SENG 275

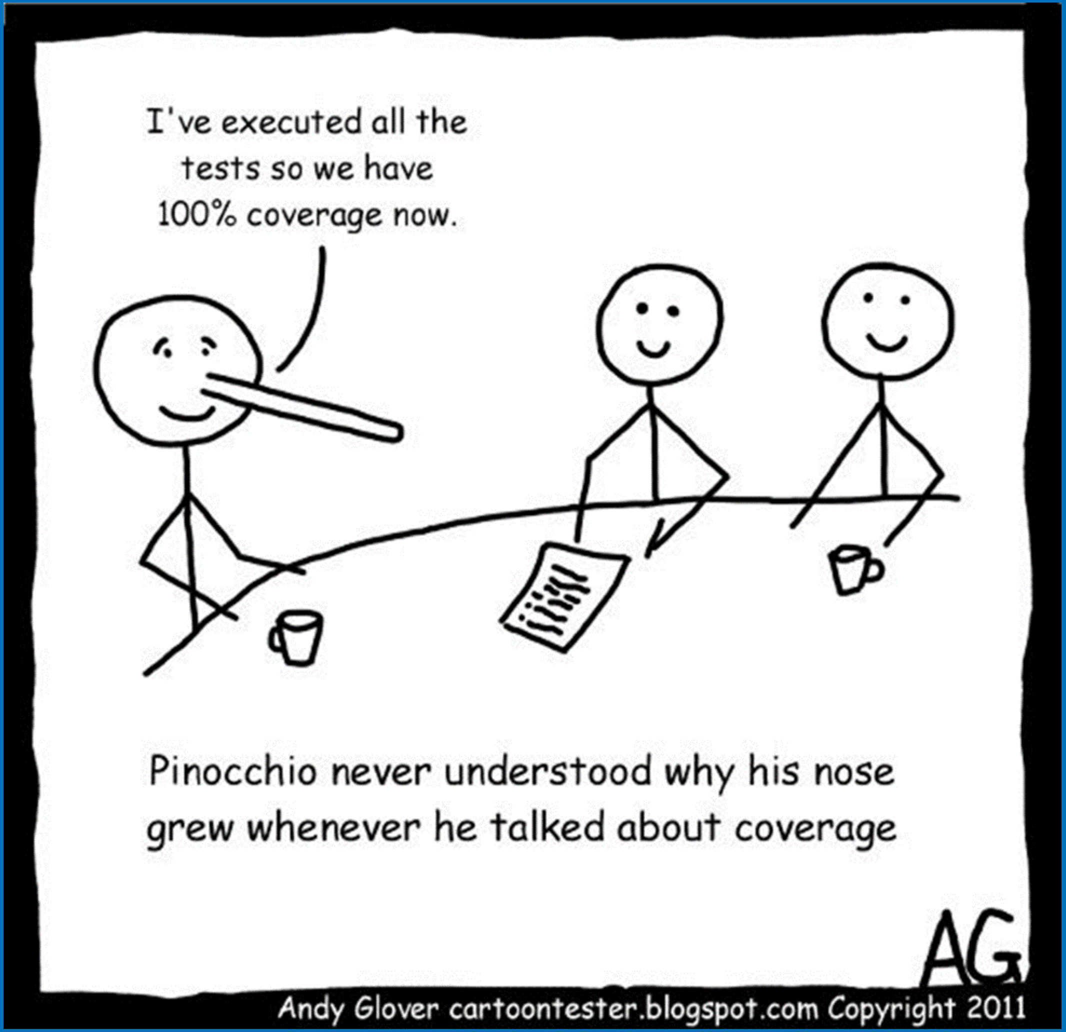
SOFTWARE TESTING

DR. NAVNEET KAUR POPLI

DEPT. OF ELECTRICAL AND COMPUTER ENGINEERING

University of Victoria

STRUCTURAL TESTING


I've executed all the tests so we have 100% coverage now.
Pinocchio never understood why his nose grew whenever he talked about coverage
AG
Andy Glover cartoontester.blogspot.com Copyright 2011

University of Victoria

Structural Testing

- Previously we discussed how to test software using **requirements** as the main element to guide the testing (Black box testing).
- Now, we will use the source code itself as a source of information to create tests (white box testing).
- Techniques that use the **structure of the source code** as a way to guide the testing, are called **structural testing techniques**.
- Understanding structural testing techniques means understanding the different **coverage criteria**.
- These coverage criteria relate closely to test coverage, a concept that many developers know.
- By **test coverage, we mean the amount (or percentage) of production code that is exercised by the tests.**

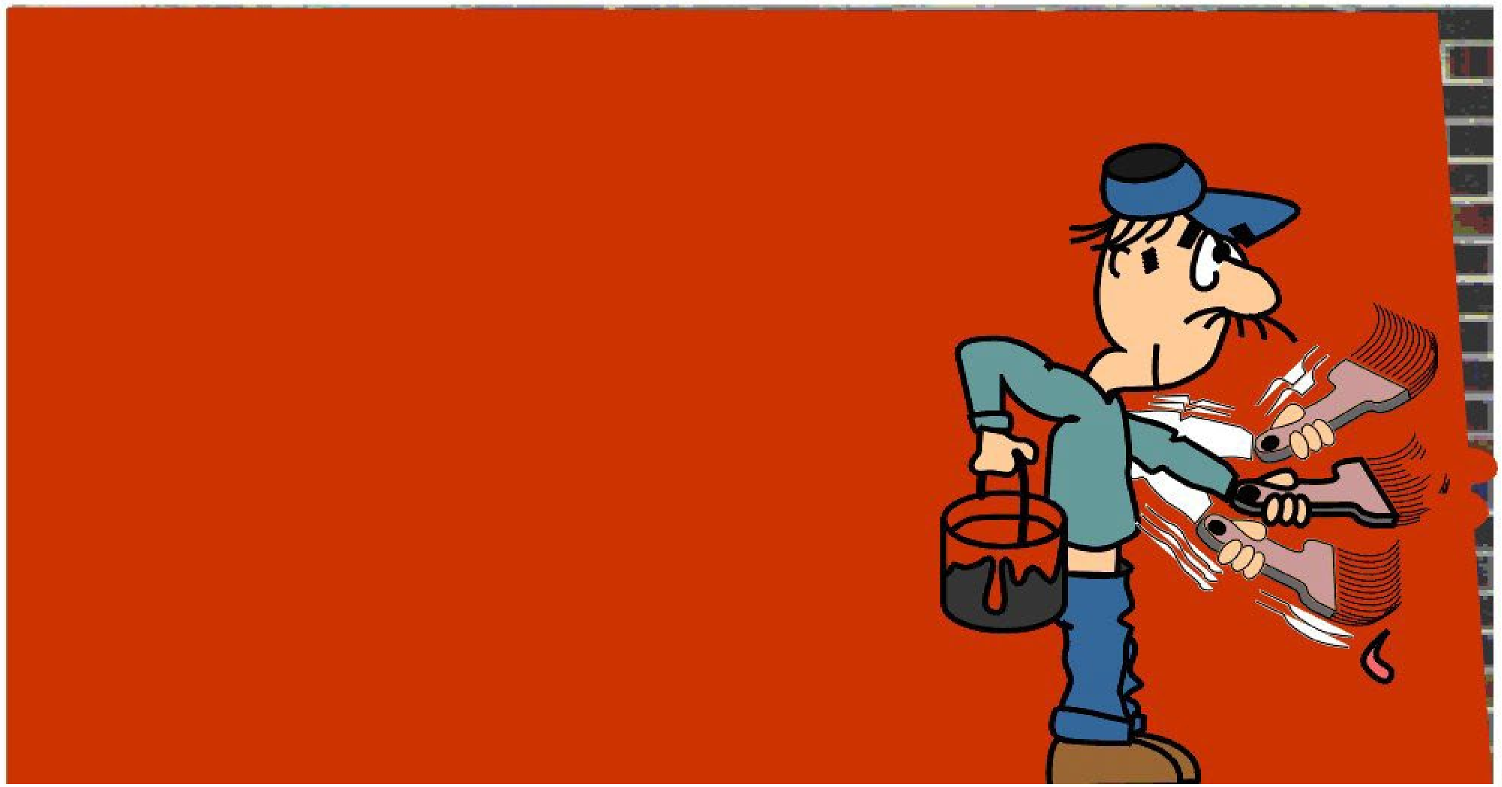
Structural testing should complement your requirements-based testing

- The first step of a tester should be to derive test cases out of any requirements-based technique.
- Once requirements are fully covered, testers then perform structural testing to cover what is missing from the structural-point of view.
- Any divergences should be brought back to the requirements-based testing phase: why did we not find this class/partition before?
- Once requirements and structure are covered, one can consider the testing phase done.

•Code Coverage:

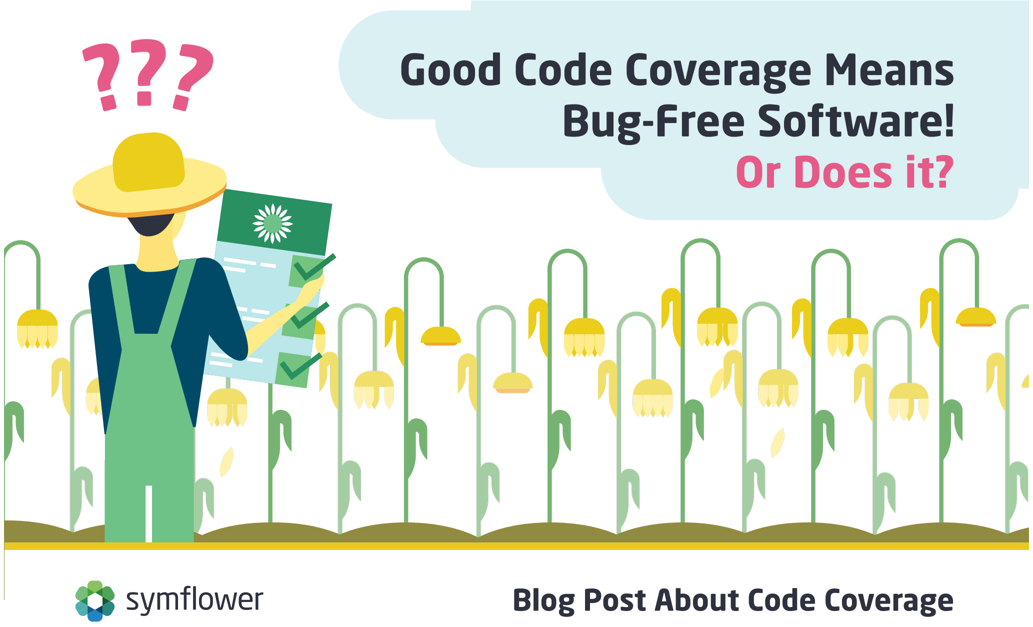
Percentage of source code that has been tested with respect to the total source code available for testing

Introduced by Miller and Maloney in 1963



Type of coverage criteria

- Line coverage (and statement coverage)
- Block coverage
- Branch/Decision coverage
- Condition (Basic and Condition + Branch) coverage
- Path coverage
- MC/DC coverage


???
Good Code Coverage Means Bug-Free Software!
Or Does it?
symflower
Blog Post About Code Coverage

University of Victoria

Why do we need structural testing?

