it looks like there is more code, and now a chargeAccepted method, this method would have had to have been implemented in NoSendCreditCardProcessor, along with any other functionality needed by the CreditCardProcessor interface. This is a lot less code, and it's more to the point.

# Measuring Test Quality: Coverage



# Coverage: Basic Idea

Run tests, and see which parts of the code tests touch.

# Coverage: Basic Idea

Run tests, and see which parts of the code tests touch.

```
public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```

-Let's take this code

# Coverage: Basic Idea

Run tests, and see which parts of the code tests touch.

```
public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```

```
@Test  
public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```

-...along with this test

# Coverage: Basic Idea

Run tests, and see which parts of the code tests touch.

```
public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```

```
@Test  
public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```

-If we run the test, it will touch part of the code

# Coverage: Basic Idea

Run tests, and see which parts of the code tests touch.

```
→ public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```

```
@Test  
public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```

-Hits method entry

# Coverage: Basic Idea

Run tests, and see which parts of the code tests touch.

```
→ public int min(int x, int y) {  
→   if (x < y) { return x; }  
     else { return y; }  
}
```

```
@Test  
public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```

-Hits the if, specifically the true branch

# Coverage: Basic Idea

Run tests, and see which parts of the code tests touch.

```
→ public int min(int x, int y) {  
→   if (x < y) { return x; }  
   X else { return y; }  
}
```

```
@Test  
public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```

-Does not hit the false branch

# Coverage Intuition

Touching more code = better tests.



# Coverage Intuition

Touching more code = better tests.

```
public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```

-I'll take the same code...

# Coverage Intuition

Touching more code = better tests.

```
public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```

```
@Test  
public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```

-...along with the same test

# Coverage Intuition

Touching more code = better tests.

```
public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```

```
@Test  
public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}  
  
@Test  
public void minWorksIfMinSecond() {  
    assertEquals(0, min(1, 0));  
}
```

-And add a second test, which tests something different from the first

# Coverage Intuition

Touching more code = better tests.

```
→ public int min(int x, int y) {  
→   if (x < y) { return x; }  
  X   else { return y; }  
}
```

```
@Test  
public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```

```
@Test  
public void minWorksIfMinSecond() {  
    assertEquals(0, min(1, 0));  
}
```

-First test hits only the first two lines

# Coverage Intuition

Touching more code = better tests.

```
→ public int min(int x, int y) {  
  × if (x < y) { return x; }  
  → else { return y; }  
}
```

```
@Test  
public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```

```
@Test  
public void minWorksIfMinSecond() {  
    assertEquals(0, min(1, 0));  
}
```

-Second test misses the true branch of the if

# Coverage Intuition

Touching more code = better tests.

```
→ public int min(int x, int y) {  
→   if (x < y) { return x; }  
→   else { return y; }  
}
```

```
@Test  
public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}  
  
@Test  
public void minWorksIfMinSecond() {  
    assertEquals(0, min(1, 0));  
}
```

-Together, however, they hit everything

# Coverage Reveals

# Missed Behaviors

More tests != better tests

# Coverage Reveals Missed Behaviors

More tests != better tests

---

```
public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```



# Coverage Reveals Missed Behaviors

More tests != better tests

```
public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```

```
@Test public void minWorksIfMinFirst1() {  
    assertEquals(0, min(0, 1));  
}  
@Test public void minWorksIfMinFirst2() {  
    assertEquals(2, min(2, 3));  
}
```

# Coverage Reveals Missed Behaviors

More tests != better tests

```
→ public int min(int x, int y) {  
→   if (x < y) { return x; }  
   else { return y; }  
}
```

```
@Test public void minWorksIfMinFirst1() {  
    assertEquals(0, min(0, 1));  
}  
  
@Test public void minWorksIfMinFirst2() {  
    assertEquals(2, min(2, 3));  
}
```

-Coverage of only the first test

# Coverage Reveals Missed Behaviors

More tests != better tests

```
→ public int min(int x, int y) {  
→   if (x < y) { return x; }  
   X else { return y; }  
}
```

```
@Test public void minWorksIfMinFirst1() {  
    assertEquals(0, min(0, 1));  
}  
  
@Test public void minWorksIfMinFirst2() {  
    assertEquals(2, min(2, 3));  
}
```

-Coverage of only the second test

# Coverage Reveals Missed Behaviors

More tests != better tests

```
→ public int min(int x, int y) {  
→   if (x < y) { return x; }  
  X else { return y; }  
}
```

```
@Test public void minWorksIfMinFirst1() {  
    assertEquals(0, min(0, 1));  
}  
@Test public void minWorksIfMinFirst2() {  
    assertEquals(2, min(2, 3));  
}
```

-Coverage of both tests combined

# Coverage Metric: Line

How many lines did the tests touch?

# Coverage Metric: Line

How many lines did the tests touch?

