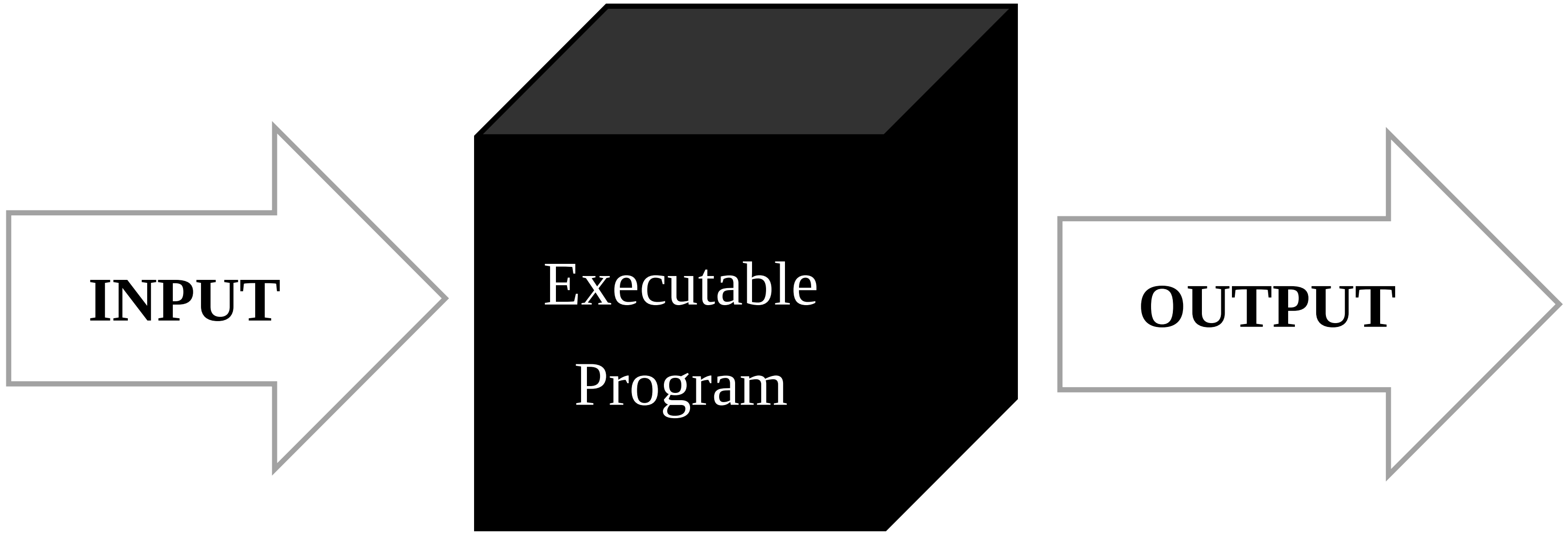
Software Testing
Course's Code: CSE 453
Test Design Techniques – Structure-based (White-Box) Technique
(Chapter 4)

Categories of Test Case Design Techniques

- Two Primary Categories are:
  - Black-Box Testing or Specification-Based Testing
  - White-Box Testing or Structure-Based Testing
  - Experience based Testing
- One more additional category is Grey-Box Testing

Black-Box Testing



Black-Box Testing

- ❖ Black Box testing refers to a software testing method where the SUT (Software Under Test) functionality is tested without worrying about its details of implementation, internal path knowledge and internal code structure of the software

- ❖ Black Box testing techniques are also known as Specification-Based Techniques

- ❖ In specification-based techniques, Test Cases are derived directly from the specification or from some other kind of model of what the system should do.

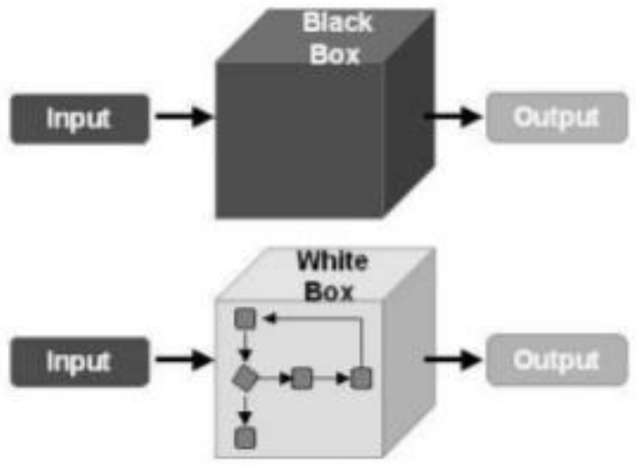
- ❖ The source of information on which to base testing is known as the 'test basis'. It includes both functional and non-functional aspects

White-Box Testing

White box testing Vs. Black box testing

In **Black box** testing, we test the software from a user's point of view.

In **White box** testing, we evaluates the code and internal structure of the program.



White-Box Testing

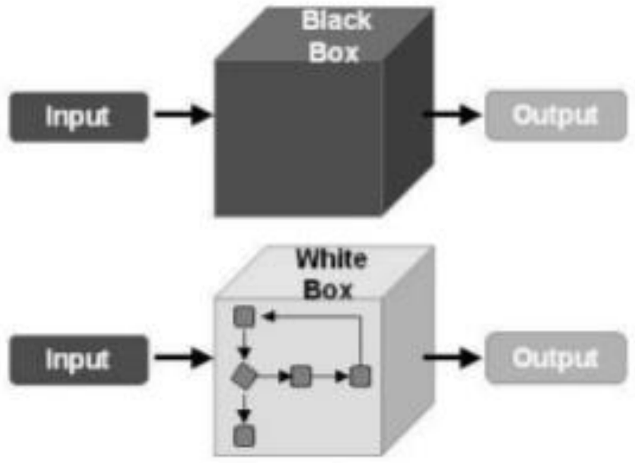
- White Box testing is based on specific knowledge of the source code to define the test cases and to examine outputs
- These kinds of techniques help to identify which line of code is actually executed and which is not
- This indicates that there is either missing logic or a typo.
- This technique exercises a path of code
- This kind of testing is also known as Structure-based Testing or Glass-box Testing

White-Box Testing vs. Black-Box Testing

White box testing Vs. Black box testing

In **Black box** testing, we test the software from a user's point of view.

In **White box** testing, we evaluates the code and internal structure of the program.

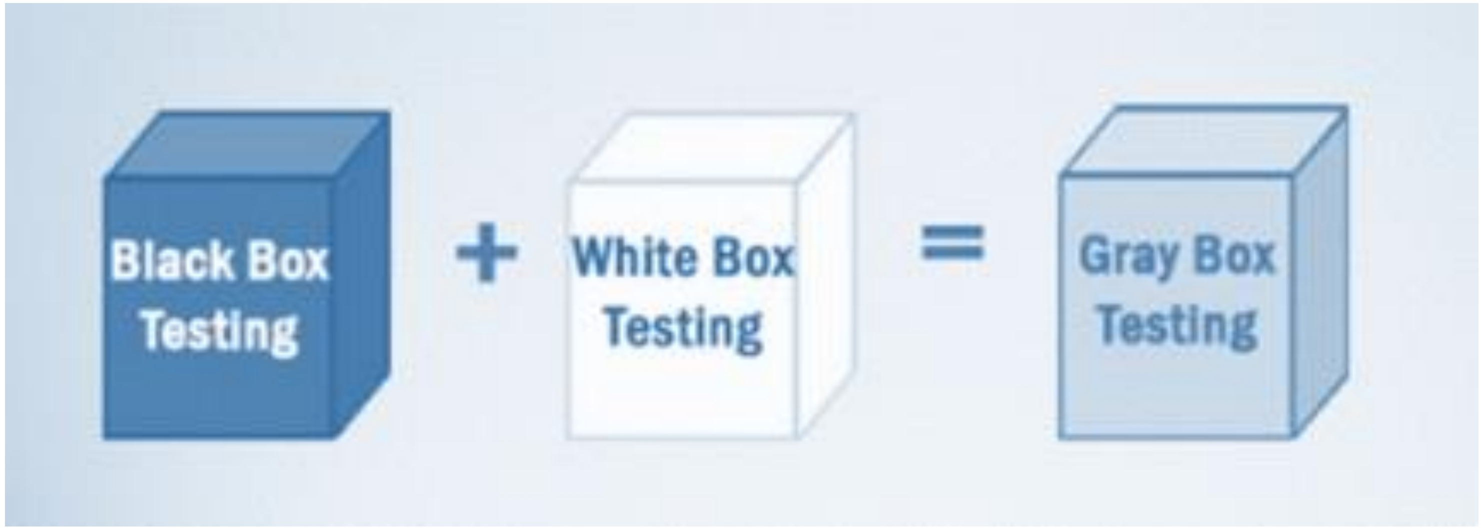


Experience based Testing

- This kind of technique leverage the experience of developers, testers and users to design, implement, and execute tests.
- These techniques are often combined with black-box and white-box test techniques.
- Test conditions, test cases, and test data are derived from a test basis that may include knowledge and experience of testers, developers, users and other stakeholders

Grey-Box Testing

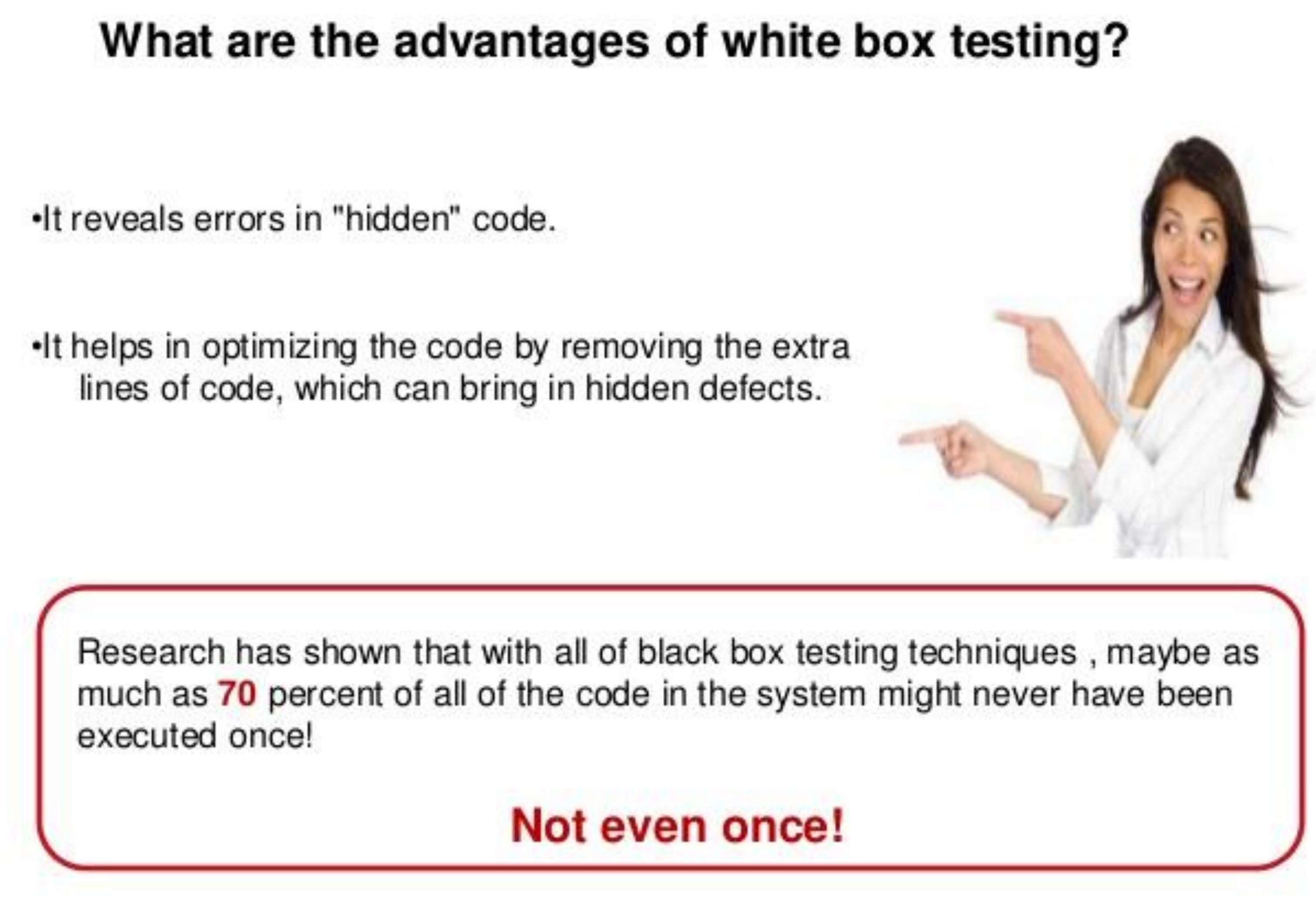
- Gray Box Testing is a software testing method, which is a combination of both White Box Testing and Black Box Testing method.
  - In White Box testing internal structure (code) is known
  - In Black Box testing internal structure (code) is unknown
  - In Grey Box Testing internal structure (code) is partially known



Grey-Box Testing

- Example of Gray Box Testing: While testing websites feature like links or orphan links, if tester encounters any problem with these links, then he can make the changes straightaway in HTML code and can check in real time.
- Why Gray Box Testing
  - It provides combined benefits of both black box testing and white box testing both
  - It combines the input of developers as well as testers and improves overall product quality
  - It reduces the overhead of long process of testing functional and non-functional types
  - It gives enough free time for a developer to fix defects
  - Testing is done from the user point of view rather than a designer point of view

Advantages of White-Box Testing



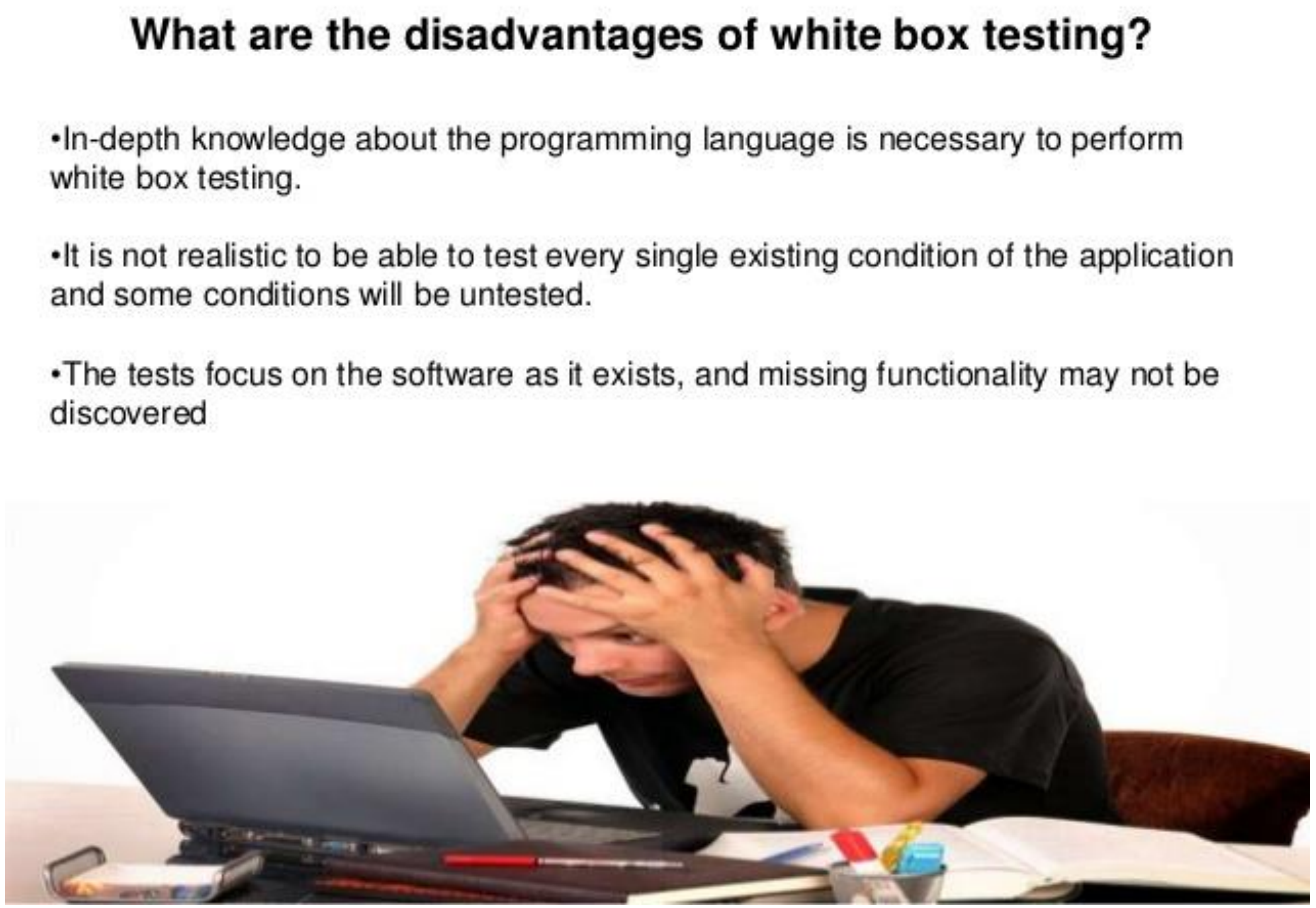
What are the advantages of white box testing?

- It reveals errors in "hidden" code.
- It helps in optimizing the code by removing the extra lines of code, which can bring in hidden defects.

Research has shown that with all of black box testing techniques , maybe as much as **70** percent of all of the code in the system might never have been executed once!

Not even once!

Disadvantages of White-Box Testing



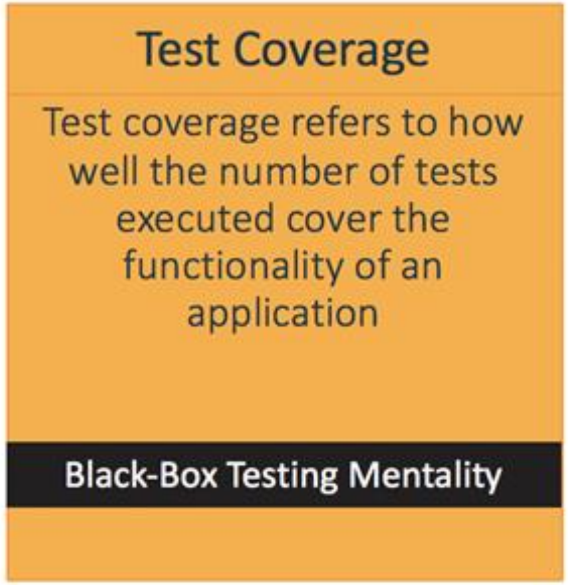
What are the disadvantages of white box testing?

•In-depth knowledge about the programming language is necessary to perform white box testing.

•It is not realistic to be able to test every single existing condition of the application and some conditions will be untested.

•The tests focus on the software as it exists, and missing functionality may not be discovered

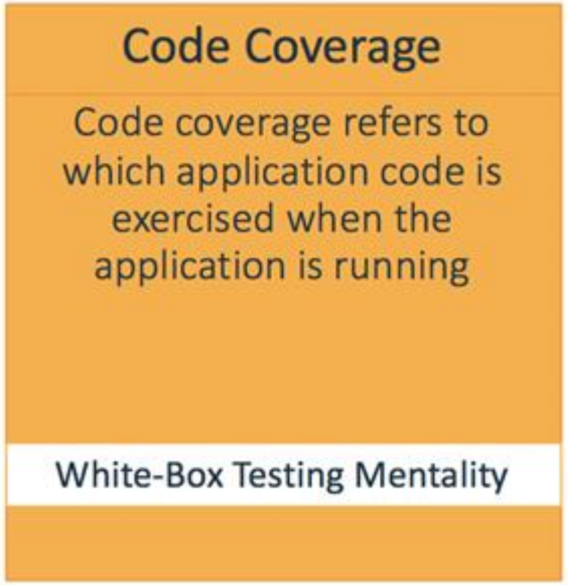
Test Coverage vs. Code Coverage



Test Coverage

Test coverage refers to how well the number of tests executed cover the functionality of an application

Black-Box Testing Mentality



Code Coverage

Code coverage refers to which application code is exercised when the application is running

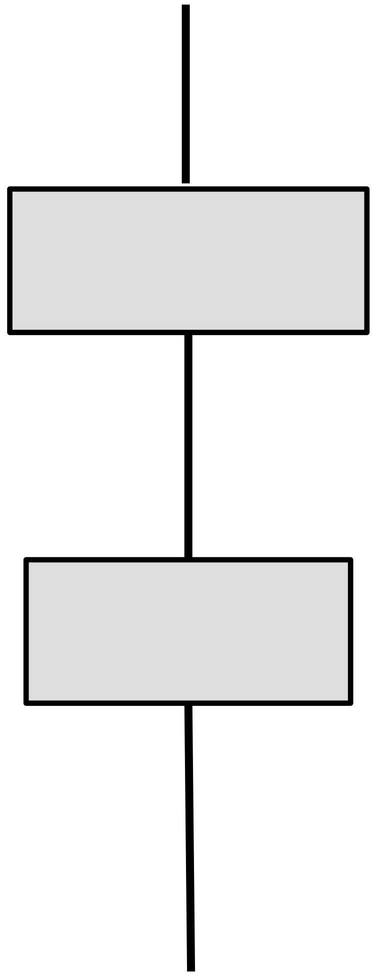
White-Box Testing Mentality

General Structure of a Computer Program

- Code can be of two types
  - Executable
  - Non-executable
- Executable code instructs the computer to take some actions
- Non-executable code is used to prepare the computer to do its calculations
  - It does not involve any actions
  - For example declaration statement

General Structure of a Computer Program

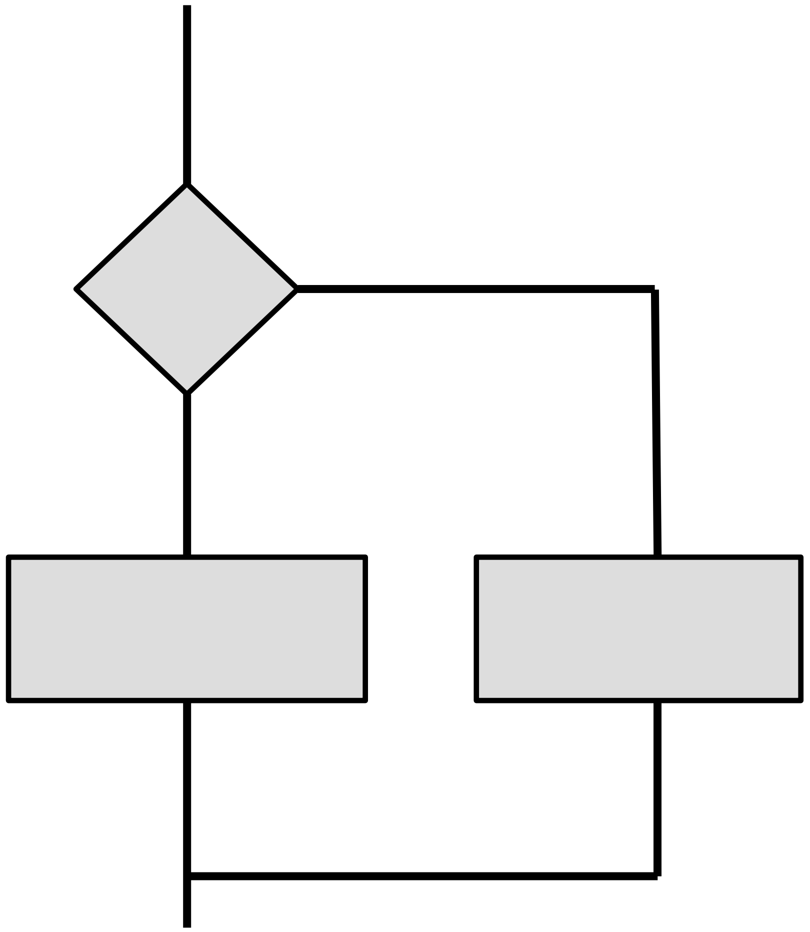
- There are only three ways that executable code can be structured
  - ❑ Sequence
  - ❑ Selection
  - ❑ Iteration
- Sequence Statements



Selection Statements

- Selection: Selection structures involve decisions
- The computer has to decide if a condition (known as a Boolean condition) is true or false

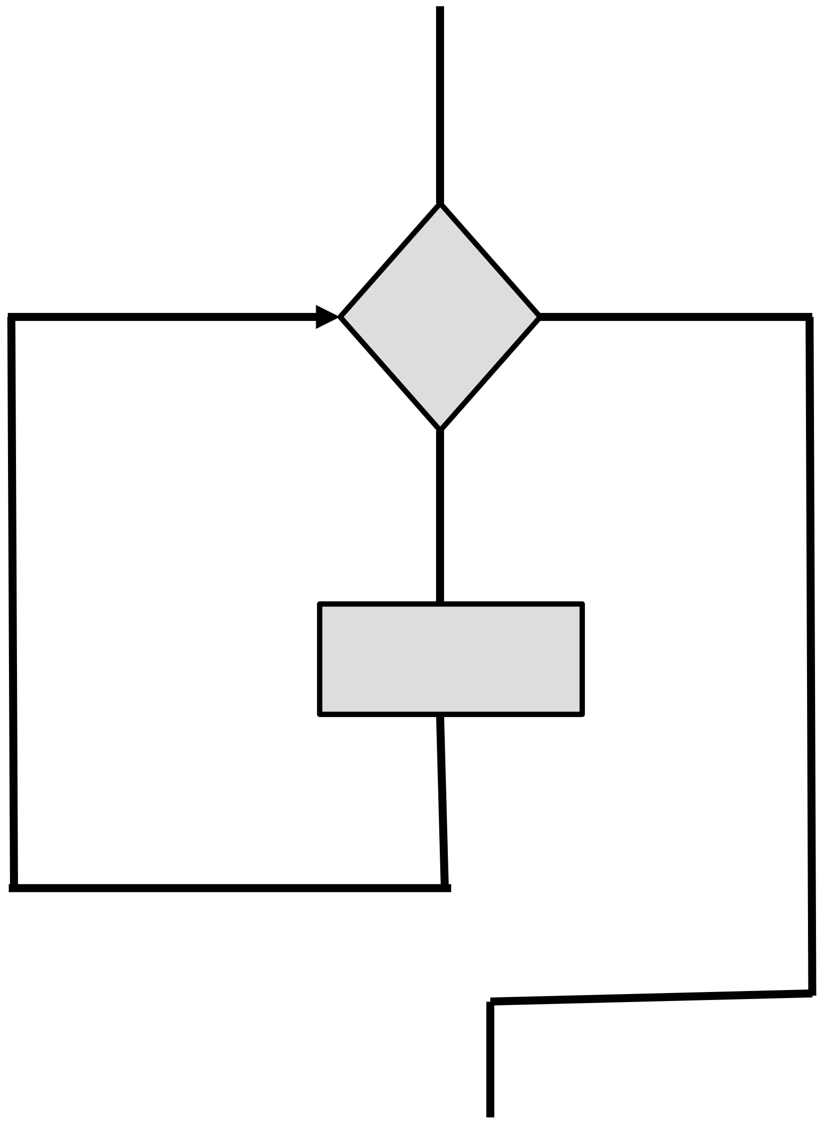
```
1   If p>3
2    Then
3          x=x+y
4     Else
5       x=x-y
6   End If
```



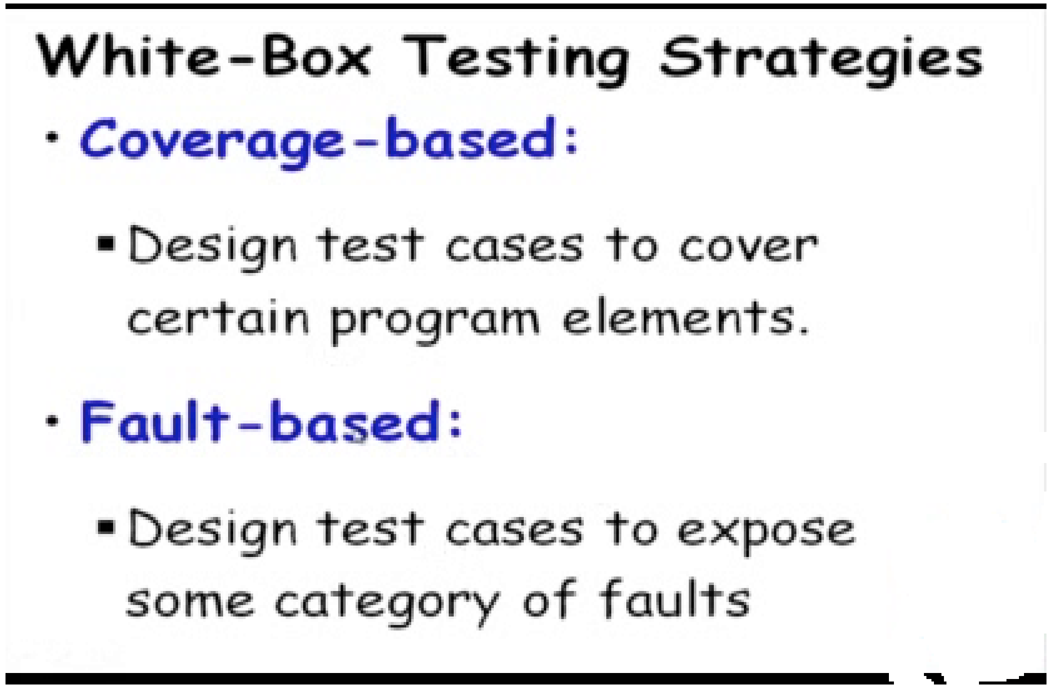
Iteration Statements

- Iteration: It simply involves the computer exercising a chunk of code more than once
- These types of structures are also known as loop

```
x=15
Count=0
 While x<20 Do
     x=x+1
     Count=count+1
  End do
```

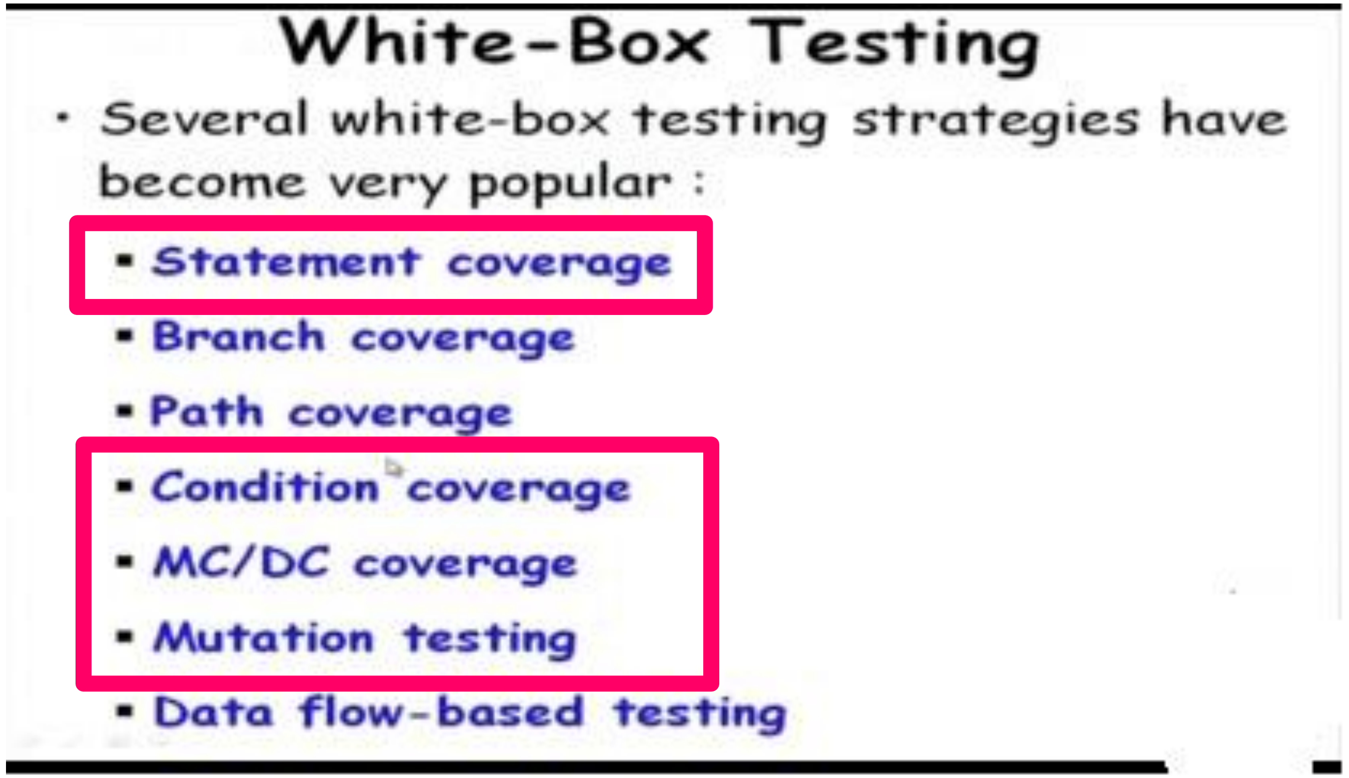


White Box Testing Strategies



White-Box Testing Strategies

- Coverage-based:
  - Design test cases to cover certain program elements.
- Fault-based:
  - Design test cases to expose some category of faults

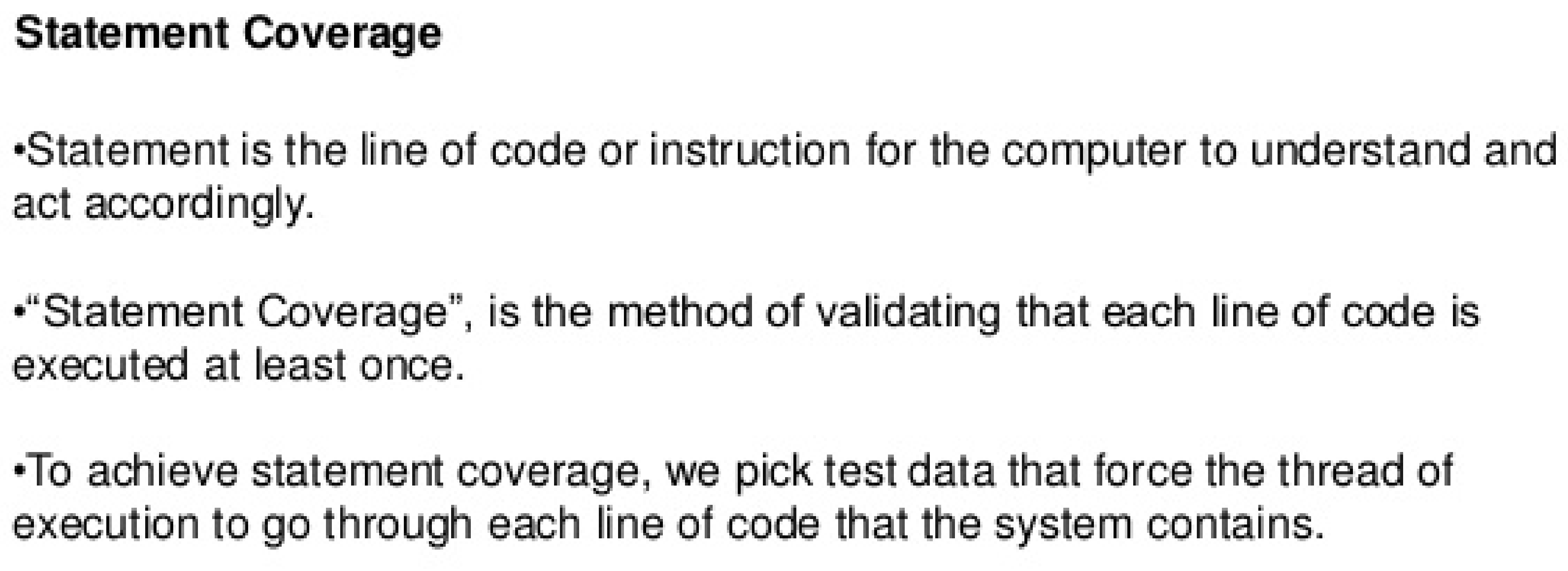


White-Box Testing

- Several white-box testing strategies have become very popular :
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Condition coverage
  - MC/DC coverage
  - Mutation testing
  - Data flow-based testing

- Mutation testing is an example of Fault-based testing
- Rest are examples of coverage-based Testing

Statement Coverage



Statement Coverage

•Statement is the line of code or instruction for the computer to understand and act accordingly.

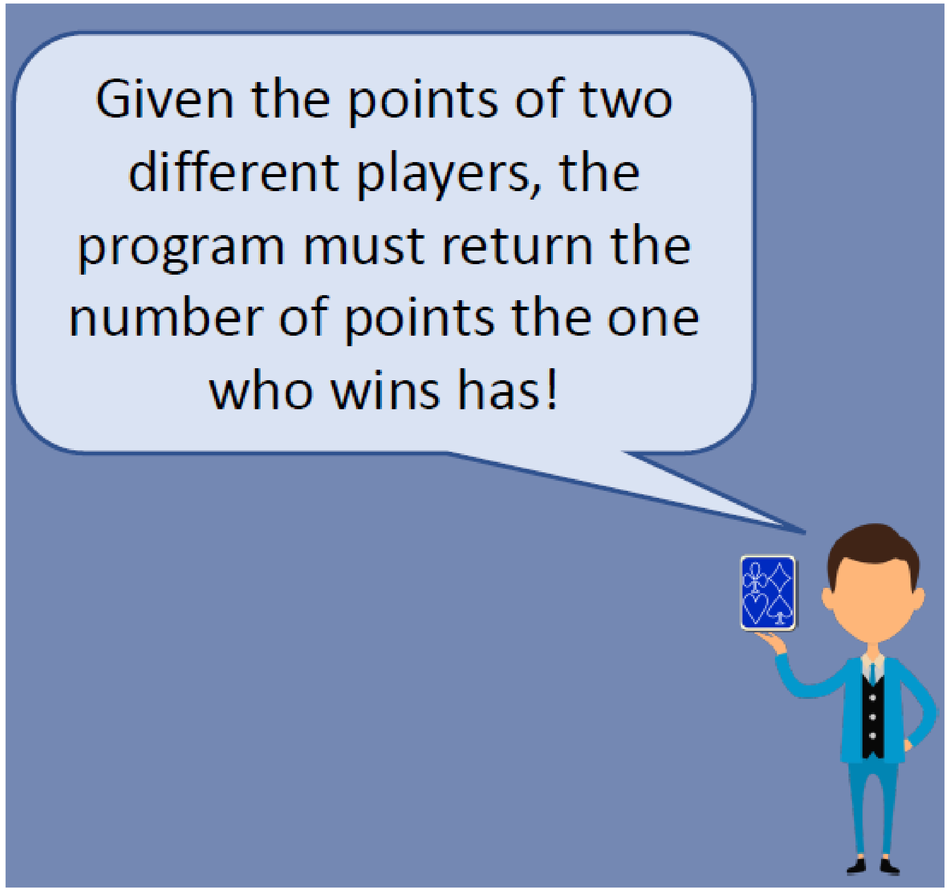
•"Statement Coverage", is the method of validating that each line of code is executed at least once.