To systematically derive tests from source code;

Why do we need structural testing?

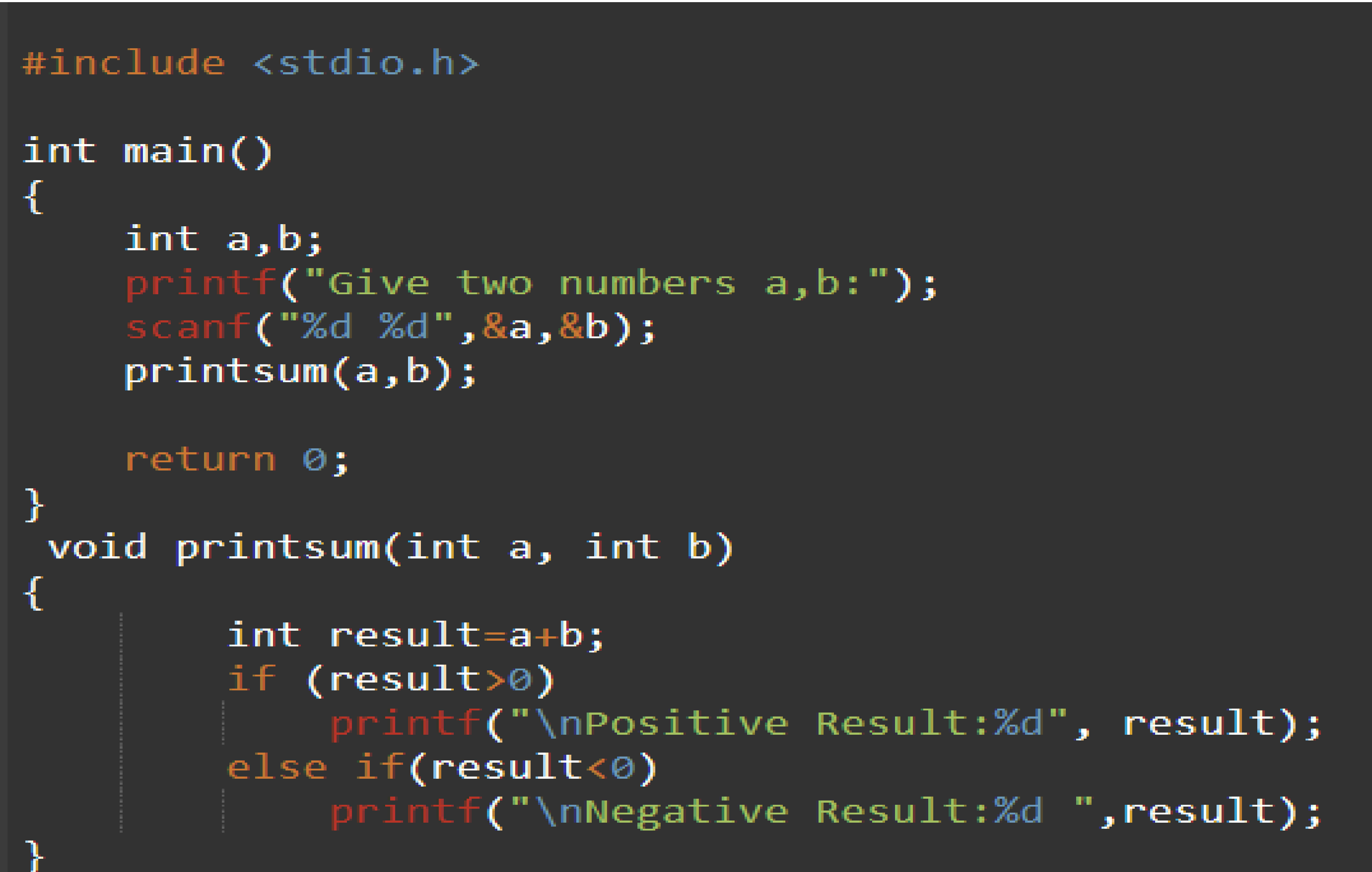
1. To systematically derive tests from source code;
2. To know when to stop testing;

- As a tester, when performing specification-based testing, your goal was clear: to derive classes out of the requirement specifications, and then to derive test cases for each of the classes. **You were satisfied once all the classes and boundaries were systematically exercised**.
- The same idea applies to structural testing. First, it gives us a systematic way to devise tests. As we will see, a tester might focus on testing **all the lines** of a program; or focus on the **branches** and **conditions** of the program. Different criteria produce different test cases.
- Second, to know when to stop. It is easy to imagine that the number of possible paths in a mildly complex piece of code is just too large, and exhaustive testing is impossible. Therefore, having clear criteria on when to stop helps testers in understanding the costs of their testing.

Advantages of structural testing

- Based on the code
  - Can be measured objectively
  - Can be measured automatically
- It verifies that the code function is working as intended.
- Reveals errors in "**hidden**" code
- It checks for **vulnerabilities** in the code.
- Spots the **Dead Code** or other issues with respect to best programming practices.
- Allows for **covering** the coded behaviour.

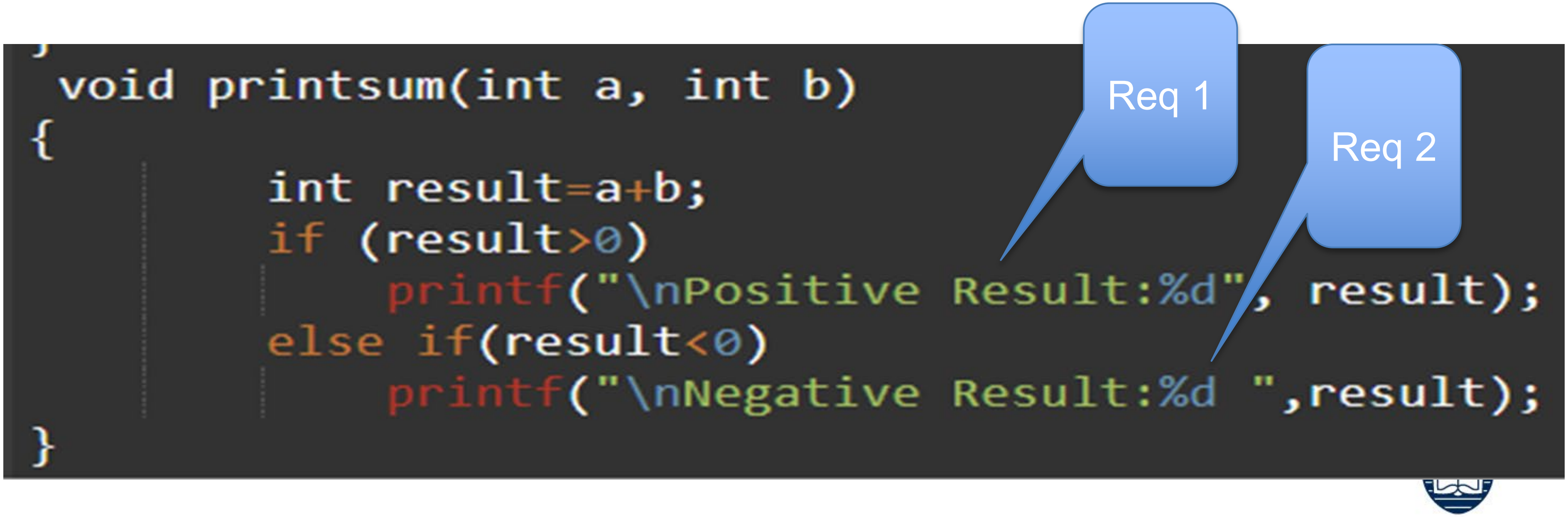
Add two numbers- develop test cases for printsum()



```c
#include <stdio.h>

int main()
{
    int a,b;
    printf("Give two numbers a,b:");
    scanf("%d %d",&a,&b);
    printsum(a,b);

    return 0;
}
 void printsum(int a, int b)
{
        int result=a+b;
        if (result>0)
            printf("\nPositive Result:%d", result);
        else if(result<0)
            printf("\nNegative Result:%d ",result);
}
```