```
1: public int min(int x, int y) {  
2:     if (x < y) {  
3:         return x;  
4:     } else {  
5:         return y;  
6:     }  
7: }
```

- Now line numbers have been added
- Explicit moving of code to lines

# Coverage Metric: Line

How many lines did the tests touch?

```
1: public int min(int x, int y) {  
2:     if (x < y) {  
3:         return x;  
4:     } else {  
5:         return y;  
6:     }  
7: }
```

```
@Test  
public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```

-Same test as before

# Coverage Metric: Line

How many lines did the tests touch?

```
→ 1: public int min(int x, int y) {  
→ 2:     if (x < y) {  
→ 3:         return x;  
X 4:     } else {  
X 5:         return y;  
X 6:     }  
X 7: }
```

```
@Test  
public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```

-Running that test touches these lines



# Coverage Metric: Line

How many lines did the tests touch?

```
→ 1: public int min(int x, int y) {  
→ 2:     if (x < y) {  
→ 3:         return x;  
✗ 4:     } else {  
✗ 5:         return y;  
✗ 6:     }  
✗ 7: }
```

Coverage: 3/7 (~43%)

```
@Test  
public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```

-Running that test touches these lines

# Coverage Metric: Instruction

- Like line coverage, but based on the compiled instructions emitted instead of raw source code
- Avoids counting oddities due to formatting

# Coverage Metric: Instruction

- Like line coverage, but based on the compiled instructions emitted instead of raw source code
- Avoids counting oddities due to formatting

---

```
1: x = 1; y = 2; z = 3;
```

-I might only have one line of code

# Coverage Metric: Instruction

- Like line coverage, but based on the compiled instructions emitted instead of raw source code
- Avoids counting oddities due to formatting

---

```
1: x = 1; y = 2; z = 3;
```

---

```
1: x = 1;  
2: y = 2;  
3: z = 3;
```

-I might format the same code differently to get three lines

# Coverage Metric: Instruction

- Like line coverage, but based on the compiled instructions emitted instead of raw source code
- Avoids counting oddities due to formatting

```
1: x = 1; y = 2; z = 3;
```

```
1: x = 1;  
2: y = 2;  
3: z = 3;
```

```
1: li $t0, 1  
2: li $t1, 2  
3: li $t2, 3
```

- No matter the formatting, I get the same instructions emitted
- As such, measuring coverage over instructions is more robust

# Coverage Metric: Branch

How many *branches* did the tests touch?

# Coverage Metric: Branch

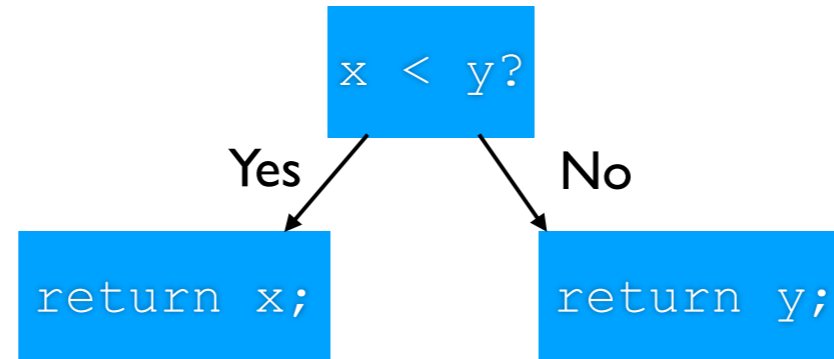
How many *branches* did the tests touch?

```
public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```

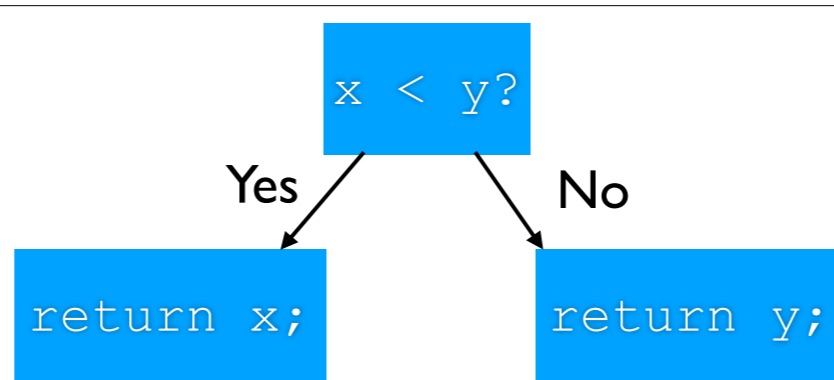
# Coverage Metric: Branch

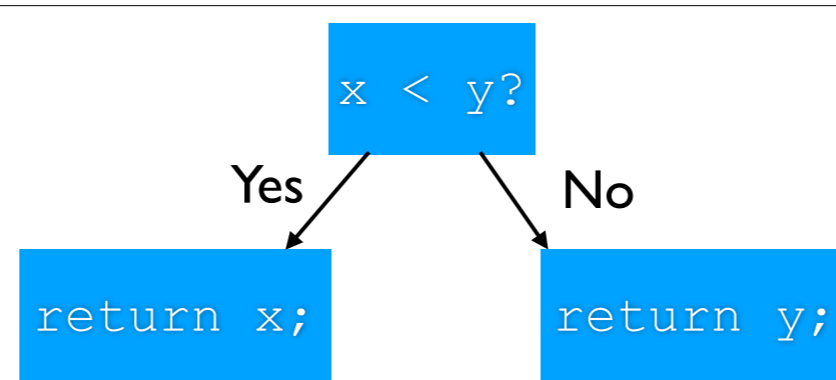
How many *branches* did the tests touch?

