Objectives

- Testing
- Collaboration
- Coverage

Review

- What are the steps to a JUnit test case?
  - How do we implement them?
- What approaches did you use to write good test cases to reveal the mutants?
  - How is programming tests the same/different from programming generally?
  - What are examples of different kinds of assert statements you can make?
- Recall the characteristics of good unit tests
  - How did you achieve them in your testing?
- What are the benefits of unit testing/using JUnit?
  - Consider if you were developing/maintaining the thirdShortest method. How would your testing/development process change?
- True or False. Unit testing is all the testing that needs to be done for an application.

Think about team/group projects

- Why did some work well?
- Why were some disasters?

Catch the Mutants: Analysis

- Focus on the API and how you *want* the code to work under a variety of circumstances
  - Your tests are what the code needs to pass to know that it works
- One test case can reveal multiple bugs
  - There is not a one-to-one mapping

Project: Test-Driven Development

- Given: a `Car` class that only has enough code to compile
- Your job: Create a ***good*** set of test cases that ***thoroughly***/***effectively*** test `Car` class
  - Find faults in my faulty version of `Car` class
  - Start: look at API, think about how to test, set up JUnit tests
  - Written analysis of process
- First team project: teams of **3**
  - Practice collaboration
  - Every student must commit code to the repository

Approaching the Testing Project

- **THINK** first (and often)
  - Use paper/a document to structure your thoughts and systematically consider what you need to test
  - Decide on assumptions about specification
- Iterative approach
  - Each team member implements a few tests
    - Put into repository
  - Discuss as a team
    - Reconsider/confirm your assumptions, how you want to break up the work

Testing Recommendations

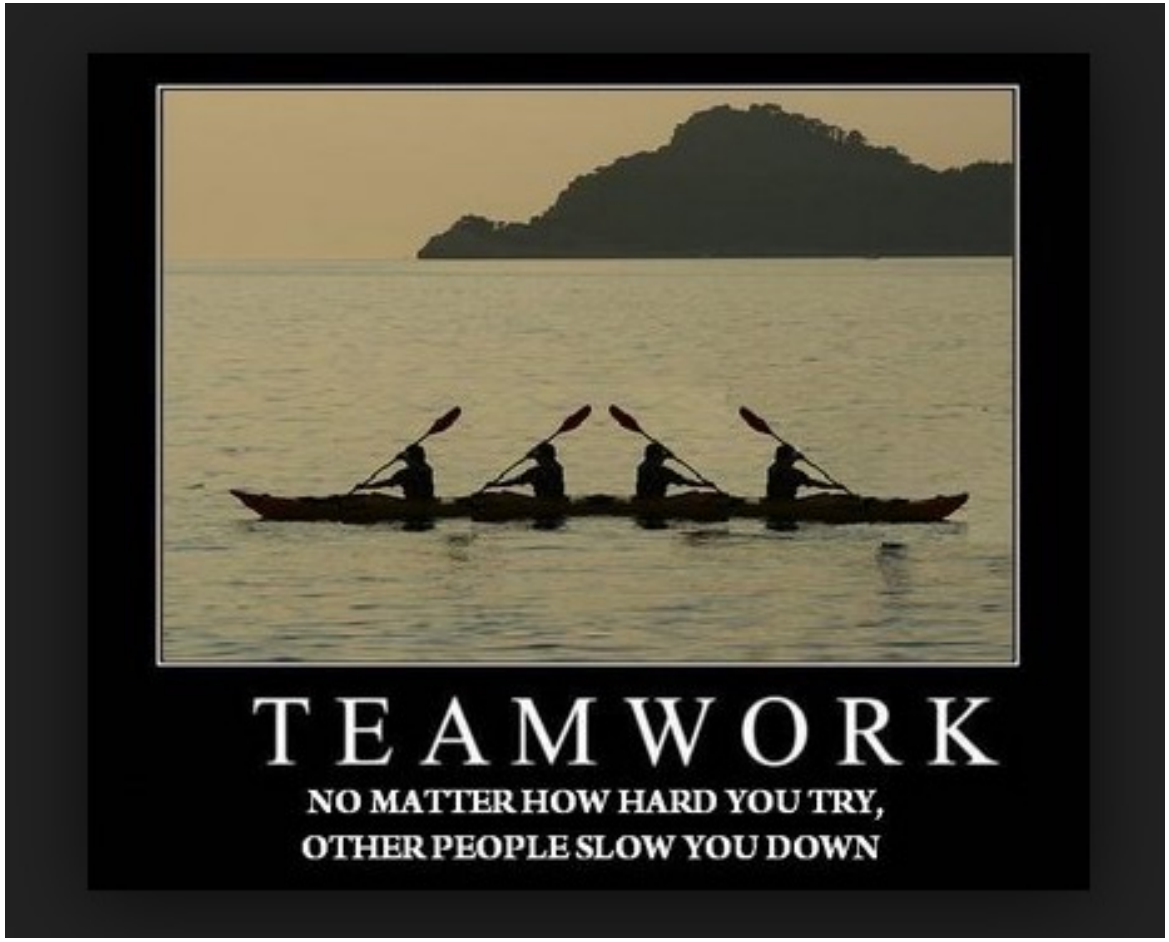
- Do what you did to test classes previously, but adapt for JUnit framework
- Create your testing process
- Decide on your assumptions
  - Be consistent
- Encode the specifications for the code in your tests
  - Code must pass these to show that it is correct
- Check the FAQ

Guidance for Writing JUnit Tests

- A test method should focus on one behavior
  - If test case fails, the test case should be helpful in narrowing down where the problem is
- Testing isn't typically "creative" and doesn't need to be generalizable
  - Code should be straightforward
- See examples linked from course schedule page

Guidance for Organizing JUnit Tests

- Group tests in methods, classes
- Classes could be distinguished by behavior, by error conditions, by set up method…
- Name methods based on what they test
  - Template: `functionality_state_expectedresult`
  - Example: `go_fulltank_forward_moves`


TEAMWORK
NO MATTER HOW HARD YOU TRY,
OTHER PEOPLE SLOW YOU DOWN

Think about Team (Group) Projects

- Why did some work well?
- Why were some disasters?

Teams Work Best When They are **Interdependent**

- In code terms, we want *loose coupling*
  - Depend on each other but don't depend on their details
- Consider
  - Are you allowing your team to truly be interdependent?
  - Who might be you be ignoring?
  - Who might be allowing themselves to feel inadequate?
  - How do you show appreciation for each other and yourself?