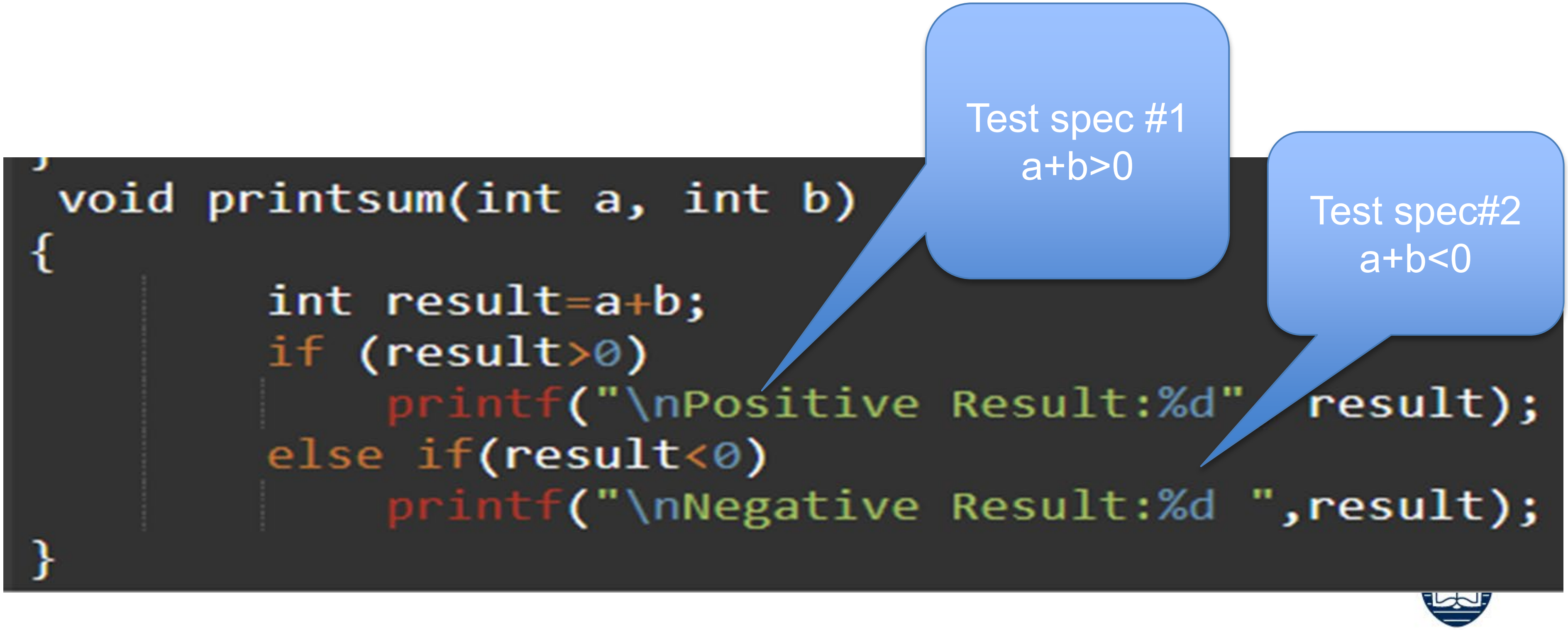
Coverage criteria

- Are defined in terms of
  - Test requirements
- Result in
  - Test specifications
  - Test cases



```
}
 void printsum(int a, int b)
{
        int result=a+b;
        if (result>0)
            printf("\nPositive Result:%d", result);
        else if(result<0)
            printf("\nNegative Result:%d ",result);
}
```

Test specifications- how do we get these statements executed?

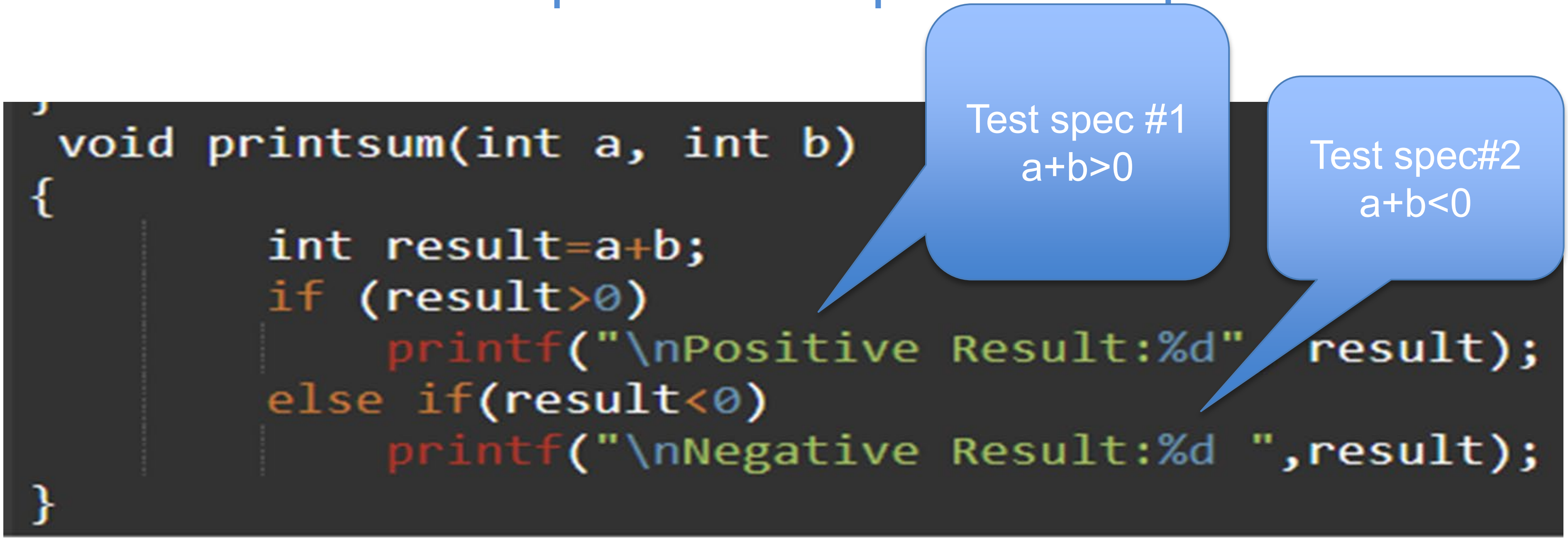


```
}
 void printsum(int a, int b)
{
        int result=a+b;
        if (result>0)
            printf("\nPositive Result:%d"  result);
        else if(result<0)
            printf("\nNegative Result:%d ",result);
}
```

Test spec #1
a+b>0

Test spec#2
a+b<0

Test cases- set of inputs and expected outputs



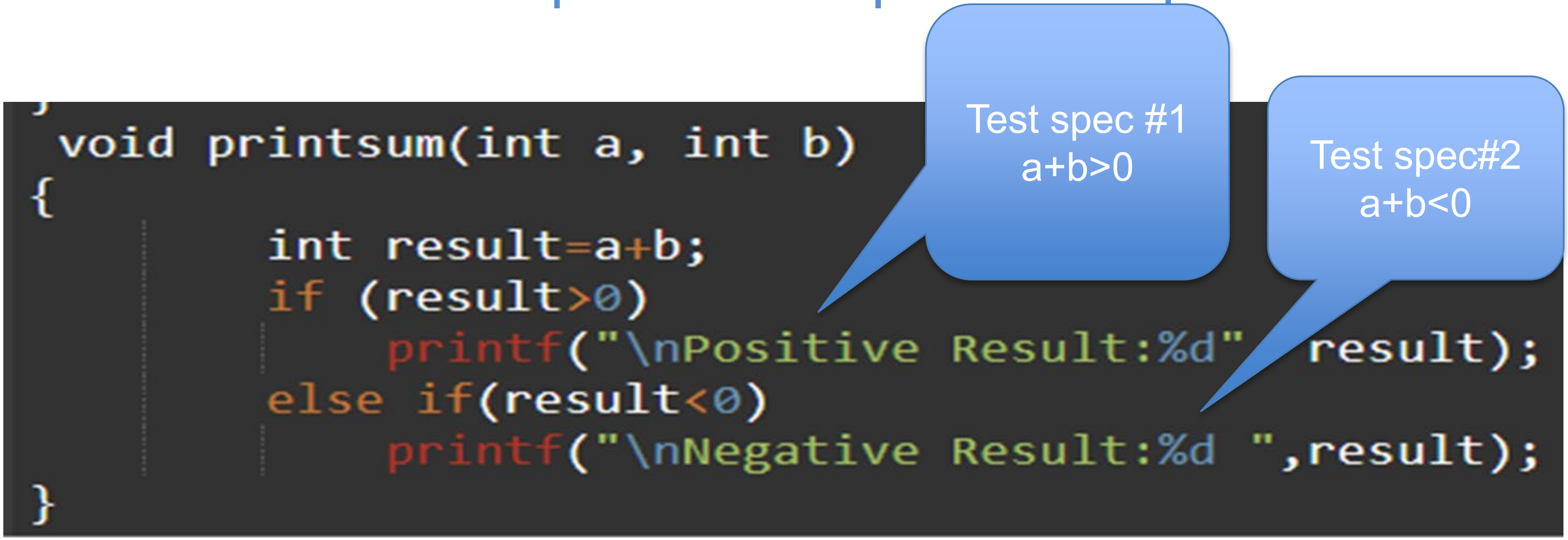
```
 void printsum(int a, int b)
{
        int result=a+b;
        if (result>0)
            printf("\nPositive Result:%d"   result);
        else if(result<0)
            printf("\nNegative Result:%d ",result);
}
```

Test spec #1
a+b>0

Test spec#2
a+b<0

1. ((a=[ ], b=[ ]),(output value=[ ]))
2. ((a=[ ], b=[ ]),(output value=[ ]))

Test cases- set of inputs and expected outputs



```
void printsum(int a, int b)
{
        int result=a+b;
        if (result>0)
            printf("\nPositive Result:%d"   result);
        else if(result<0)
            printf("\nNegative Result:%d ",result);
}
```

Test spec #1
a+b>0

Test spec#2
a+b<0

1. ((a=[3], b=[9]),(output value=[12]))
2. ((a=[-5], b=[-8]),(output value=[-13]))

LINE/STATEMENT COVERAGE

Line Coverage

- Percentage of source code lines executed by test cases.
  - For developer easiest to work with
  - Precise percentage depends on layout?
    - var x = 10; if (z++ < x) y =x+z;
  - Requires mapping back from binary?
- In practice, coverage not based on lines, but on *control flow graph*

Statement coverage

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```
void foo (z) {
 var x = 10;
 if (z++ < x) {
   x=+ z;
 }
}
```