•To achieve statement coverage, we pick test data that force the thread of execution to go through each line of code that the system contains.

- ➢ Statement Coverage can be achieved through Line Coverage and/or Instruction Coverage in Eclipse

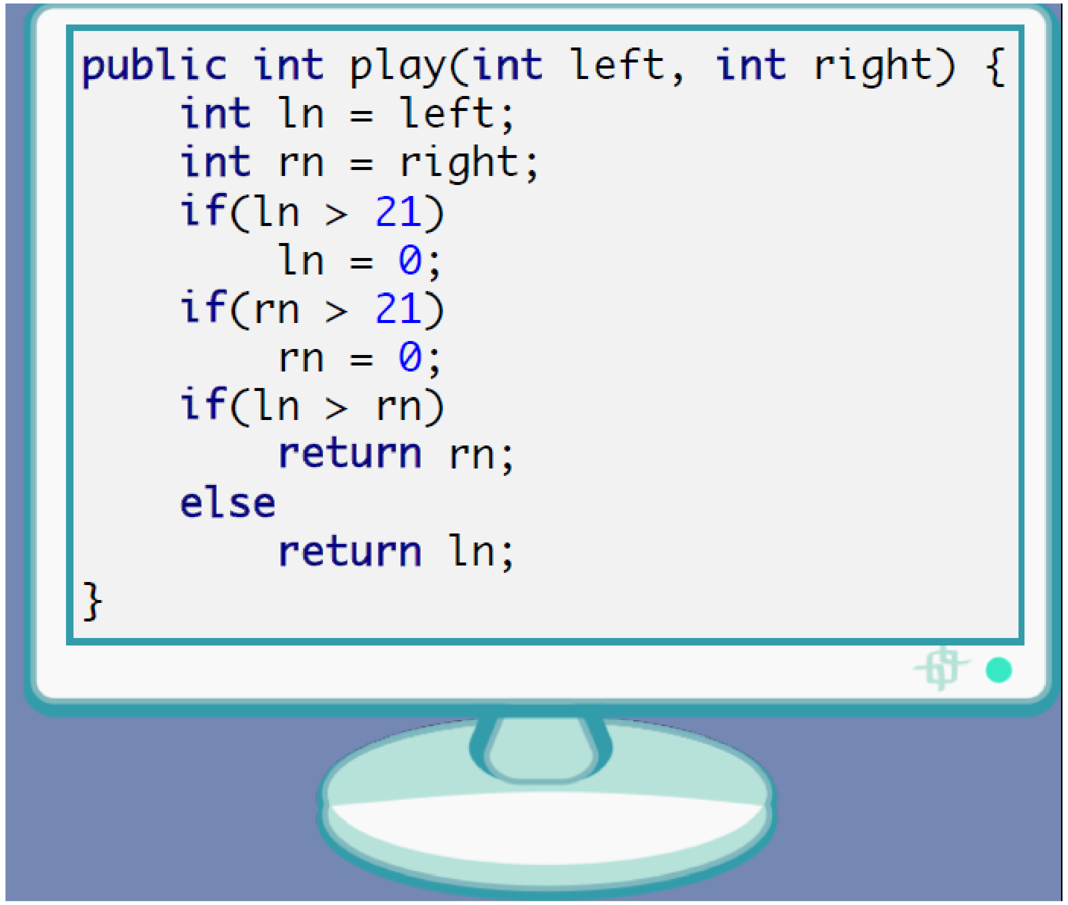
Line Coverage – An Example

- Let's see an example : Black Jack card game
- In black jack, whoever is closer to 21, wins the game.
- So, given the points of two different players, the program must return the number of points the one who wins has.
- If a player has more than 21 points, she can't win.
- If both players have more than 21 points, the program then must return zero.


Given the points of two different players, the program must return the number of points the one who wins has!

Line Coverage – An Example

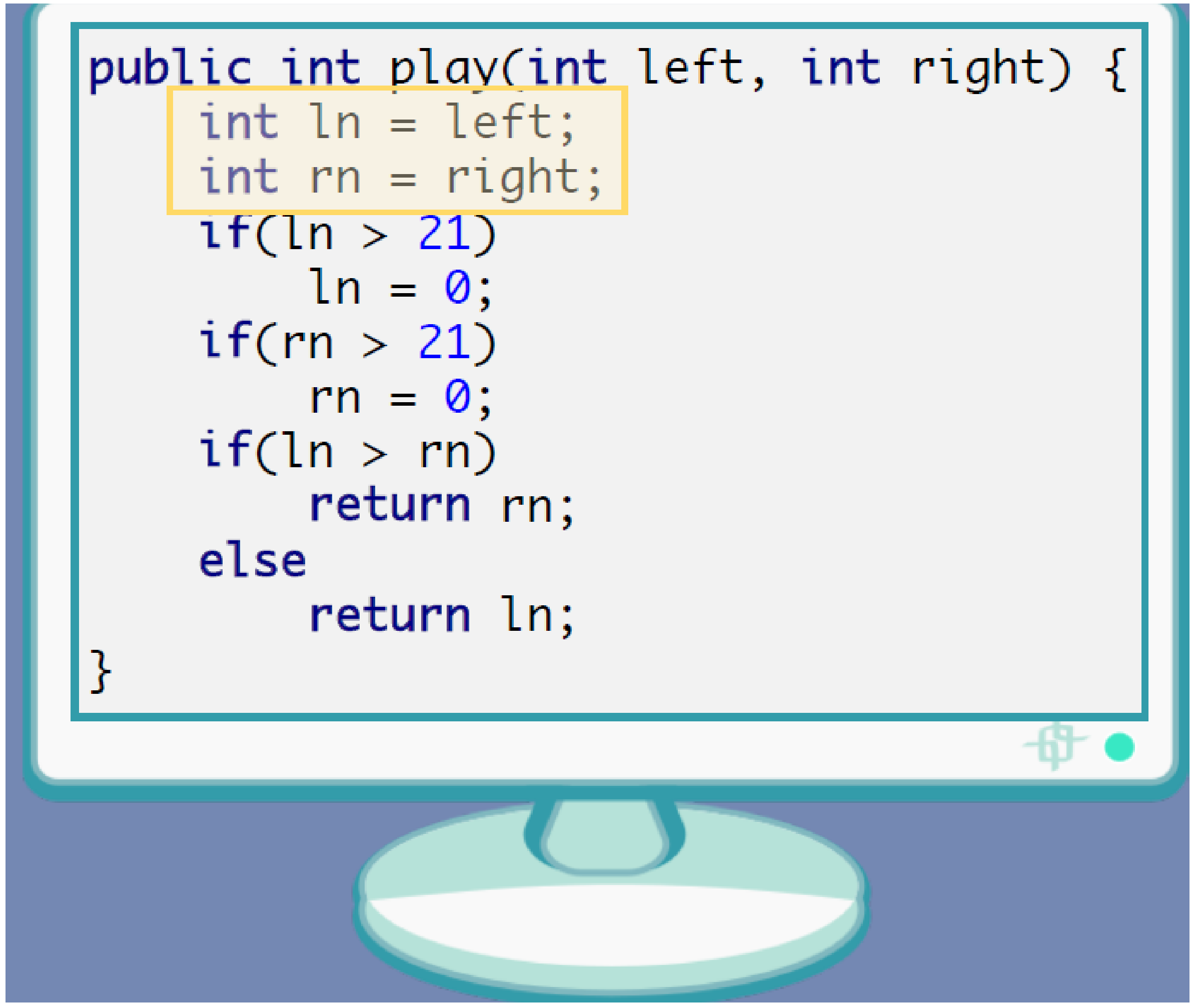
- Now, we will create test cases based on implementation



```
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 rn;
    else
        return ln;
}
```

Ok, let's see what we should test.. But looking to the implementation!

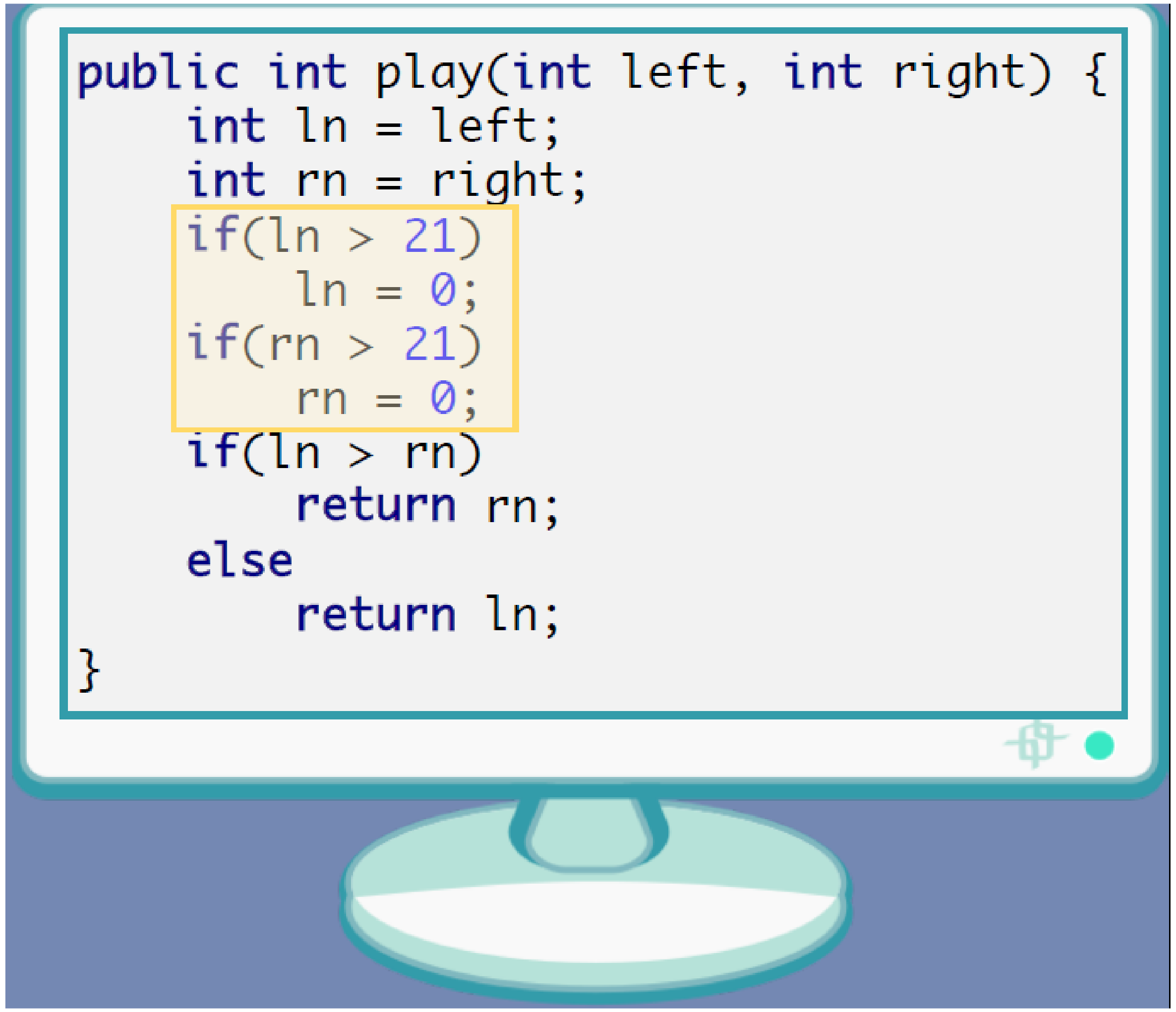
Line Coverage – An Example



```
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 rn;
    else
        return ln;
}
```

- The play() method receives two parameters: left and right, which are both integers.
- The implementation first copies both values to new variables, ln and rn.

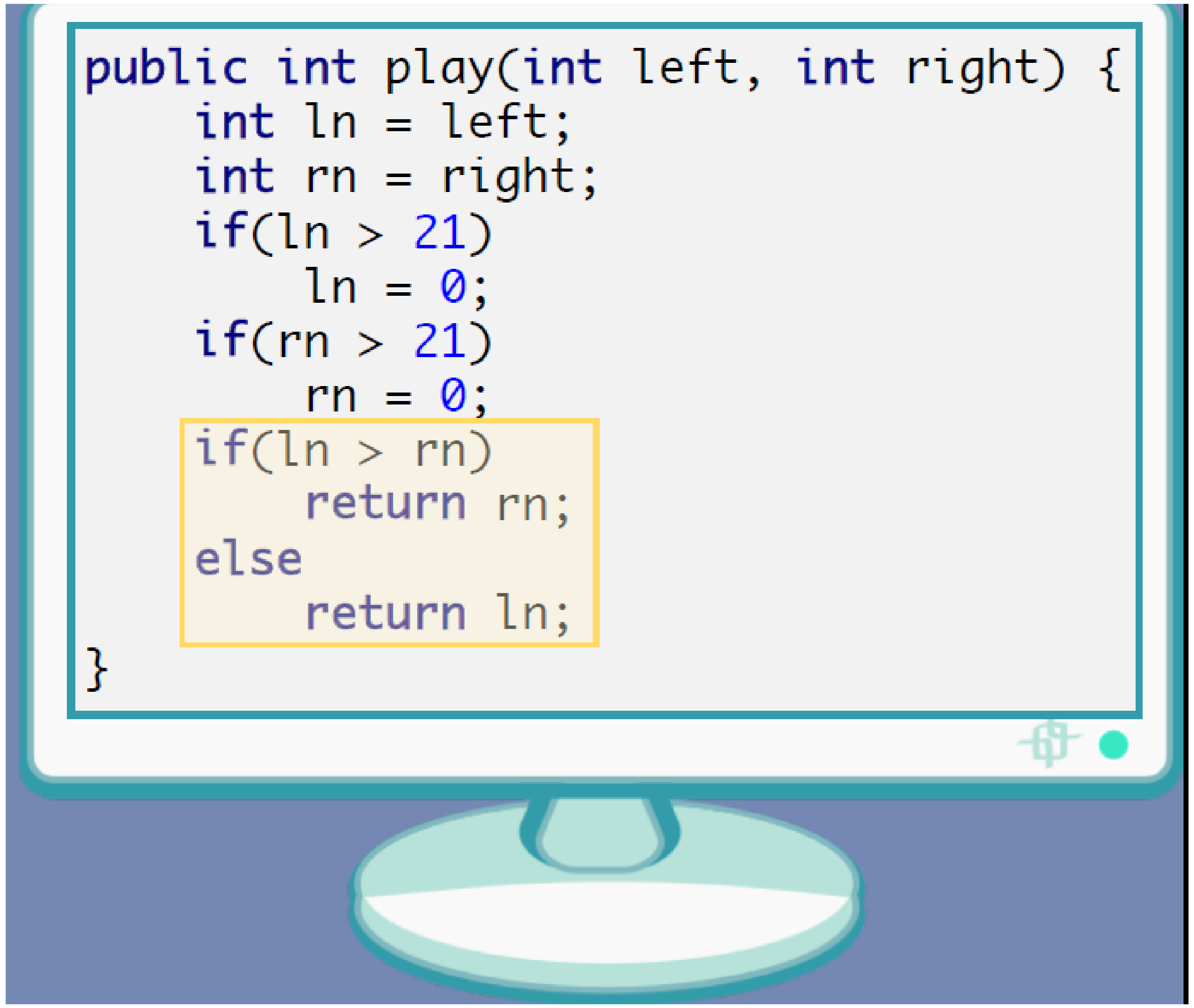
Line Coverage – An Example



```
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 rn;
    else
        return ln;
}
```

- ➢ Then, it checks whether they are greater than 21; if they are, the implementation changes it to zero.

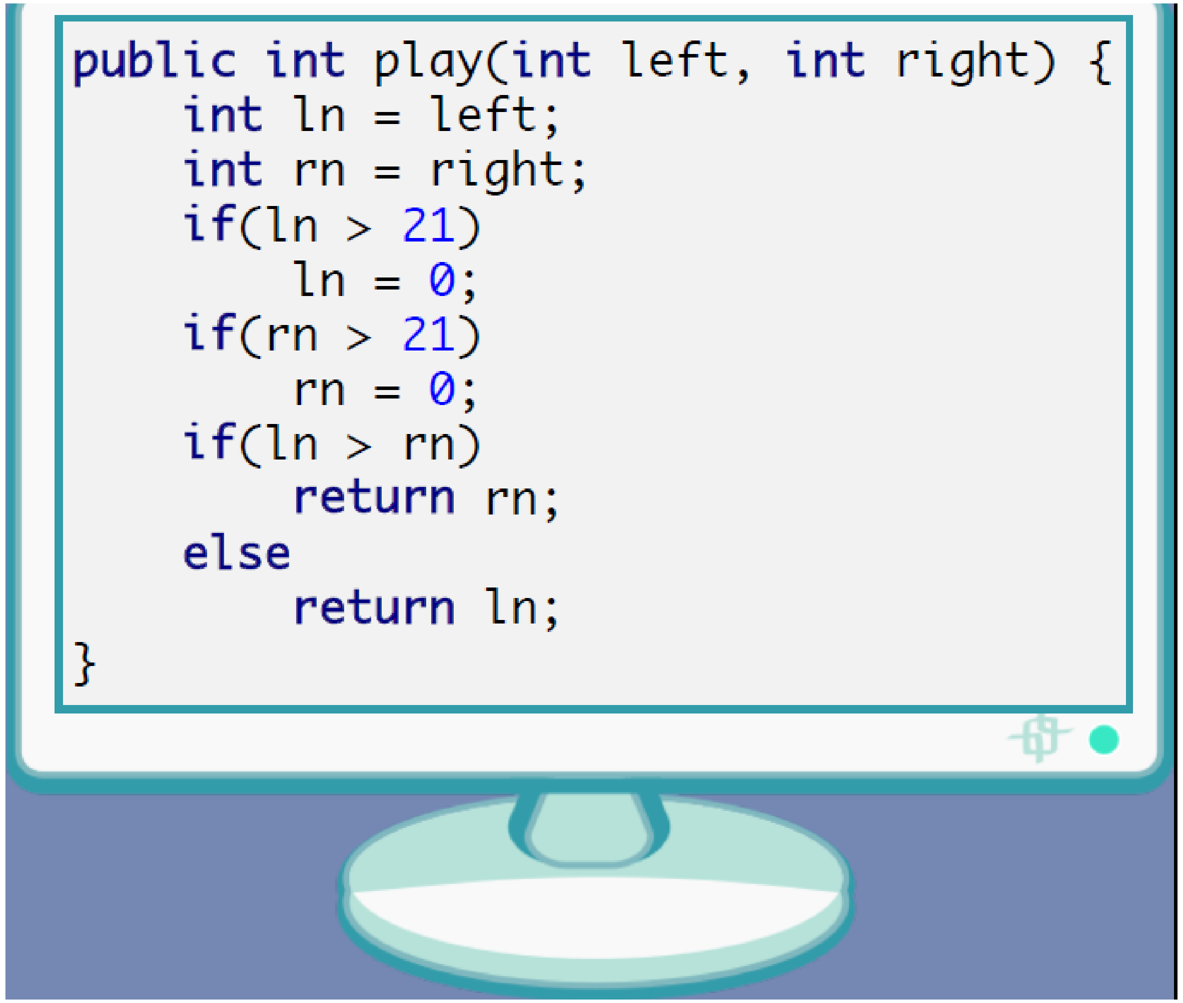
Line Coverage – An Example



```
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 rn;
    else
        return ln;
}
```

- At the end, the code returns the greatest number.

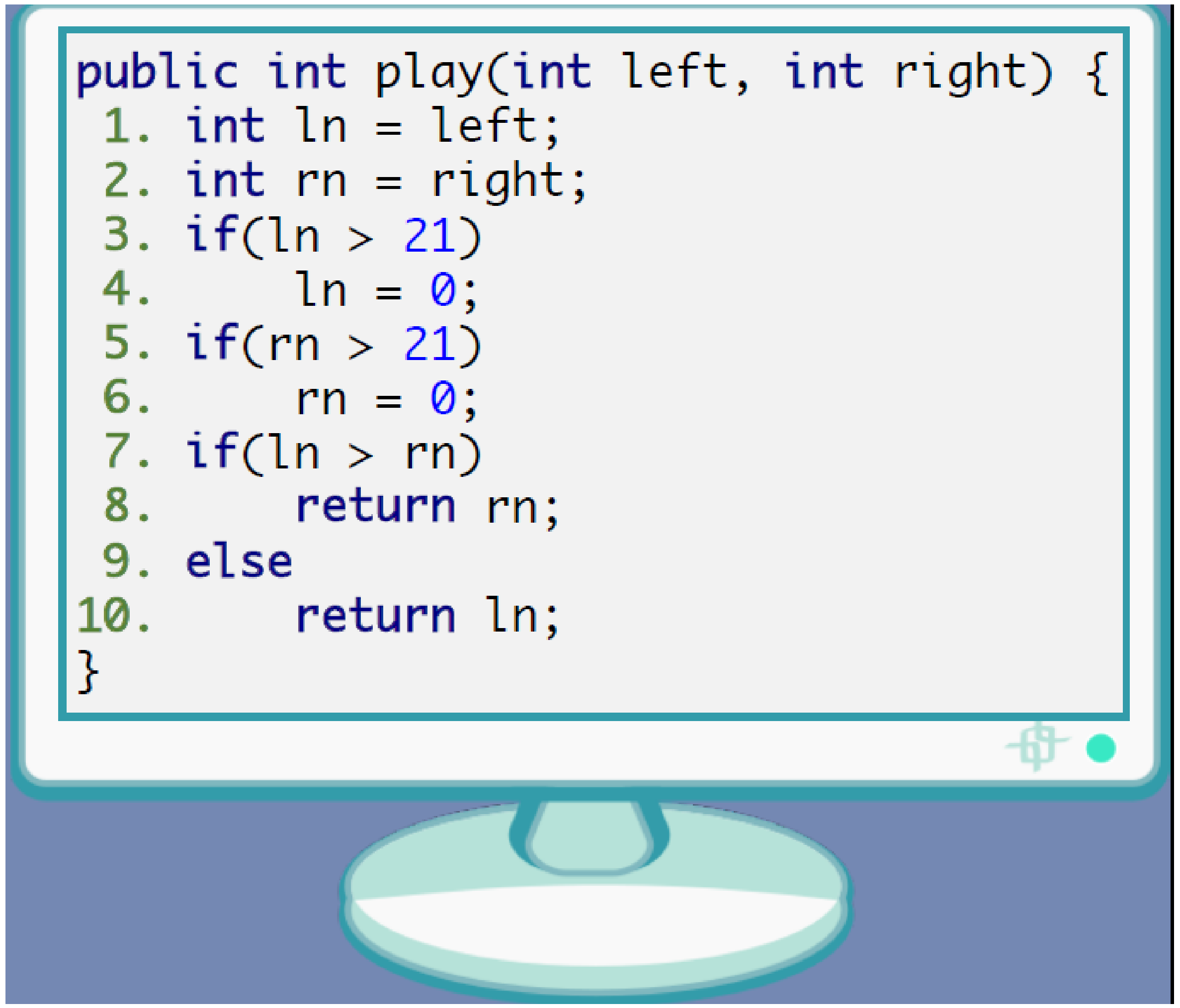
Line Coverage – An Example



```
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 rn;
    else
        return ln;
}
```

- We will create test cases by exercising as much as we can its implementation.
- By exercising, means having tests that will make all the lines of our source code to run!
- We can make all the lines of our source code to run in many ways
- A very simple one and good to get started, is for example, "line coverage"

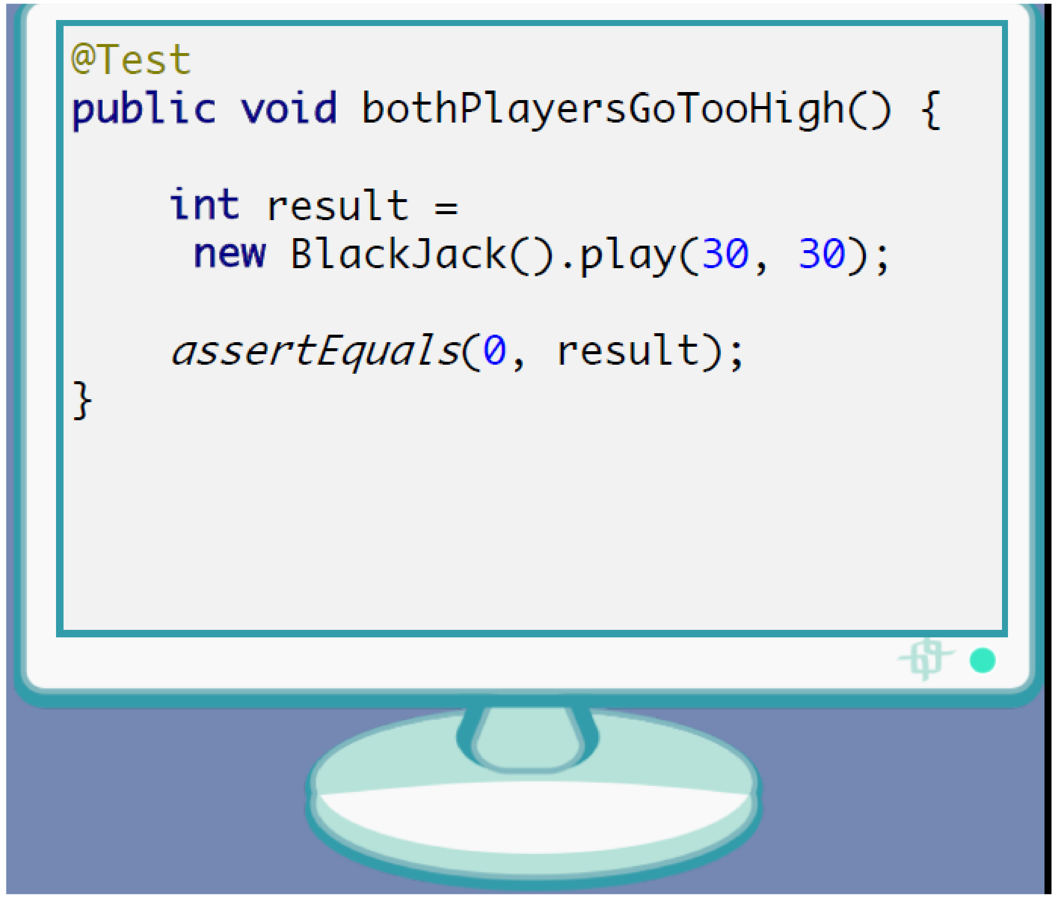
Line Coverage – An Example



```
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 rn;
 9. else
10.      return ln;
}
```

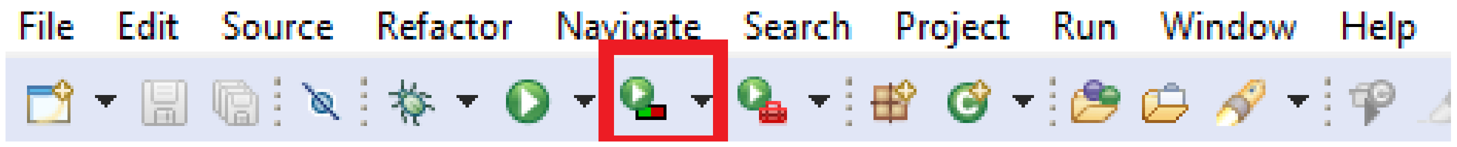
- Line coverage means that we will be happy with our tests when all the lines in the source code are exercised by at least one test.
- Black Jack example right now has 10 lines, so this means we will be happy when all these 10 lines are exercised by at least one test.
- So, here our goal is to test all the lines or, as we say, achieve 100% line coverage.
- For that, we have to make all "ifs" true.
- This happens when both left and right are greater than 21.

Line Coverage – An Example

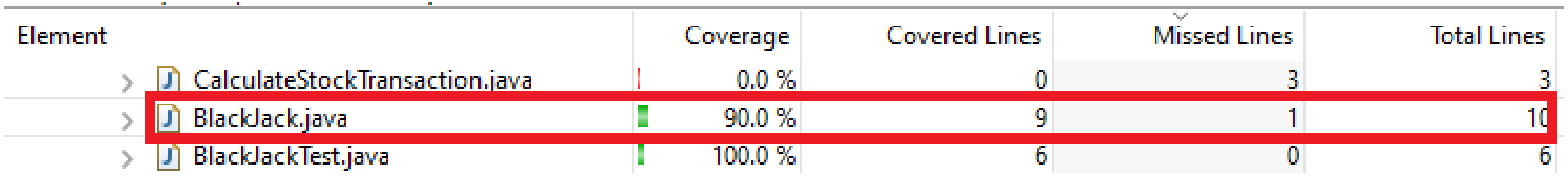


```
@Test
public void bothPlayersGoTooHigh() {

    int result =
     new BlackJack().play(30, 30);

    assertEquals(0, result);
}
```

- To run this test case we will use Junit and to measure coverage we will use coverage plugin in Eclipse

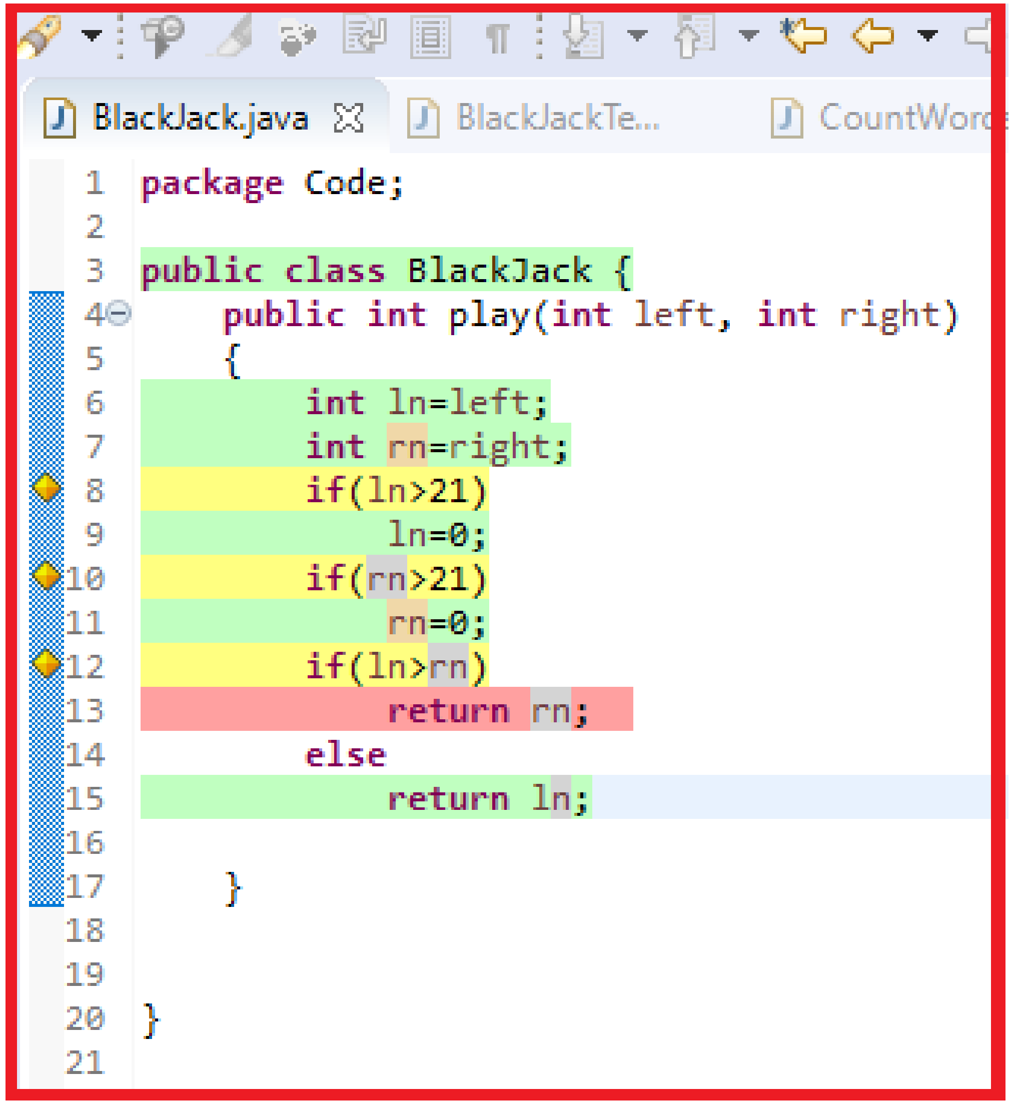


Coverage Window



Element	Coverage	Covered Lines	Missed Lines	Total Lines
CalculateStockTransaction.java	0.0 %	0	3	3
BlackJack.java	90.0 %	9	1	10
BlackJackTest.java	100.0 %	6	0	6

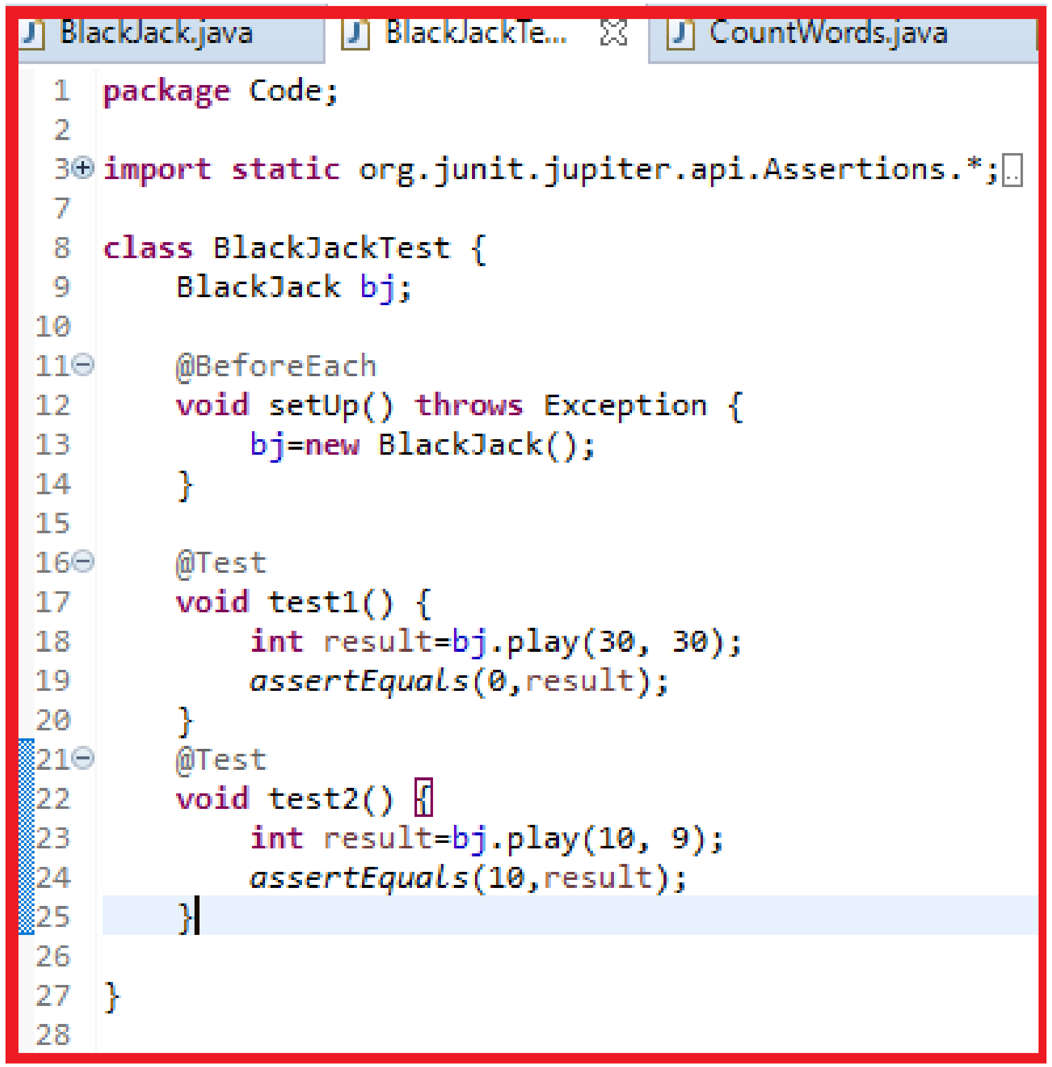
Line Coverage – An Example



```
1  package Code;
2
3  public class BlackJack {
4    public int play(int left, int right)
5    {
6      int ln=left;
7      int rn=right;
8      if(ln>21)
9        ln=0;
10     if(rn>21)
11       rn=0;
12     if(ln>rn)
13       return rn;
14     else
15       return ln;
16
17   }
18
19
20 }
21
```

- We can even see line by line in main code
- Eclipse shows, near the line numbers, colors like green, yellow and red.
- Green means that that line was covered, Yellow means partially covered and red means that the line was not covered.
- We can see that we miss one line: line 13.
- This tells us what our next test should aim for.
- Let's then exercise this line in a new test.

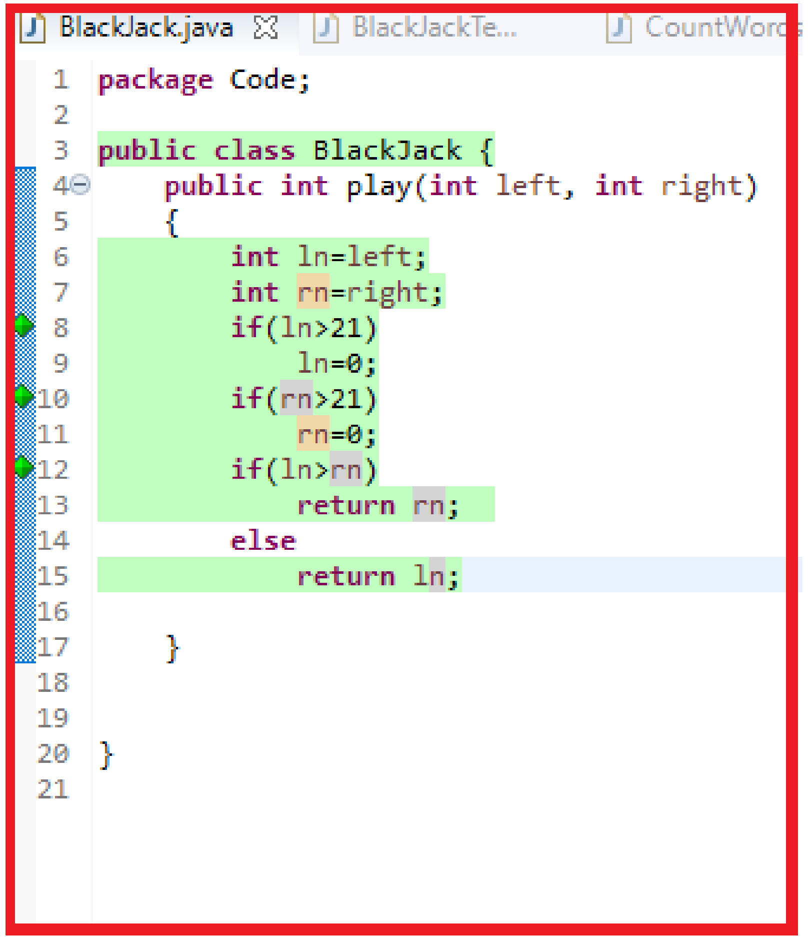
Line Coverage – An Example



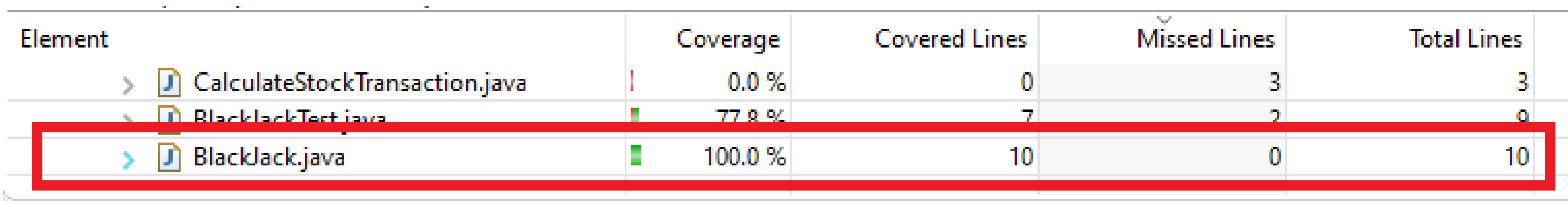
```
package Code;

import static org.junit.jupiter.api.Assertions.*;

class BlackJackTest {
    BlackJack bj;

    @BeforeEach
    void setUp() throws Exception {
        bj=new BlackJack();
    }

    @Test
    void test1() {
        int result=bj.play(30, 30);
        assertEquals(0,result);
    }
    @Test
    void test2() {
        int result=bj.play(10, 9);
        assertEquals(10,result);
    }

}
```

- For this, the left player needs to have more points than the right player, for example, 10 vs 9.
- If the left player has 10 points and the right player has 9 points, then the left player wins, and the program needs to return 10.
- If we run the tests with coverage again, we now get 100% line coverage!