Collaboration: Team Project

- Version Control does not eliminate need for communication
  - Process becomes much more difficult if developers do not communicate
- Keep the version to be graded in `main` branch
- You should work in a different branch
- Before picking up again on development, **pull** the repository
  - Get others' changes in main; merge into your branch
- Each student on team must make *significant* commits to the project's repository

Collaboration: Team Project

- Need to talk about the solution
- Discuss your plan, e.g.,
  - Your assumptions about the Car class
  - Your system for testing to make sure that you test everything
  - Organization of test cases
  - Naming
  - Division of labor
- Maintain planning documents too
  - in GitHub or elsewhere

Collaboration: Workflow – Seeking Feedback

1. Pull the `main` branch to get the latest code
2. Create a branch from `main` for your work
   - Commit periodically
   - Write descriptive comments so your team members know what you did and why
3. Push your branch
4. In GitHub, open a ***Pull Request*** on your branch
   - You can tag your teammates to let them know that you've completed your work
   - Team: discuss and review potential changes – can still update
5. Merge pull request into `main` branch (when ready)
6. Pull the `main` branch to get the latest code
   - Merge main into your branch or create a new branch from main

Don't work directly in main

Collaboration: Workflow

1. Pull the `main` branch to get the latest code
2. Create a branch from `main` for your work
   - Commit periodically
   - Write descriptive comments so your team members know what you did and why
3. Switch to `main`
4. Pull `main` branch to get most recent code
5. Merge your branch into the `main` branch
   - Handle merge conflicts
   - Commit
6. Push `main` branch

Don't work directly in main

Culture Eats Strategy for Breakfast

Your actions should match what your team says are your squad goals.

Review: Software Testing Issues

- How should you test? How often?
  - Code may change frequently
  - Code may depend on others' code
  - A lot of code to validate
- How do you know that an output is correct?
  - Complex output
  - Human judgment?
- What caused a code failure?

➥ Need a *systematic*, *automated*, *repeatable* approach

More Software Testing Issues

- How do we know if our code is correct?
  - How do we know that we've exposed all the faults?
  - How confident are we in its correctness?
- How do we know if we've tested enough?
  - Our customers want this product soon but we need product to be correct
    - Harder to fix after it has been released

More Software Testing Issues

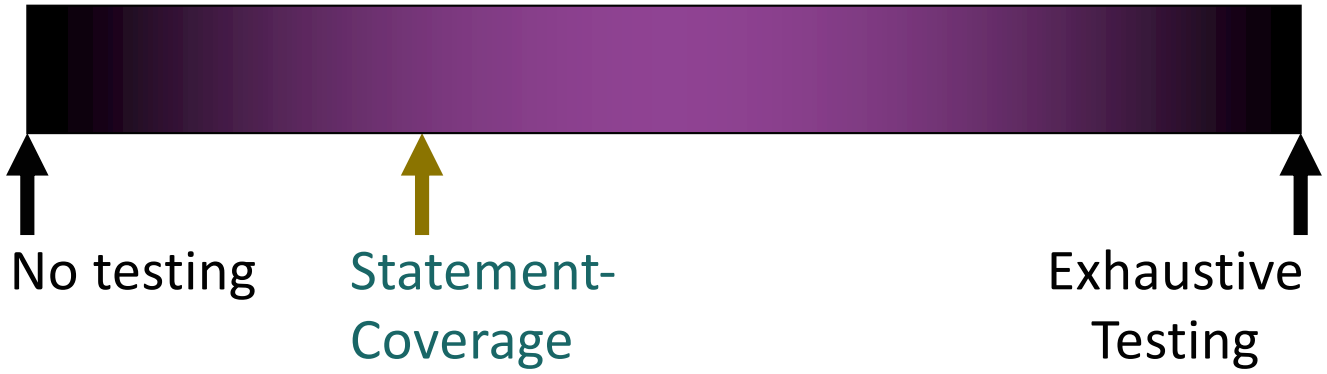
- How do we know if our code is correct?
  - How do we know that we've exposed all the faults?
  - How confident are we in its correctness?
- How do we know if we've tested enough?
  - Our customers want this product soon but we need product to be correct
    - Harder to fix after it has been released
  - Passes all of our TDD test suite
    - But did we come up with *all* the necessary test cases?
  - Time? It's been a couple hours/days/…
  - Number of test cases executed? A lot!
  - I asked my sister and she's really smart and she says that it's enough

Testing Continuum

- Give to customer immediately
- Likely buggy!
- Very little confidence in program's quality

- Test *every possible* input
- Costly, impractical
- Need to release application to customers *sometime!*

Measuring Test Quality



- Idea: Measure how much code you cover
  - If you *cover*, i.e., execute, code in your tests, you'll reveal faults if that code contains a fault.
- Metric: cover all **statements** in the program

Analogy: Map coverage

Statement Coverage

- Cover all statements in the program

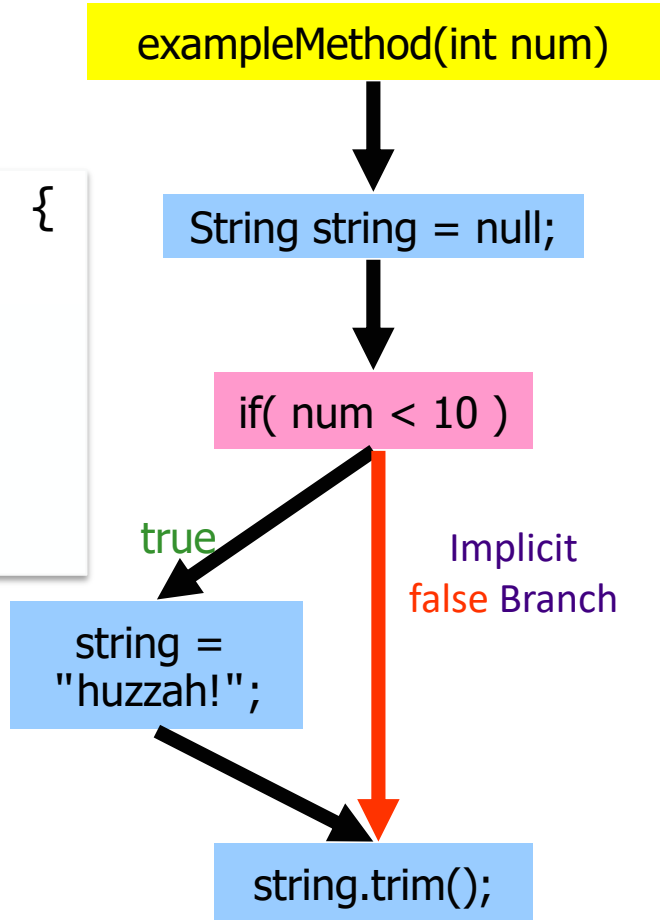
Test Suite: num=5

```
   public String exampleMethod(int num) {
✓ 1    String string = null;
✓ 2    if (num < 10) {
✓ 3        string = "huzzah!";
       }
       // remove leading & trailing whitespace
✓ 4    return string.trim();
   }
```

Is this method bug-free?