```
@Test
void testFoo() {
  foo(10);
}
```

- Coverage:

$$\frac{\text{\# executed statements}}{\text{\# statements}}$$

Statement coverage

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```
void foo (z) {
 var x = 10;
 if (z++ < x) {
   x=+ z;
 }
}
```

```
@Test
void testFoo() {
  foo(10);
}
```

- Coverage:

$$\frac{\#\text{ executed statements}}{\#\text{ statements}}$$

Statement coverage

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```
void foo (z) {
 var x = 10;
 if (z++ < x) {
   x=+ z;
 }
}
```

```
@Test
void testFoo() {
  foo(5);
}
// 100% statement coverage
```

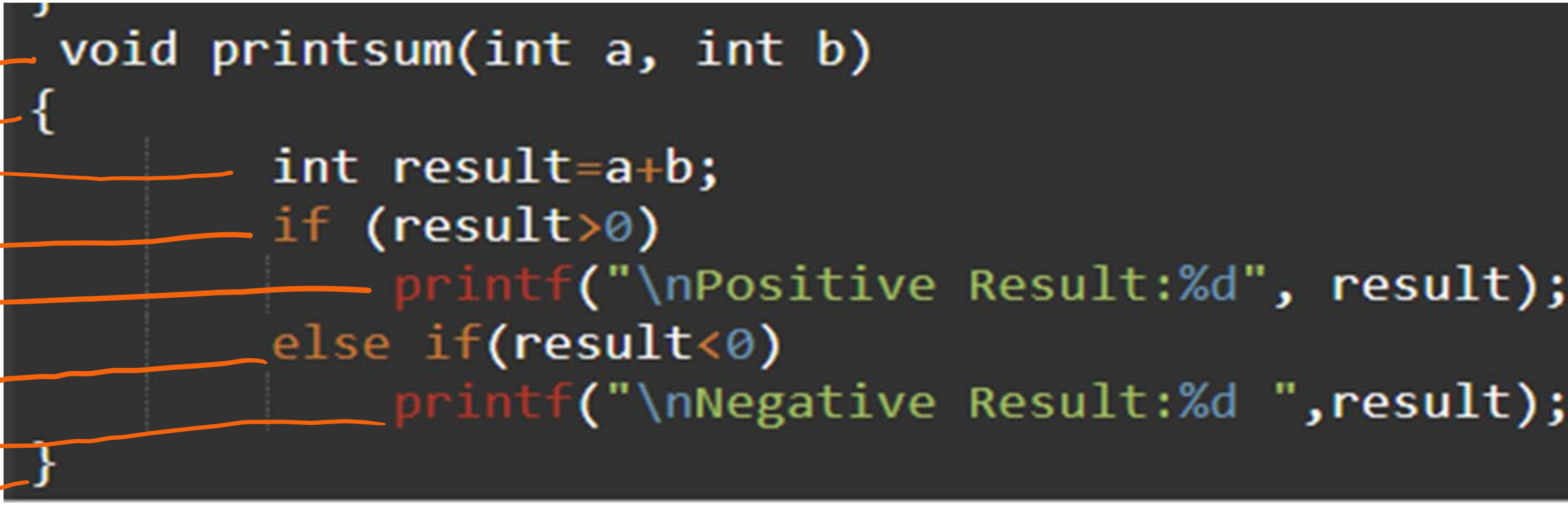
- Coverage:

$$\frac{\#\text{ executed statements}}{\#\text{ statements}}$$

Line(and Statement) coverage

- **Test requirements**: statements in the program
- **Coverage measure**: $$\frac{\text{number of executed statements}}{\text{total number of statements}}$$
- As the name suggests, when determining the line coverage, we look at the number of lines of code that are covered by the tests (more specifically, by at least one test).

Goal is 100% statement coverage



```
1  void printsum(int a, int b)
2  {
3        int result=a+b;
4        if (result>0)
5            printf("\nPositive Result:%d", result);
6        else if(result<0)
7            printf("\nNegative Result:%d ",result);
8  }
```

TC # 1
a=3
b=9

Executes statements
1,2,3,4,5,8 = 6/8 = 75% coverage

TC # 2
a= -5
b= -8

→ Executes statements
1,2,3,4,6,7,8

So together both give 100% coverage.

Let’s take another example: Requirement: Black-jack

- The program receives the number of points of two blackjack players.
- The **program must return the number of points of the winner**. In blackjack, whoever gets **closer to 21 points wins**.
- If a player goes **over 21 points, the player loses**. If both players lose, the program must return 0.

Implementation

```
public class BlackJack {
  public int play(int left, int right) {
1.  int ln = left;
2.  int rn = right;
3.  if (ln > 21)
4.    ln = 0;
5.  if (rn > 21)
6.    rn = 0;
7.  if (ln > rn)
8.    return ln;
9.  else
10.   return rn;
  }
}
```

For 100% line coverage, test cases:

- (ln=, rn=), lines:
- (ln=, rn=), lines:

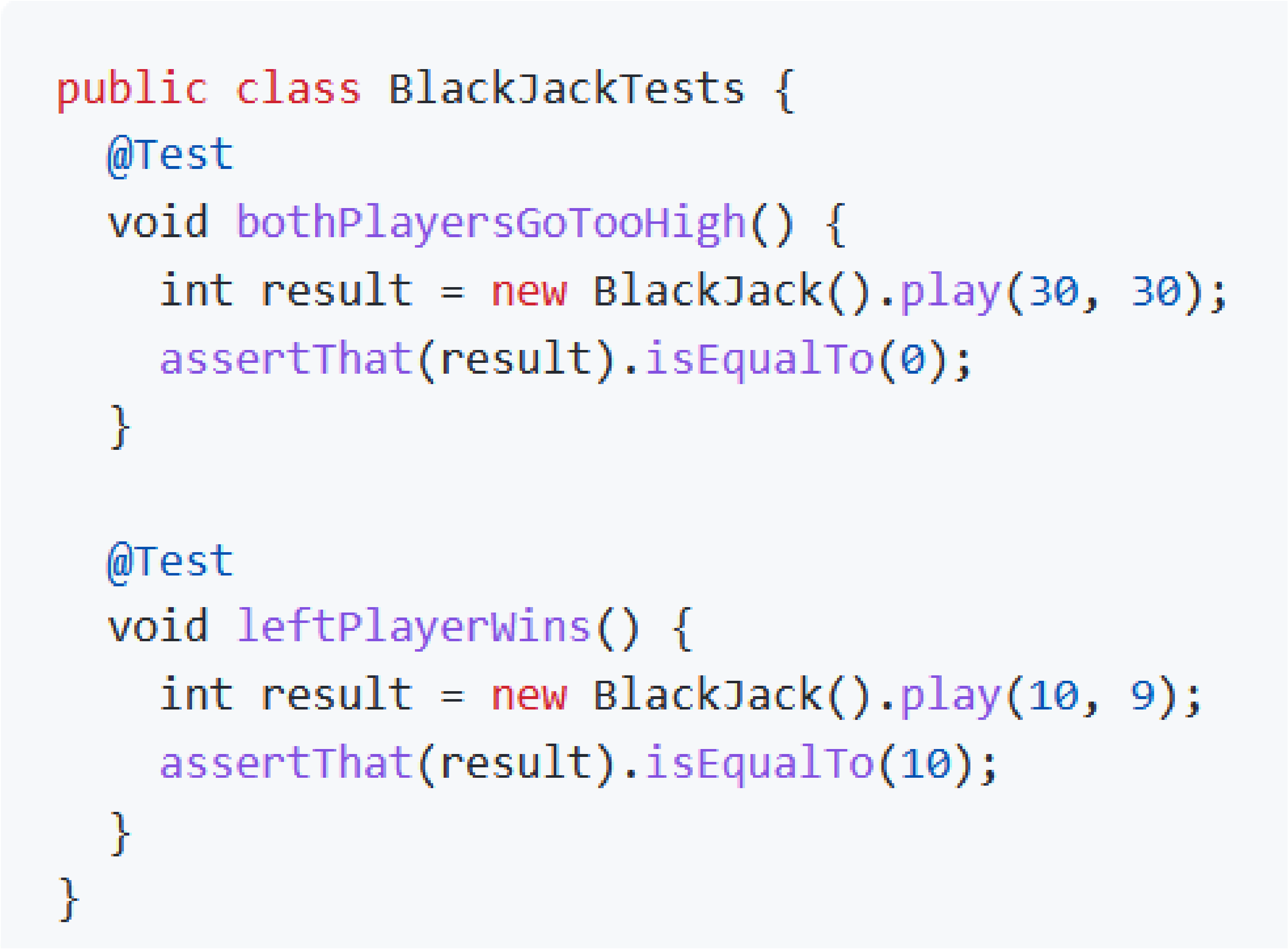
Implementation