Line Coverage – An Example

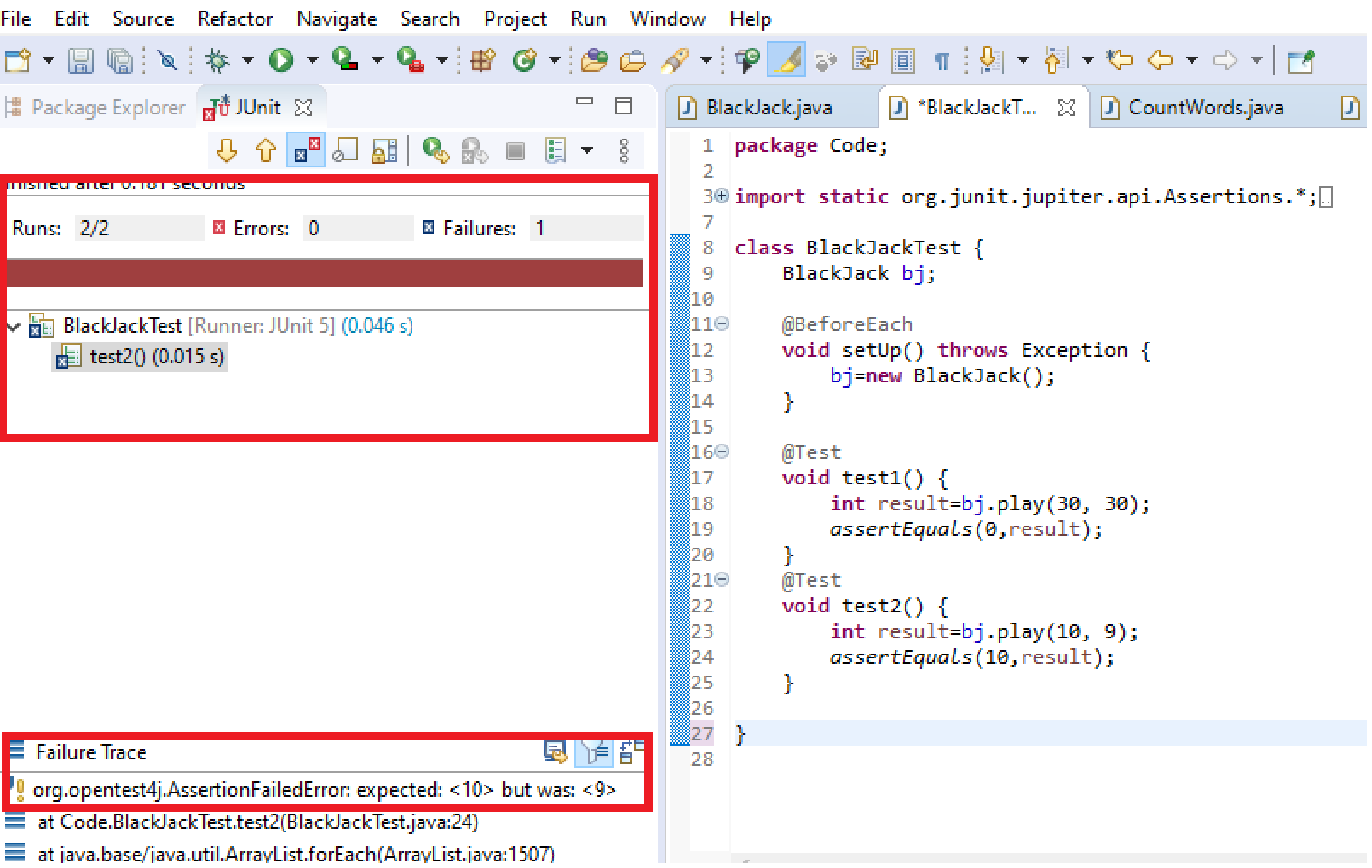


```java
package Code;

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 rn;
        else
            return ln;

    }


}

```



Element	Coverage	Covered Lines	Missed Lines	Total Lines
CalculateStockTransaction.java	0.0 %	0	3	3
BlackJackTest.java	77.8 %	7	2	9
BlackJack.java	100.0 %	10	0	10

Line Coverage – An Example

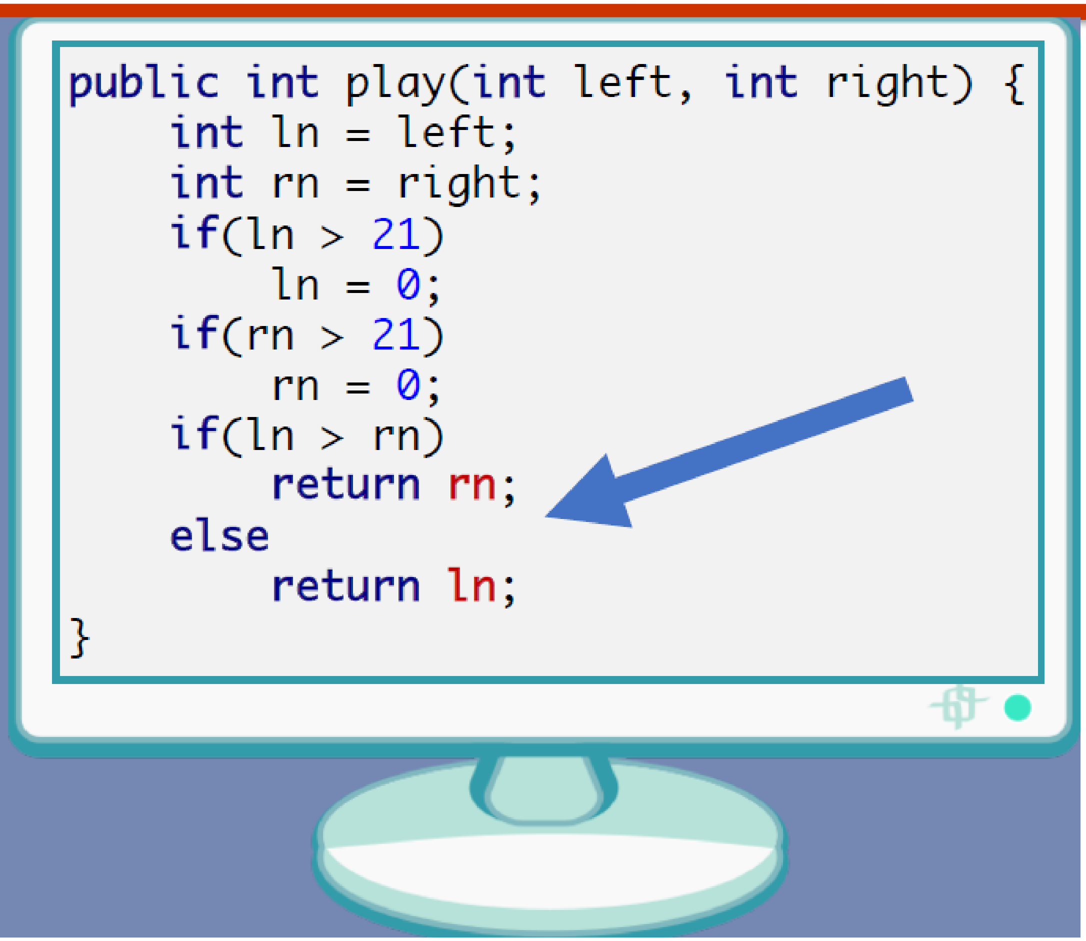


```
Runs: 2/2   Errors: 0   Failures: 1

BlackJackTest [Runner: JUnit 5] (0.046 s)
    test2() (0.015 s)

org.opentest4j.AssertionFailedError: expected: <10> but was: <9>
at Code.BlackJackTest.test2(BlackJackTest.java:24)
at java.base/java.util.ArrayList.forEach(ArrayList.java:1507)
```

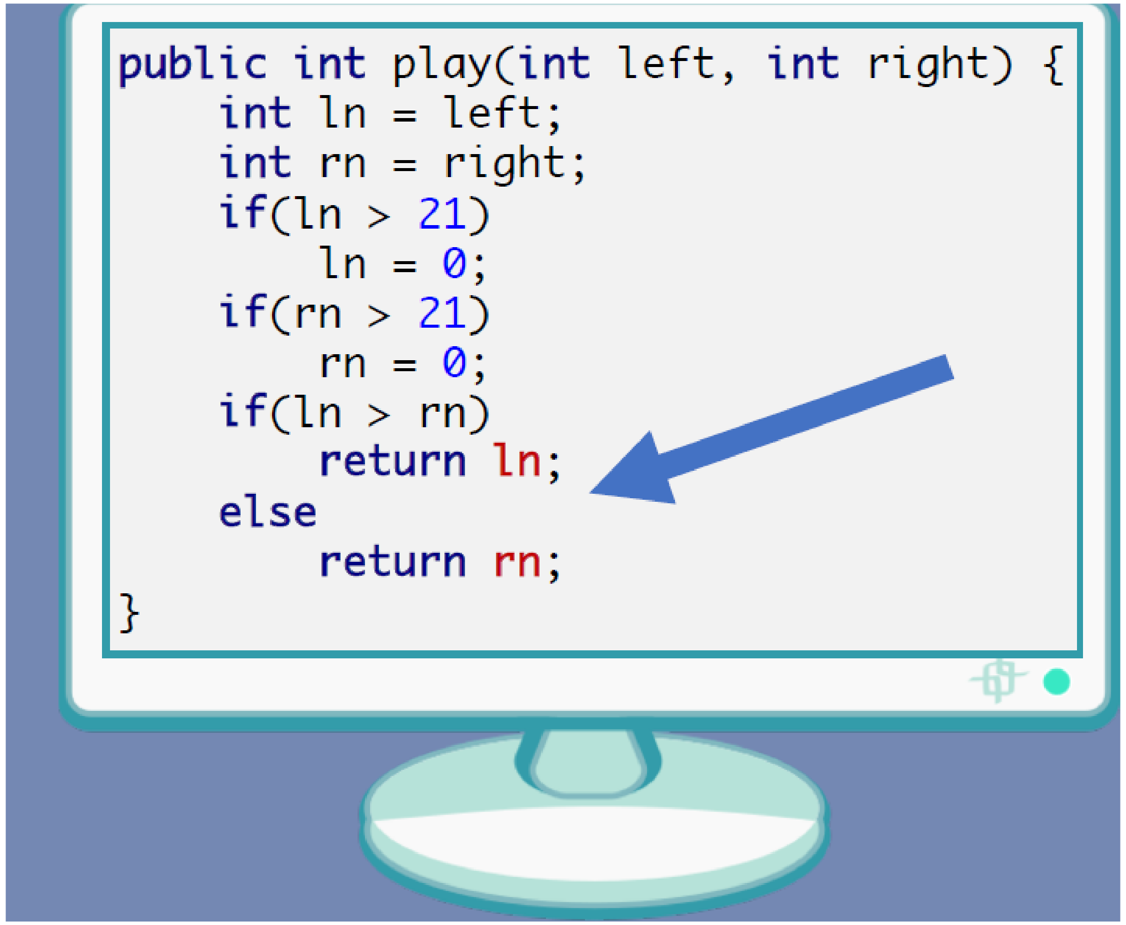
```java
package Code;

import static org.junit.jupiter.api.Assertions.*;

class BlackJackTest {
    BlackJack bj;

    @BeforeEach
    void setUp() throws Exception {
        bj=new BlackJack();
    }

    @Test
    void test1() {
        int result=bj.play(30, 30);
        assertEquals(0,result);
    }
    @Test
    void test2() {
        int result=bj.play(10, 9);
        assertEquals(10,result);
    }

}
```

- However, our test fails!!
- JUnit shows us that the program returns 9 instead of 10.
- So, there is a bug

Line Coverage – An Example



```
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 rn;
    else
        return ln;
}
```



```
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;
}
```

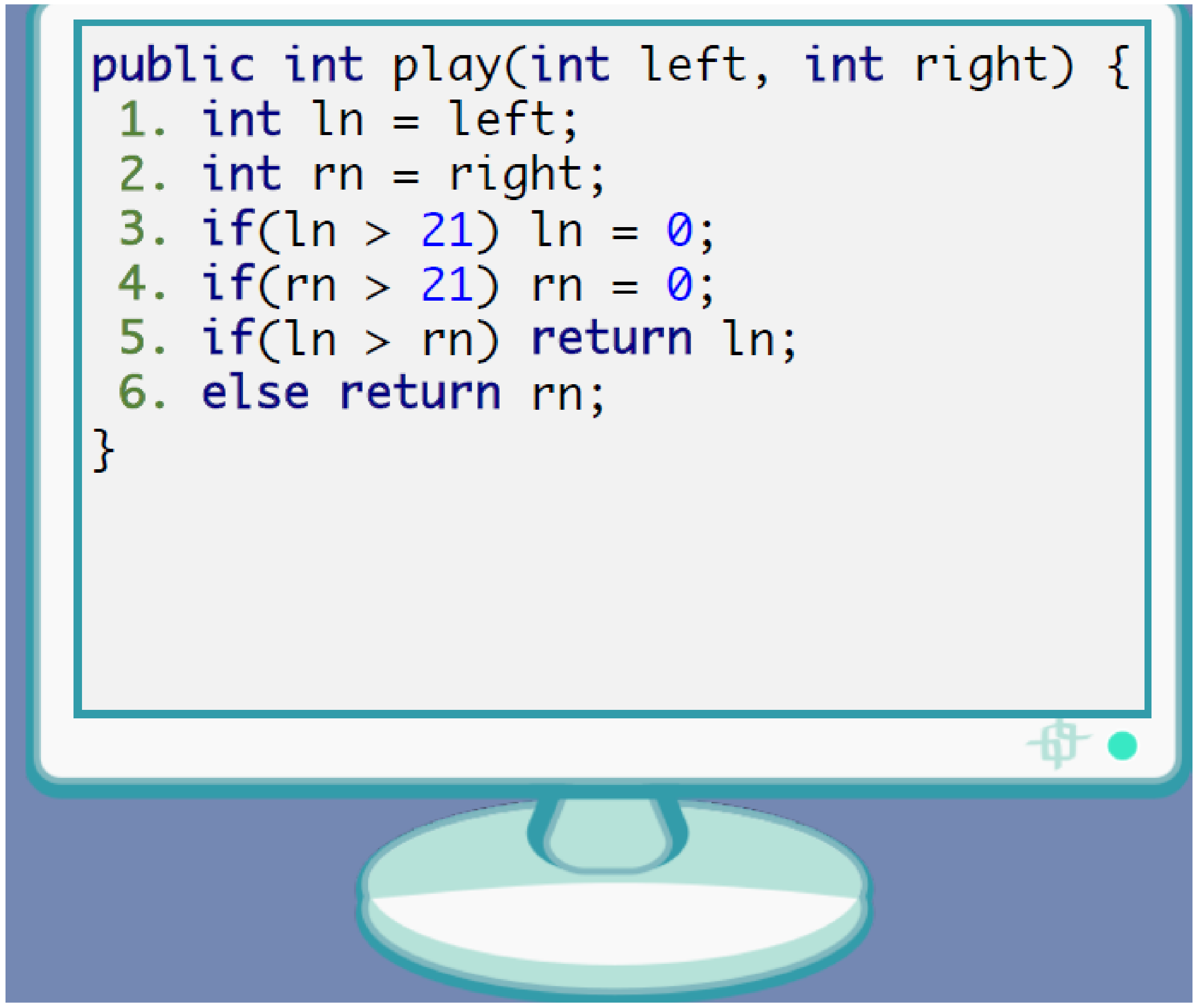
- If we go back to the implementation, we see that the developer changed the return variables!
- The first if, ln greater than rn should return "ln" and not "rn".

Line Coverage – Equation

$$\text{Line coverage} = \frac{\text{Number of lines exercised}}{\text{Total number of lines}} \times 100\%$$

Instruction Coverage

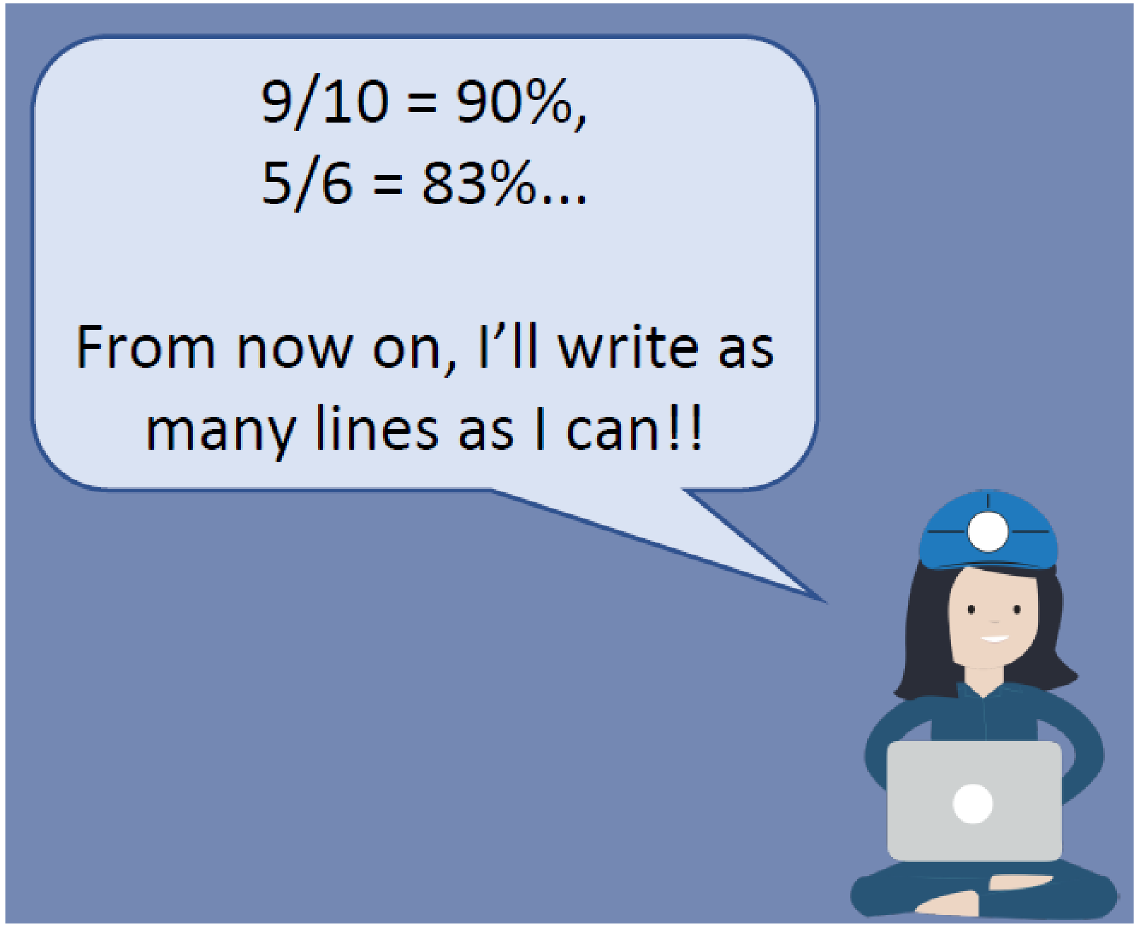
- ➢ Lines might not even be a totally good one.
- ➢ First, it can depend on how developers write their code.
- ➢ For example, the same code can have 10 lines, or 6 lines if we put the ifs in a single line...
- ➢ It really depends on the developer.



```
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;
}
```

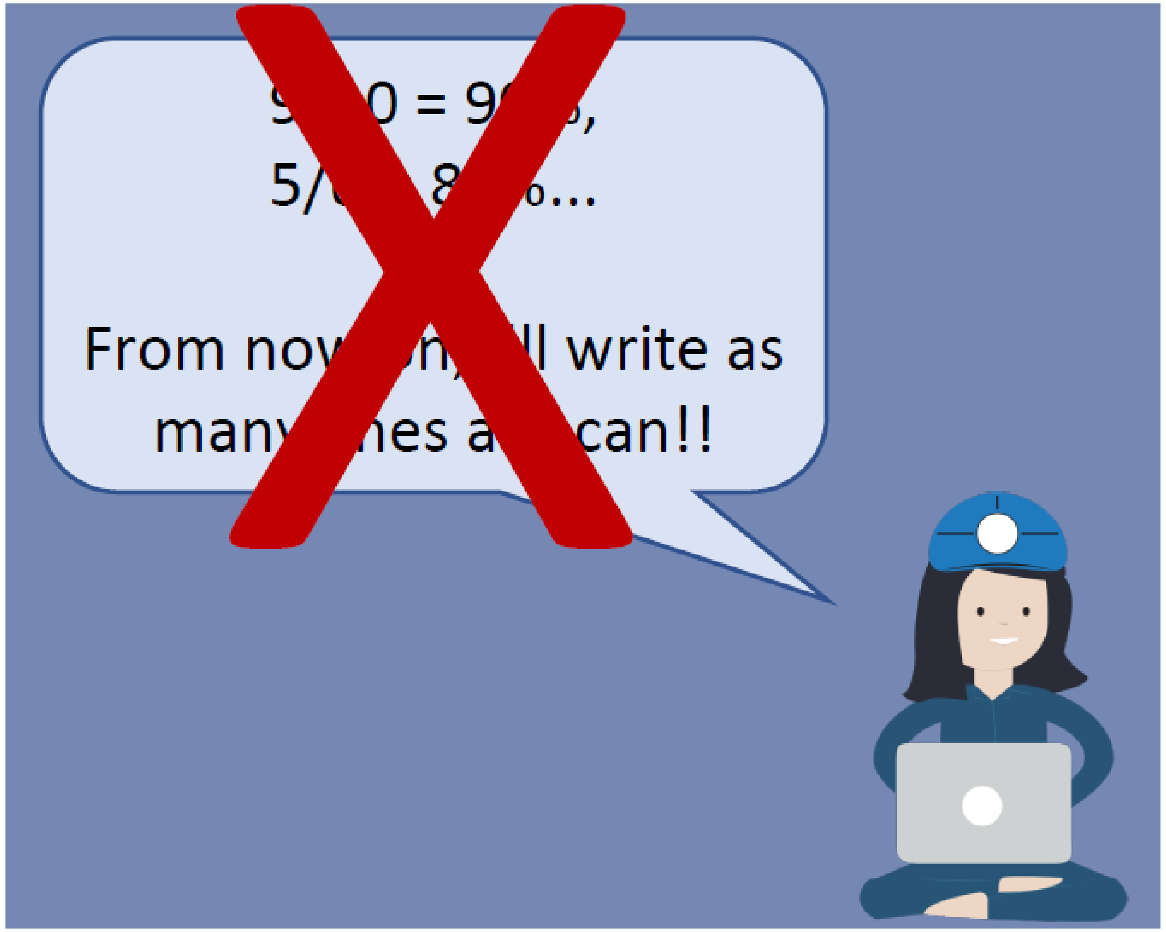
Instruction Coverage

- it completely changes the final outcome of our formula.
- If we don't test a line, we may have 9 divided by 10, which is 90% line coverage for the code with 10 lines of code
- and 5 divided by 6, which is 83% for the code with 6 lines...
- It seems that having more lines is better!!
- **Ok, Don't ever think like this again!.**



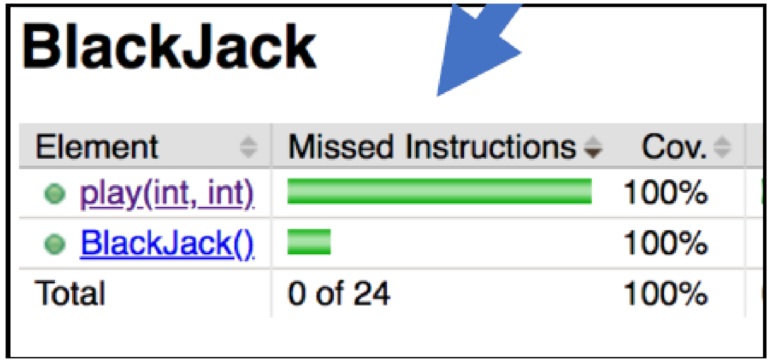
Instruction Coverage

- it completely changes the final outcome of our formula.
- If we don't test a line, we may have 9 divided by 10, which is 90% line coverage for the code with 10 lines of code
- and 5 divided by 6, which is 83% for the code with 6 lines...
- It seems that having more lines is better!!
- **Ok, Don't ever think like this again!.**



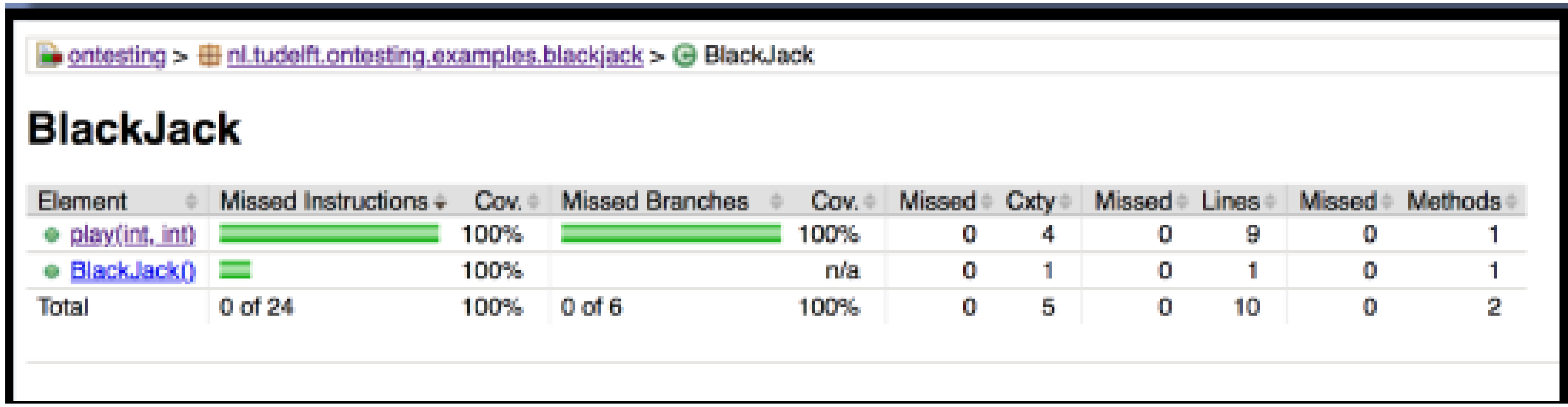
Instruction Coverage

- That's why some tools offer not only "line coverage", but also "statement coverage", meaning,
  - counting the number of different statements, regardless of the number of lines the developer uses to write them.
- Some other tools even show "instruction coverage", as they do it at byte code level!
- Eclipse use Instruction Coverage since its coverage plug in is based on Jacoco tool that supports instruction coverage



BlackJack

Element	Missed Instructions	Cov.
play(int, int)	🟩	100%
BlackJack()	🟩	100%
Total	0 of 24	100%



ontesting > nl.tudelft.ontesting.examples.blackjack > BlackJack

BlackJack

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
play(int, int)	🟩	100%	🟩	100%	0	4	0	9	0	1
BlackJack()	🟩	100%		n/a	0	1	0	1	0	1
Total	0 of 24	100%	0 of 6	100%	0	5	0	10	0	2

Instruction Coverage

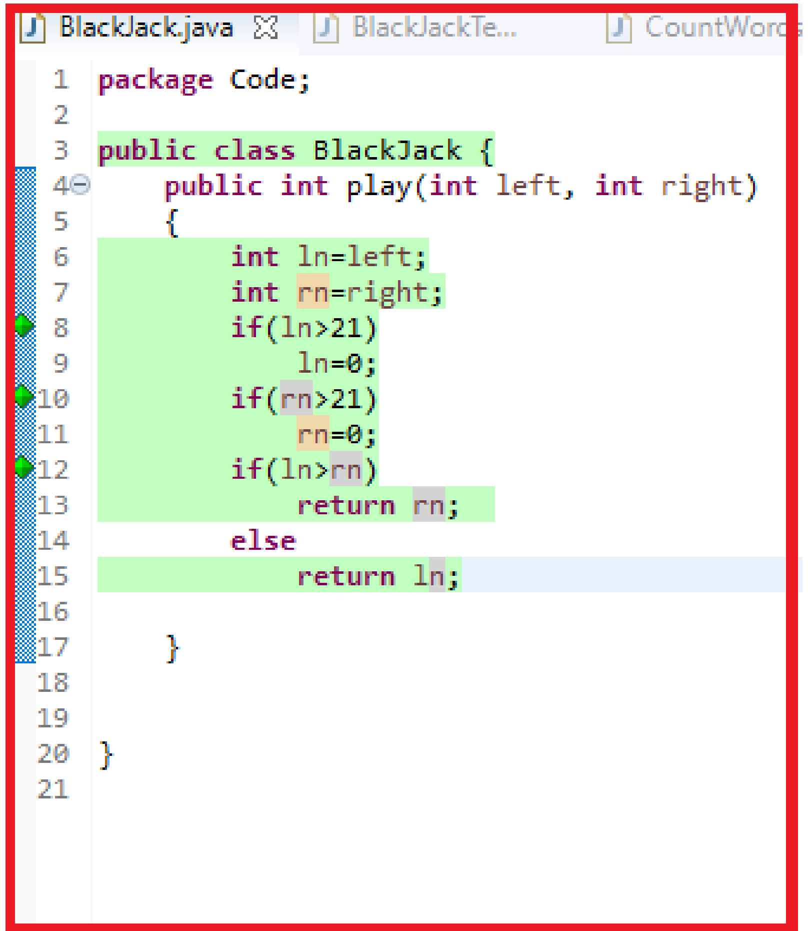
- Research and empirical experience actually show that when testers only do black box testing or, in other words, when they test without using the source code as inspiration, they only achieve 60% to 75% coverage.

- So, we should also think about structural tests when testing our systems.

- But, you probably feel that, although we have 100% line coverage, our tests are not enough.

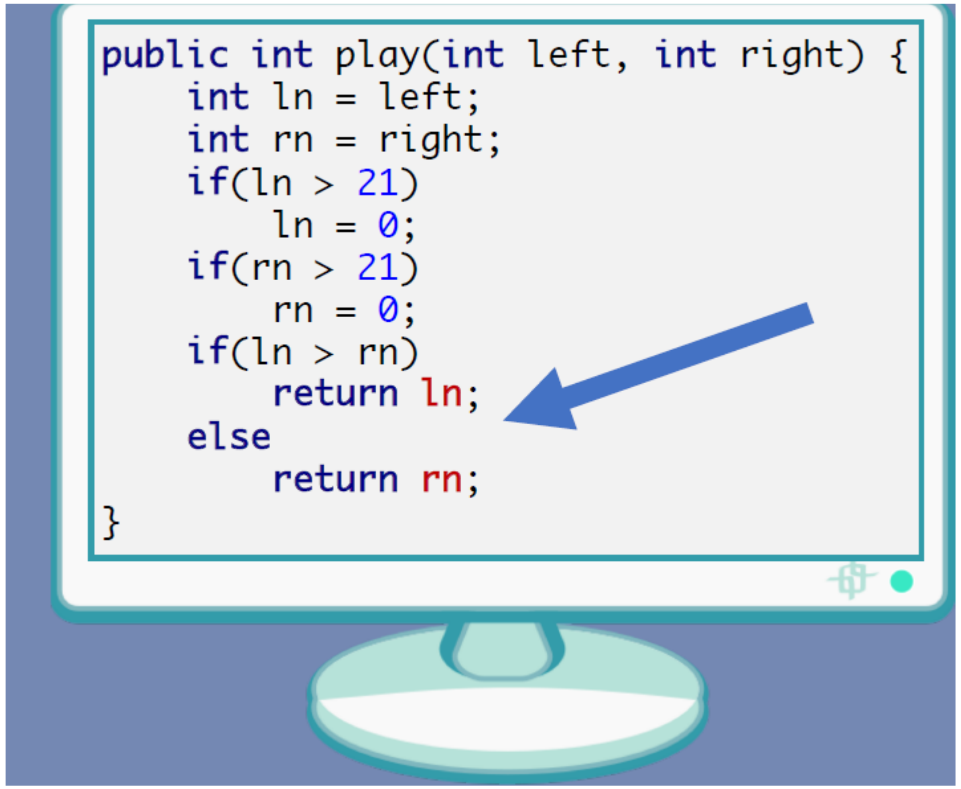
Instruction Coverage

- ➢ Right now we only test if both players have more than 21, and if the left player wins.
  - ❑ What about the right player? What about if they have the same points? What about the boundaries?



BlackJack.java | BlackJackTe... | CountWords

```java
package Code;

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 rn;
        else
            return ln;

    }


}
```

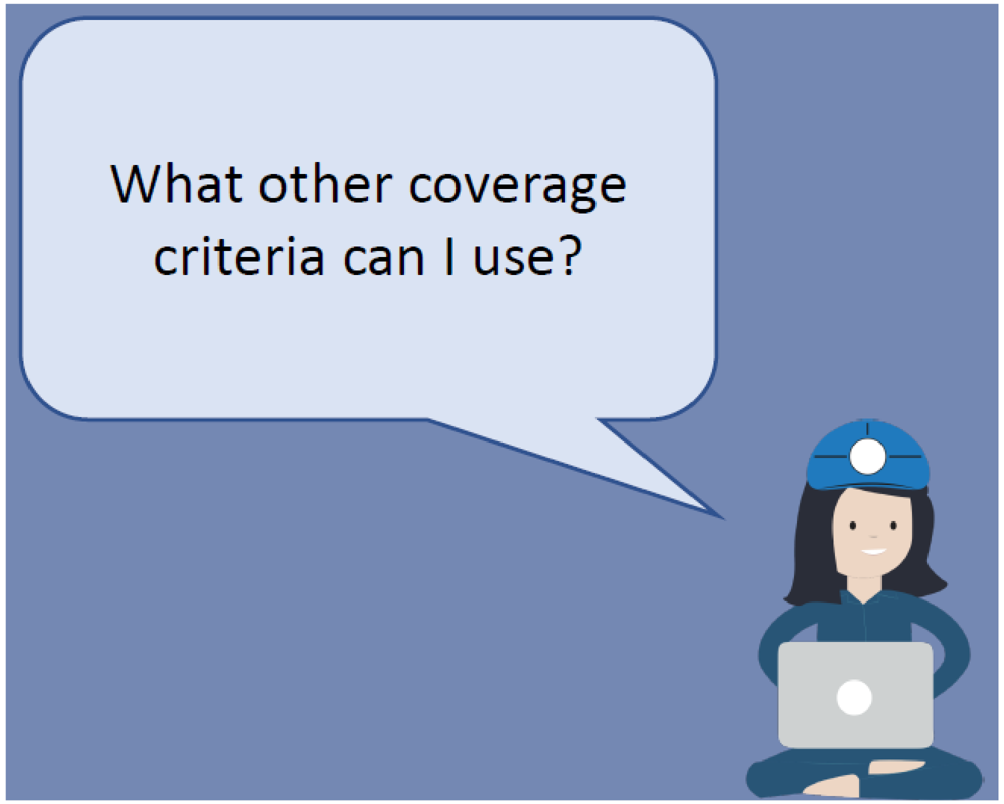


```java
public int play(int left, int right) {
    int ln = left;
    int rn = right;
    if(ln > 21)
        ln = 0;
    if(rn > 21)
        rn = 0;
    if(ln > rn)
        return ln;
    else
        return rn;
}
```

```java
@Test
void test1() {
    int result=bj.play(30, 30);
    assertEquals(0,result);
}
@Test
void test2() {
    int result=bj.play(10, 9);
    assertEquals(10,result);
}
```

Branch Coverage

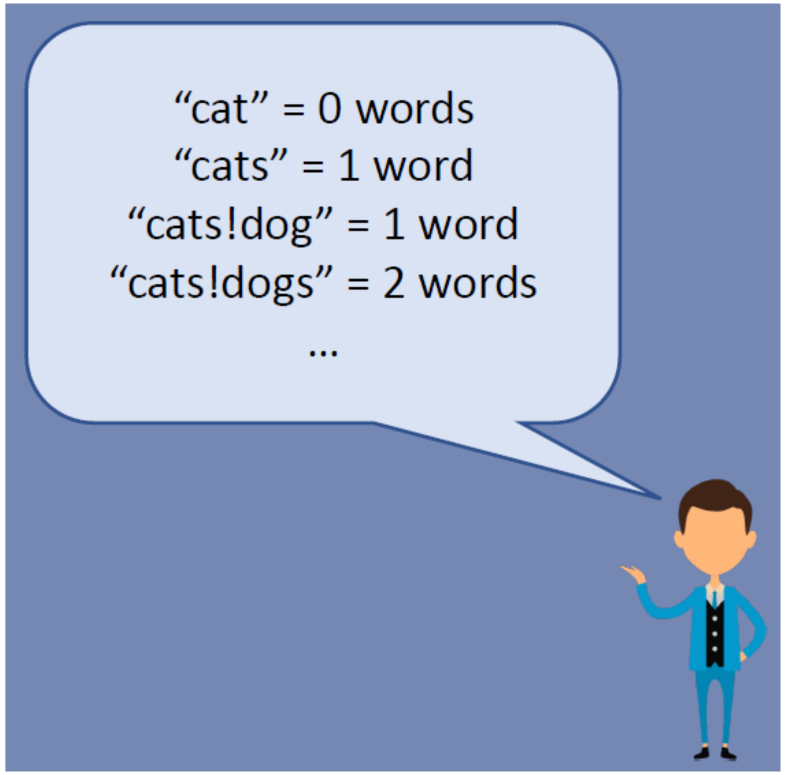
- Can we go further than line coverage?
- We sure can!
- After all, looking only at lines might not be enough when programs get more complicated.