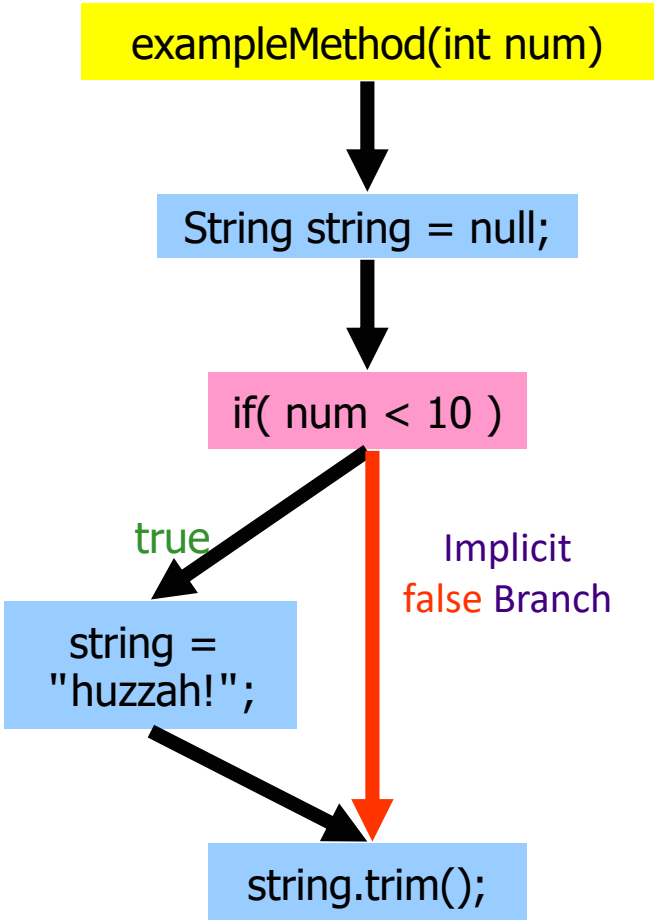
Program Flow

```
public String exampleMethod(int num) {
    String string = null;
    if (num < 10) {
        string = "huzzah!";
    }
    return string.trim();
}
```



What Went Wrong?

- Test suite had 100% statement coverage but missed a **branch/edge**
- Try covering all **edges** in program's flow
  - Also covers all **nodes**
  - Called **Branch Coverage**



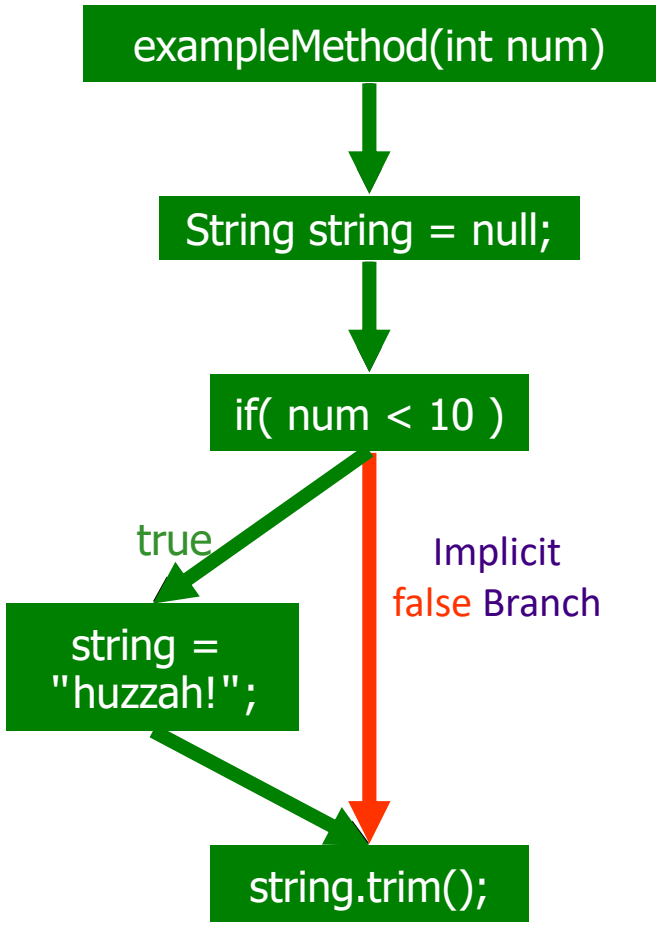
Branch Coverage

- Cover all **branches** in the program

Test Suite:

num=5,

num=10



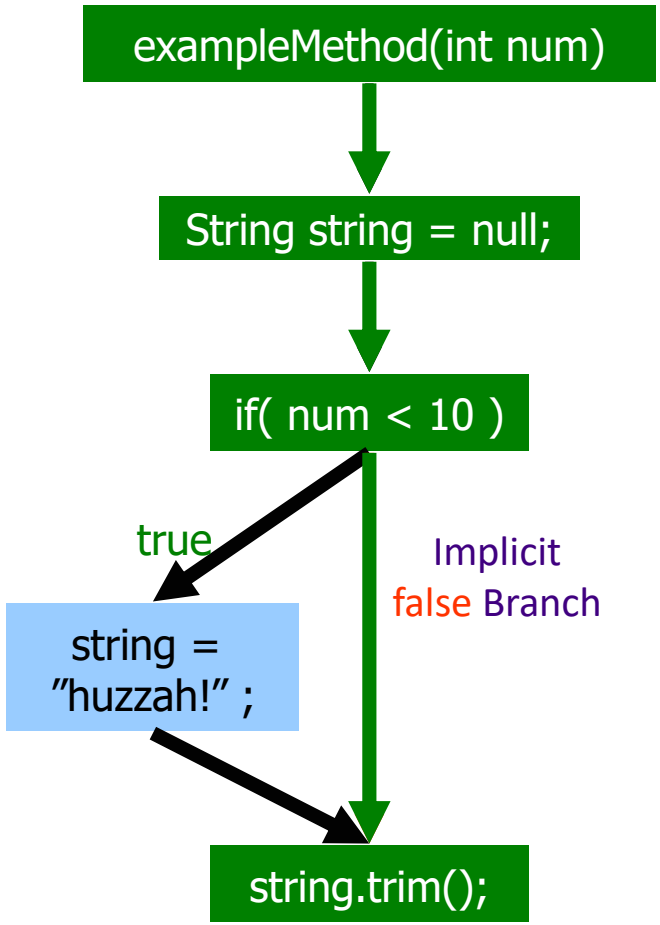
Branch Coverage

- Cover all **branches** in the program

Test Suite:

num=5,

num=10



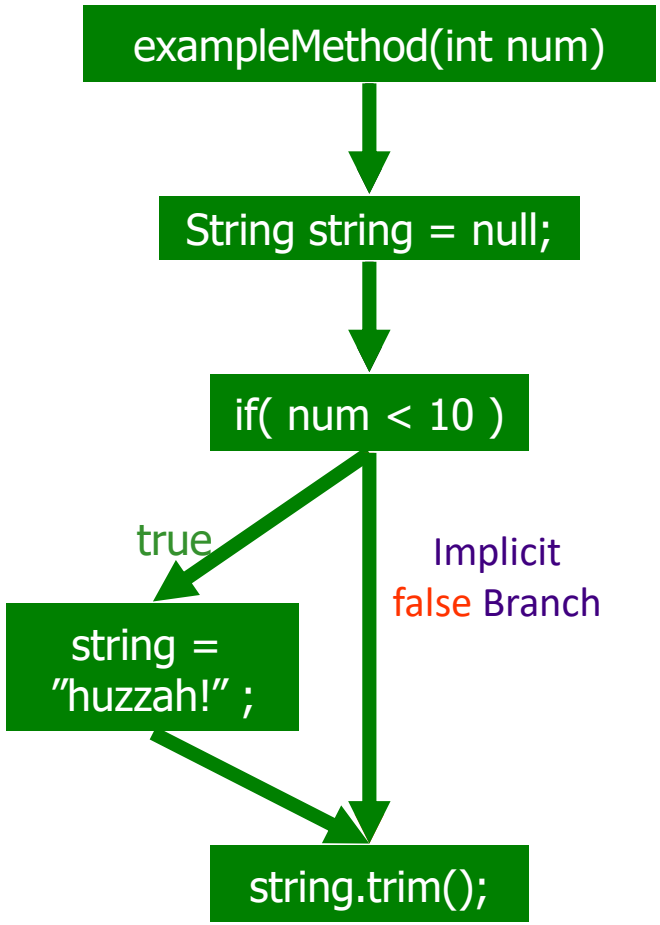
Branch Coverage

- Cover all **branches** in the program

Test Suite:

num=5,

num=10



Example 2

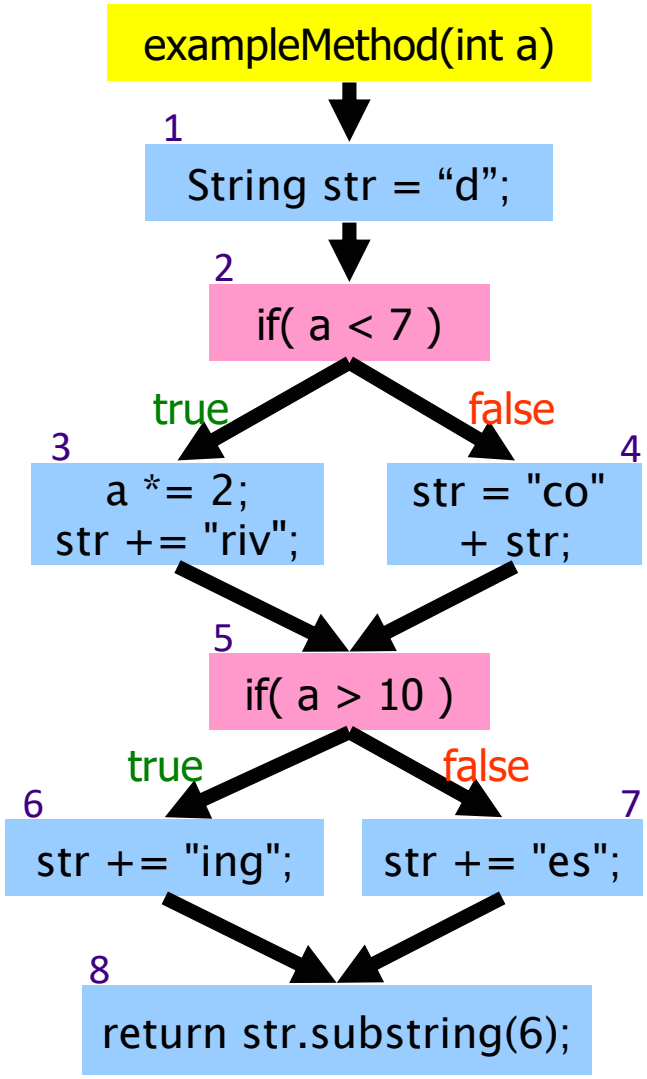
```
public static String exampleMethod(int a) {
   String str = "d";
   if ( a < 7 ) {
      a *= 2;
      str += "riv";
    } else {
      str = "co" + str;
    }

    if( a > 10 ) {
      str += "ing";
    } else {
      str += "es";
    }
   return str.substring(6);
}
```

Example 2

```
public String exampleMethod(int a) {
   String str = "d";
   if ( a < 7 ) {
      a *= 2;
      str += "riv";
    } else {
      str = "co" + str;
    }

    if( a > 10 ) {
      str += "ing";
    } else {
      str += "es";
    }
   return str.substring(6);
}
```