```
public class BlackJack {
  public int play(int left, int right) {
1.  int ln = left;
2.  int rn = right;
3.  if (ln > 21)
4.    ln = 0;
5.  if (rn > 21)
6.    rn = 0;
7.  if (ln > rn)
8.    return ln;
9.  else
10.   return rn;
  }
}
```

For 100% line coverage, test cases:

- (ln=30, rn=30)-1,2,3,4,5,6,7,9,10
- (ln=10, rn=9)-1,2,3,5,7,8

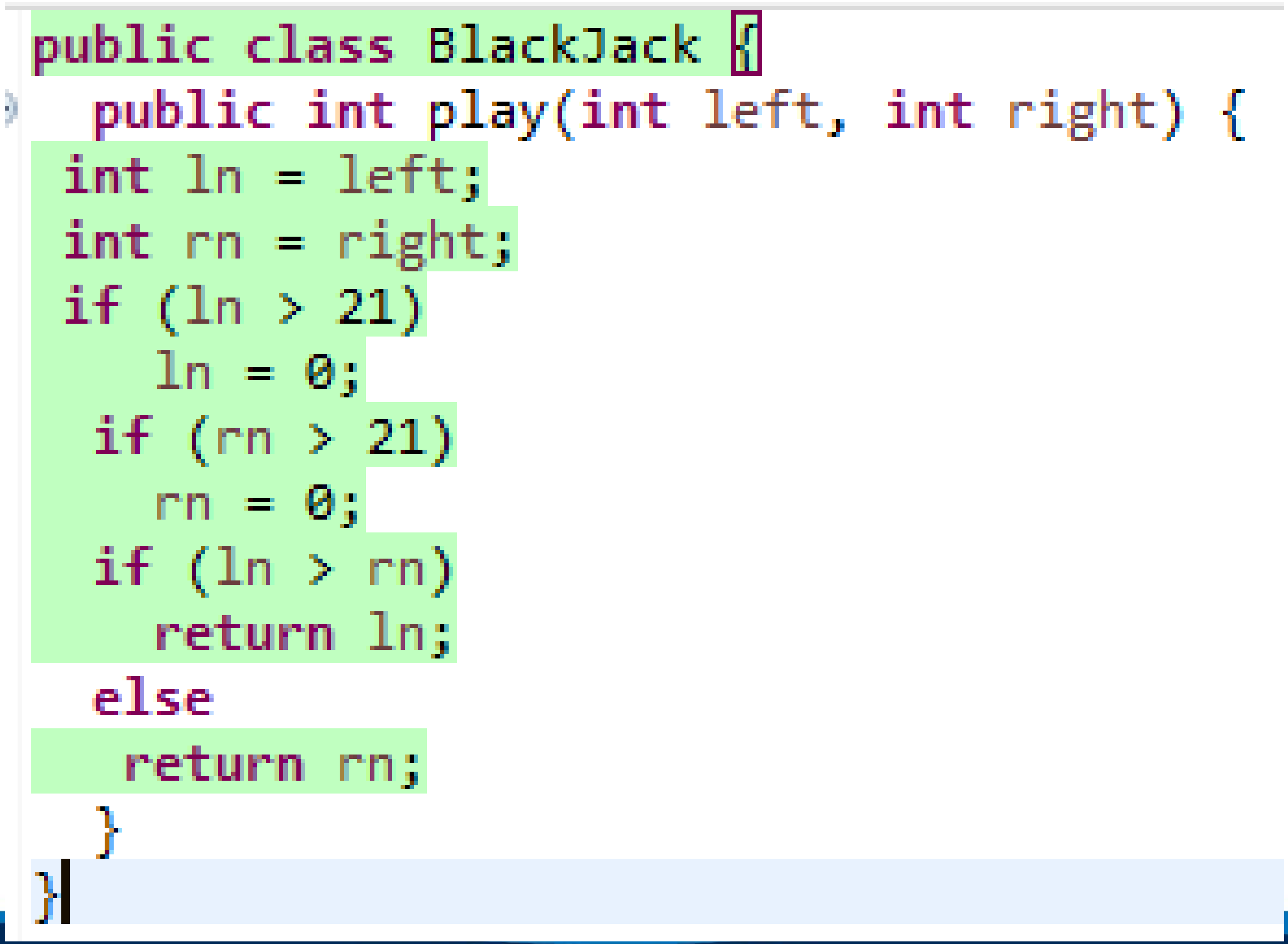
Devise and implement two test cases for this method:



```
public class BlackJackTests {
  @Test
  void bothPlayersGoTooHigh() {
    int result = new BlackJack().play(30, 30);
    assertThat(result).isEqualTo(0);
  }

  @Test
  void leftPlayerWins() {
    int result = new BlackJack().play(10, 9);
    assertThat(result).isEqualTo(10);
  }
}
```

University of Victoria

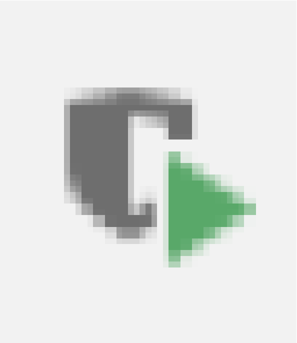
Visual Test coverage using TestNG and Jacoco



```
public class BlackJack {
  public int play(int left, int right) {
 int ln = left;
 int rn = right;
 if (ln > 21)
    ln = 0;
  if (rn > 21)
    rn = 0;
  if (ln > rn)
    return ln;
  else
   return rn;
  }
}
```

In IntelliJ

- Run... With Coverage feature of IntelliJ – click on the icon in the upper right that looks like a black-and-white shield with a green triangle pointing right (or choose Run... with Coverage from the Run menu at the top of the IDE).
- If these options are grayed out, run your tests first – they need to run normally once before we can Run With Coverage.



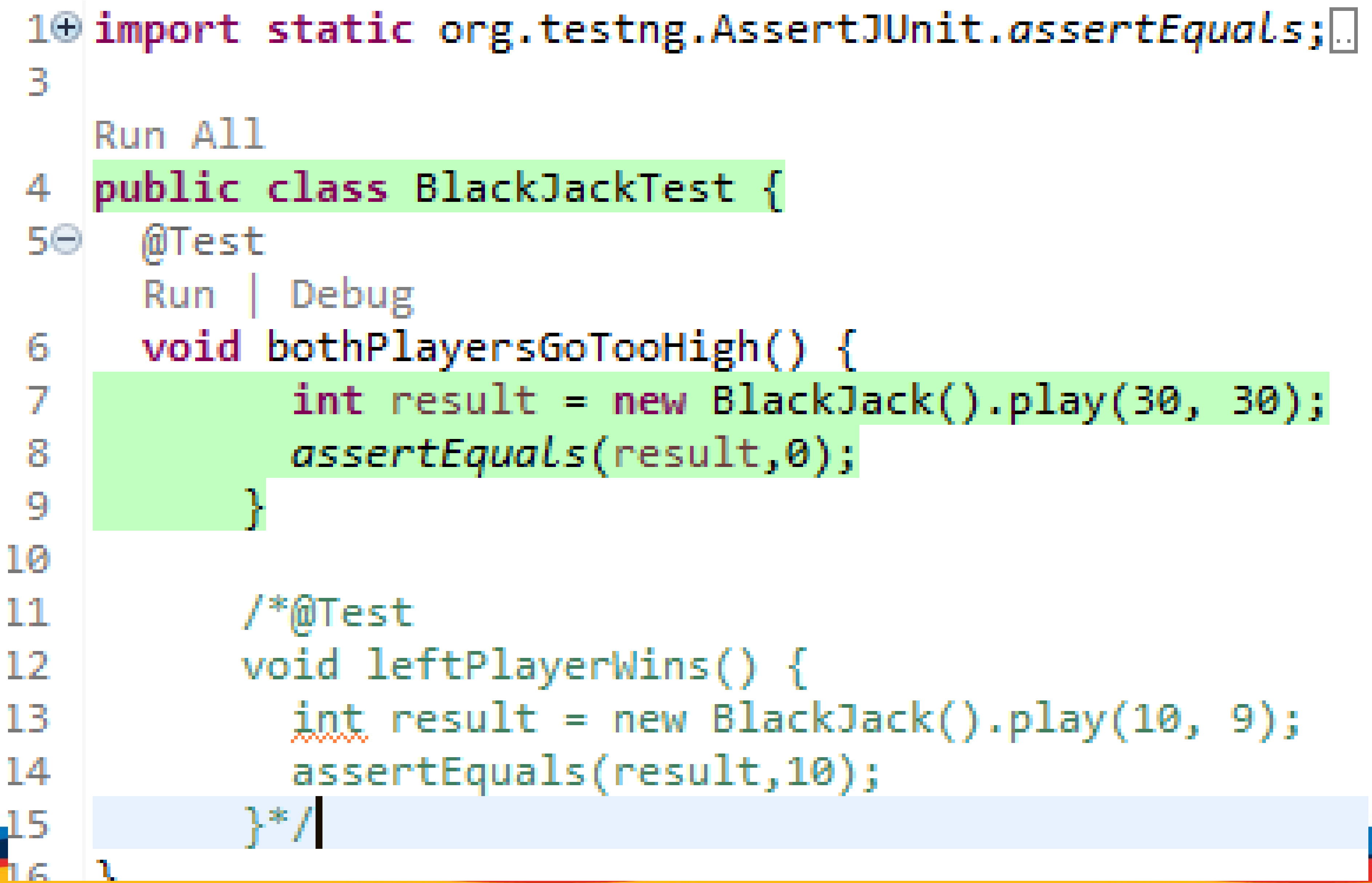
Coverage

- The first test executes lines 1-7, 9, and 10 as both values are higher than 21.
- This means that, after the bothPlayersGoTooHigh test, 9 out of the 10 lines are covered. Thus, line coverage is:

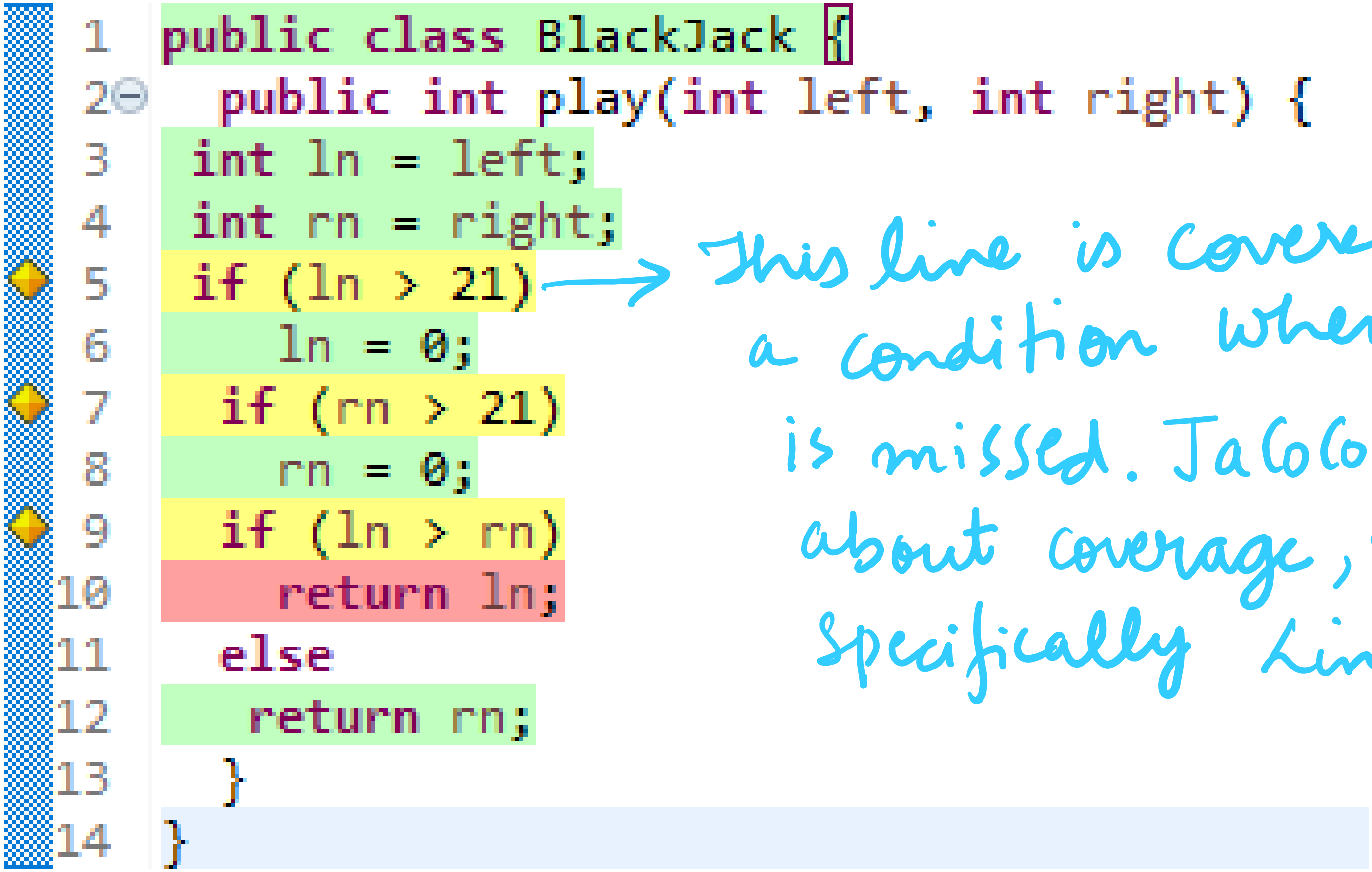
$$\frac{9}{10} \cdot 100$$

- Line 8 is therefore the only line that the first test does not cover. The second test, leftPlayerWins, complements the first test, and executes lines 1-3, 5, 7 and 8. Both tests together now achieve a line coverage of 100, as together they cover all the 10 different lines of the program.
- More formally, we can compute line coverage as:
- $$\text{line coverage} = \frac{\text{lines covered}}{\text{lines total}} \cdot 100$$

Remove one test

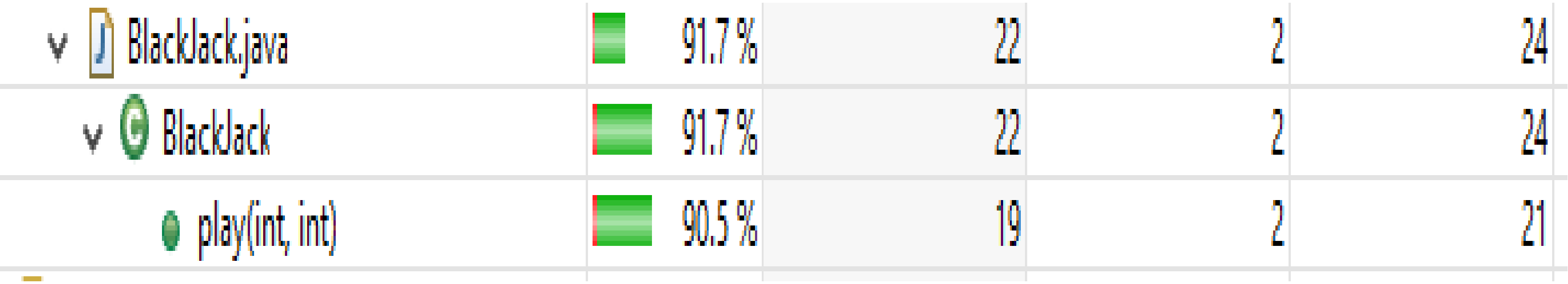