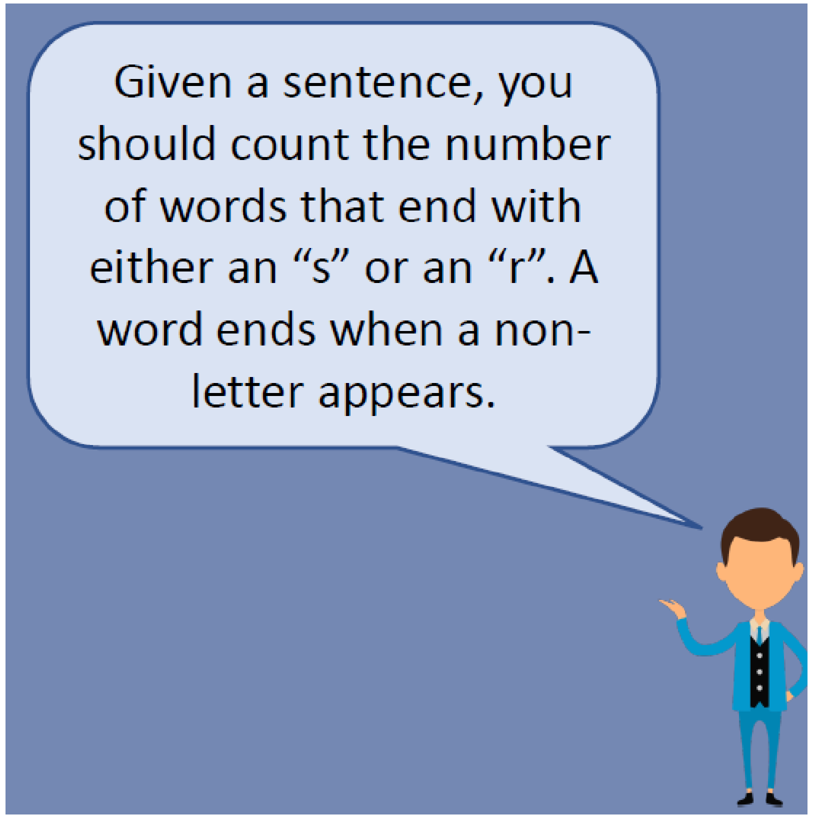


Branch Coverage

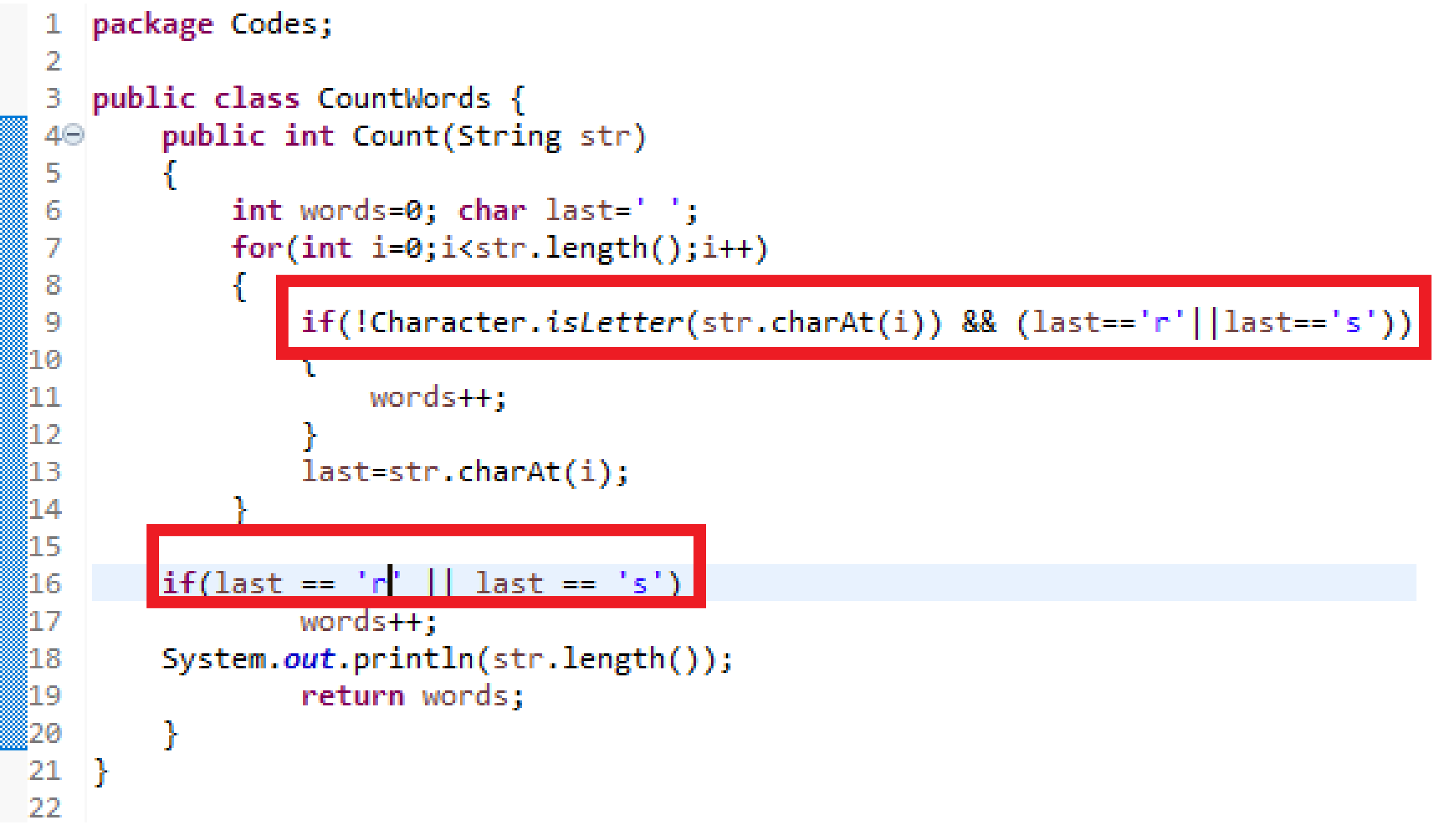
- Take a look at this problem: we should count the number of words in a sentence that end with "s" or "r".

- And we consider a new word to start when there's a non letter character in the middle.




Given a sentence, you should count the number of words that end with either an "s" or an "r". A word ends when a non-letter appears.

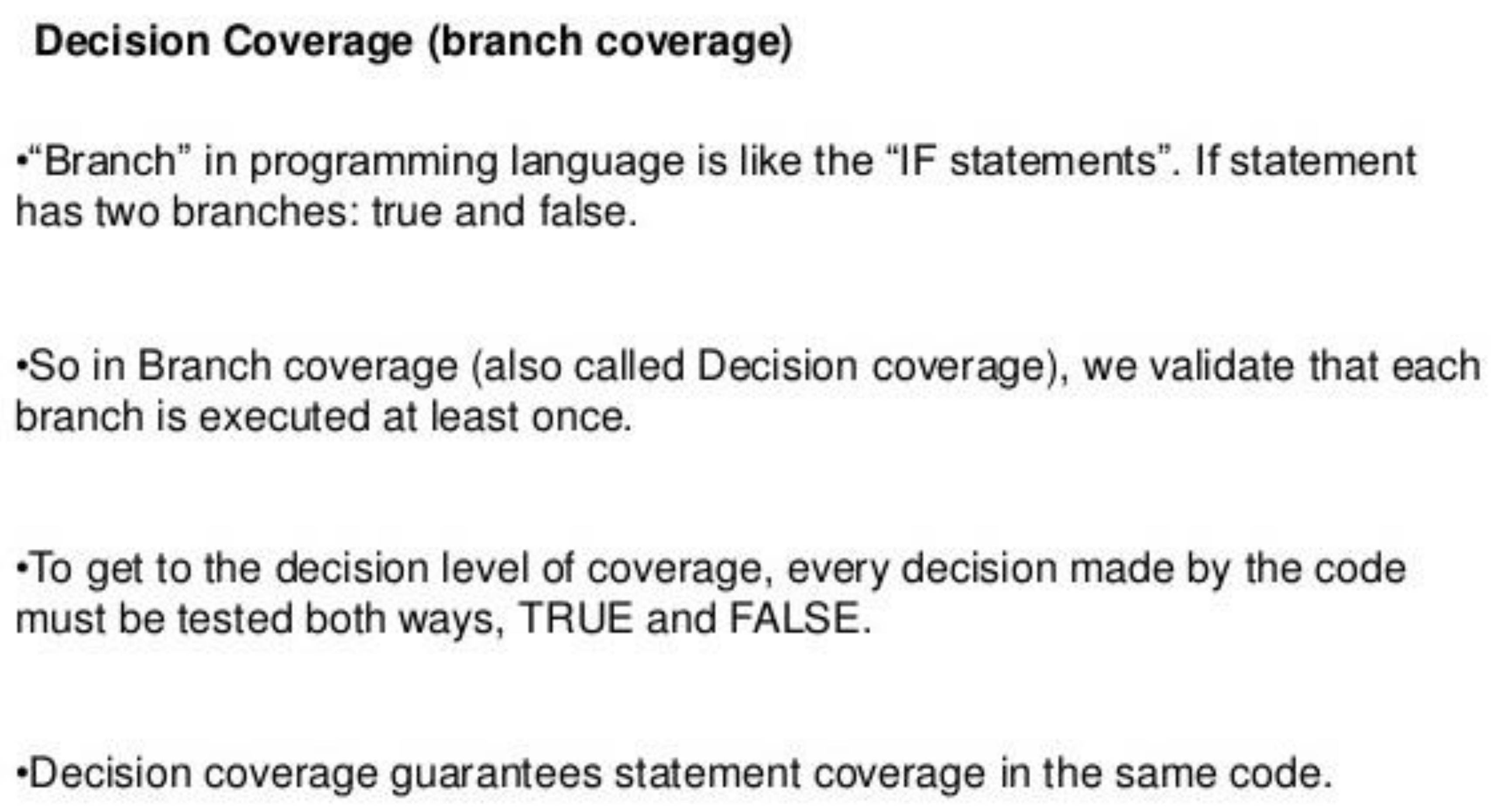
Branch Coverage



```
package Codes;

public class CountWords {
    public int Count(String str)
    {
        int words=0; char last=' ';
        for(int i=0;i<str.length();i++)
        {
            if(!Character.isLetter(str.charAt(i)) && (last=='r'||last=='s'))
            {
                words++;
            }
            last=str.charAt(i);
        }

    if(last == 'r' || last == 's')
            words++;
    System.out.println(str.length());
            return words;
    }
}
```

- Although this problem might look simple, the implementation might need a few tricks
- For example, while visiting the string, we should keep track of the last character visited
- As soon as we find something that is not a letter, we check if the last character was either an "s" or an "r“
- As you can see, the "if" expressions we have there are quite complex
- They create a good number of different paths that our program can take

Branch Coverage



Decision Coverage (branch coverage)

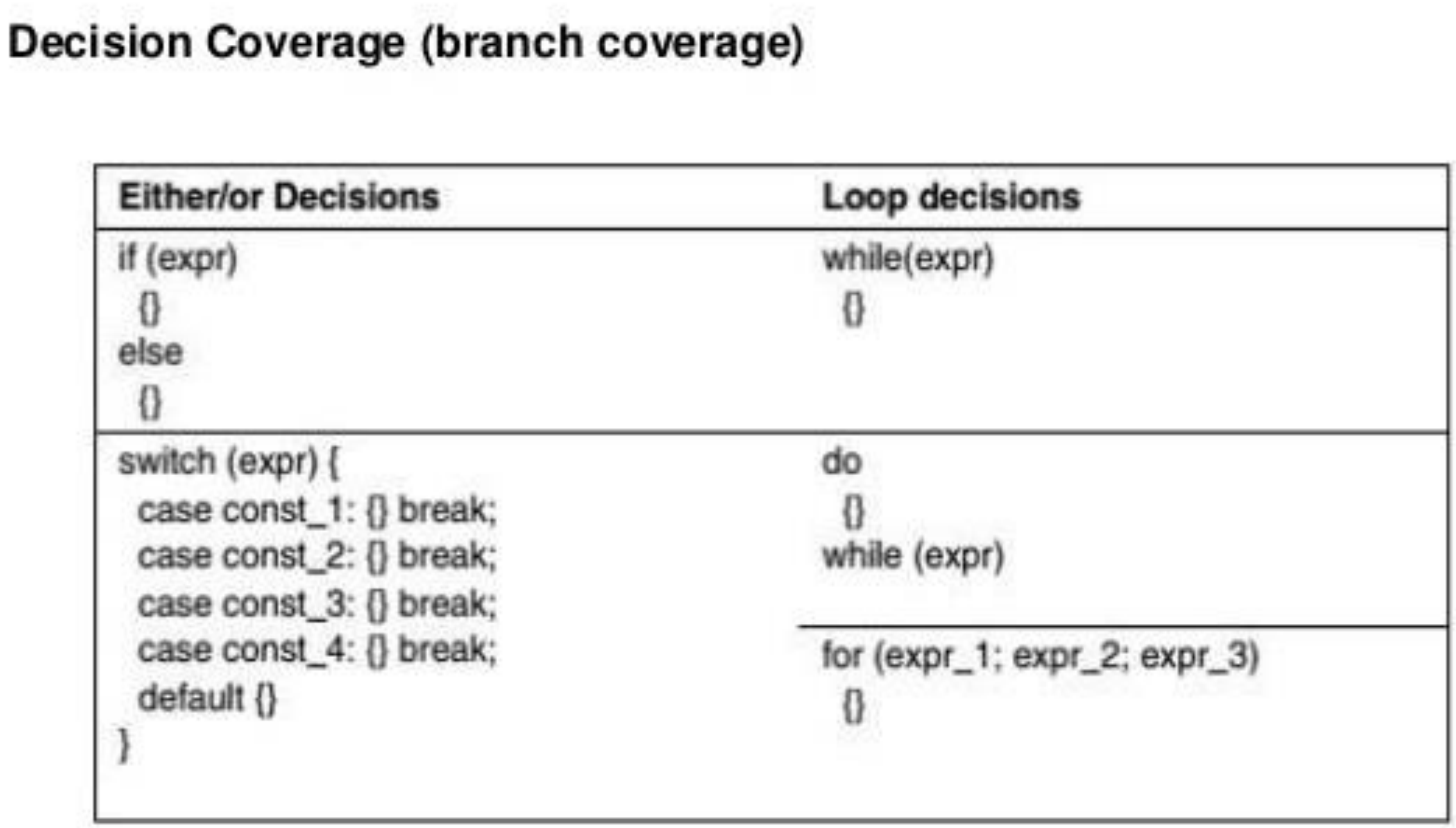
•“Branch” in programming language is like the “IF statements”. If statement has two branches: true and false.

•So in Branch coverage (also called Decision coverage), we validate that each branch is executed at least once.

•To get to the decision level of coverage, every decision made by the code must be tested both ways, TRUE and FALSE.

•Decision coverage guarantees statement coverage in the same code.

Branch Coverage



Decision Coverage (branch coverage)

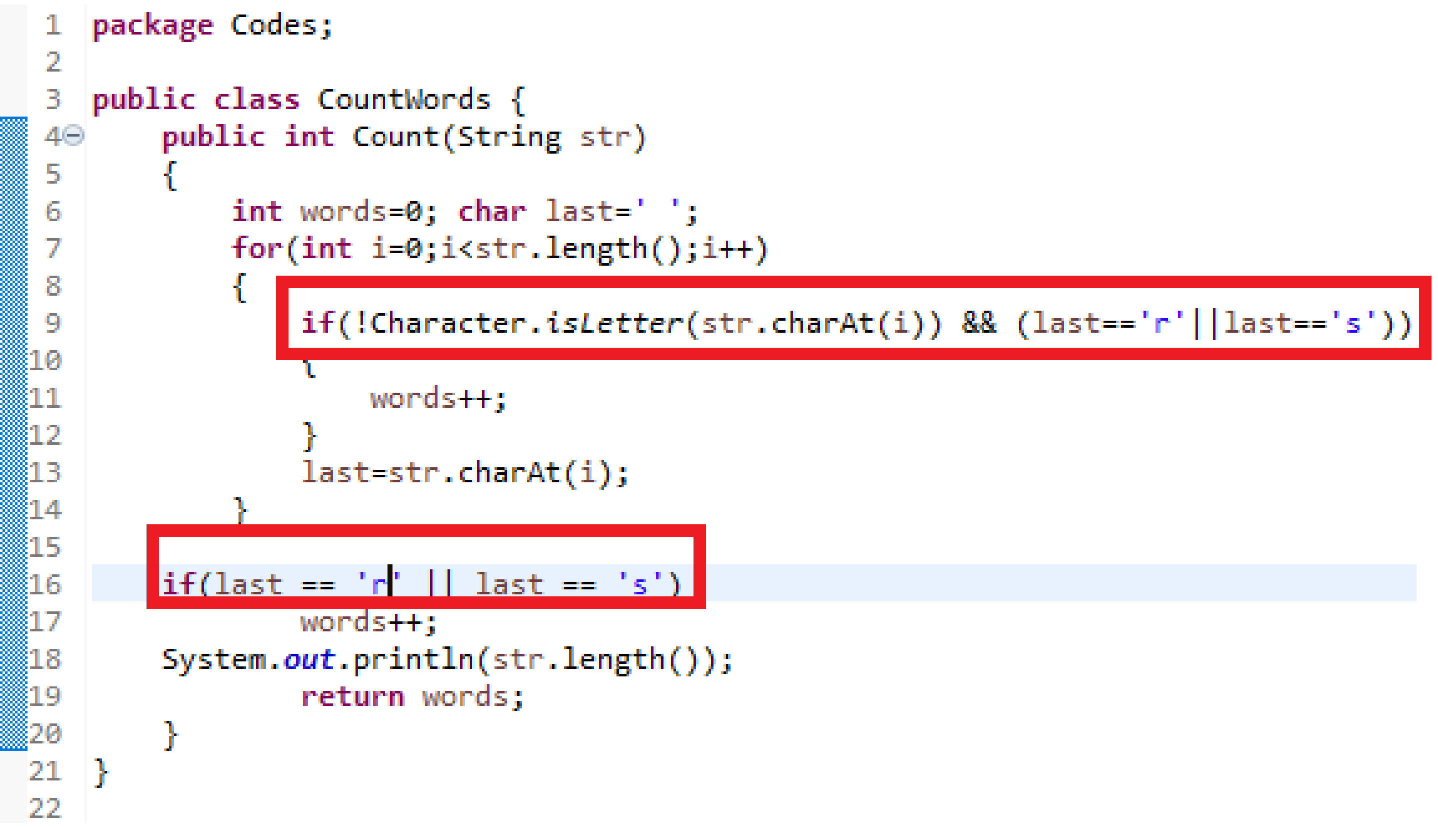
Either/or Decisions	Loop decisions

if (expr)
  {}
else
  {}
	while(expr)
	  {}

switch (expr) {
  case const_1: {} break;
  case const_2: {} break;
  case const_3: {} break;
  case const_4: {} break;
  default {}
}
	do
	  {}
	while (expr)

	for (expr_1; expr_2; expr_3)
	  {}

Branch Coverage

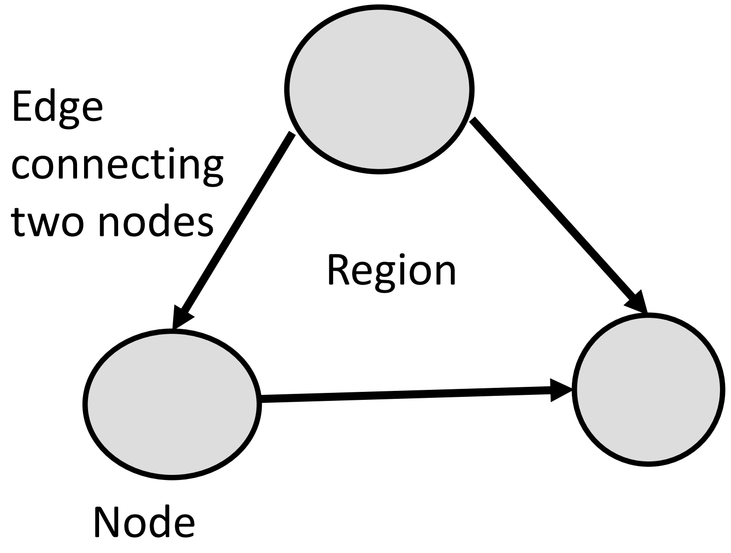


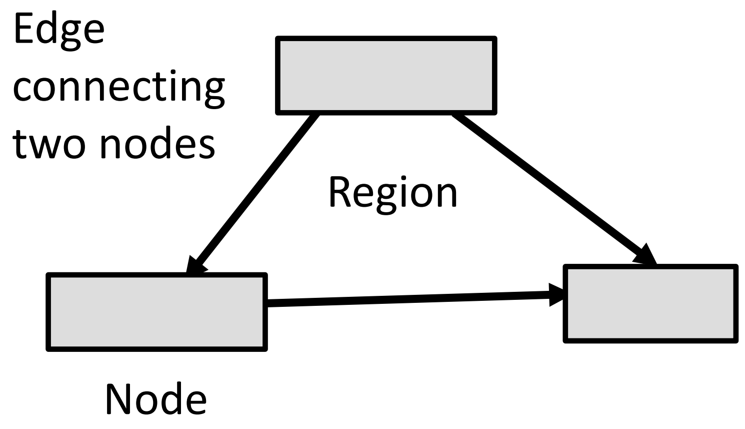
```
package Codes;

public class CountWords {
    public int Count(String str)
    {
        int words=0; char last=' ';
        for(int i=0;i<str.length();i++)
        {
            if(!Character.isLetter(str.charAt(i)) && (last=='r'||last=='s'))
            {
                words++;
            }
            last=str.charAt(i);
        }

    if(last == 'r' || last == 's')
            words++;
    System.out.println(str.length());
            return words;
    }
}
```

- To create test cases for achieving branch coverage we will draw control flow graph to see the different paths (or, as we also call, branches) that are possible in our implementation.

Control Flow Graph

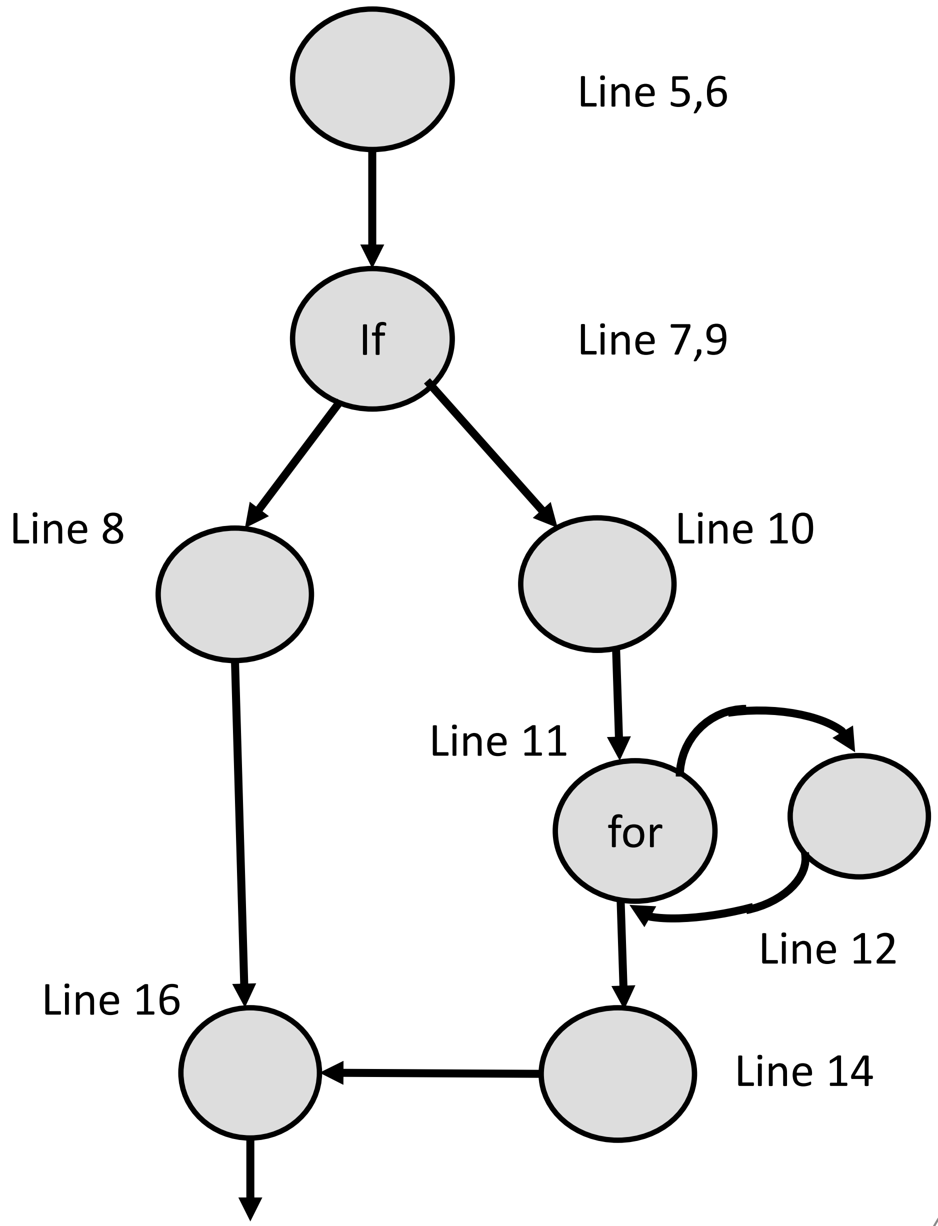
- A control flow graph provides a method for representing the decision points and the flow of control within a piece of code
- like a flow chart, but only shows decisions
- It is produced by looking only at the statements affecting the flow of control
- The graph itself is made up of two symbols
- A node represents any point where the flow of control can be modified
- An edge is a line connecting any two nodes
- The closed area contained within a collection of nodes and edges, as shown in the diagram, is known as a region



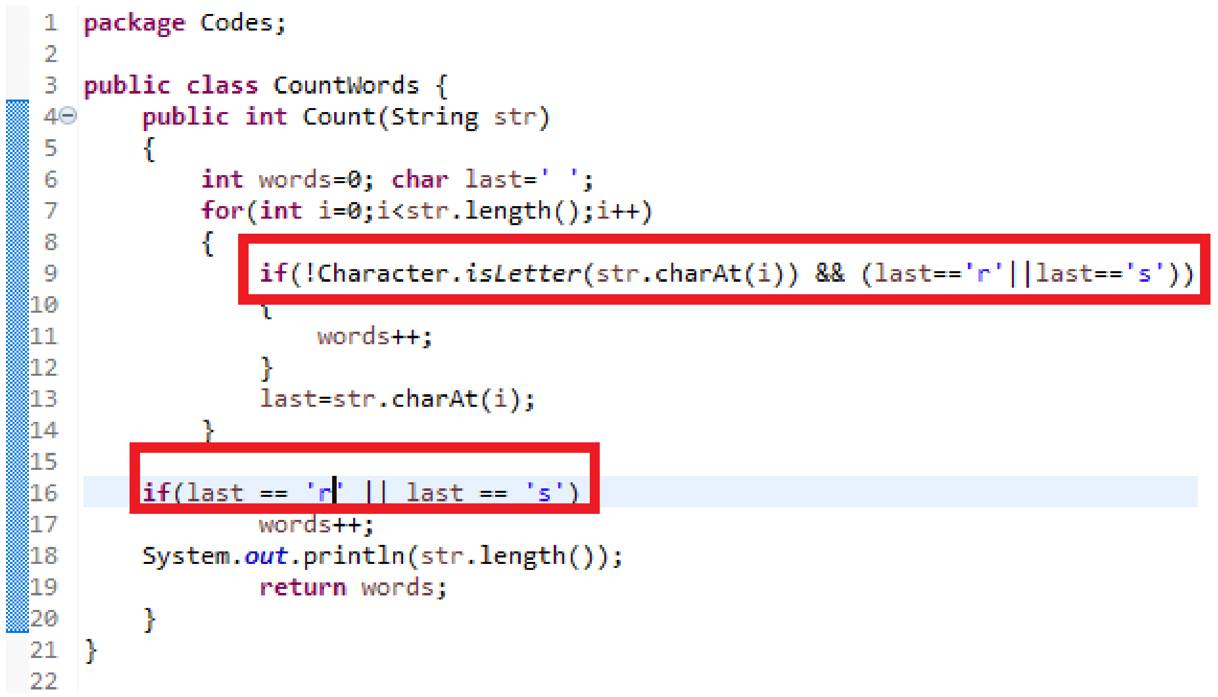


Control Flow Graph – An Example

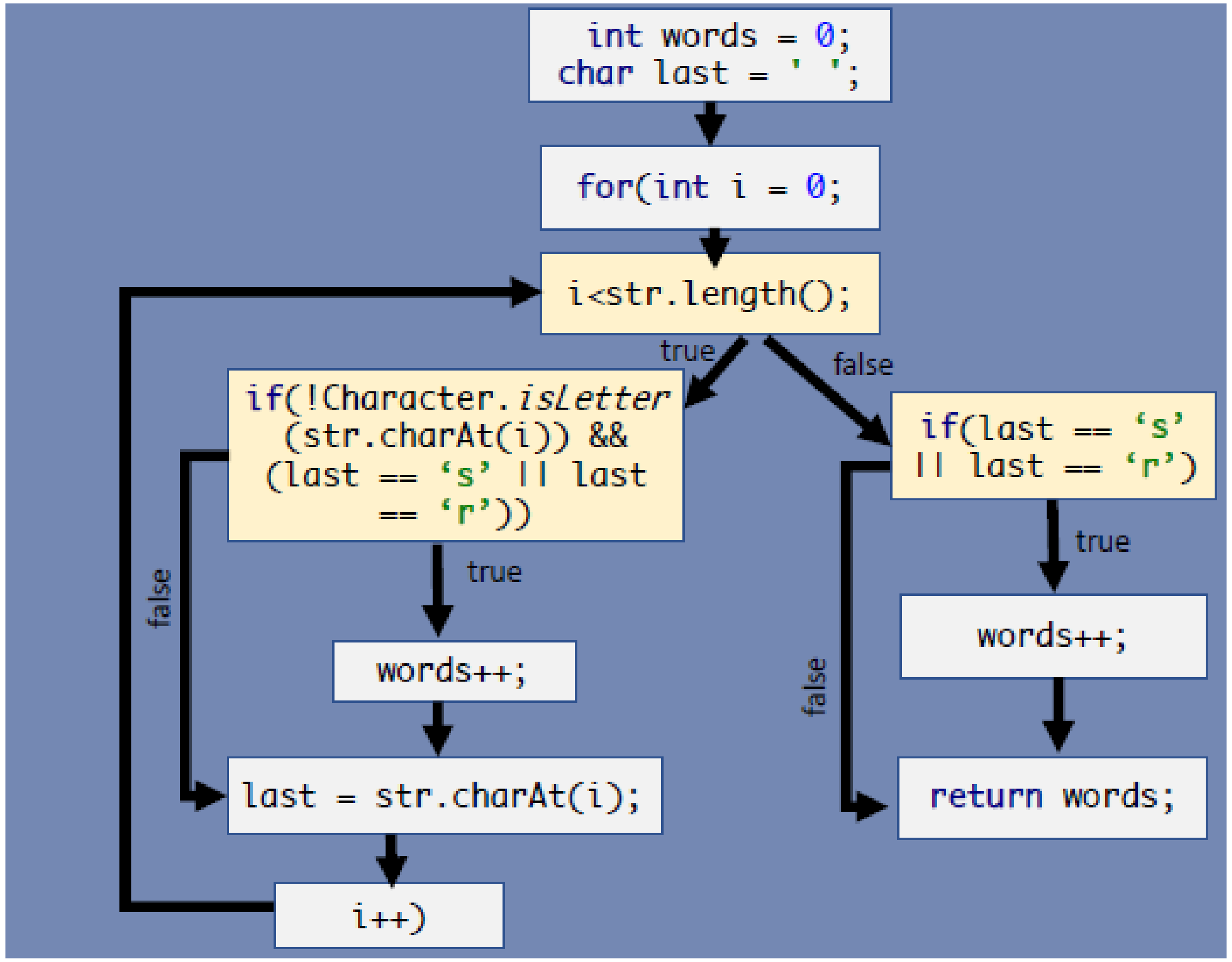
```
1.  class FactorialExample{
2.  public static void calFact( ){
3.    int i,n, fact;
4.  Scanner scan = new Scanner(System.in);
5.  System.out.println("Enter value of n");
6.  n=scan.nextInt();
7.   if(n<0) {
8.              System.out.println("Invalid value of"+n);
9.  } else  {
10.             fact=1;
11.             for(i=1;i<=n;i++){
12.                 fact=fact*i;
13.             }
14.             System.out.println("Factorial of "+n+" is: "+fact);
15.         }
16. }
17. }
```



Back to Branch Coverage Example



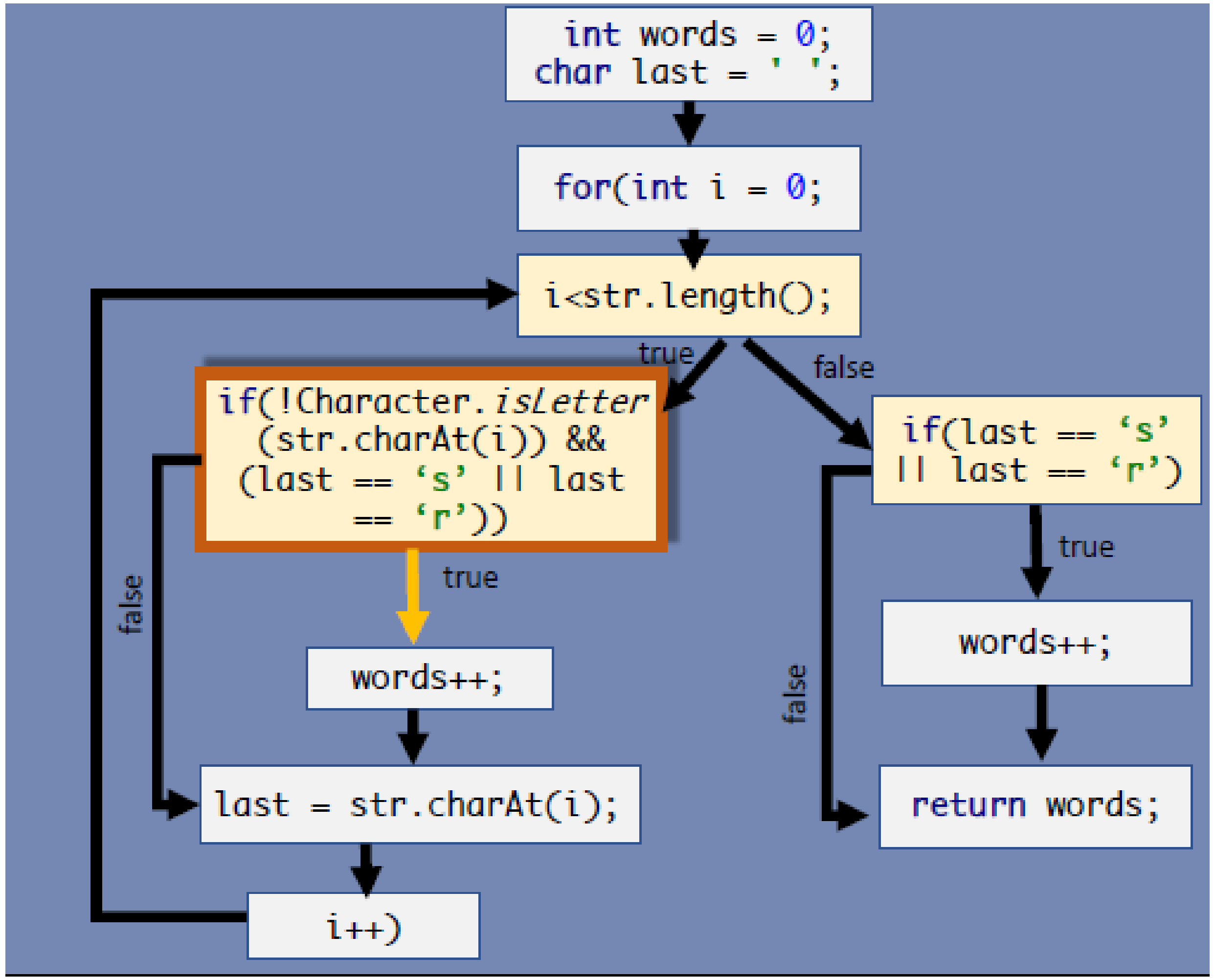
```
package Codes;

public class CountWords {
    public int Count(String str)
    {
        int words=0; char last=' ';
        for(int i=0;i<str.length();i++)
        {
            if(!Character.isLetter(str.charAt(i)) && (last=='r'||last=='s'))
            {
                words++;
            }
            last=str.charAt(i);
        }

    if(last == 'r' || last == 's')
            words++;
    System.out.println(str.length());
            return words;
    }
}
```



- ➢ In this graph, the yellow boxes represent decision points (meaning, ifs, fors, etc).
- ➢ As you see, after every yellow box, we have two arrows.
- ➢ One points to where the program goes when that condition is evaluated to true,
- ➢ The other one to where the program goes when that condition is evaluated to false.