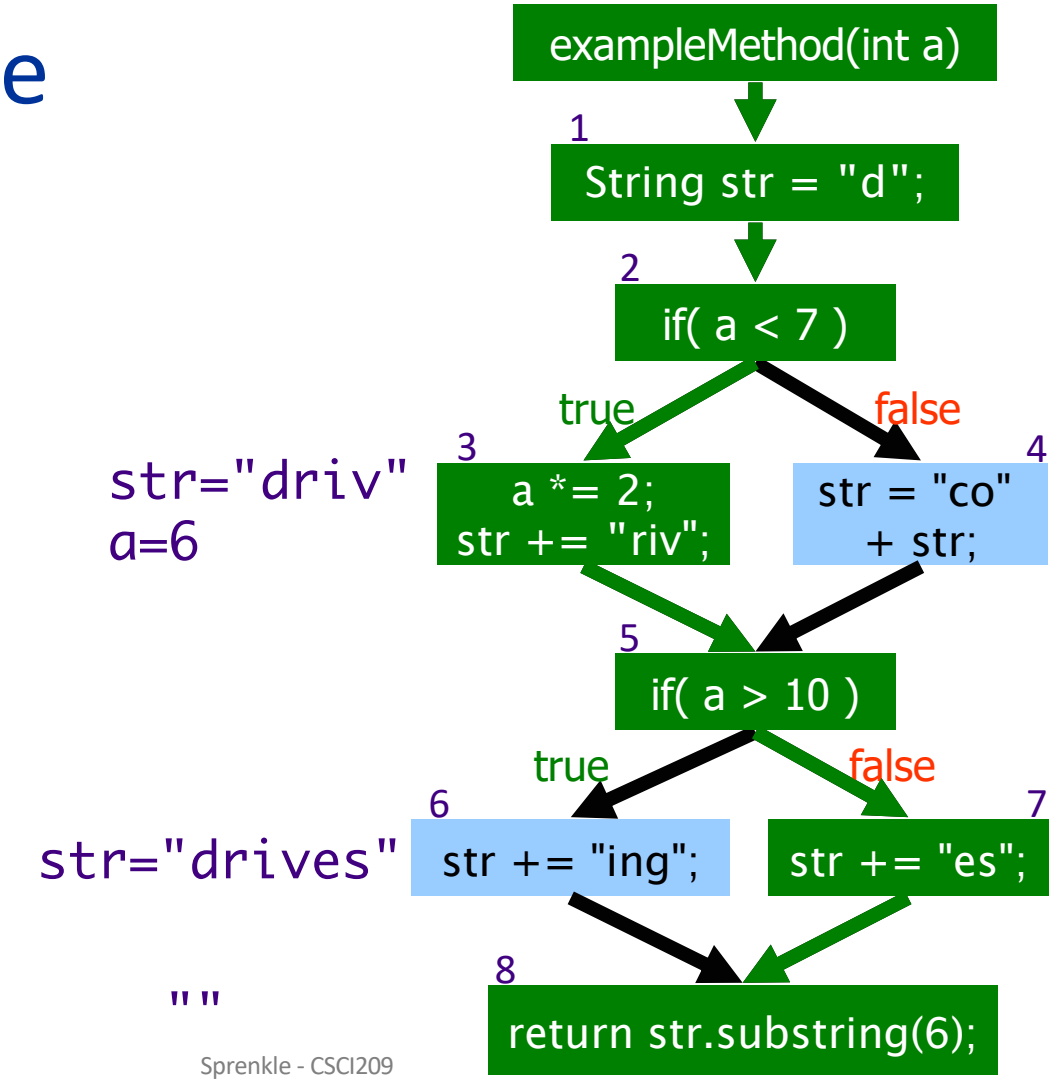


Branch Coverage

Test Suite:

a=3,

a=30

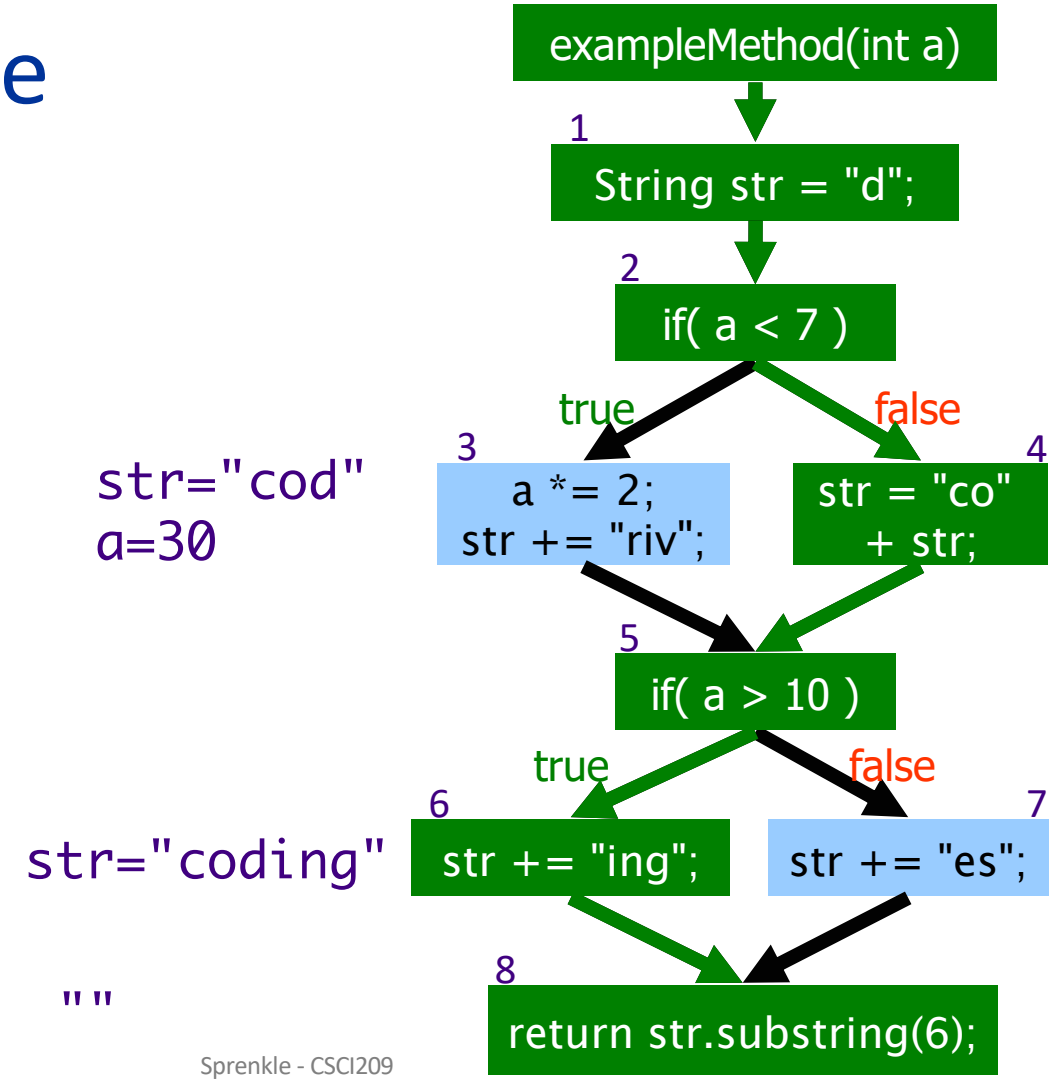


Branch Coverage

Test Suite:

a=3,

a=30



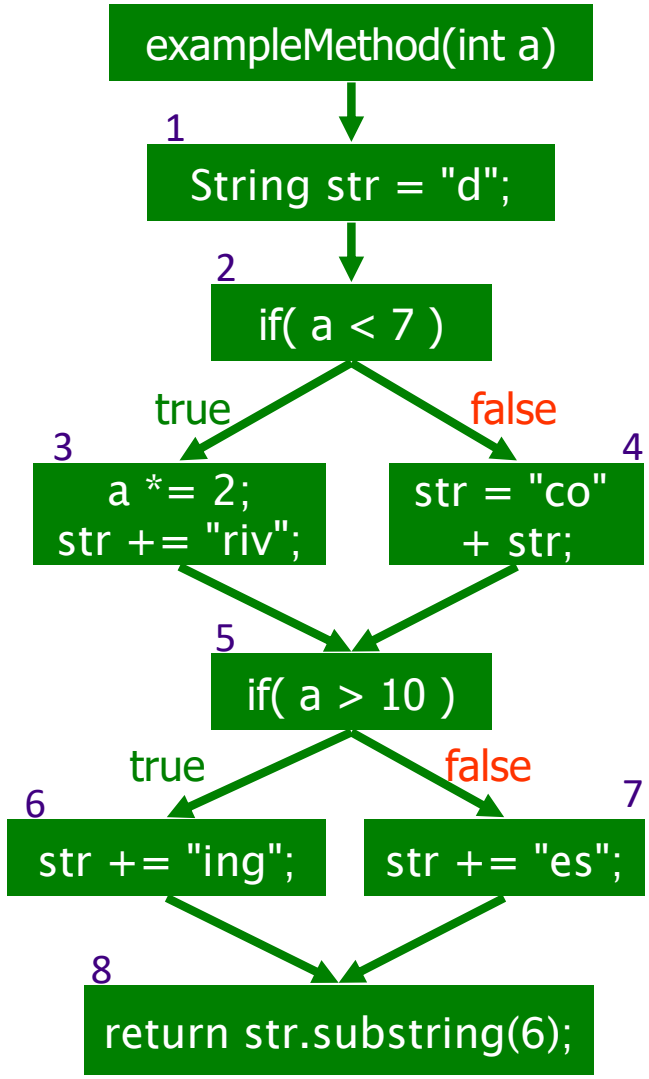
Branch Coverage

Test Suite:

a=3,

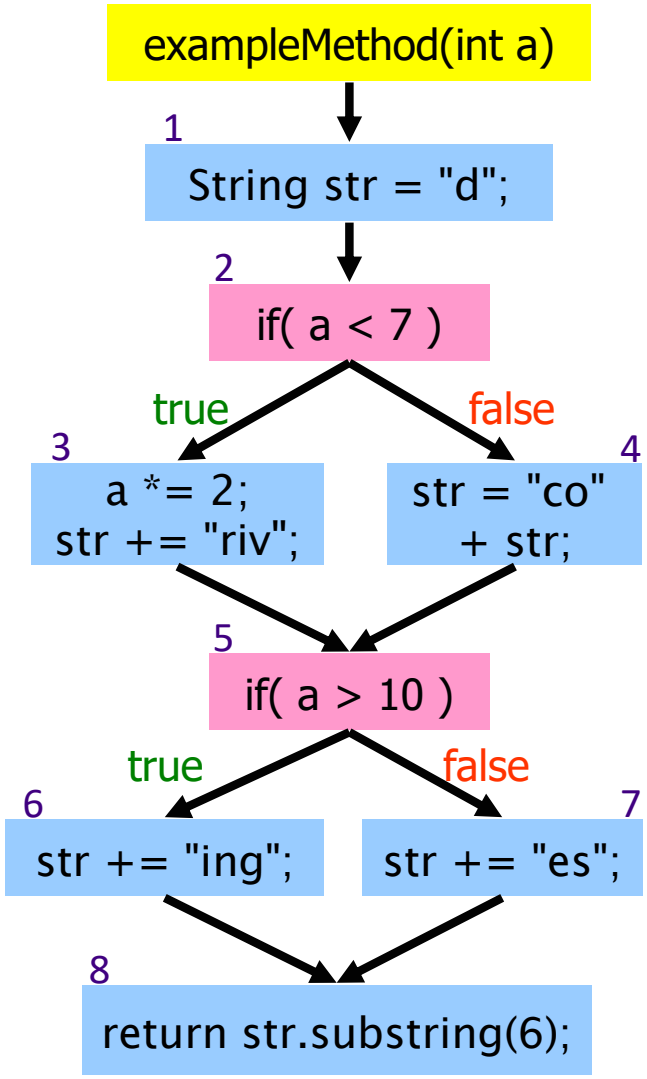
a=30

Is this method bug free?



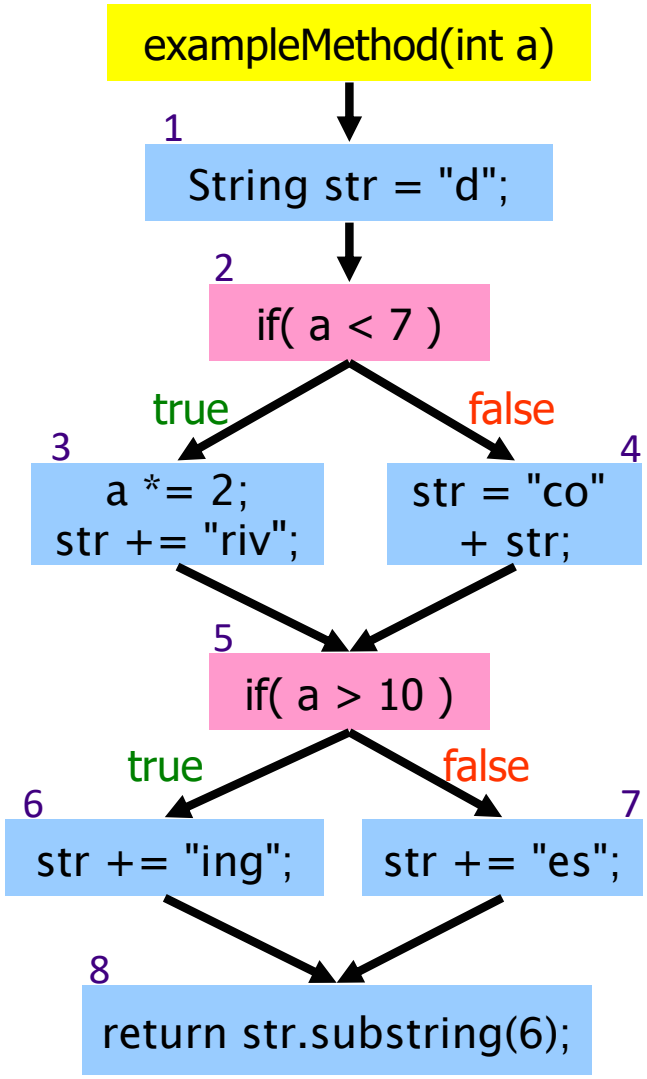
What Went Wrong?