```
import static org.testng.AssertJUnit.assertEquals;

public class BlackJackTest {
  @Test
  void bothPlayersGoTooHigh() {
        int result = new BlackJack().play(30, 30);
        assertEquals(result,0);
      }

      /*@Test
      void leftPlayerWins() {
        int result = new BlackJack().play(10, 9);
        assertEquals(result,10);
      }*/
}
```



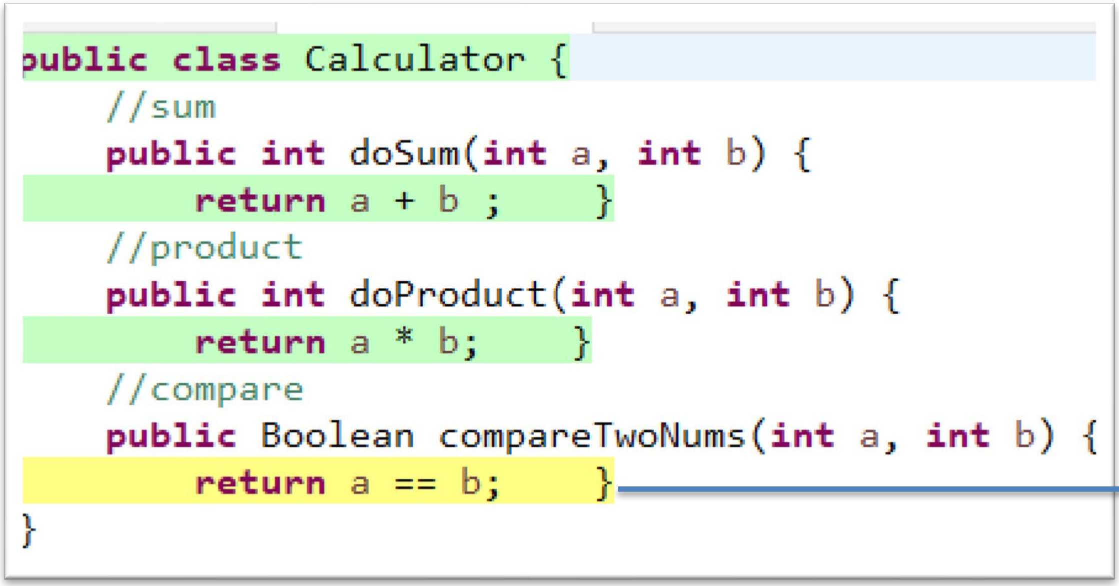
```
public class BlackJack {
  public int play(int left, int right) {
 int ln = left;
 int rn = right;
 if (ln > 21)
    ln = 0;
  if (rn > 21)
    rn = 0;
  if (ln > rn)
    return ln;
  else
   return rn;
  }
}
```

→ This line is covered but a condition when ln<20 is missed. JaCoCo talks about coverage, not specifically Line Coverage.



BlackJack.java	🟩	91.7 %	22	2	24
BlackJack	🟩	91.7 %	22	2	24
play(int, int)	🟩	90.5 %	19	2	21

85%coverage in Calculator class



```
public class Calculator {
    //sum
    public int doSum(int a, int b) {
        return a + b ;      }
    //product
    public int doProduct(int a, int b) {
        return a * b;      }
    //compare
    public Boolean compareTwoNums(int a, int b) {
        return a == b;      }
}
```

```
@Test
public void testCompareTrue()
{
boolean
actual=C.compareTwoNums(12,12);
assertTrue(actual);
System.out.println("The
Comparison is: "+actual);
}
```

Statement coverage in practice

- Mostly used in industry
- “Typical coverage” target is 80-90%

shutterstock.com · 1982860883

Statement coverage

```
static int test(int x, boolean c1, boolean c2)
{
1.  if (c1)
2.      System.out.println("In c1");
3. if ( c2)
4.      System.out.println("In c1");
5.  return x;
}
```

$$\text{Score} = \frac{\#\text{covered}}{\#\text{ statements}}$$

Q) What would be the test score for this test case:
test(0, false, false)

a) **5/5 (100%)**

b) **4/5 (80%)**

c) **3/5 (60%)**

d) **2/5 (40%)**

e) **1/5 (20%)**

Statement coverage

```
static int test(int x, boolean c1, boolean c2)
{
  if (c1)
      System.out.println(“In c1”);
  if ( c2)
      System.out.println(“In c1”);
  return x;
}
```

$$\text{Score} = \frac{\text{\#covered}}{\text{\# statements}}$$

Q) What would be the test score for this test case:
test(0, false, false)

a) 5/5 (100%)

b) 4/5 (80%)

c) 3/5 (60%)

d) 2/5 (40%)

e) 1/5 (20%)

Statement coverage

```
static int test(int x, boolean c1, boolean c2)
{
  if (c1)
      System.out.println(“In c1”);
  if ( c2)
      System.out.println(“In c1”);
  return x;
}
```

$$\text{Score}= \frac{\#\text{covered}}{\#\text{ statements}}$$

Q) What would be the test score for this test case:
test(0, true, true)

a) **5/5 (100%)**

b) **4/5 (80%)**

c) **3/5 (60%)**

d) **2/5 (40%)**

e) **1/5 (20%)**

Statement coverage