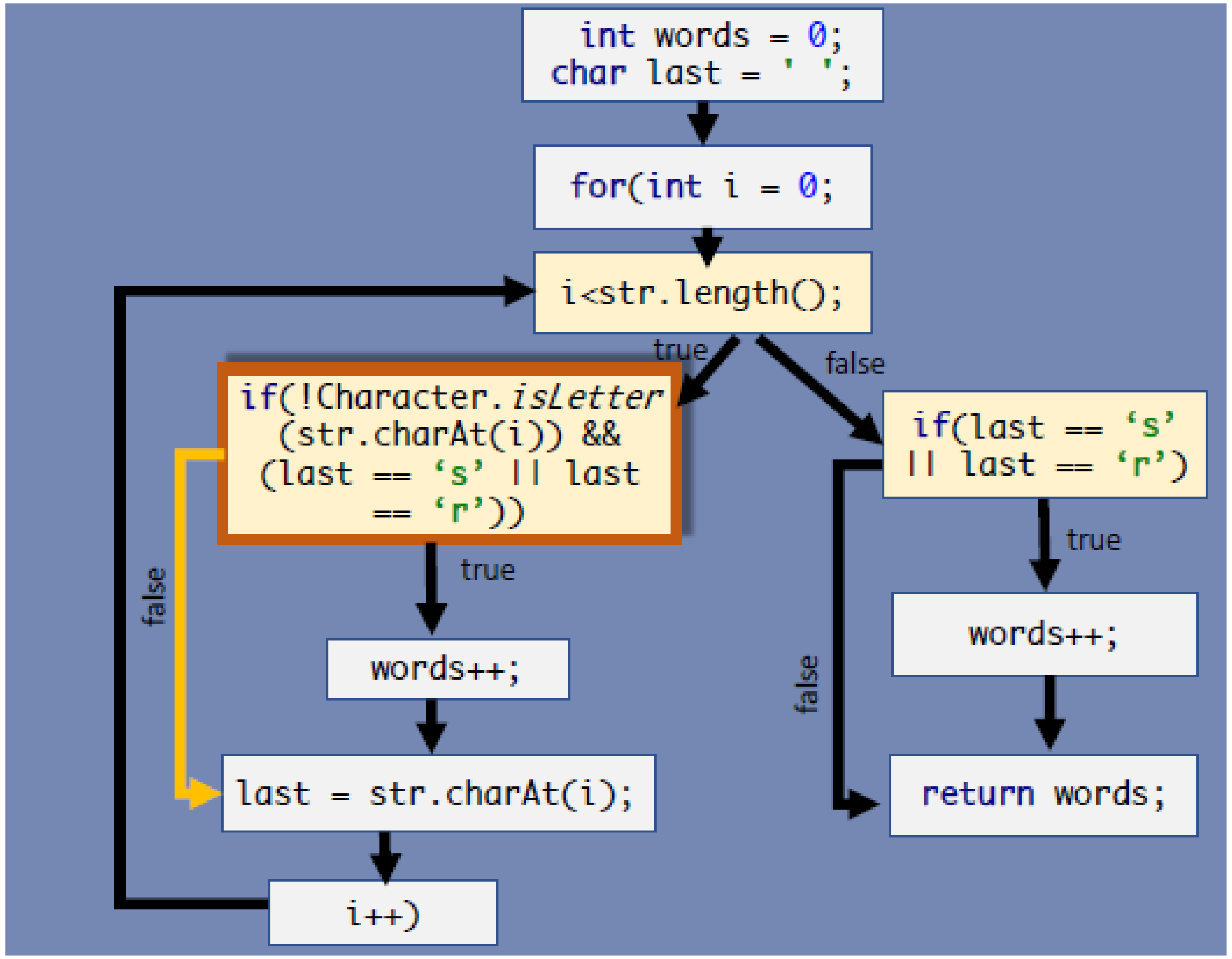
Branch Coverage Example

- we have at least one test for each branch or for each arrow there.
- For example, the big if we have: one of the tests should exercise the true path.
- In this one, the words++ is executed.



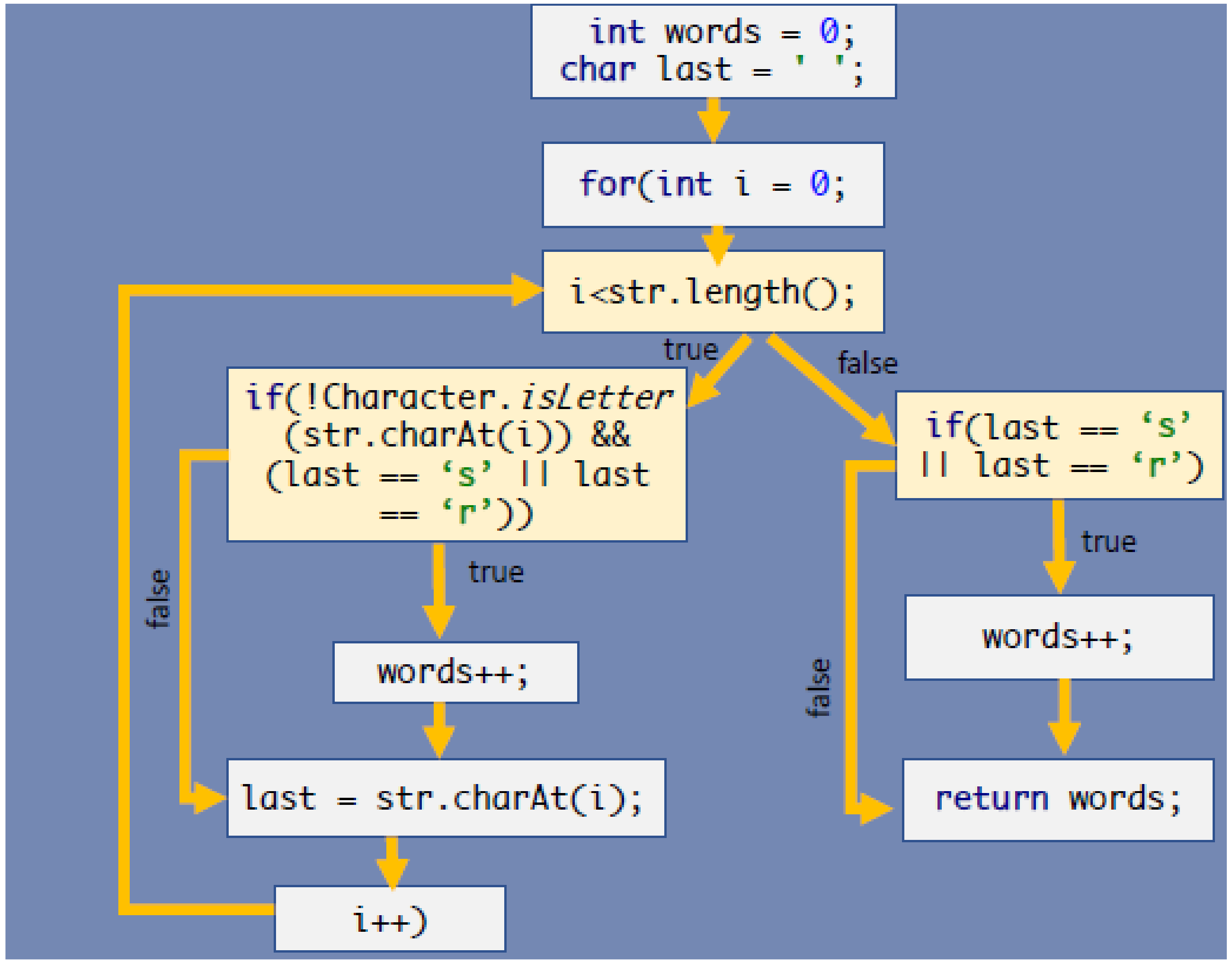
Branch Coverage Example

- Then, another test should exercise the false path.
- In this one, the words++ is not executed.

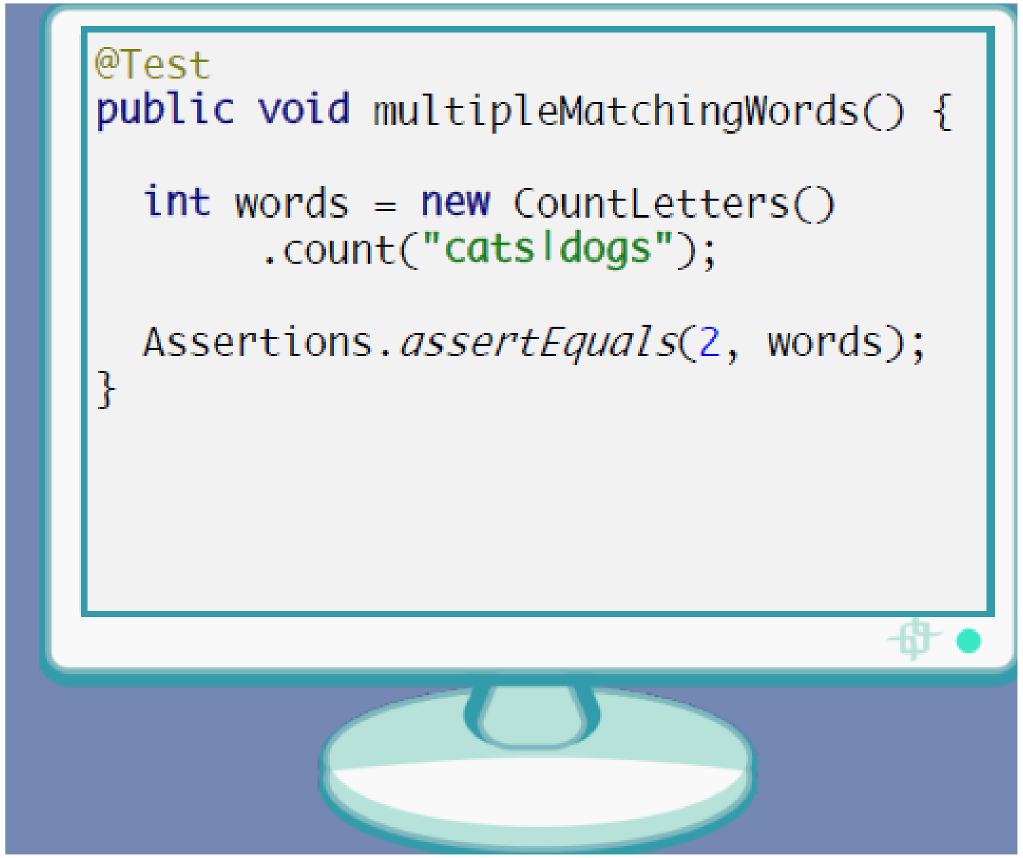


Branch Coverage Example

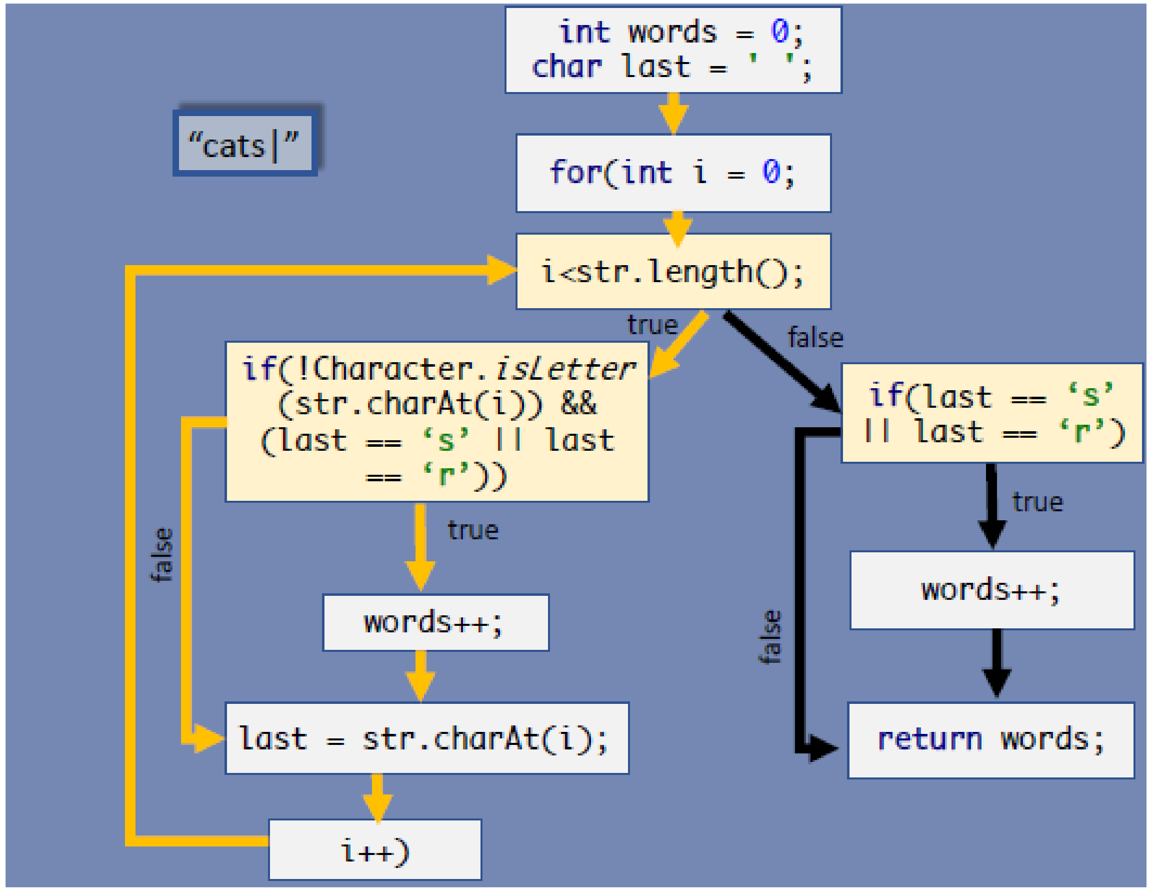
- We then repeat it until all the branches are covered!



Branch Coverage Example



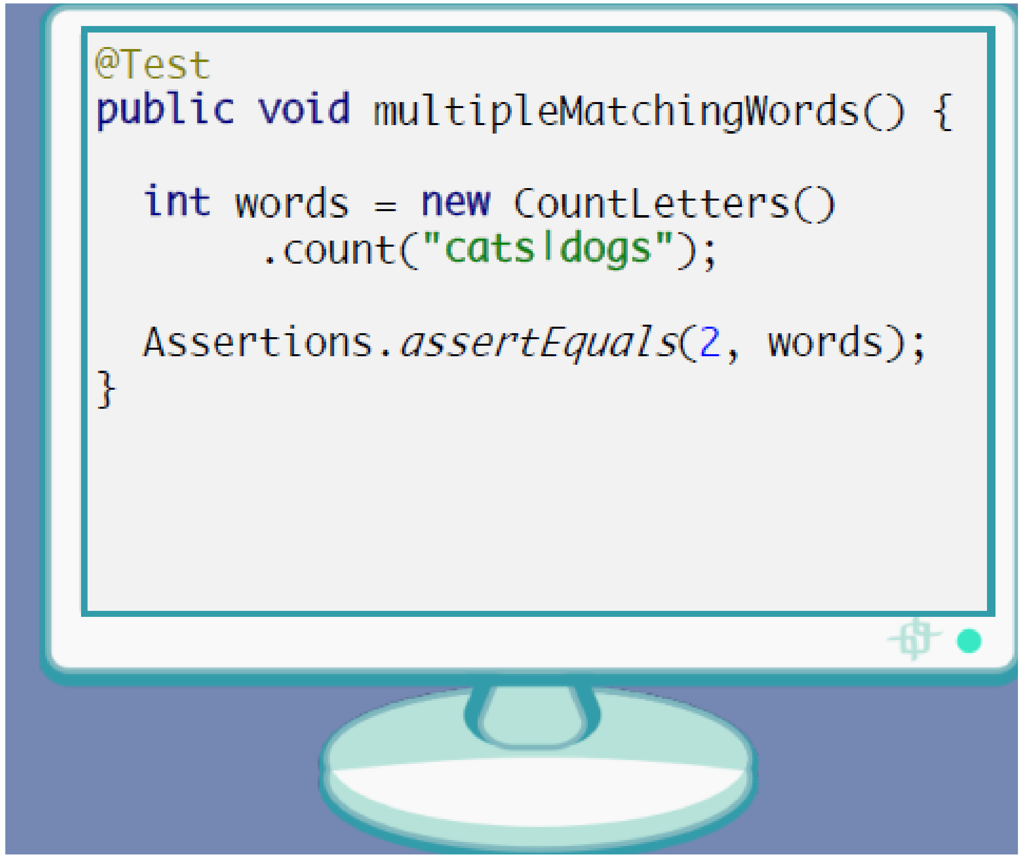
```
@Test
public void multipleMatchingWords() {

  int words = new CountLetters()
        .count("cats|dogs");

  Assertions.assertEquals(2, words);
}
```



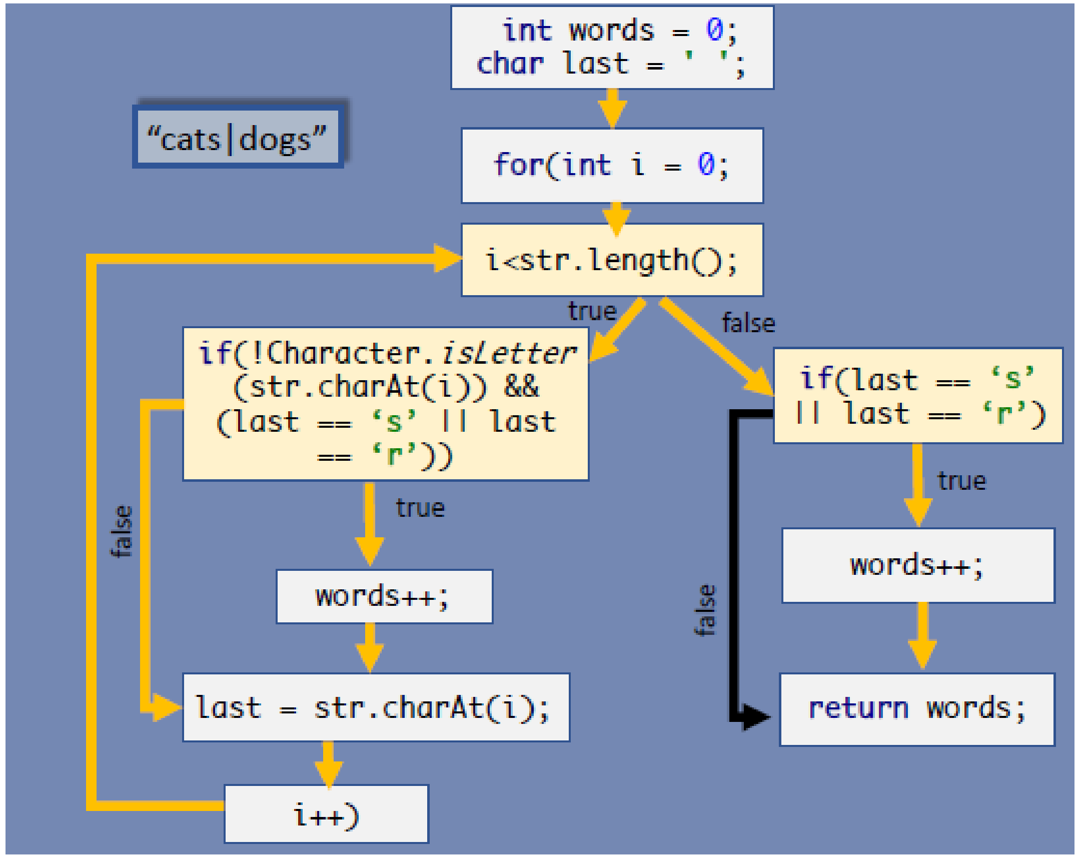
- Take a moment to follow the execution of this test in the control flow graph.

- We see that, after the string "cats|", the left part of our control flow graph is already completely covered.

Branch Coverage Example

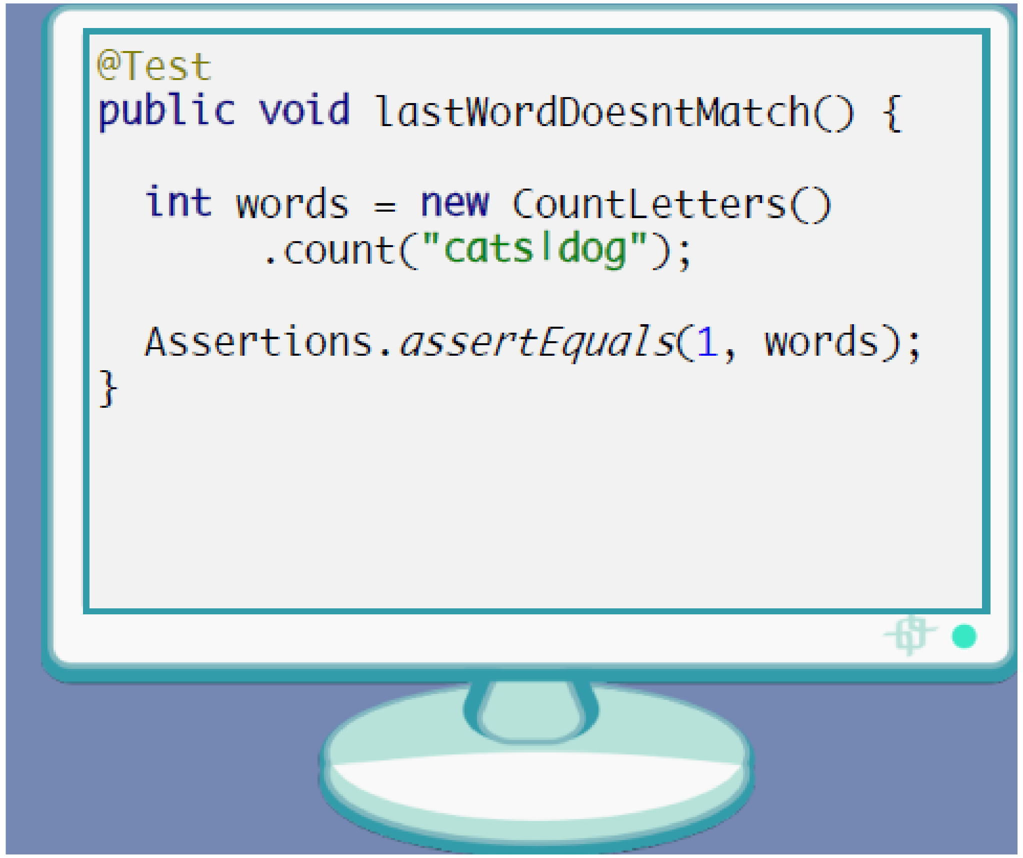


```
@Test
public void multipleMatchingWords() {

  int words = new CountLetters()
        .count("cats|dogs");

  Assertions.assertEquals(2, words);
}
```

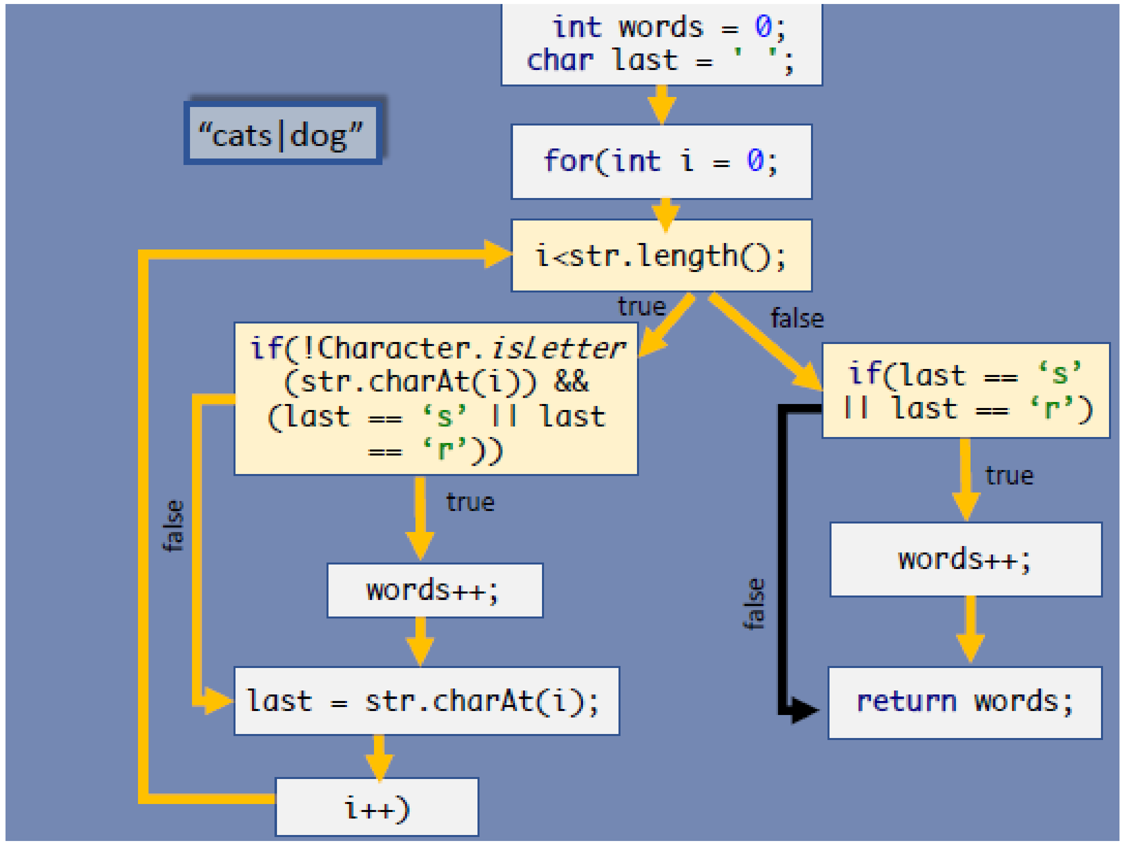


- ➢ Then, after "dogs", then the other if is evaluated to true, which means we cover the "true branch".
- ➢ Ok, almost there...
- ➢ But we still need to exercise the false branch.
- ➢ This means we need one more test.

Branch Coverage Example

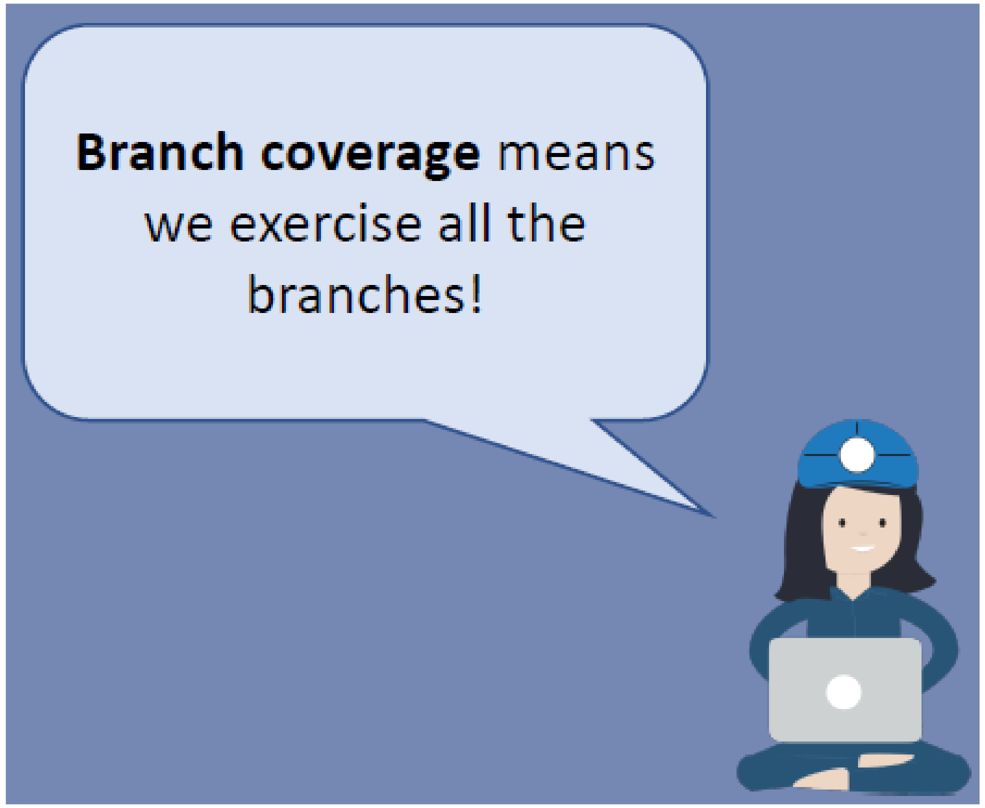


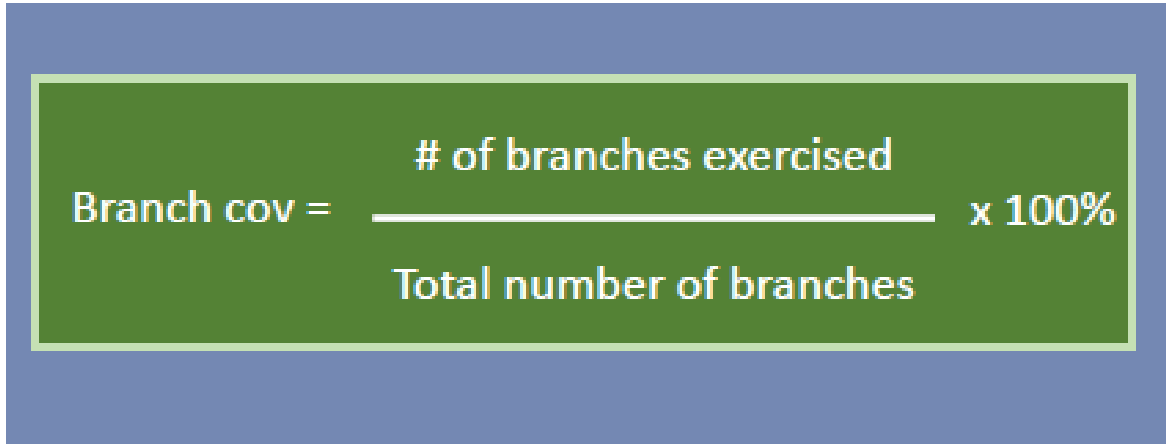
```
@Test
public void lastWordDoesntMatch() {

  int words = new CountLetters()
        .count("cats|dog");

  Assertions.assertEquals(1, words);
}
```



➢ As "dog", which is the last word in the string, doesn't end with an "s" or an "r", then the false branch will be executed.

Branch Coverage Equation





$$\text{Branch cov} = \frac{\text{\# of branches exercised}}{\text{Total number of branches}} \times 100\%$$

- This is very similar to the line coverage formula; the only difference is now that we count branches, and not lines.

- But...
- can we still do better?

Limitation of Branch Coverage

- When there are multiple conditions - sub conditions in a conditional expression then just achieving branch coverage may not be able to detect all bugs.

```
void main()
{ float x, y;
Scanner sc=new Scanner(System.in);
x=sc.nextFloat();
 y=sc.nextFloat();
if((x==0)||(y>0))
    y=y/x;
else
    x=y+2;
System.out.println(x);
System.out.println(y);
}
```

- Branch Coverage Test Case:

x=5,y=6;   x=5,y=-6;

Branch Coverage 100%

- What happens If x==0 ?

Condition Coverage

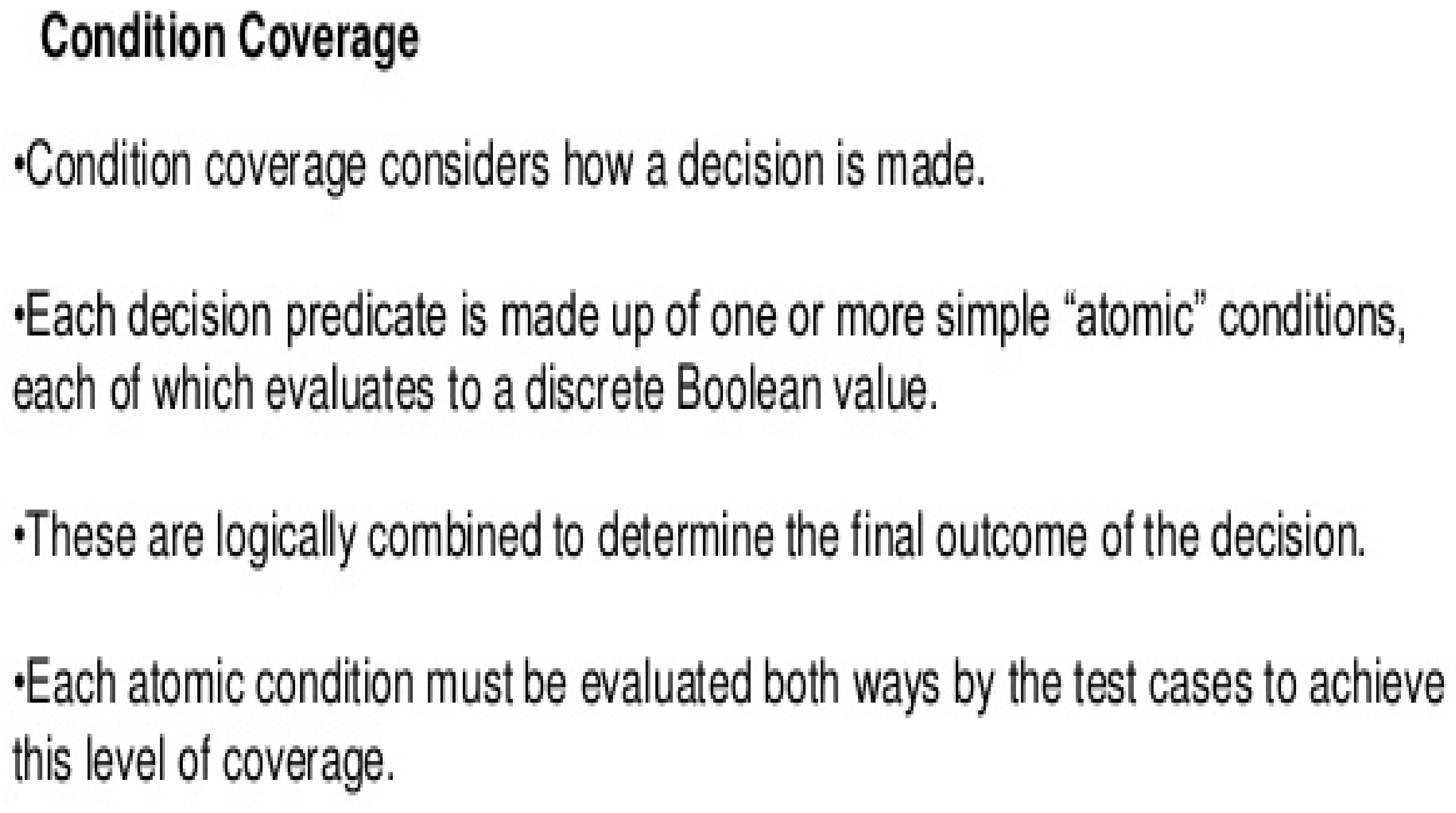
- We have to consider the individual component conditions - the sub conditions to be given true and false values.

- The third type of code coverage technique is Condition Coverage

- when there is only one condition in expression then branch coverage and condition coverage are same

Condition Coverage

- So far, a decision block has two outcomes: true and false.
- What we "kindly ignored" was that the decision block can be made of multiple conditions.
- Meaning, there are different ways for that block to be evaluated to true and to false.


Condition Coverage

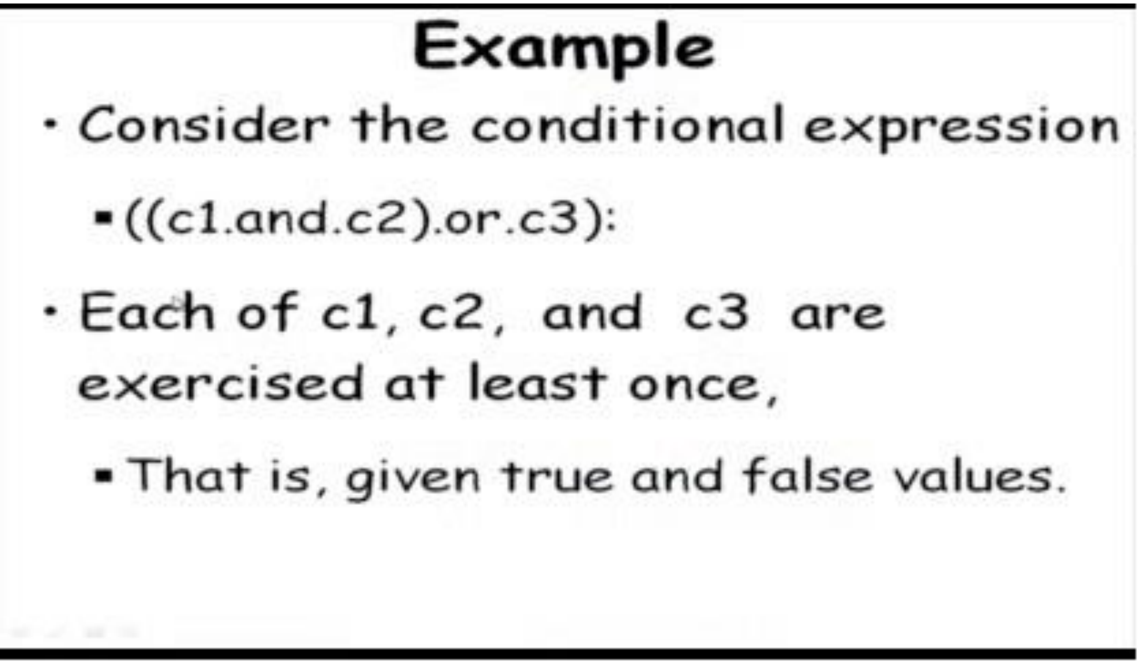
•Condition coverage considers how a decision is made.

•Each decision predicate is made up of one or more simple "atomic" conditions, each of which evaluates to a discrete Boolean value.

•These are logically combined to determine the final outcome of the decision.

•Each atomic condition must be evaluated both ways by the test cases to achieve this level of coverage.

Condition Coverage


Example

- Consider the conditional expression
  - ((c1.and.c2).or.c3):
- Each of c1, c2, and c3 are exercised at least once,
  - That is, given true and false values.