- Test suite had 100% branch (and statement) coverage but missed a **path**
- Try to cover all **paths** in program's flow
  - Also gets all **branches, nodes**
  - Called **Path Coverage**



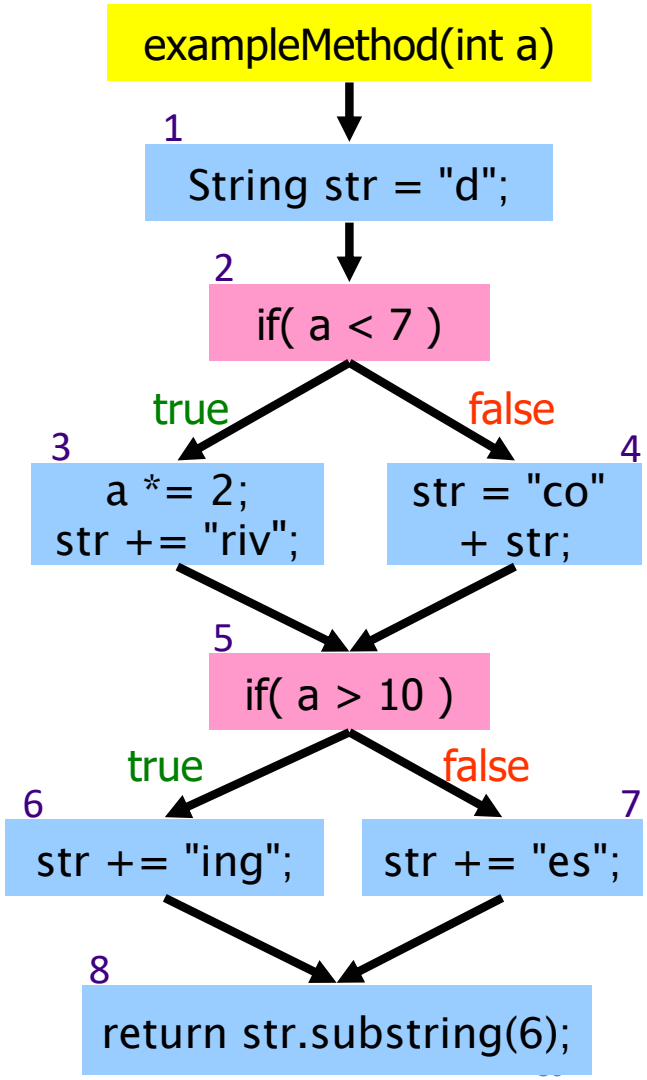
Path Coverage

- Cover all **paths** in program's flow
- How many paths through this method?
- What test cases would give us path coverage?



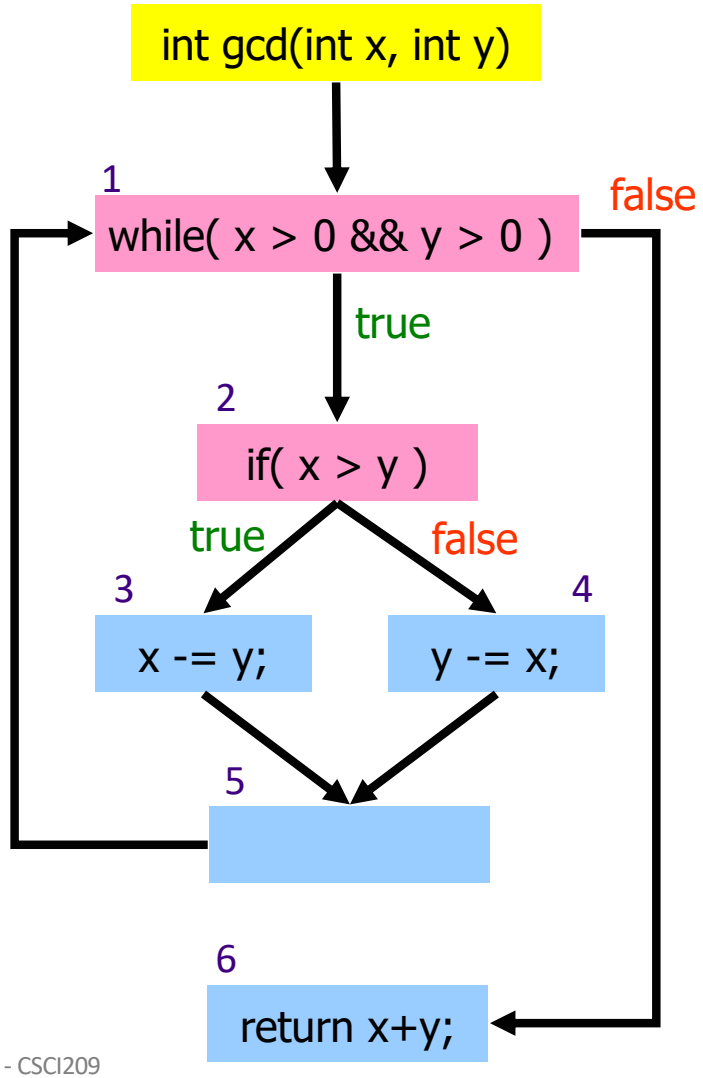
Path Coverage

- Cover all **paths** in program's flow
- How many paths through this method? 4
  - 1-2-**3**-5-**6**-8
  - 1-2-**3**-5-**7**-8
  - 1-2-**4**-5-**6**-8
  - 1-2-**4**-5-**7**-8
- What test cases would give us path coverage?
  - One possibility: a = 3, 30, 6, 10