```
static int test(int x, boolean c1, boolean c2)
{
  if (c1)
      System.out.println("In c1");
  if ( c2)
      System.out.println("In c1");
  return x;
}
```

$$\text{Score}= \frac{\#\text{covered}}{\#\text{ statements}}$$

Q) What would be the test score for this test case:
test(0, true, true)

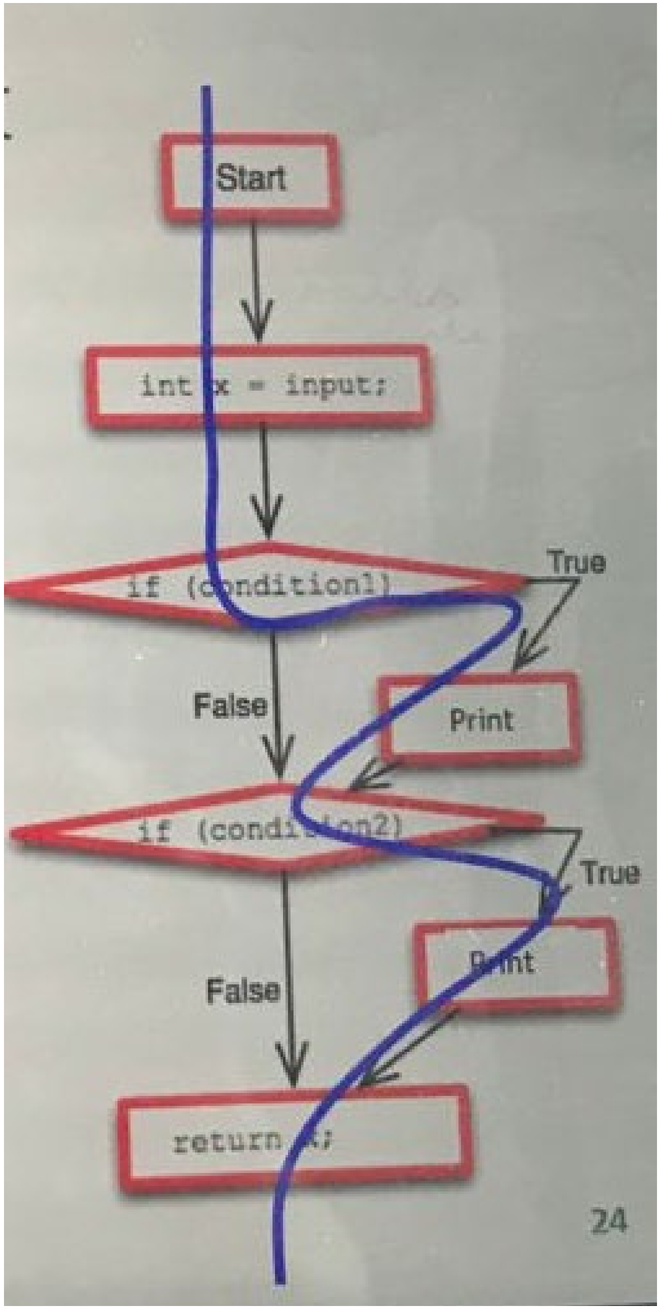
a) **5/5 (100%)**

b) **4/5 (80%)**

c) **3/5 (60%)**

d) **2/5 (40%)**

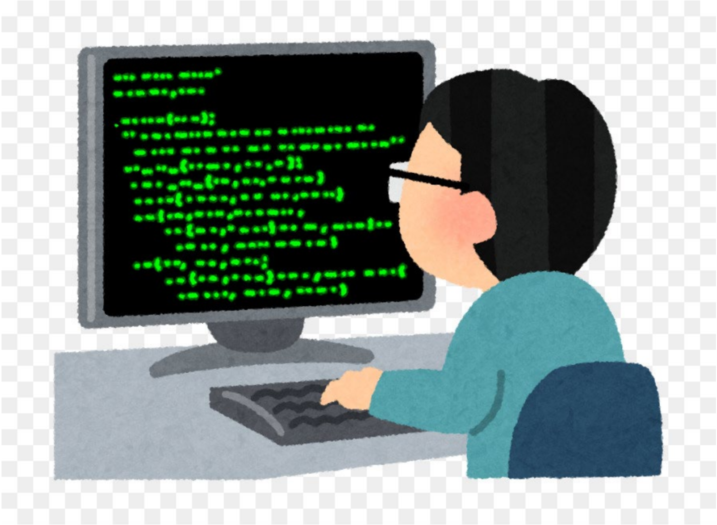
e) **1/5 (20%)**



test(0,true,true)

Why is line coverage problematic?

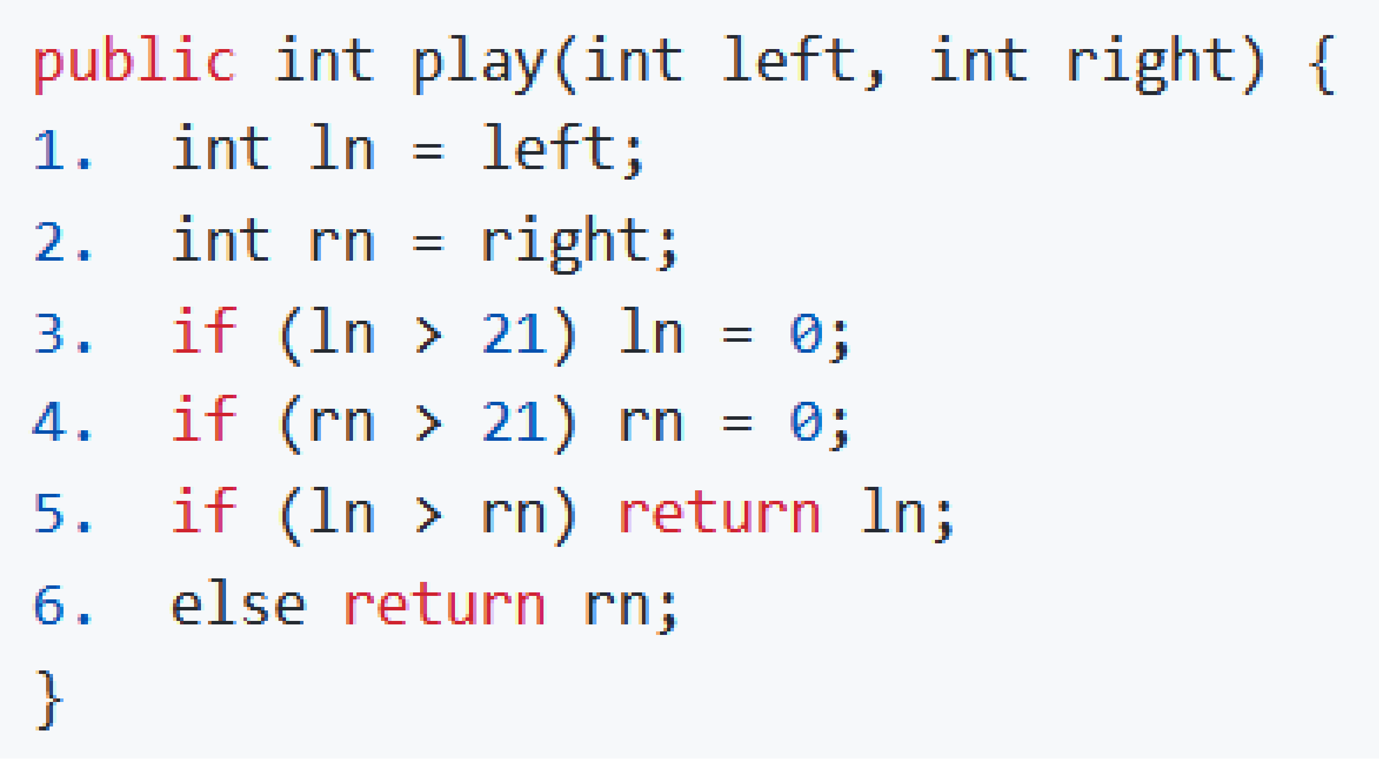
- Using lines of code as a way to determine line coverage is a simple and straightforward idea.
- However, counting the covered lines is not always a good way of calculating the coverage.
- The **number of lines** in a piece of code depends on the **decisions taken by the programmer who writes the code.**



University of Victoria

Different number of lines

- Let us look again at the Black Jack example. The play method can also be written in 6 lines, instead of 10:



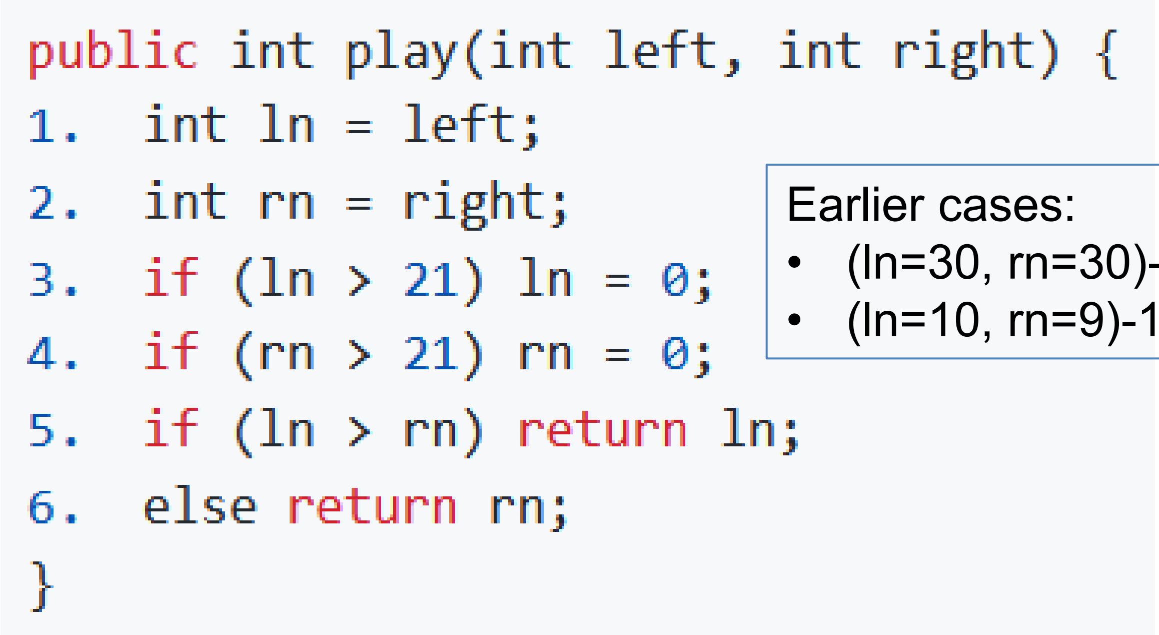
```
public int play(int left, int right) {
    int ln = left;
    int rn = right;
    if (ln > 21) ln = 0;
    if (rn > 21) rn = 0;
    if (ln > rn) return ln;
    else return rn;
}
```

Tell me…………..

- How many lines does leftPlayerWins test covers?
- What is the line coverage now?

Different number of lines

- Let us look again at the Black Jack example. The play method can also be written in 6 lines, instead of 10:



```
public int play(int left, int right) {
1.   int ln = left;
2.   int rn = right;
3.   if (ln > 21) ln = 0;
4.   if (rn > 21) rn = 0;
5.   if (ln > rn) return ln;
6.   else return rn;
}
```

Earlier cases:

- (ln=30, rn=30)-1,2,3,4,5,6,7,9,10=9/10=90%
- (ln=10, rn=9)-1,2,3,5,7,8=6/10=60%

For 100% line coverage, test cases:

- (ln=30, rn=30)-1,2,3,4,6=5/6=83%
- (ln=10, rn=9)-1,2,3,4,5=5/6=83%

You got that right !!

- The leftPlayerWins test covered: 6/10 lines in the previous implementation of the play method.
- In this new implementation, it covers lines 1-5, or **5/6 lines**.
- The line coverage went up from **60 to 83** , while testing the same method with the same input.
- This urges for a **better representation of source code**.
- One that is **independent of the developers' personal code styles**.

Sometimes 100% statement coverage might not be possible.