- We have to consider all possible combinations of conditions of c1, c2, c3
- If there are n component conditions, each one taking two values true and false, we need 2^n possible test cases

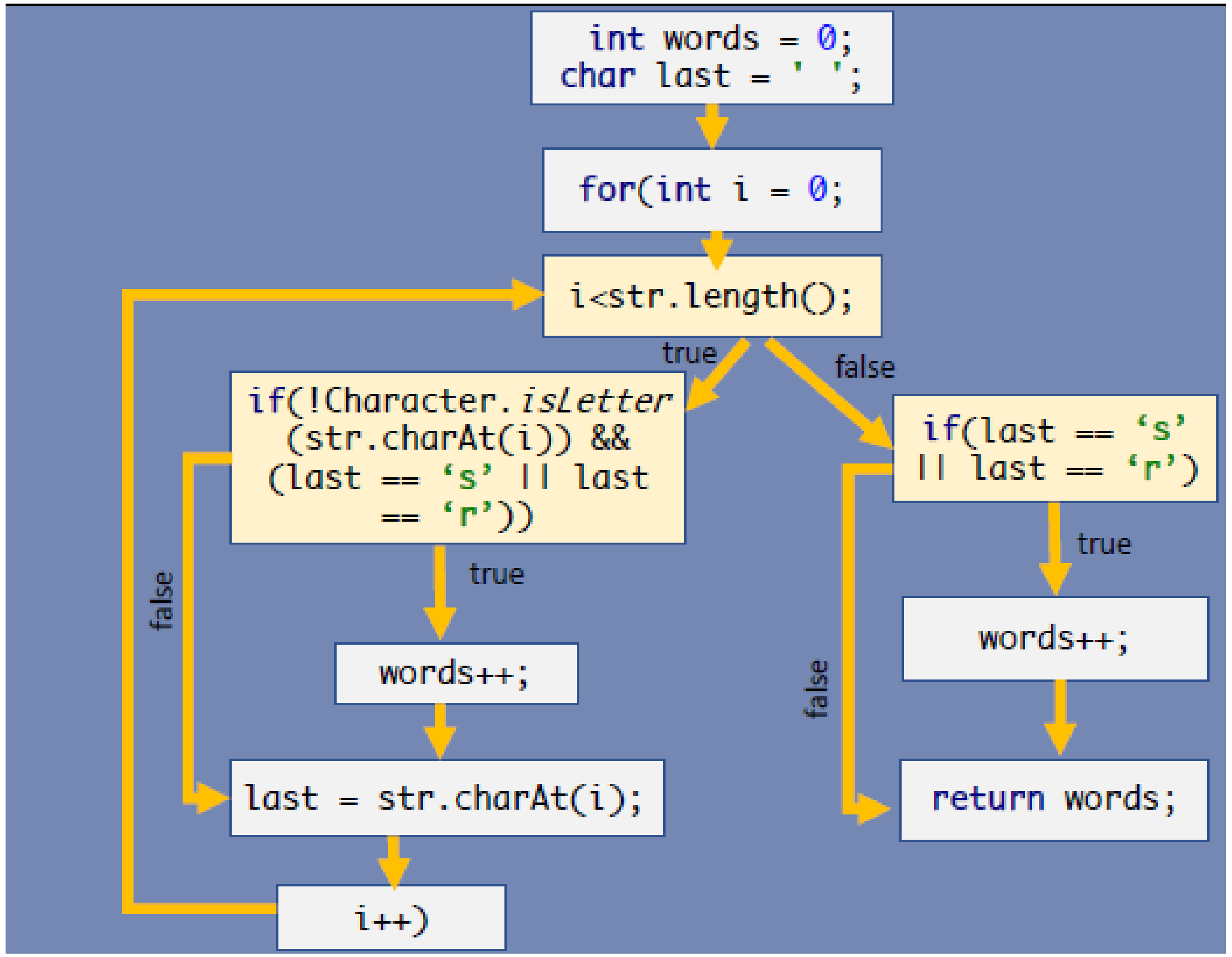
c1	c2	c3
T	T	T
T	F	T
T	T	F
T	F	F
F	T	T
F	T	F
F	F	T
F	F	F

Coverage Measure for condition coverage:

$$\frac{\text{Number of conditions that are both true and false}}{\text{Total Number of Conditions}} X100\%$$

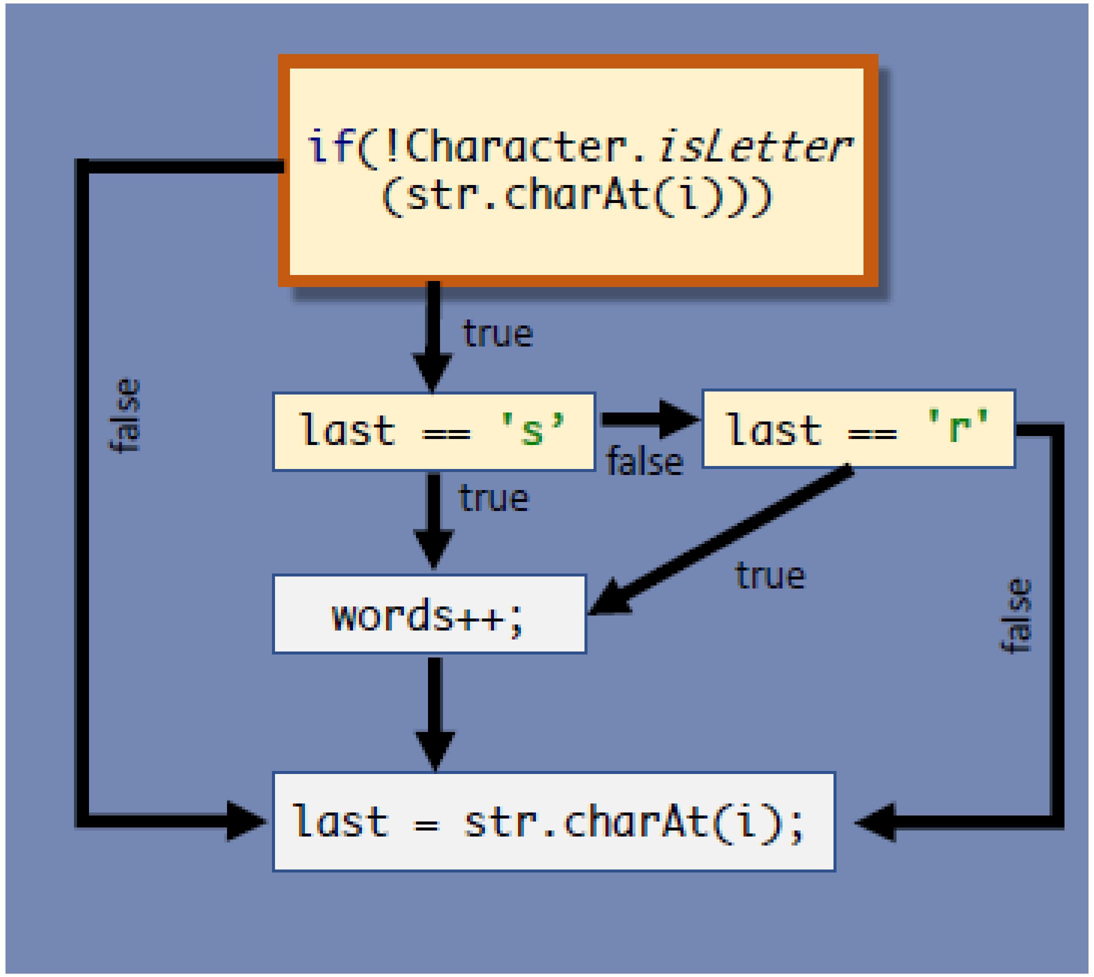
Condition Coverage

- Let's take as example the "big if" block that we have.



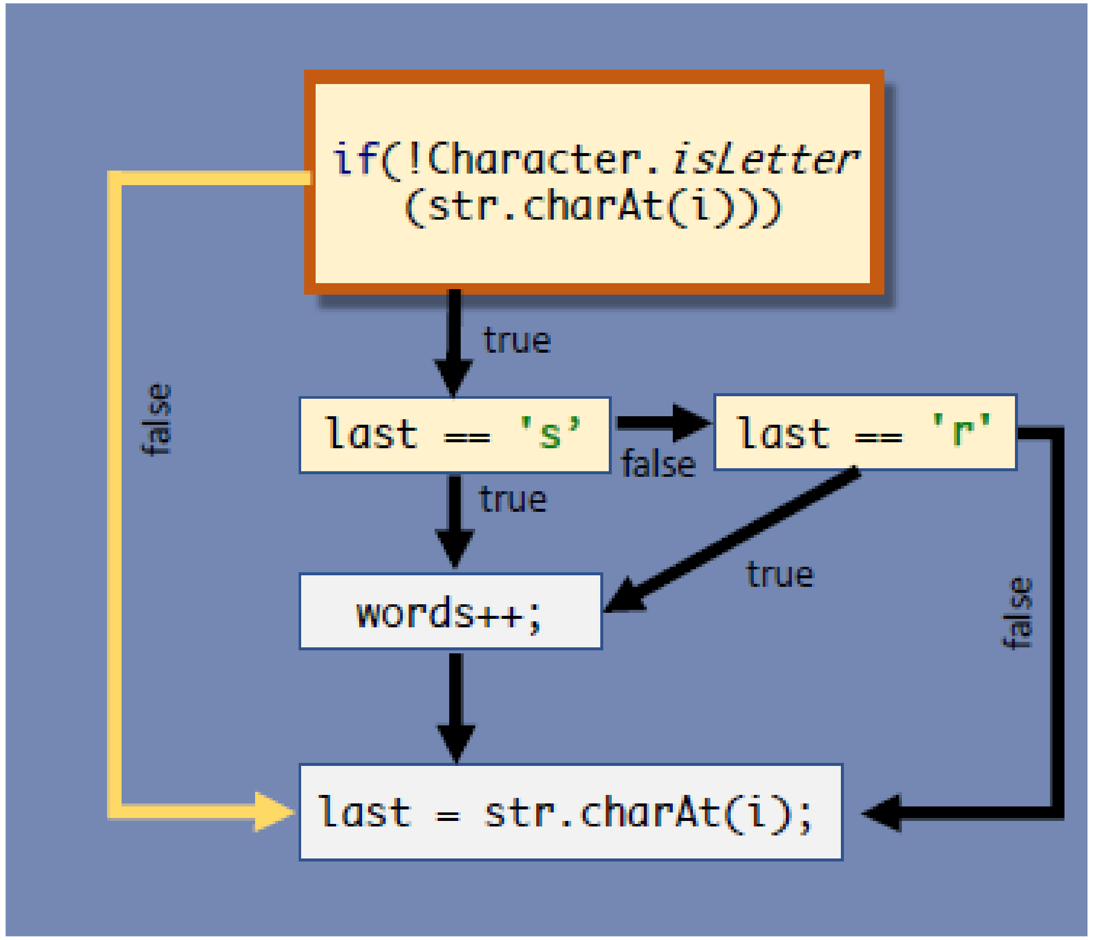
Condition Coverage

- If we zoom in closely, we see that the block can divided into multiple conditions:
- The first one, the Character.isLetter() check
- The second one, the letter inside of the last variable
- In there, we have one condition for the 's' and one condition for the 'r'‘
- If the first condition, the Character.isLetter(), is evaluated to false, the program jumps directly to the "last = str.charAt()" statement



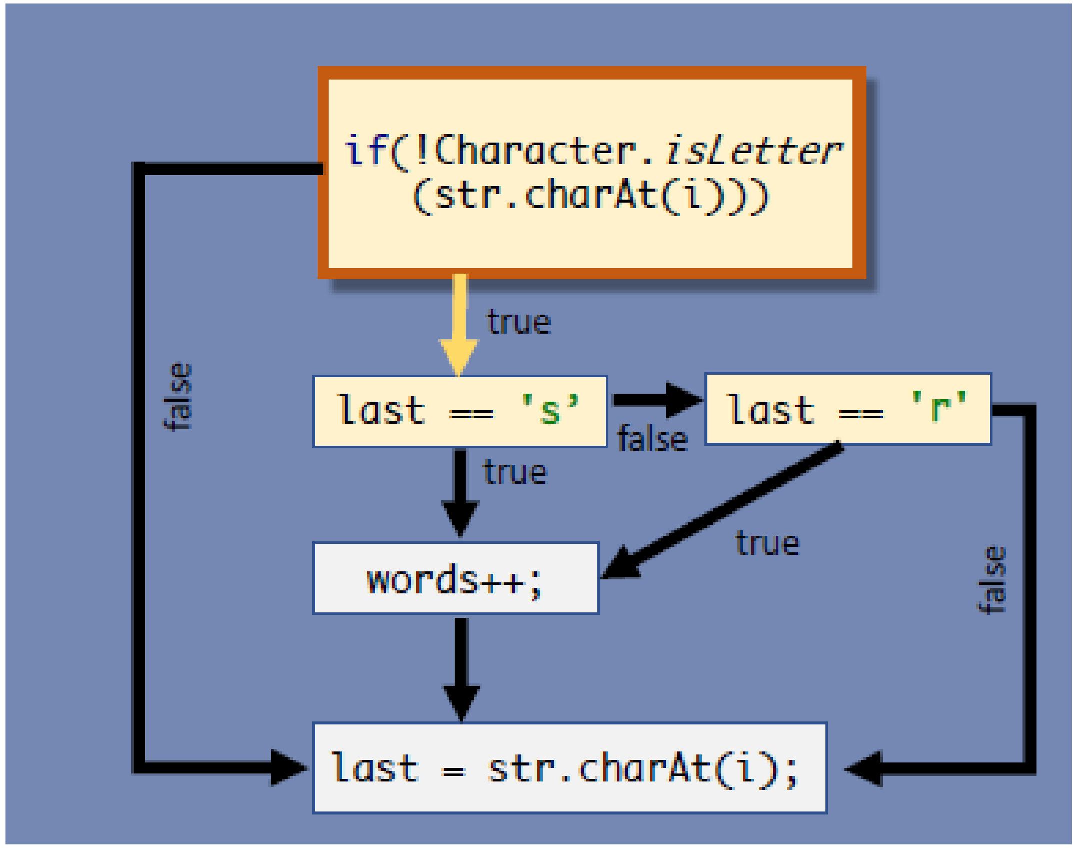
Condition Coverage

- If we zoom in closely, we see that the block can divided into multiple conditions:
- The first one, the Character.isLetter() check
- The second one, the letter inside of the last variable
- In there, we have one condition for the 's' and one condition for the 'r‘
- If the first condition, the Character.isLetter(), is evaluated to false, the program jumps directly to the "last = str.charAt()" statement

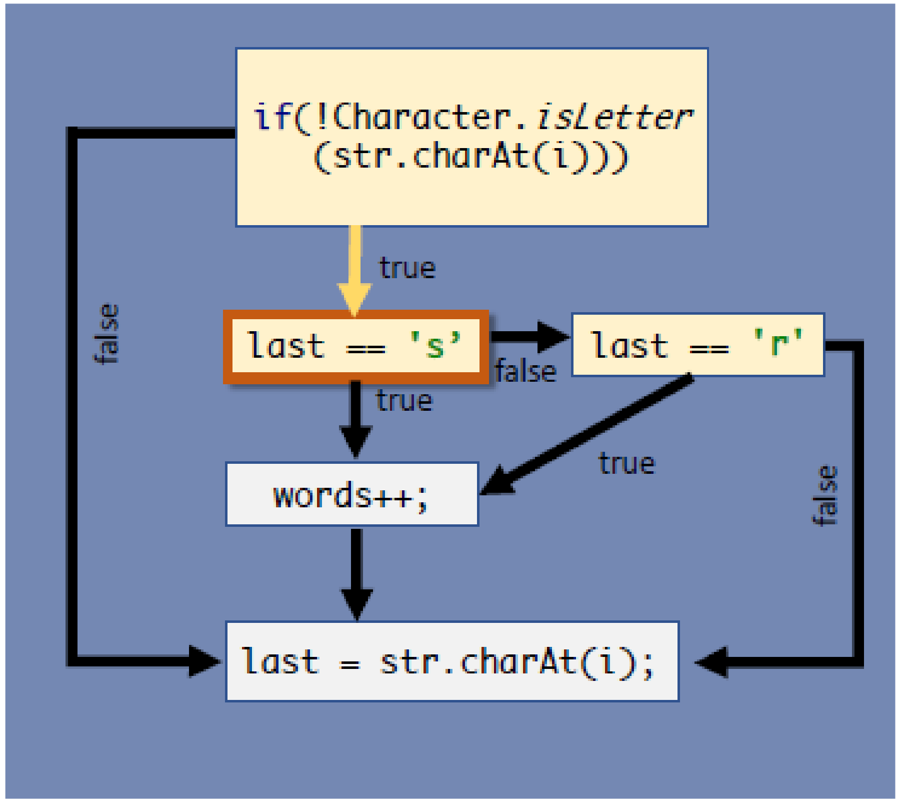


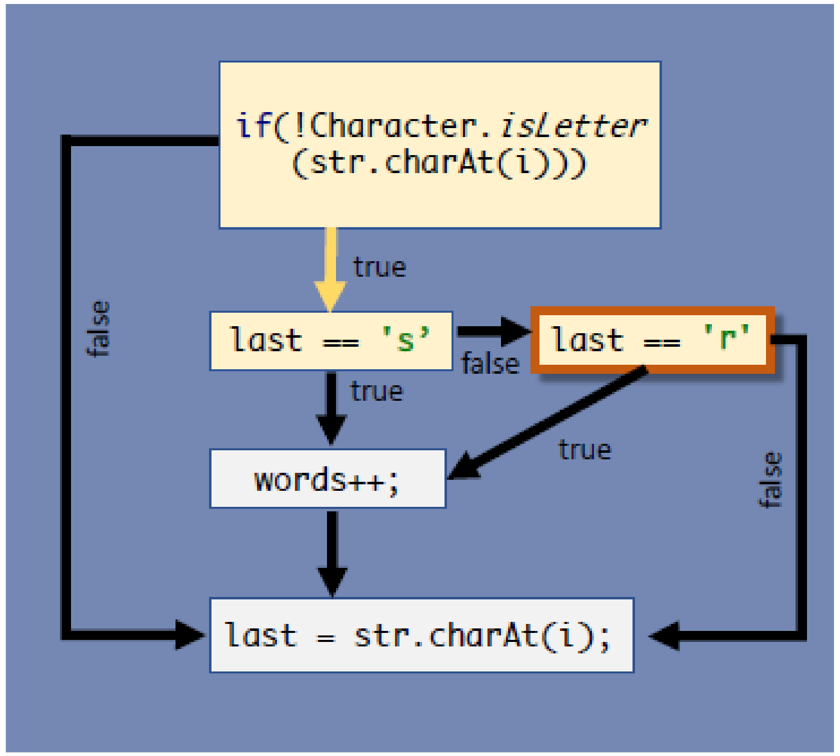
Condition Coverage

- ➢ However, if it gets evaluated to true, then the second condition should now be evaluated.



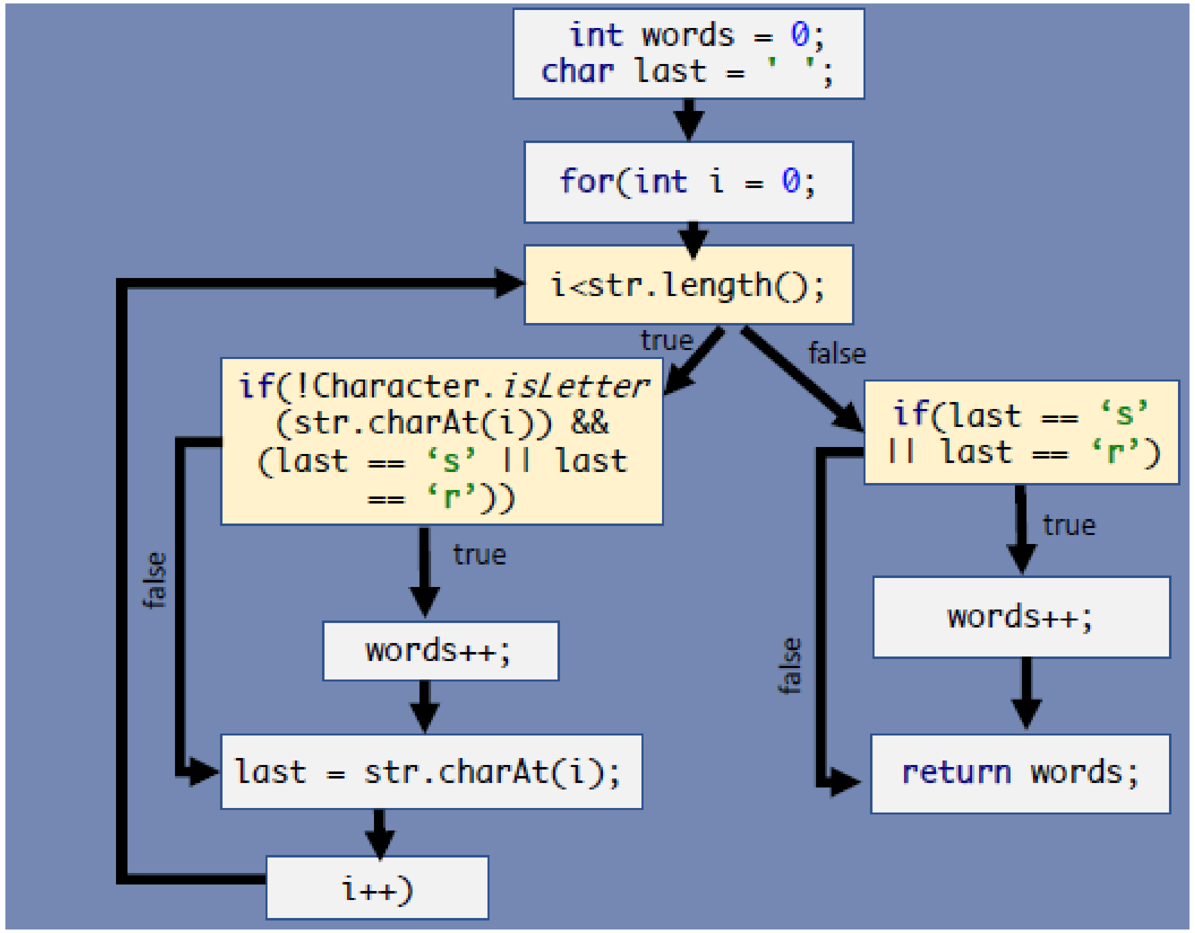
Condition Coverage

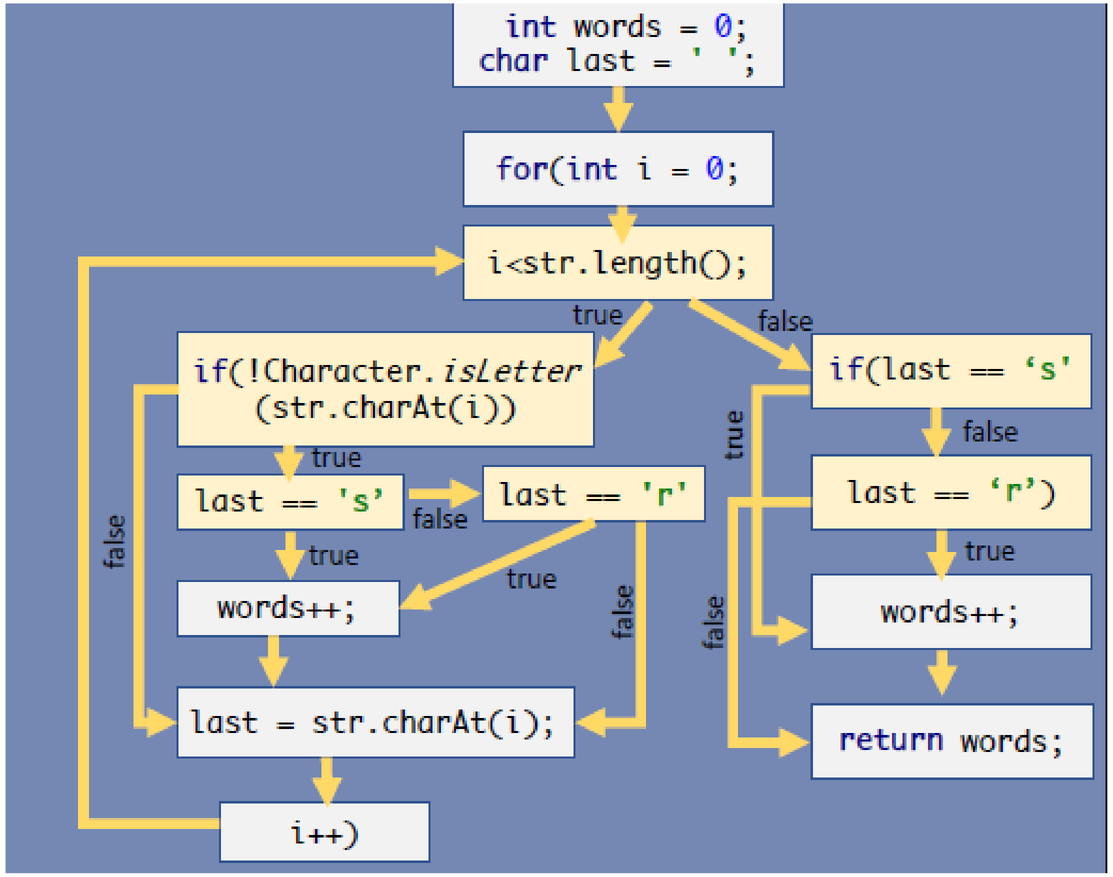




- Now, there are two ways for this second one to be true: either the last variable is equals to 's' or it is equals to 'r'.
- And, if one of them, any one of them, is true, the program does one thing; if they are both false, the program does a different thing.

Condition Coverage





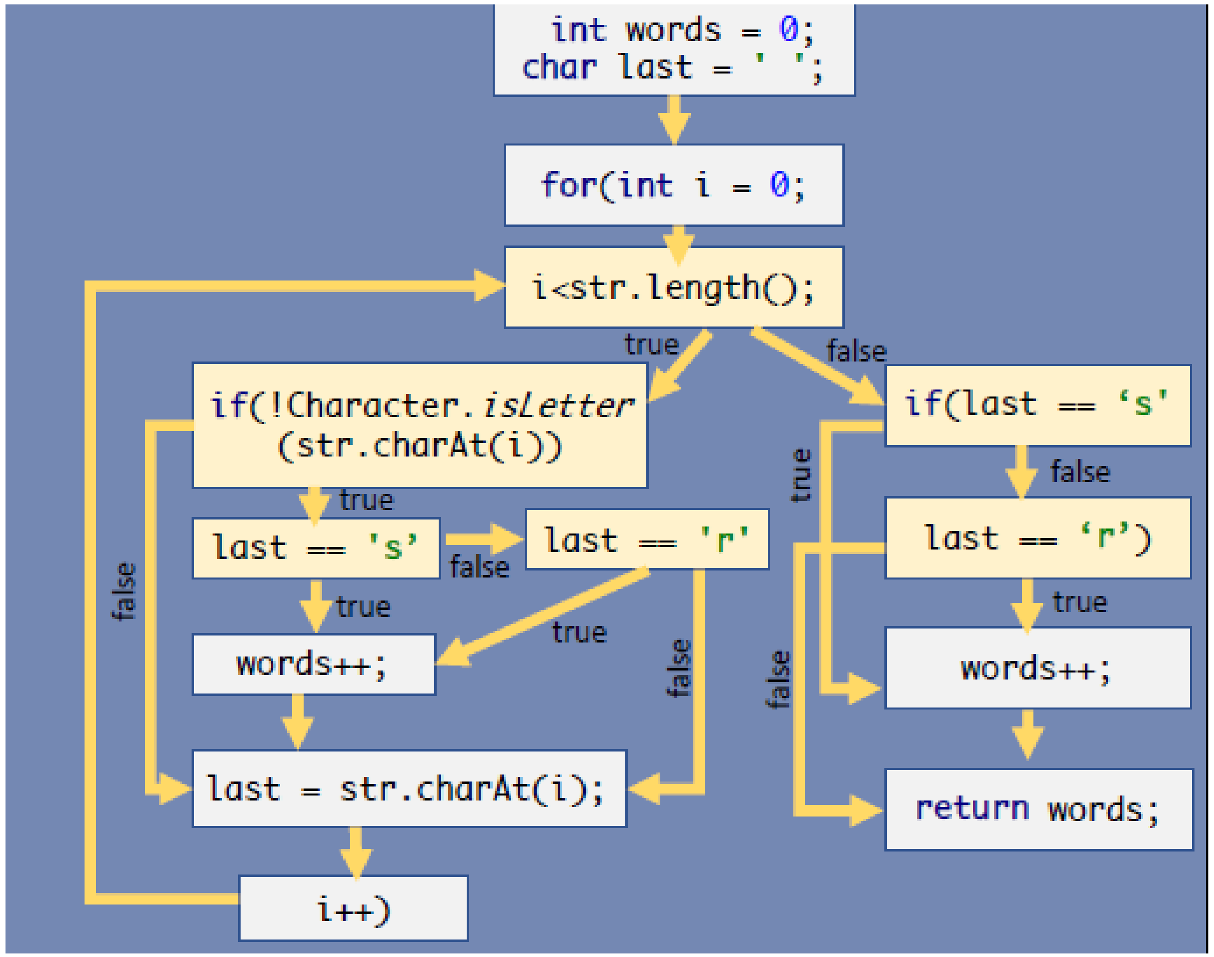
- ➢ If we break that big block and separate the conditions, we have more paths.
- ➢ And, of course, we should exercise and test all of them!

Condition Coverage

- Now, our goal here is the same as before: cover all the edges.

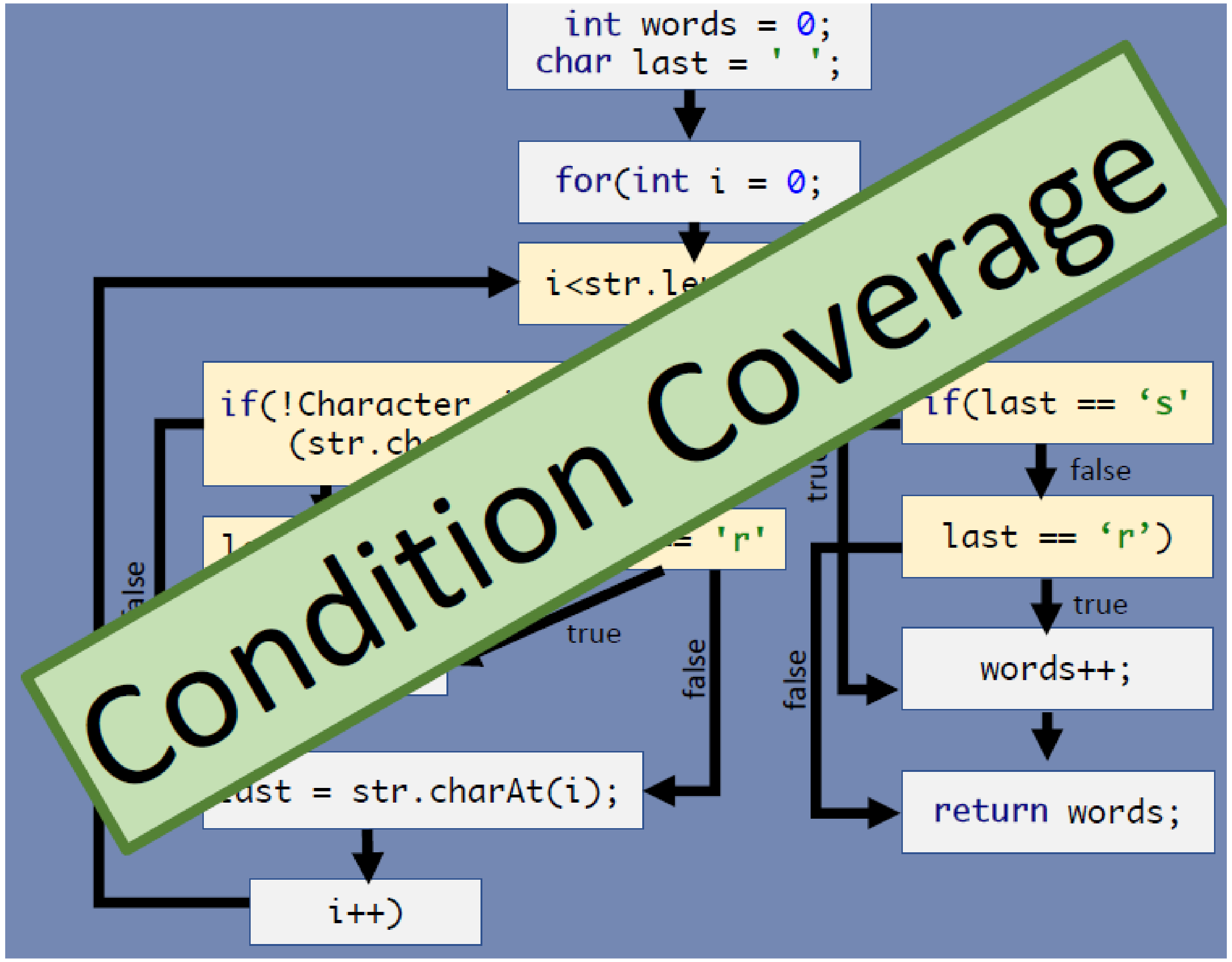
- Make sure that we have at least one test for each outcome (meaning, test the true and test the false) is what we call "condition coverage".

- condition coverage seems to cover more cases than branch coverage...



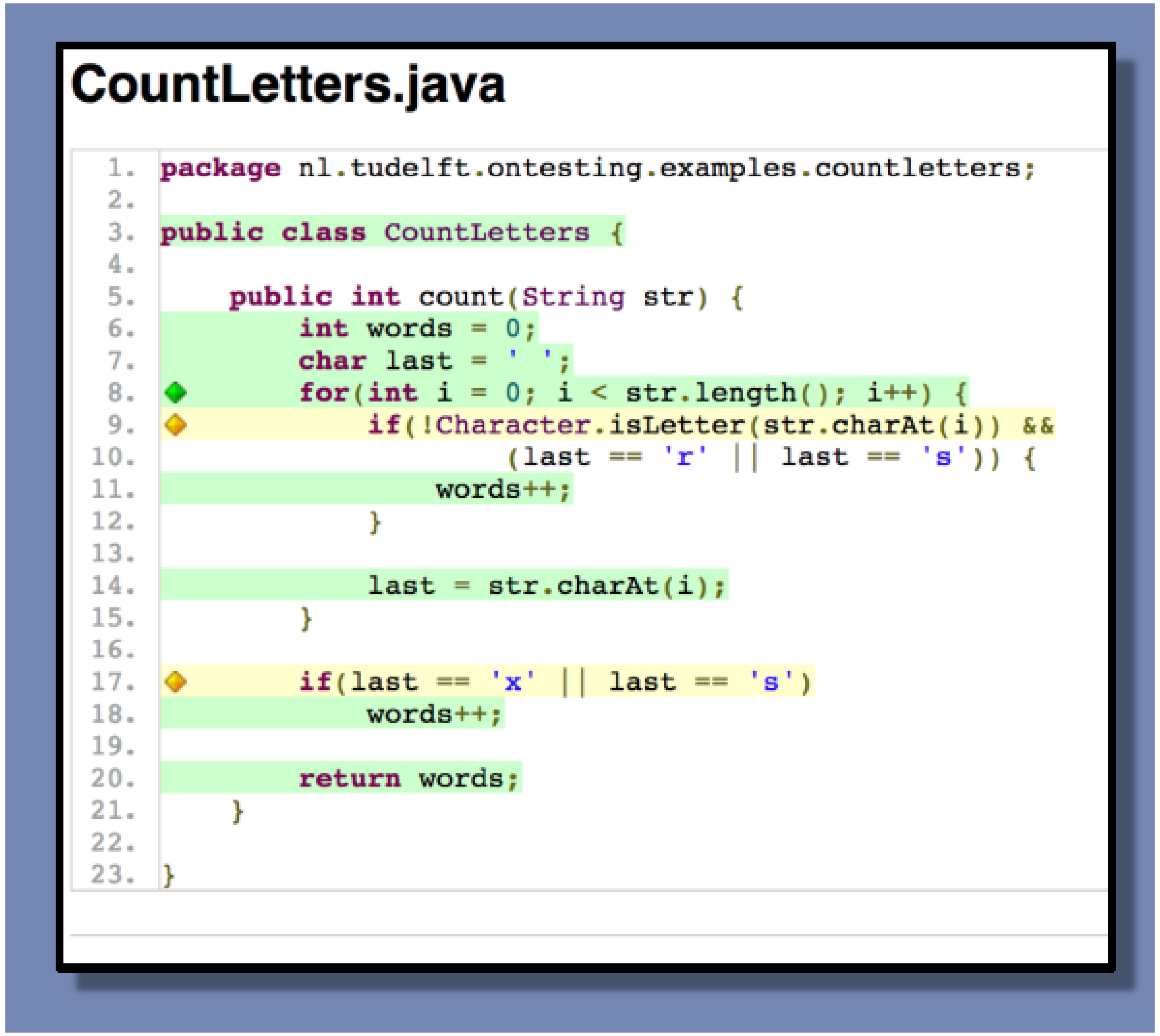
Condition Coverage

- Now, our goal here is the same as before: cover all the edges.
- Make sure that we have at least one test for each outcome (meaning, test the true and test the false) is what we call "condition coverage".



Condition Coverage

- In Eclipse, Coverage tool (backed by Jacoco tool), calls it "branch coverage”
- If you actually hover your mouse over the yellow diamonds, the first one for example, it says the number of total branches that condition has, and the number of conditions we actually missed to exercise.
- But if you think about it, if this was branch coverage, the total number of branches should be two: one true, one false...
- That's condition coverage,


CountLetters.java

```
package nl.tudelft.ontesting.examples.countletters;

public class CountLetters {

    public int count(String str) {
        int words = 0;
        char last = ' ';
        for(int i = 0; i < str.length(); i++) {
            if(!Character.isLetter(str.charAt(i)) &&
                    (last == 'r' || last == 's')) {
                words++;
            }

            last = str.charAt(i);
        }

        if(last == 'x' || last == 's')
            words++;

        return words;
    }

}
```

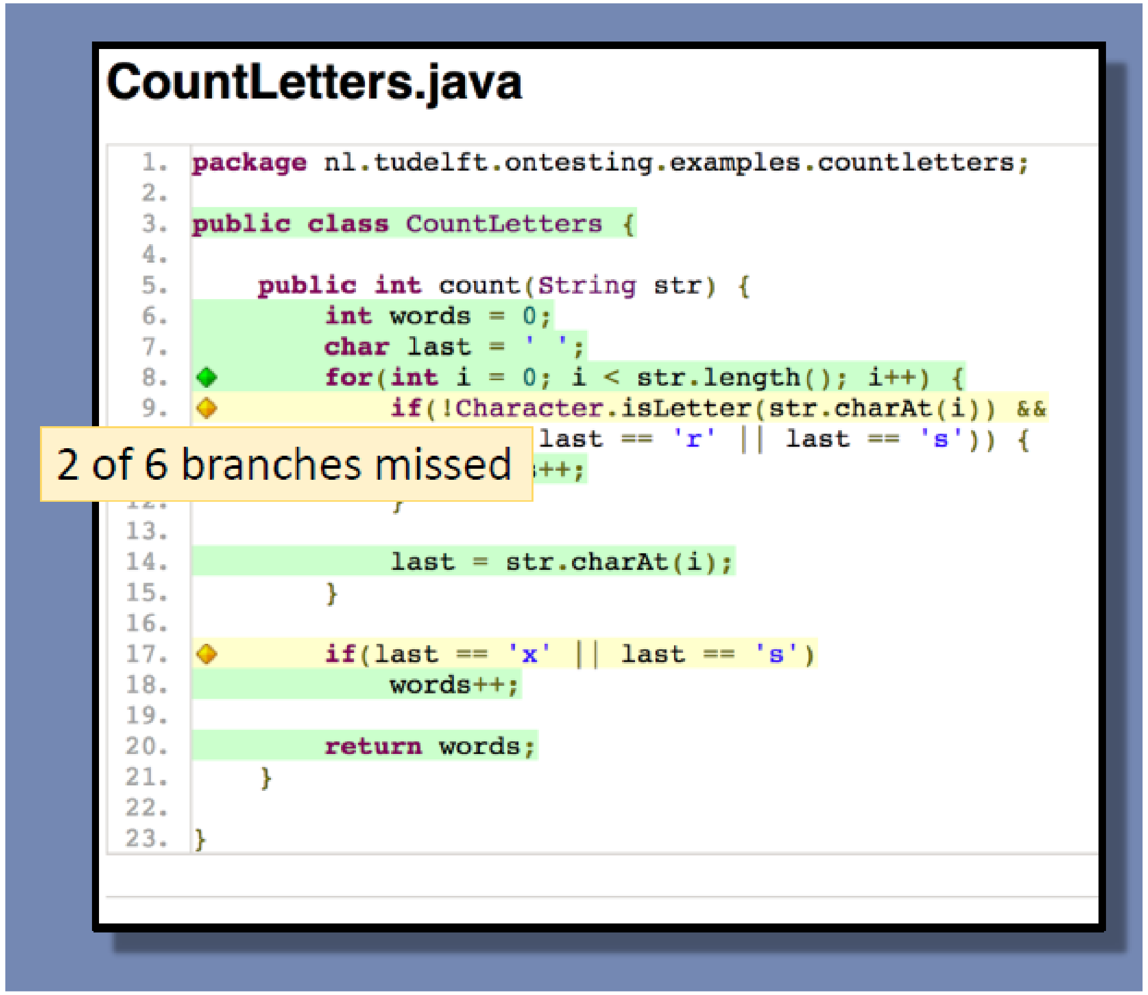
Condition Coverage

- In Eclipse, Coverage tool (backed by Jacoco tool), calls it "branch coverage"

- If you actually hover your mouse over the yellow diamonds, the first one for example, it says the number of total branches that condition has, and the number of conditions we actually missed to exercise.