Example 3

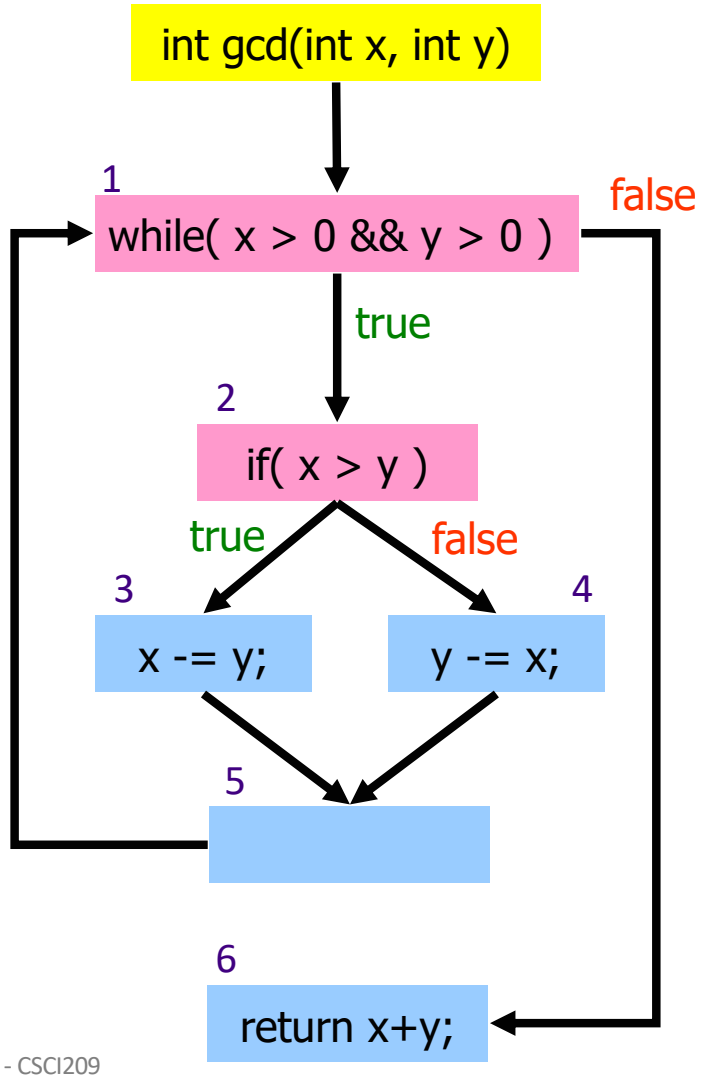
```
/**
 * Euclid's algorithm to calculate
 * greatest common divisor
 */
public int gcd( int x, int y ) {
    while ( x > 0 && y > 0 ) {
        if( x > y ) {
                x -=y ;
        } else {
                y -=x;
        }
    }
    return  x+y;
}
```



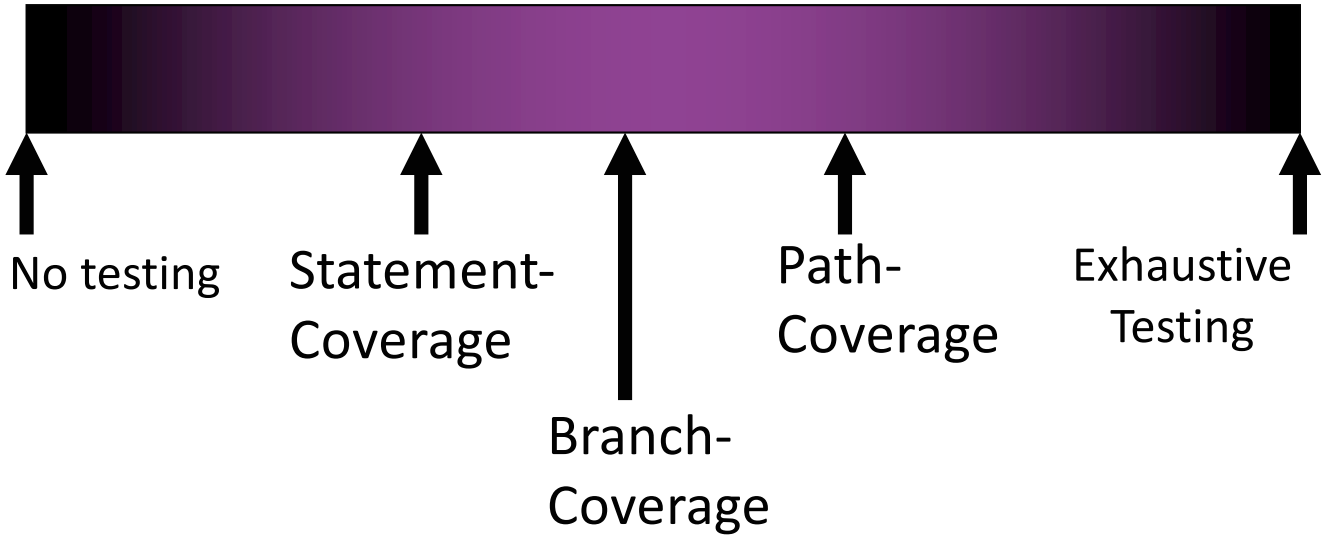
Path Coverage

- How many paths through this method?
  - Too many to count, test them all!

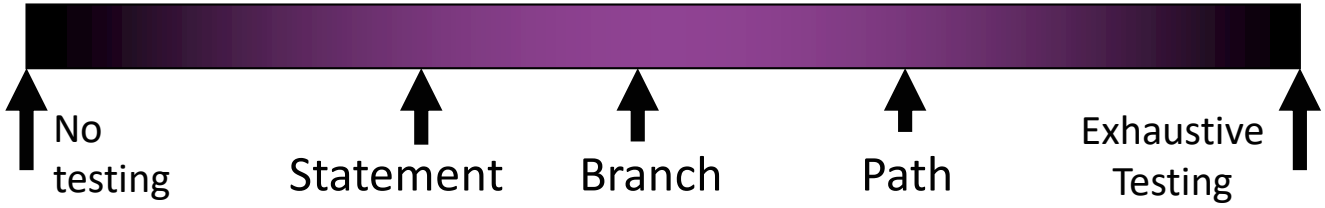
1-6
1-2-3-5-1-6
1-2-4-5-1-6
1-2-3-5-1-2-3-5-1-6
1-2-4-5-1-2-4-5-1-6
1-[2-(3|4)-5-1]*-6



Testing Continuum



Comparison of Coverage Criteria



Coverage Criterion	Advantages	Disadvantages
Statement		
Branch		
Path		

Looking Ahead

- Wednesday: Testing project