```
int i=0;
int j;
read(j);
if((j>5) && (i<0))
        print (i)      →      Dead code
```

No test suite can be created to achieve 100% statement coverage.

Dead code needs refactoring

- If there is some unreachable code, we will never be able to reach 100% statement coverage.
- And that's to account for the fact that there are **infeasible paths, inexecutable statements**, conditions that can never be true, and so on.
- Also, there might be pieces of code that are added, but they're **still not activated**.
- Refactoring can help here.

Practice question

- Req: Given a sentence, the program should count the number of words that end with either “s” or “r”. A word ends when a non-letter appears. The program returns the number of words.

```
public class CountWords {
  public int count(String str) {
    int words = 0;
    char last = ' ';

    for (int i = 0; i < str.length(); i++) {

      if (!isLetter(str.charAt(i)) &&
        (last == 's' || last == 'r')) {
          words++;
      }

      last = str.charAt(i);
    }

    if (last == 'r' || last == 's') {
      words++;
    }

    return words;
  }
}
```

Loops through each character in the string

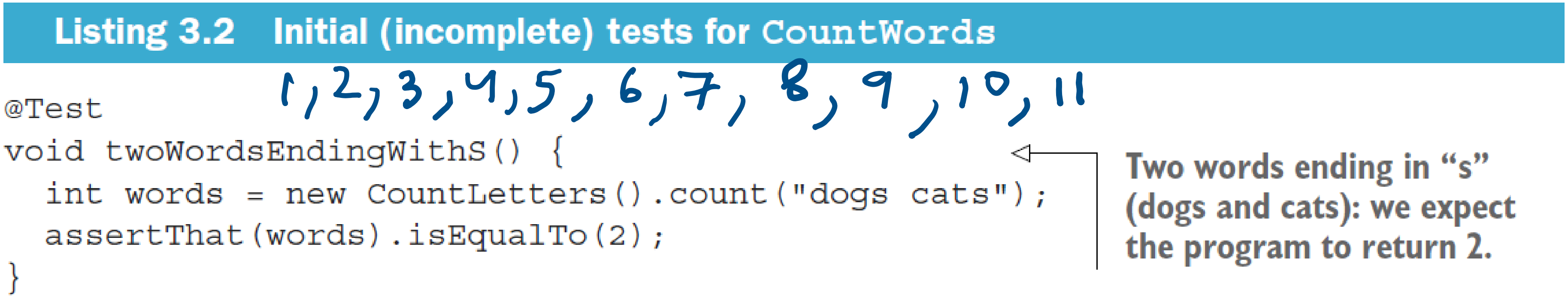
If the current character is a non-letter and the previous character was “s” or “r”, we have a word!

Stores the current character as the “last” one

Counts one more word if the string ends in “r” or “s”

index	last	w
0	d	0
1	o	0
2	g	0
3	s	0
4	' '	1
5	c	0
6	a	0
7	t	0
8	s	0
		2

Write one test case which gives 100% statement coverage.



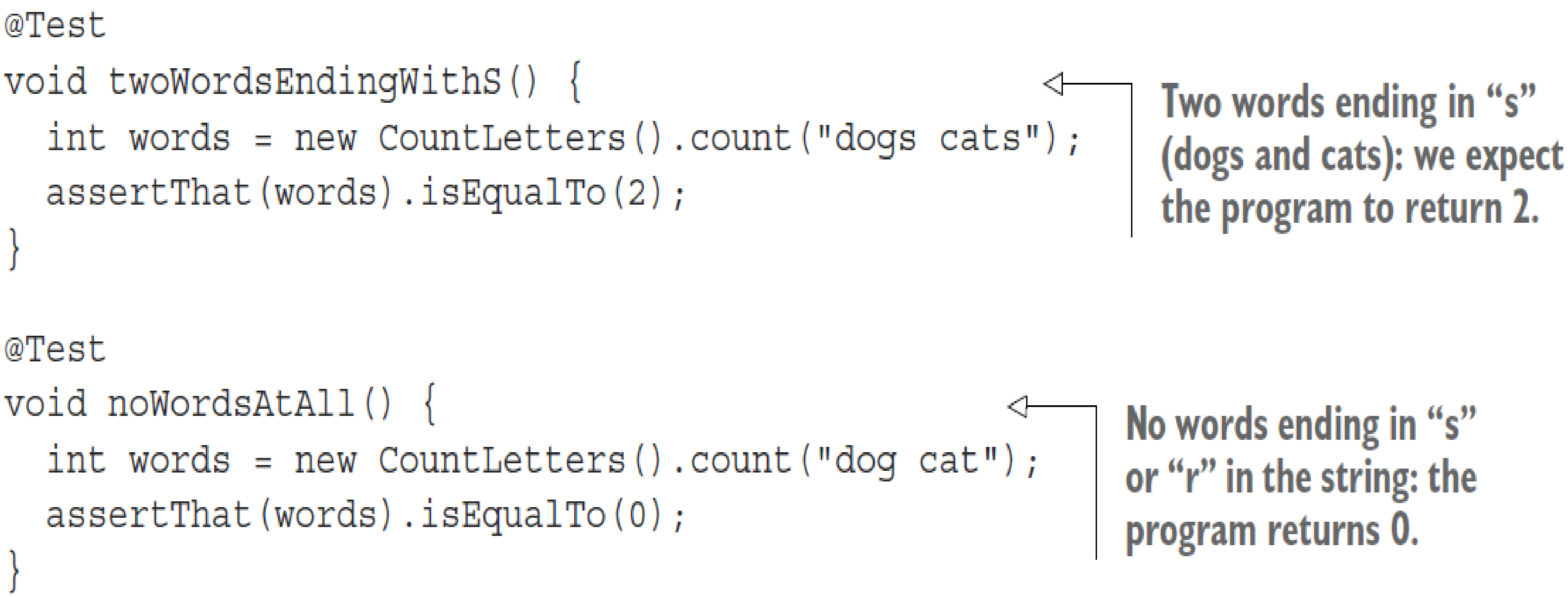
Listing 3.2 Initial (incomplete) tests for `CountWords`

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

```
@Test
void twoWordsEndingWithS() {
  int words = new CountLetters().count("dogs cats");
  assertThat(words).isEqualTo(2);
}
```

Two words ending in "s" (dogs and cats): we expect the program to return 2.

Is line coverage tests enough?

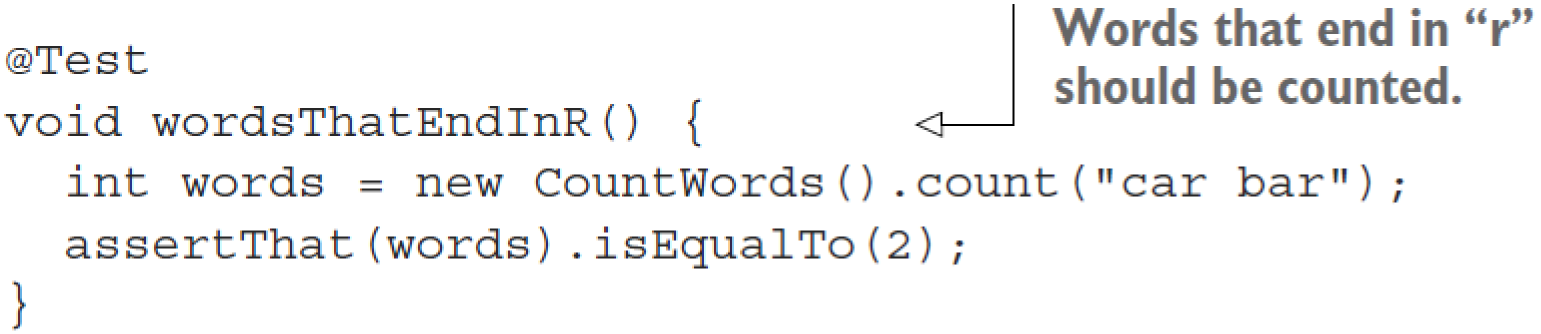


```
@Test
void twoWordsEndingWithS() {
  int words = new CountLetters().count("dogs cats");
  assertThat(words).isEqualTo(2);
}
```

Two words ending in “s” (dogs and cats): we expect the program to return 2.

```
@Test
void noWordsAtAll() {
  int words = new CountLetters().count("dog cat");
  assertThat(words).isEqualTo(0);
}
```

No words ending in “s” or “r” in the string: the program returns 0.



```
@Test
void wordsThatEndInR() {
  int words = new CountWords().count("car bar");
  assertThat(words).isEqualTo(2);
}
```

Words that end in “r” should be counted.