- But if you think about it, if this was branch coverage, the total number of branches should be two: one true, one false...

- That's condition coverage,



CountLetters.java

```
package nl.tudelft.ontesting.examples.countletters;

public class CountLetters {

    public int count(String str) {
        int words = 0;
        char last = ' ';
        for(int i = 0; i < str.length(); i++) {
            if(!Character.isLetter(str.charAt(i)) &&
                                  last == 'r' || last == 's')) {
                                 ++;
            }

            last = str.charAt(i);
        }

        if(last == 'x' || last == 's')
            words++;

        return words;
    }

}
```

2 of 6 branches missed

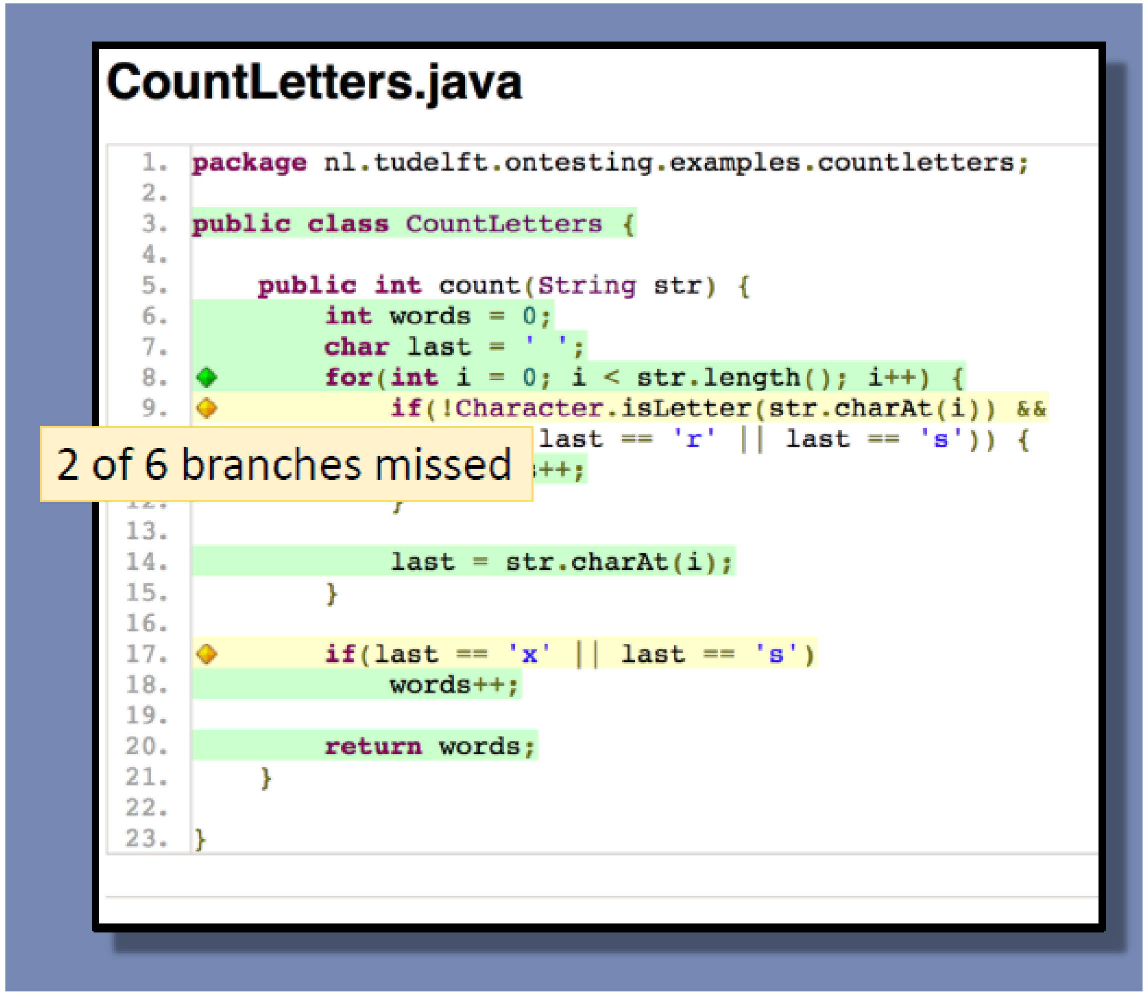
Condition Coverage

- In Eclipse, Coverage tool (backed by Jacoco tool), calls it "branch coverage”

- If you actually hover your mouse over the yellow diamonds, the first one for example, it says the number of total branches that condition has, and the number of conditions we actually missed to exercise.

- But if you think about it, if this was branch coverage, the total number of branches should be two: one true, one false...

- That's condition coverage,

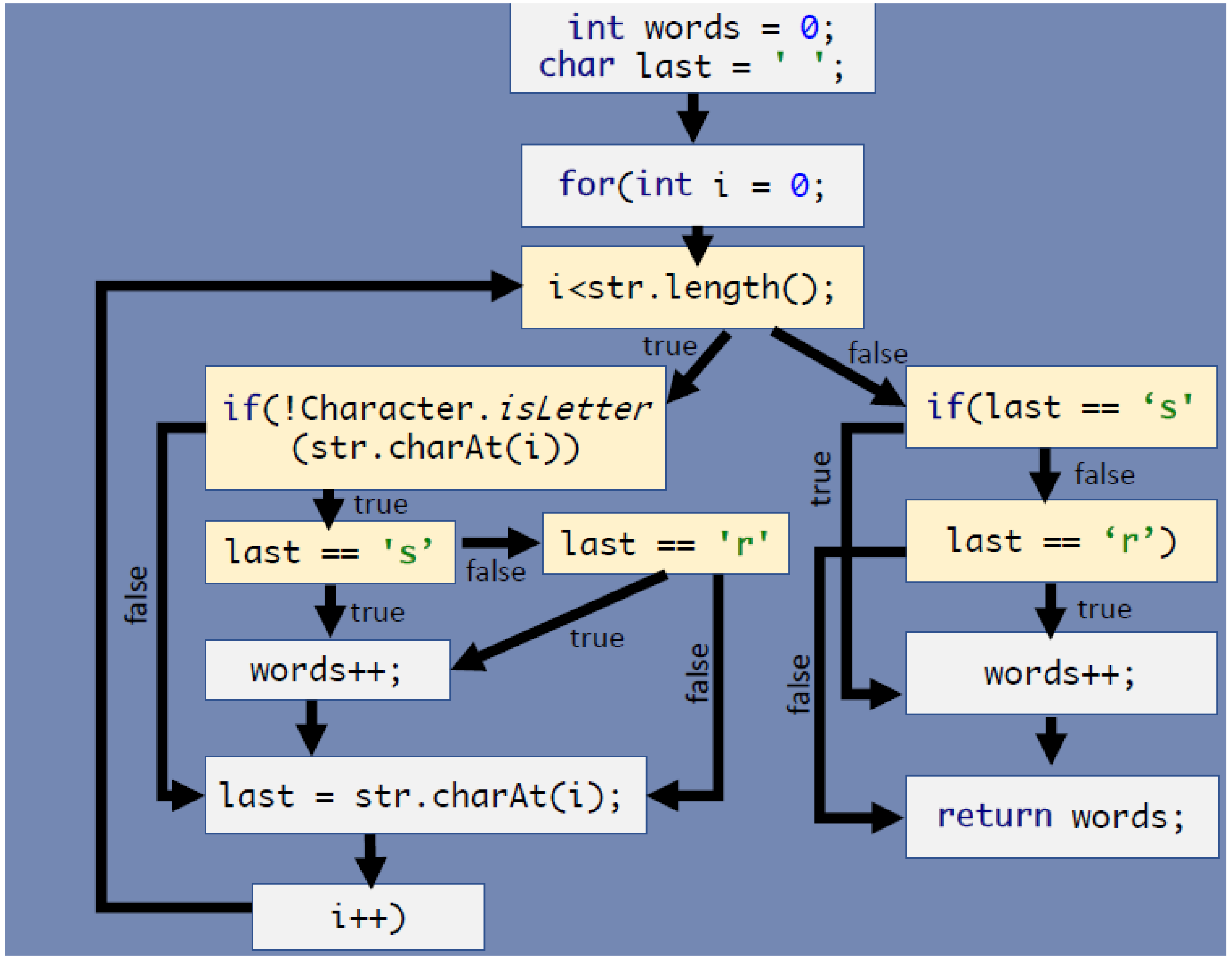

CountLetters.java

```
package nl.tudelft.ontesting.examples.countletters;

public class CountLetters {

    public int count(String str) {
        int words = 0;
        char last = ' ';
        for(int i = 0; i < str.length(); i++) {
            if(!Character.isLetter(str.charAt(i)) &&
                          last == 'r' || last == 's')) {
                          ++;
                }

                last = str.charAt(i);
        }

        if(last == 'x' || last == 's')
            words++;

        return words;
    }

}
```

2 of 6 branches missed

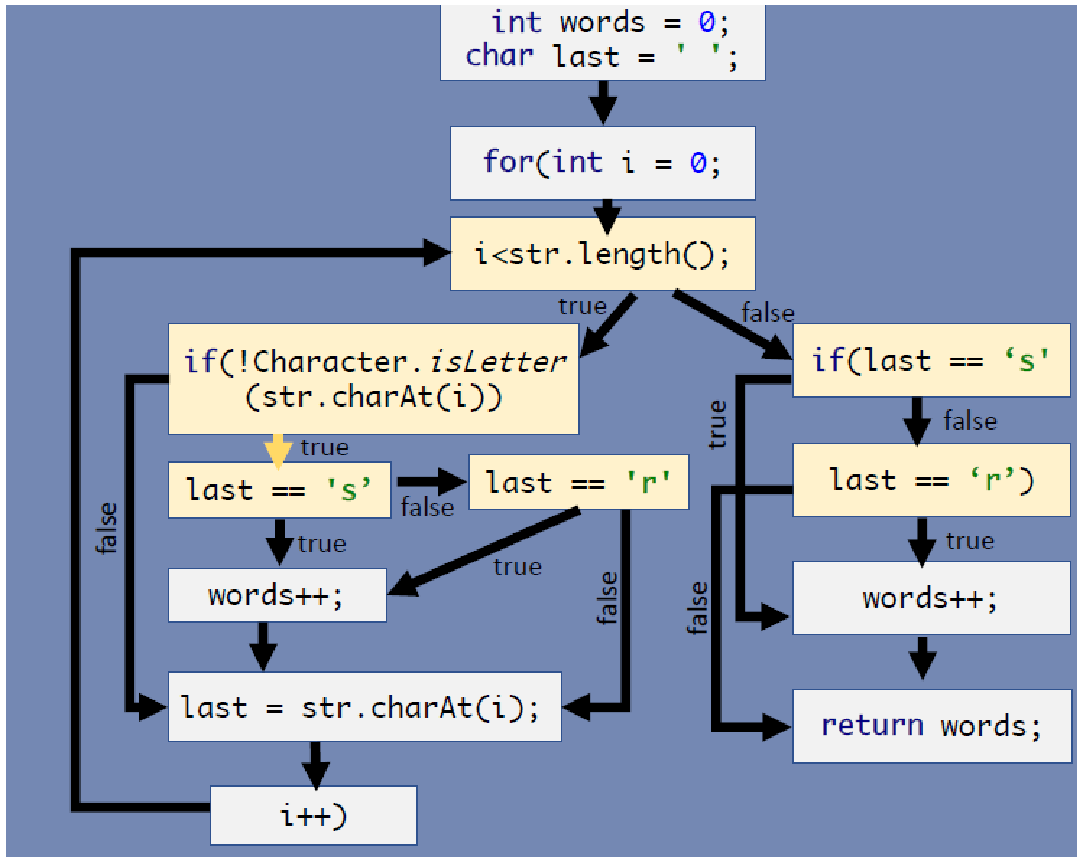
Limitation of Condition Coverage

- Right now, we look to each condition, and we make sure that we have at least one test for the "true" case and one for the "false" case.

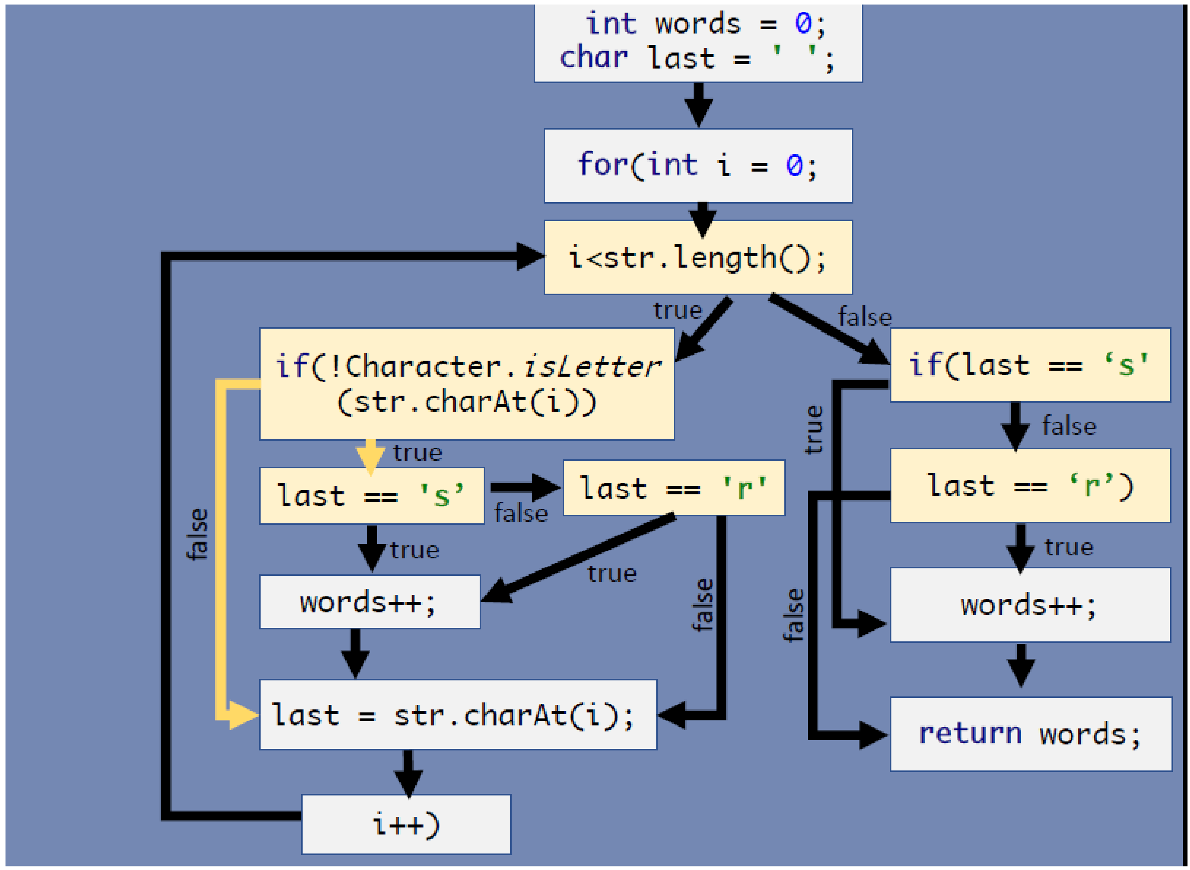


Limitation of Condition Coverage

- if we focus on the first if there, the one with the Character isLetter(), we need to have a test for the true

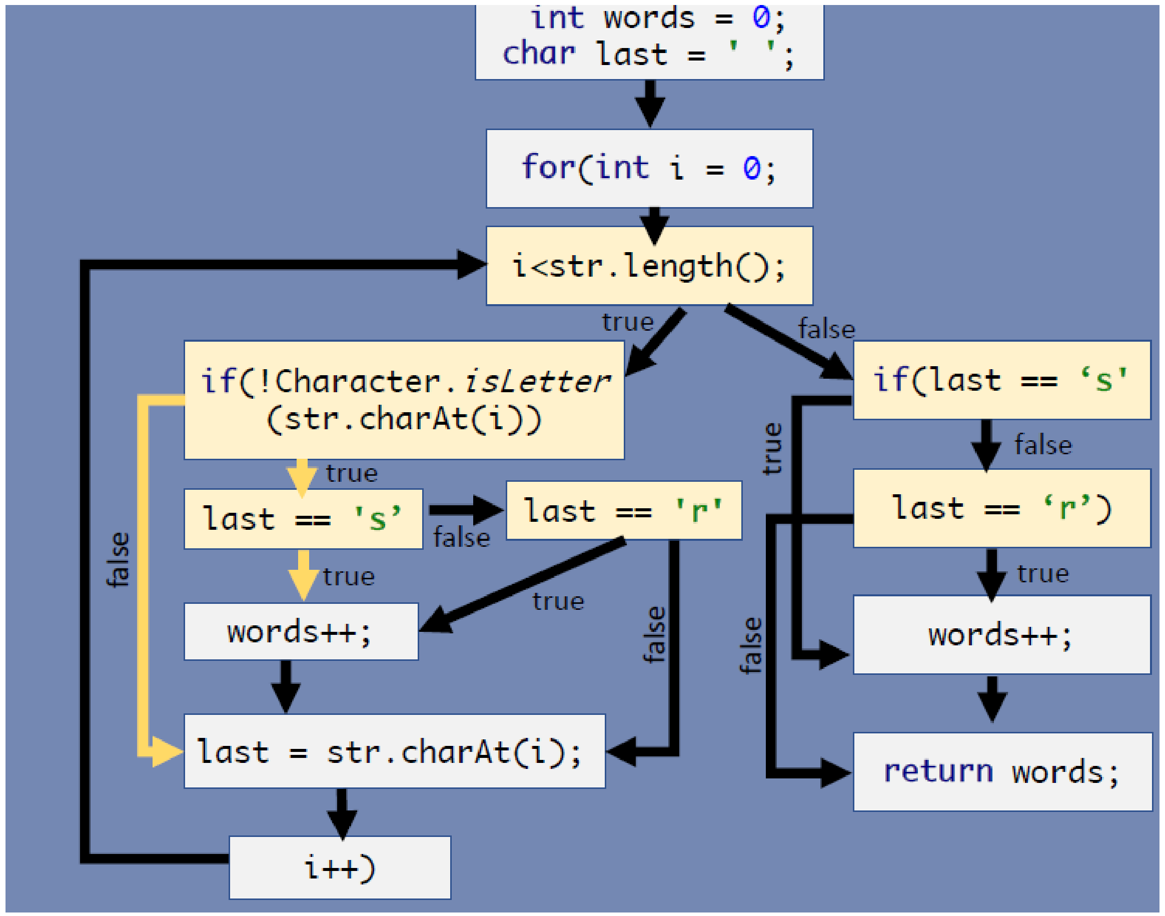


- And for the false

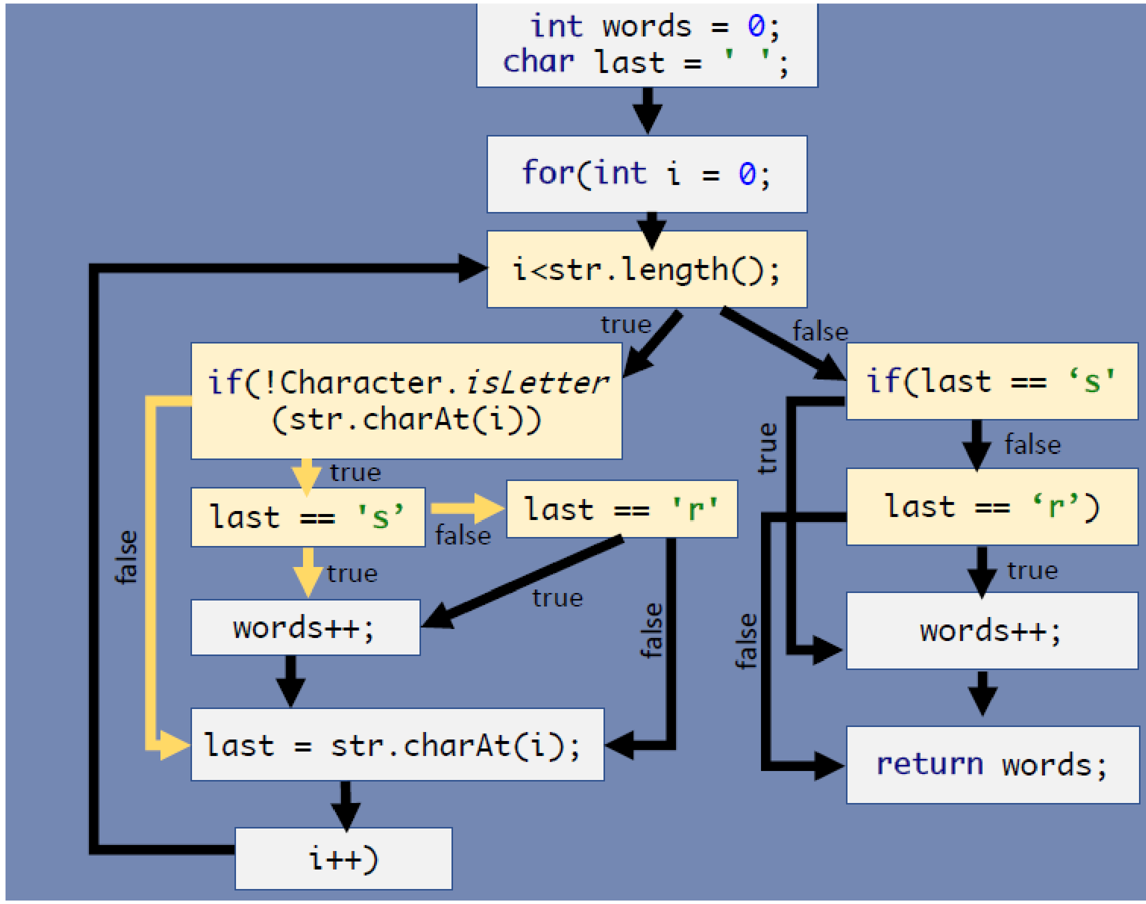


Limitation of Condition Coverage

- Same thing for the last equals to s.
- We cover the true,

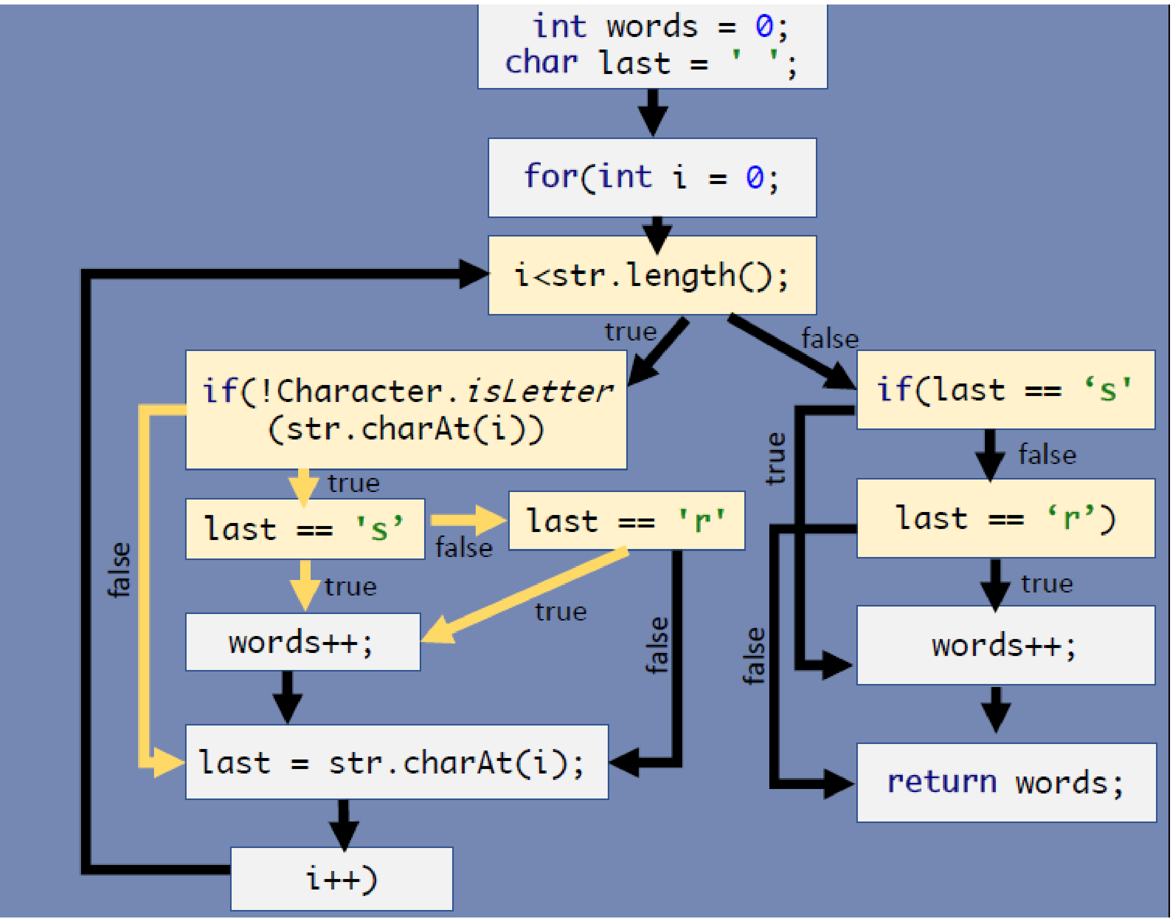


- we cover the false.

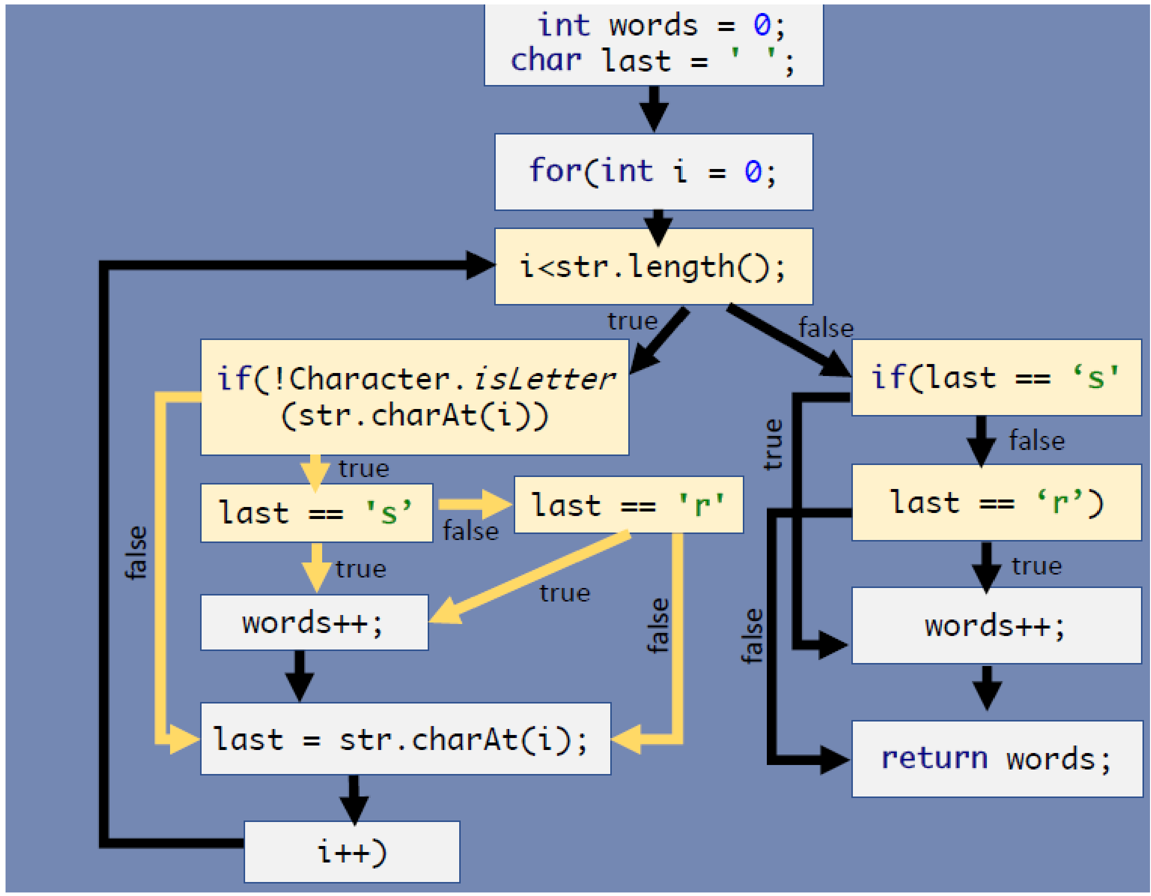


Limitation of Condition Coverage

- The last equals to r.
- We cover the true,



- we cover the false.

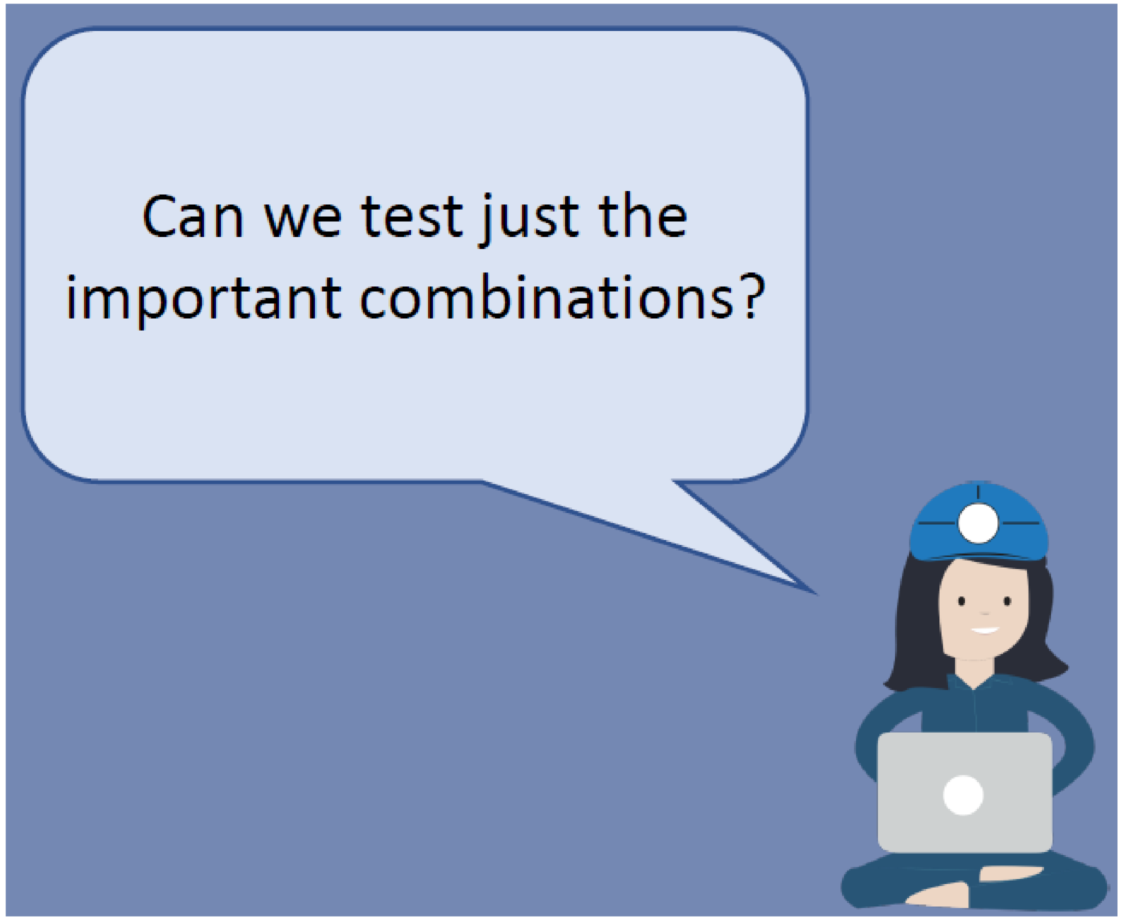


Limitation of Condition Coverage

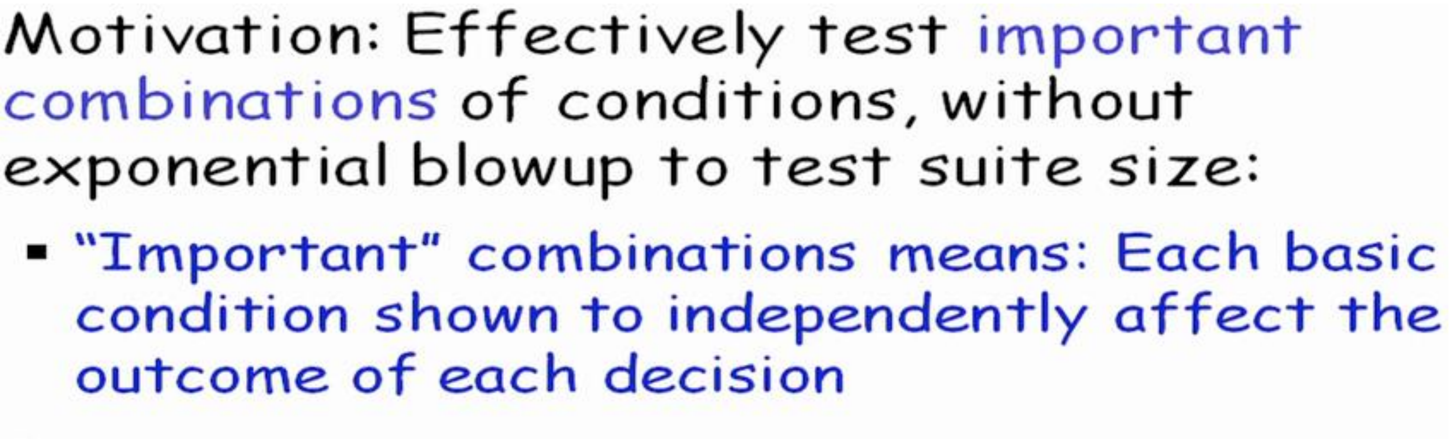
- The problem is: exercising all the conditions, the way we did so far with condition coverage, is the same thing as exercising all the paths!

Limitation of Condition Coverage

- When number of conditions in a program is large, then it requires too many test cases
- for example, embedded control applications. We might have a conditional expression involving 20 or 25 variables, and in that case, we need 220 or 225 test cases is just too many
- Can we achieve testing which is you must as effective as branch and condition testing and still not have an exponential number of test cases?