```
public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```



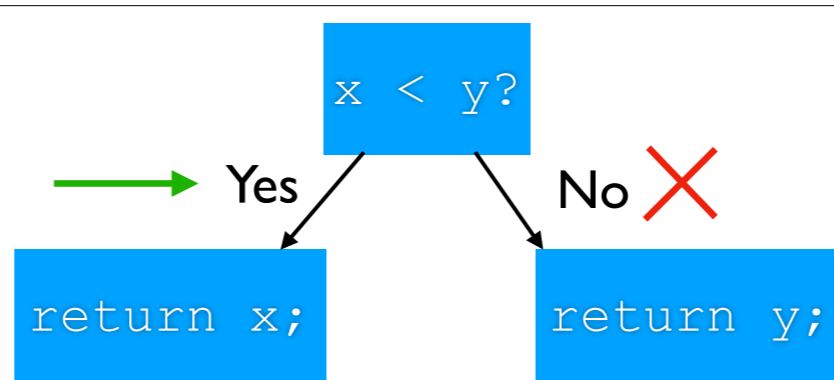






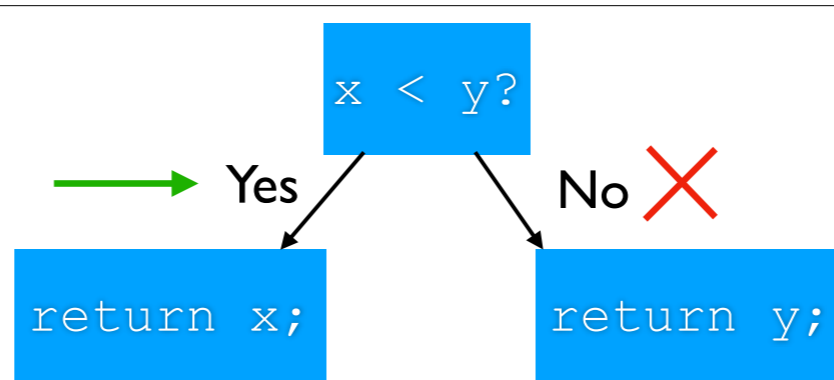
---

```
@Test public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```



```
@Test public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```

-Running this test hits the yes, but not the no



```
@Test public void minWorksIfMinFirst() {  
    assertEquals(0, min(0, 1));  
}
```

Coverage: 1/2 (50%)

-Running this test hits the yes, but not the no

# Coverage Demo: JaCoCo

# Coverage Caveat

100% coverage does **not** mean bug-free

# Coverage Caveat

100% coverage does **not** mean bug-free

---

```
public int brokenMin(int x, int y) {  
    if (x < y) { return y; }  
    else { return x; }  
}
```

# Coverage Caveat

100% coverage does **not** mean bug-free

```
public int brokenMin(int x, int y) {  
    if (x < y) { return y; }  
    else { return x; }  
}
```

```
@Test public void runMin() {  
    min(0, 1);  
    min(1, 0);  
}
```



# Coverage Caveat

100% coverage does **not** mean bug-free

```
public int brokenMin(int x, int y) {  
    if (x < y) { return y; }  
    else { return x; }  
}
```

```
@Test public void runMin() {  
    min(0, 1);  
    min(1, 0);  
}
```

100% coverage, but tests nothing

# Coverage Caveat

< 100% coverage does **not** mean buggy

# Coverage Caveat

< 100% coverage does **not** mean buggy

---

```
public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```

# Coverage Caveat

< 100% coverage does **not** mean buggy

---

```
public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```

---

<<no tests written>>

# Coverage Caveat

< 100% coverage does **not** mean buggy

---

```
public int min(int x, int y) {  
    if (x < y) { return x; }  
    else { return y; }  
}
```

---

<<no tests written>>

0% coverage, but no bugs

# Coverage Caveats in Practice

- Automated testing techniques
  - Find tons of bugs
  - Usually don't improve coverage much
- Striving for 100% sees diminishing returns
  - May need extensive code modification
  - Important: what's the repercussions of a bug being in untested code?