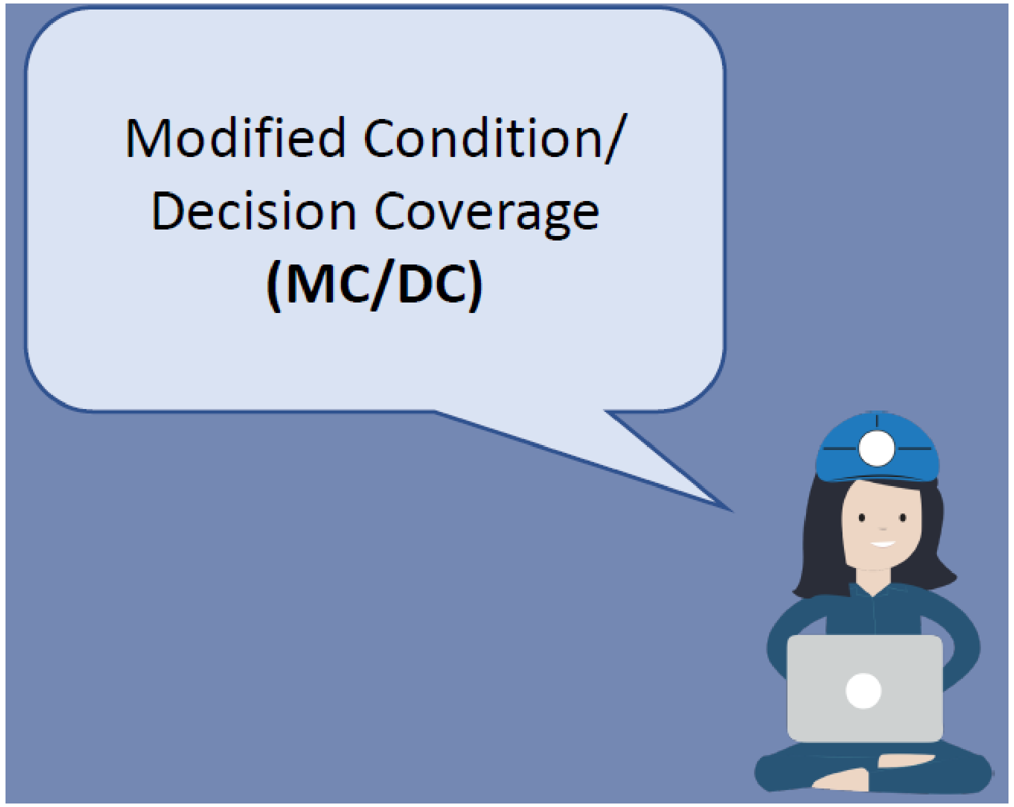


MC/DC Coverage



Motivation: Effectively test important combinations of conditions, without exponential blowup to test suite size:

- "Important" combinations means: Each basic condition shown to independently affect the outcome of each decision

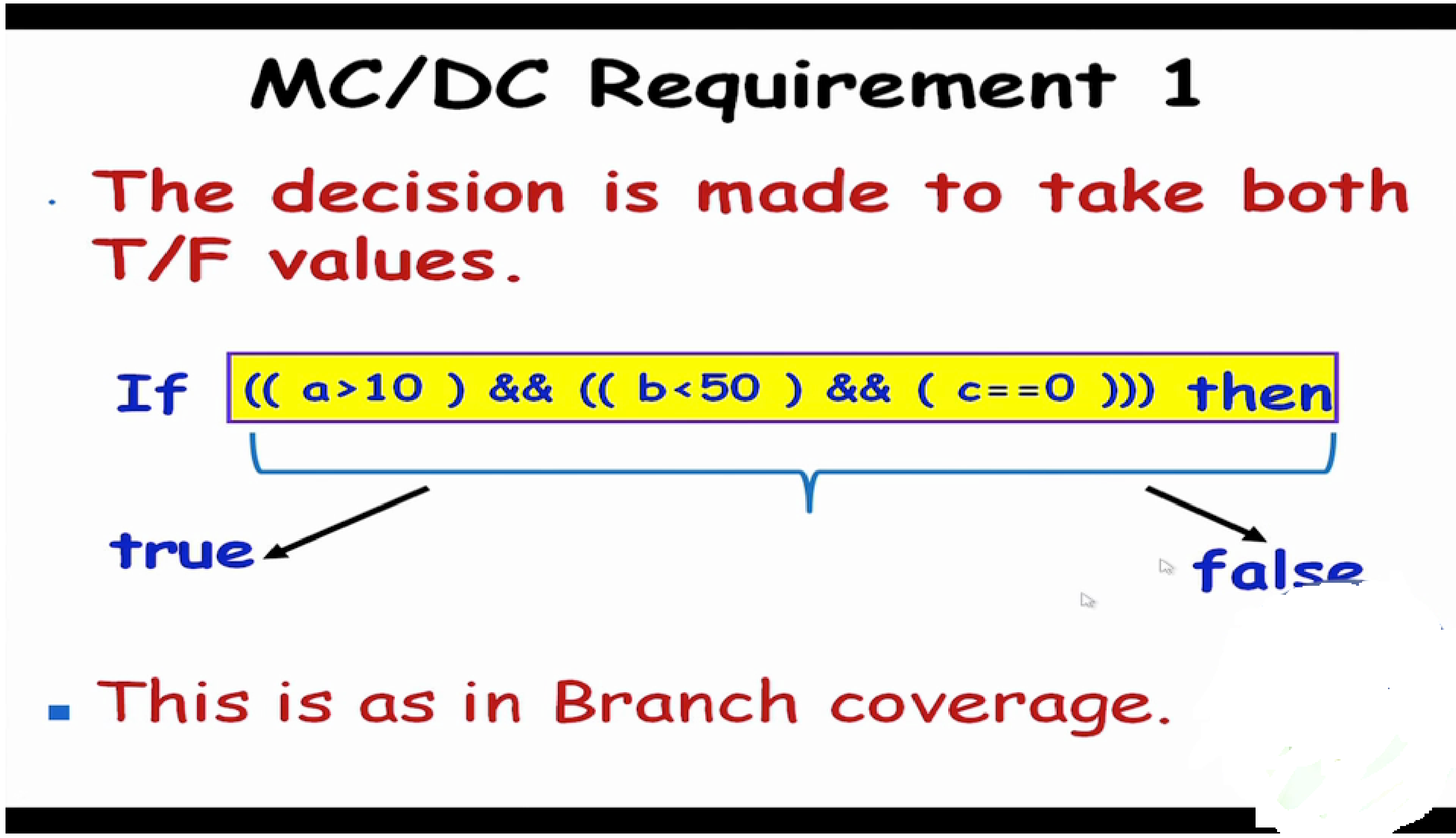


MC/DC Coverage

- MC/DC requires
  - that each decision or branch condition should execute for true and false value
  - that each basic condition in every decision should be executed for true and false value
  - each condition in a decision should affect the decision’s outcome independently

MC/DC Coverage

- MC/DC requires
  - that each decision or branch condition should execute for true and false value
  - that each basic condition in every decision should be executed for true and false value
  - each condition in a decision should affect the decision's outcome independently



MC/DC Requirement 1

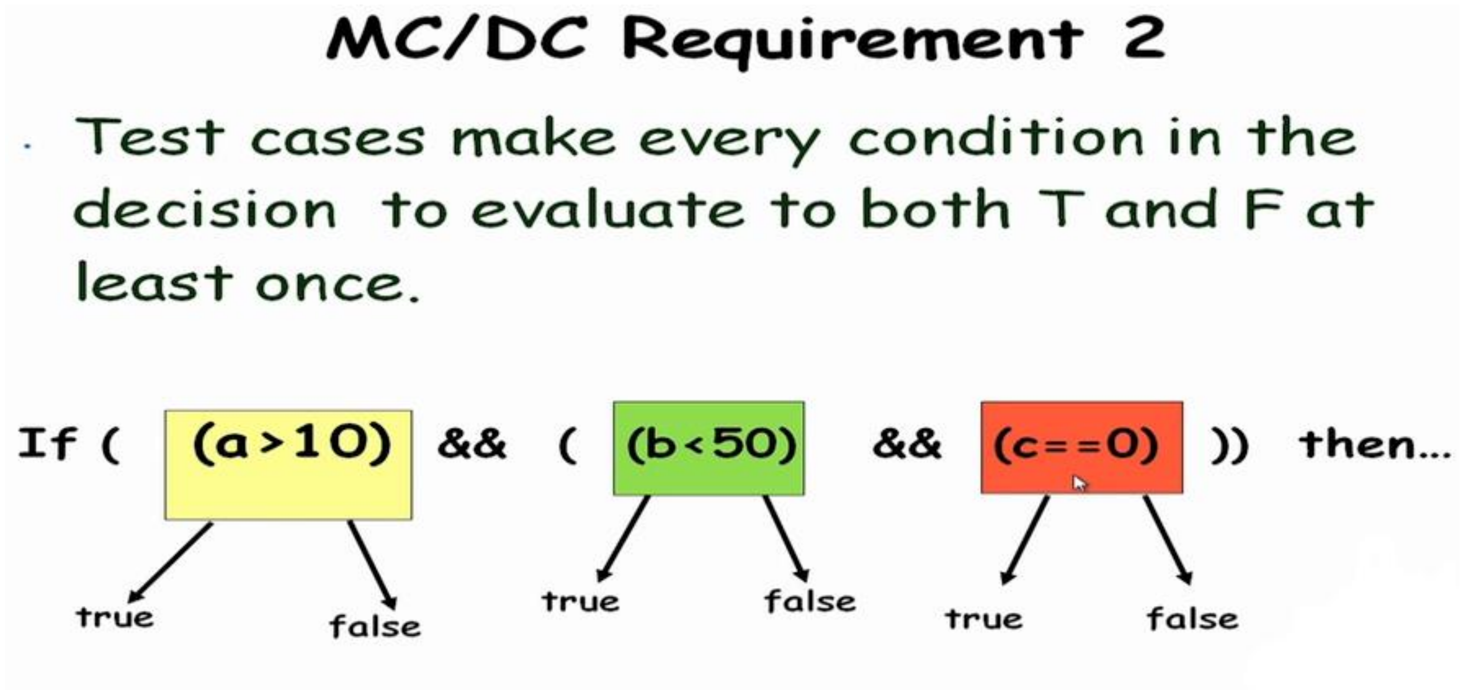
- The decision is made to take both T/F values.

If ((a>10) && ((b<50) && (c==0))) then

- This is as in Branch coverage.

MC/DC Coverage

- MC/DC requires
  - that each decision or branch condition should execute for true and false value
  - that each basic condition in every decision should be executed for true and false value
  - each condition in a decision should affect the decision's outcome independently

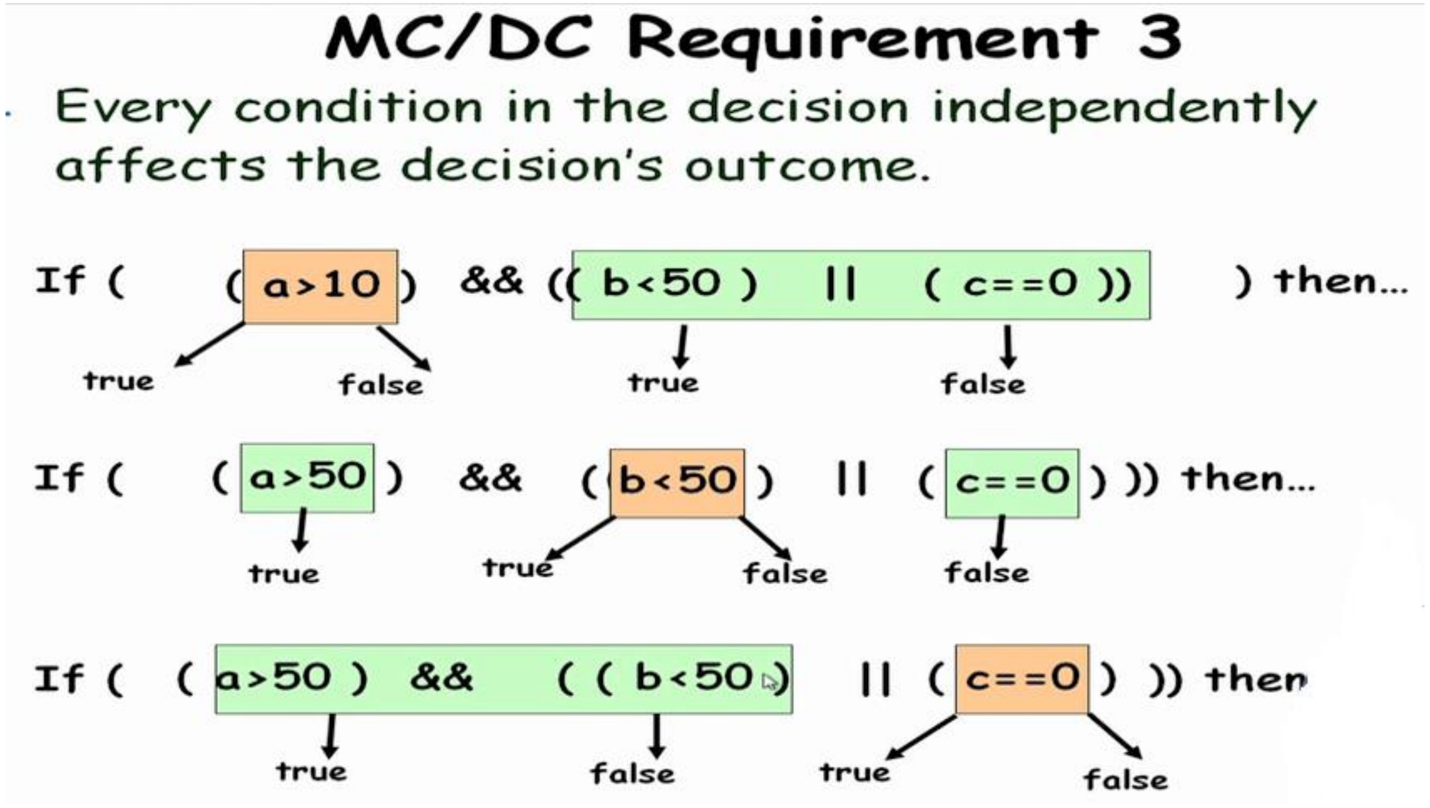


MC/DC Requirement 2

- Test cases make every condition in the decision to evaluate to both T and F at least once.

MC/DC Coverage

- MC/DC requires
  - that each decision or branch condition should execute for true and false value
  - that each basic condition in every decision should be executed for true and false value
  - each condition in a decision should affect the decision's outcome independently



MC/DC Requirement 3

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

If ((a>10) && ((b<50) || (c==0))) then...
true
false
true
false
If ((a>50) && (b<50) || (c==0))) then...
true
true
false
false
If ((a>50) && ((b<50) || (c==0))) then
true
false
true
false

MC/DC Coverage

Example : a predicate - a && b && c

Test Case	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	F
3	T	F	T	F
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

- At first consider test cases where value changes only in a (other components remain constant) make the overall outcome of the whole predicate different
  - ❑ Test Case 1 and 5
- Similarly, consider values of b
  - ❑ Test Case 1 and 3
- Similarly, consider values of c
  - ❑ Test Case 1 and 2

Test Case	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	F
3	T	F	T	F
5	F	T	T	F

MC/DC Coverage

Example : a predicate - (a||b)&&c

Test Case	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

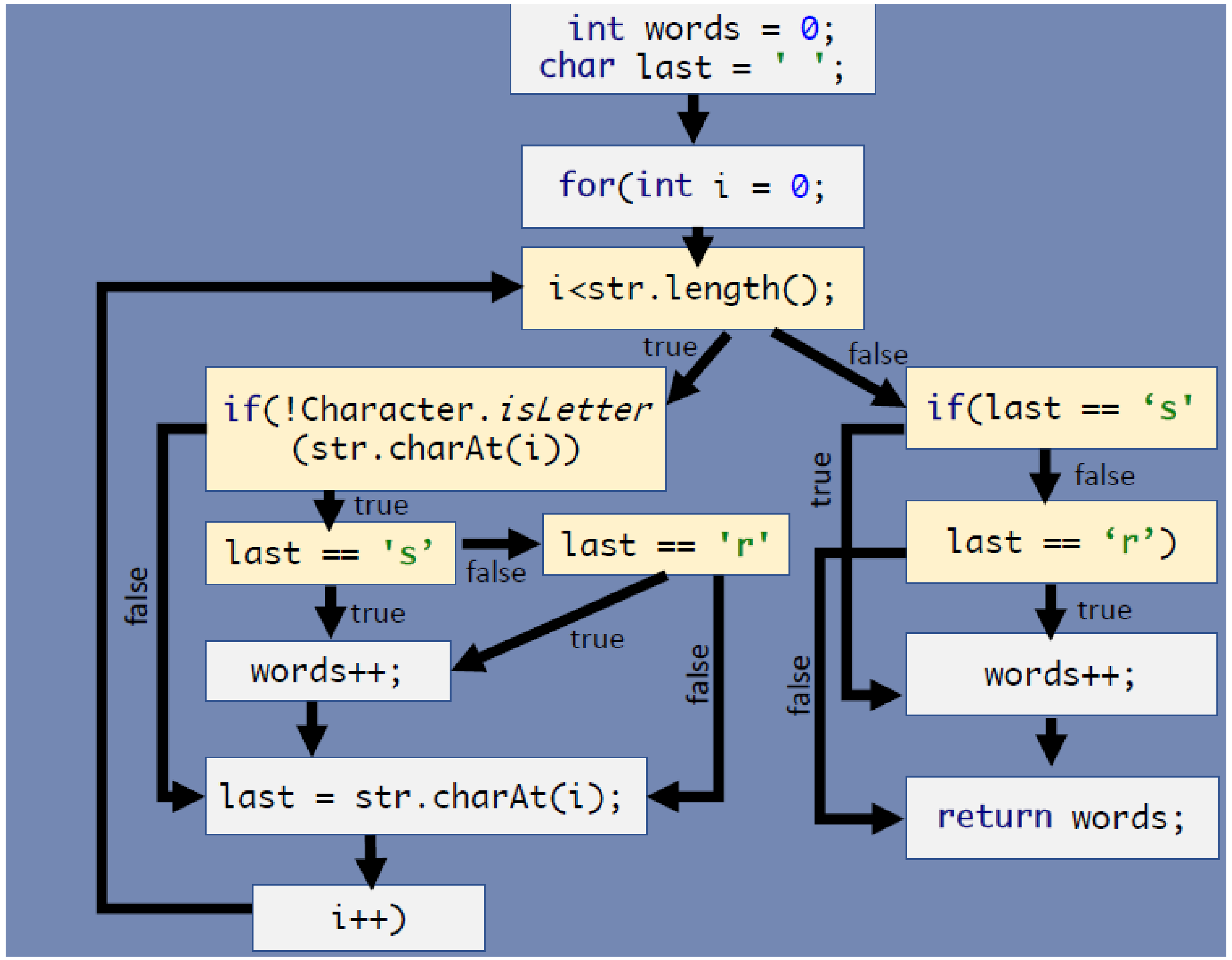
Test Case	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
5	F	T	T	T
7	F	F	T	F

Back to CountWords Problem

- In case of Condition Coverage, we look to each condition, and we make sure that we have at least one test for the "true" case and one for the "false" case.

- The number of combinations might be too large, making it just impossible to be applied in many cases.

- So, to overcome this, what we could do is to not test all the combinations, but only the important ones... (MC/DC)



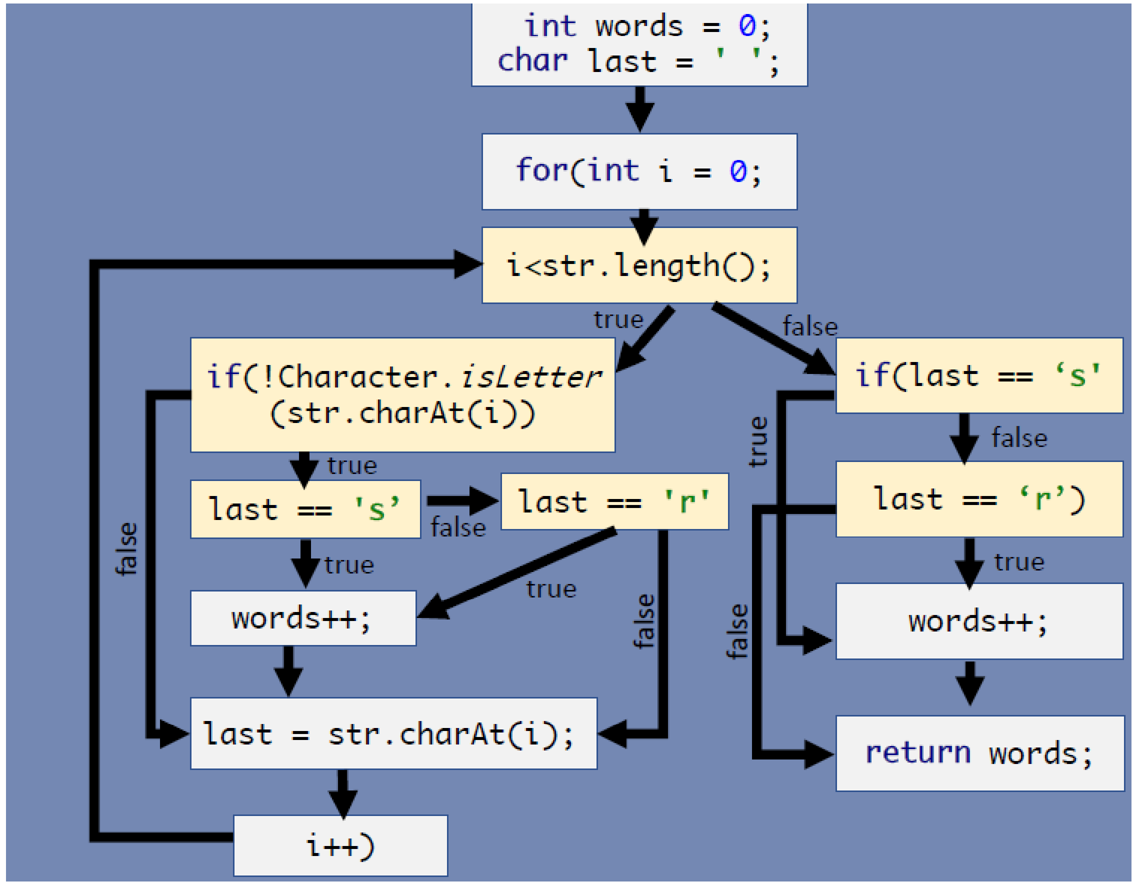
Test cases for CountWords Problem as per MC/DC

- Let's generalize the first if

(A && (B | C))

- We have three conditions, so this means that there are actually 8 different combinations

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F



Test cases for CountWords Problem as per MC/DC

(A && (B | C))

- Now, the question is: how can we identify the "useful combinations" to test?

- As per MC/DC Coverage, we will exercise each condition that could independently affect the outcome of the entire decision.

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Test cases for CountWords Problem as per MC/DC

- Let's start with A, the first condition.

(A && (B | C))

	Tests	a	b	c	Outcome
→	1	T	T	T	T
	2	T	T	F	T
	3	T	F	T	T
	4	T	F	F	F
	5	F	T	T	F
	6	F	T	F	F
	7	F	F	T	F
	8	F	F	F	F

(A && (B | C))

	Tests	a	b	c	Outcome
→	1	T	T	T	T
	2	T	T	F	T
	3	T	F	T	T
	4	T	F	F	F
→	5	F	T	T	F
	6	F	T	F	F
	7	F	F	T	F
	8	F	F	F	F

(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

- now we need to see what happens when we change A, but keep the other conditions the same.. and the outcome changed as well!

Test cases for CountWords Problem as per MC/DC

- This means that tests number 1 and number 5 are good tests for us!
- Let's now repeat it for row 2

(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Tests = {1, 5}, {2, 6}

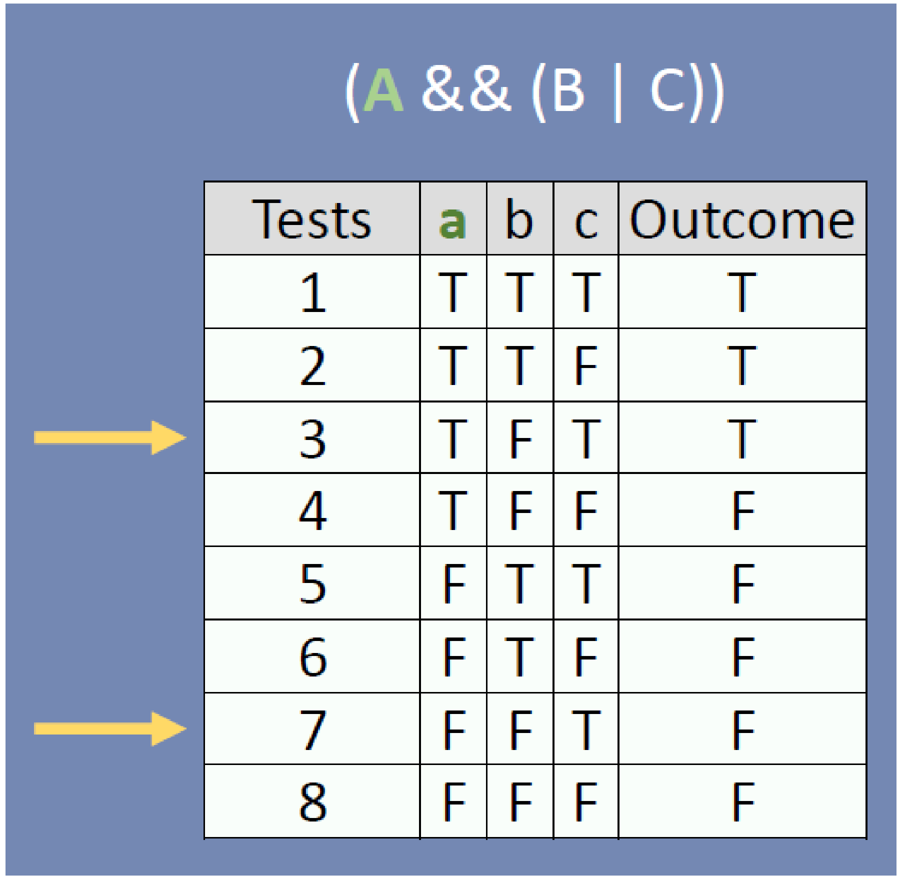
(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Tests = {1, 5}

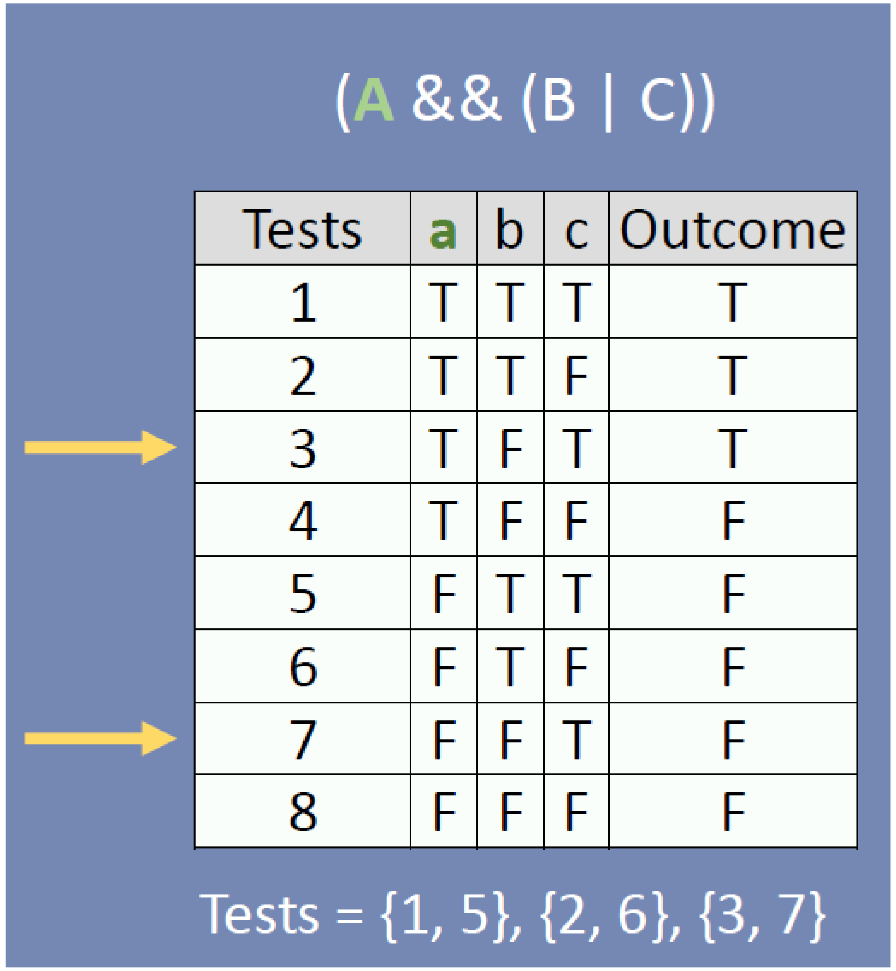
Test cases for CountWords Problem as per MC/DC

- Let's now repeat it for row 3



(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F



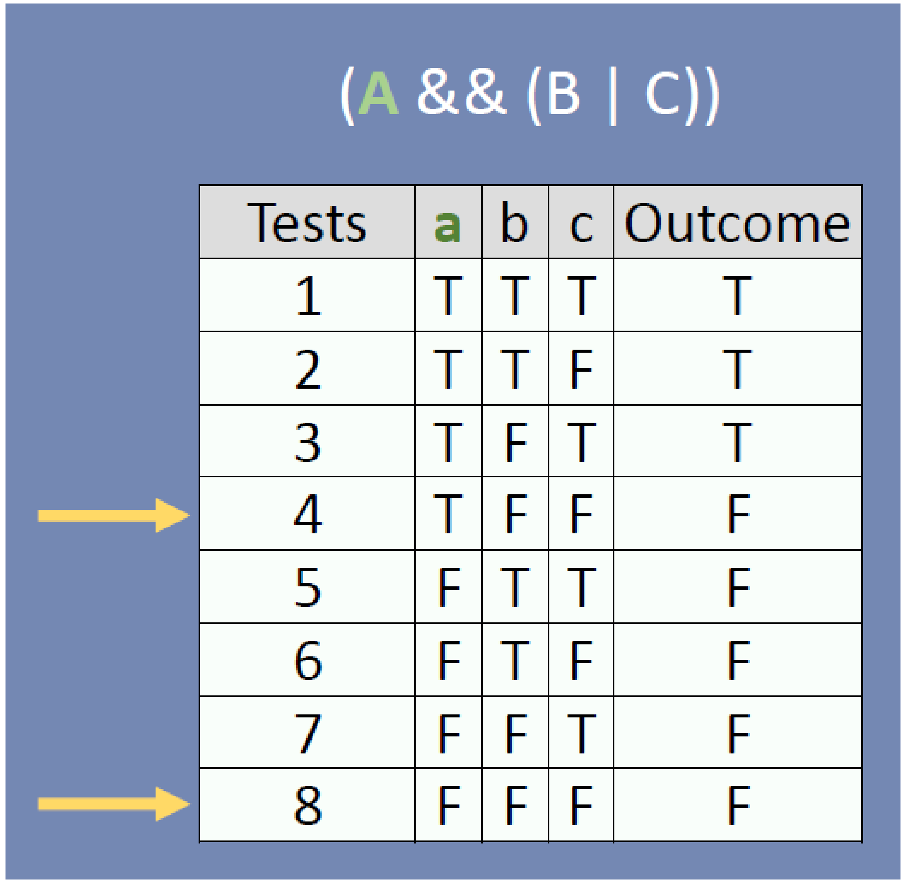
(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Tests = {1, 5}, {2, 6}, {3, 7}

Test cases for CountWords Problem as per MC/DC

- Let's now repeat it for row 4



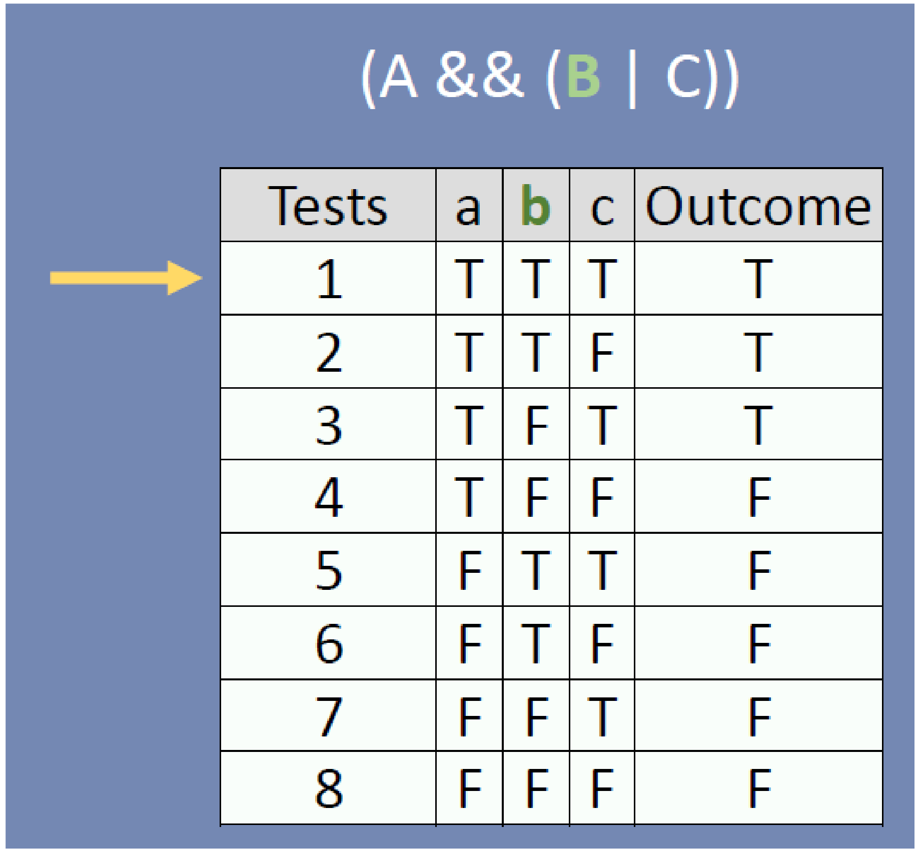
(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

- But the outcome is the same: false for both.

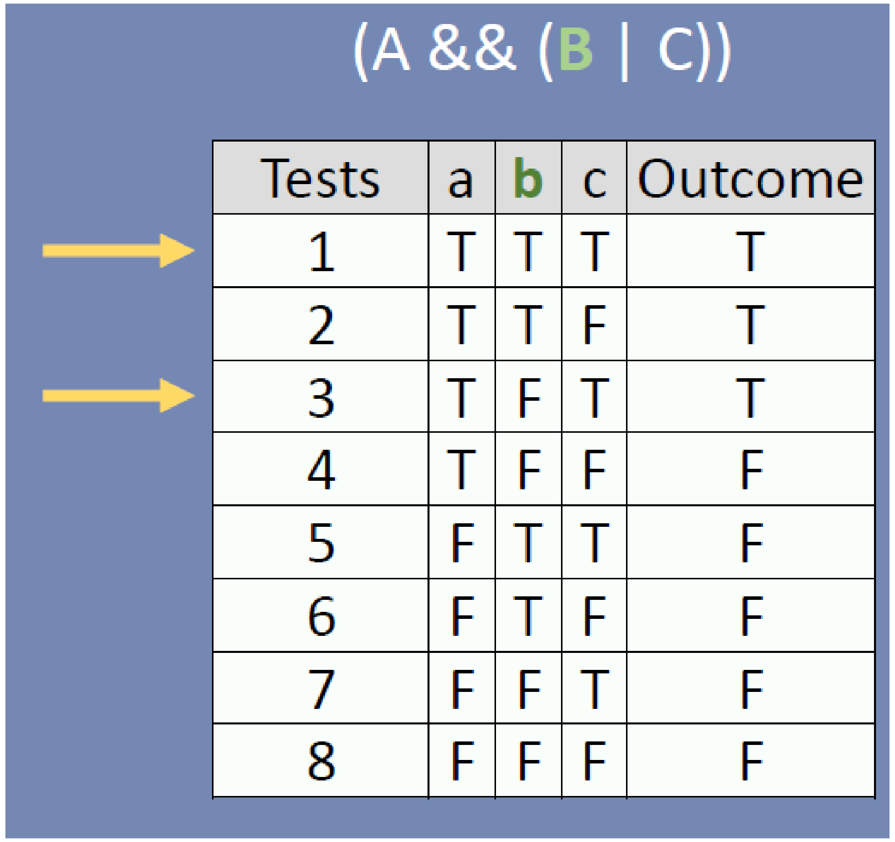
Test cases for CountWords Problem as per MC/DC

- let's go for condition b now.
- As the outcome is the same, tests 1 and 3 are not that important for B.



(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F



(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Test cases for CountWords Problem as per MC/DC

- Let's look for the row where we see the same conditions but with b flipped...
- which happens in row 4.

(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Tests = {2, 4}

Test cases for CountWords Problem as per MC/DC

- Let's go for condition C

(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Tests = {3, 4}

- Now, we see all the tests

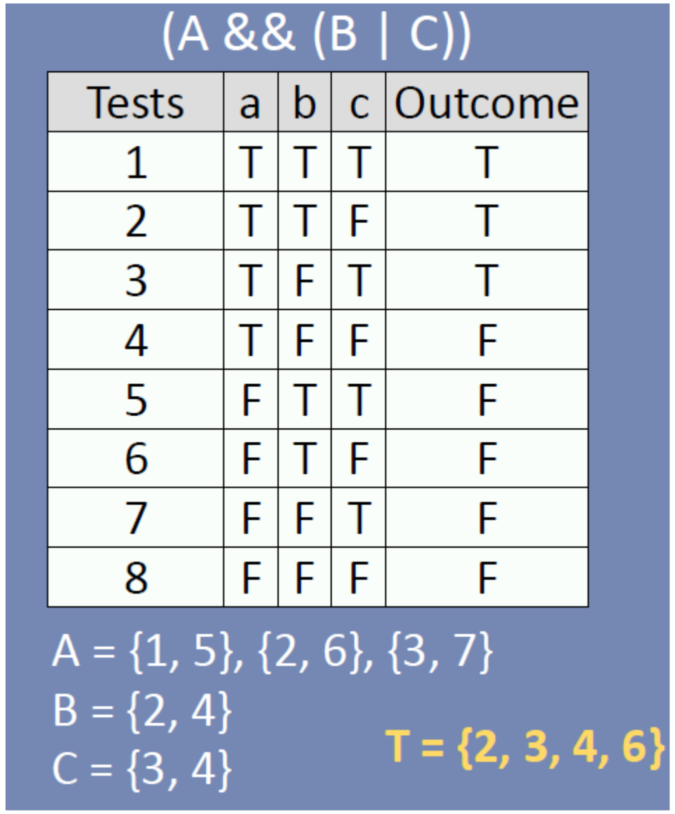
(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

A = {1, 5}, {2, 6}, {3, 7}
B = {2, 4}
C = {3, 4}

Test cases for CountWords Problem as per MC/DC

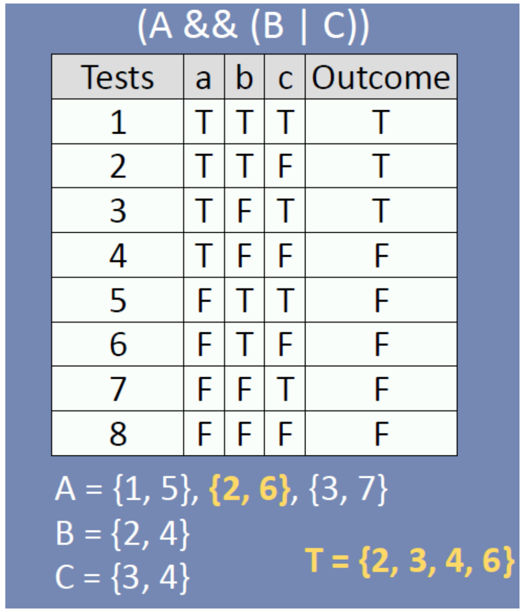
- Now, we should try to minimize the number of tests!
- Take a look: if we execute tests 2, 3, 4, and 6,



(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

A = {1, 5}, {2, 6}, {3, 7}
B = {2, 4}
C = {3, 4}
T = {2, 3, 4, 6}

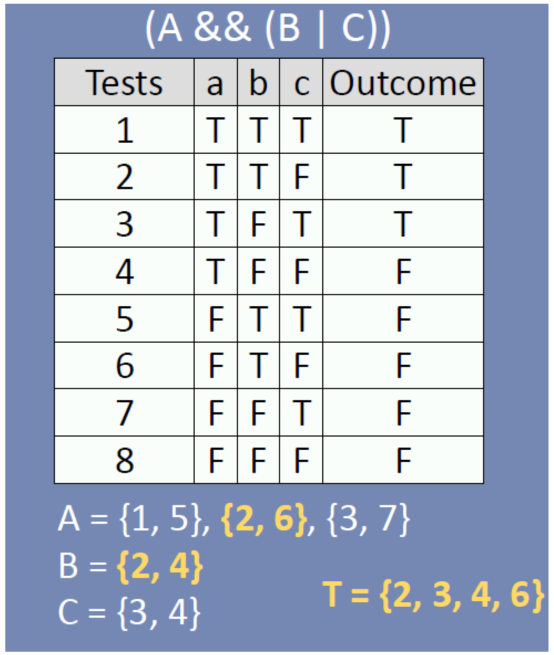


(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

A = {1, 5}, **{2, 6}**, {3, 7}
B = {2, 4}
C = {3, 4}
T = {2, 3, 4, 6}

we are selecting a pair for A

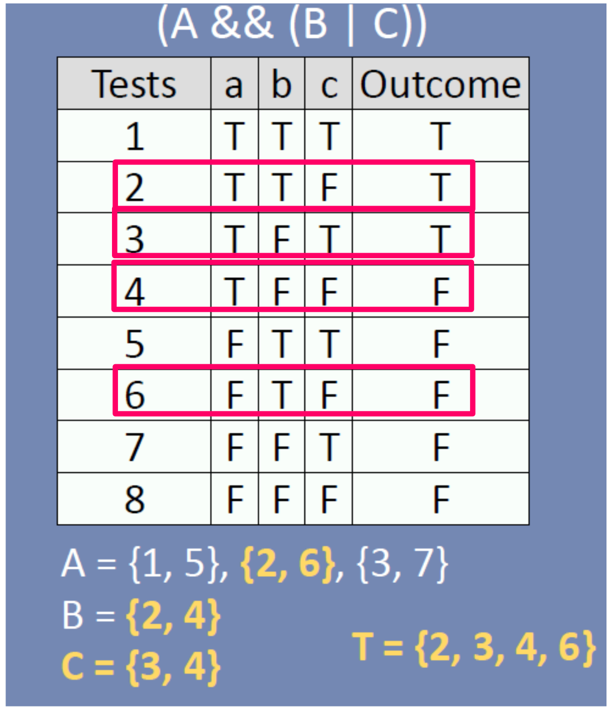


(A && (B | C))

Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

A = {1, 5}, **{2, 6}**, {3, 7}
B = **{2, 4}**
C = {3, 4}
T = {2, 3, 4, 6}

we are selecting a pair for B



(A && (B | C))

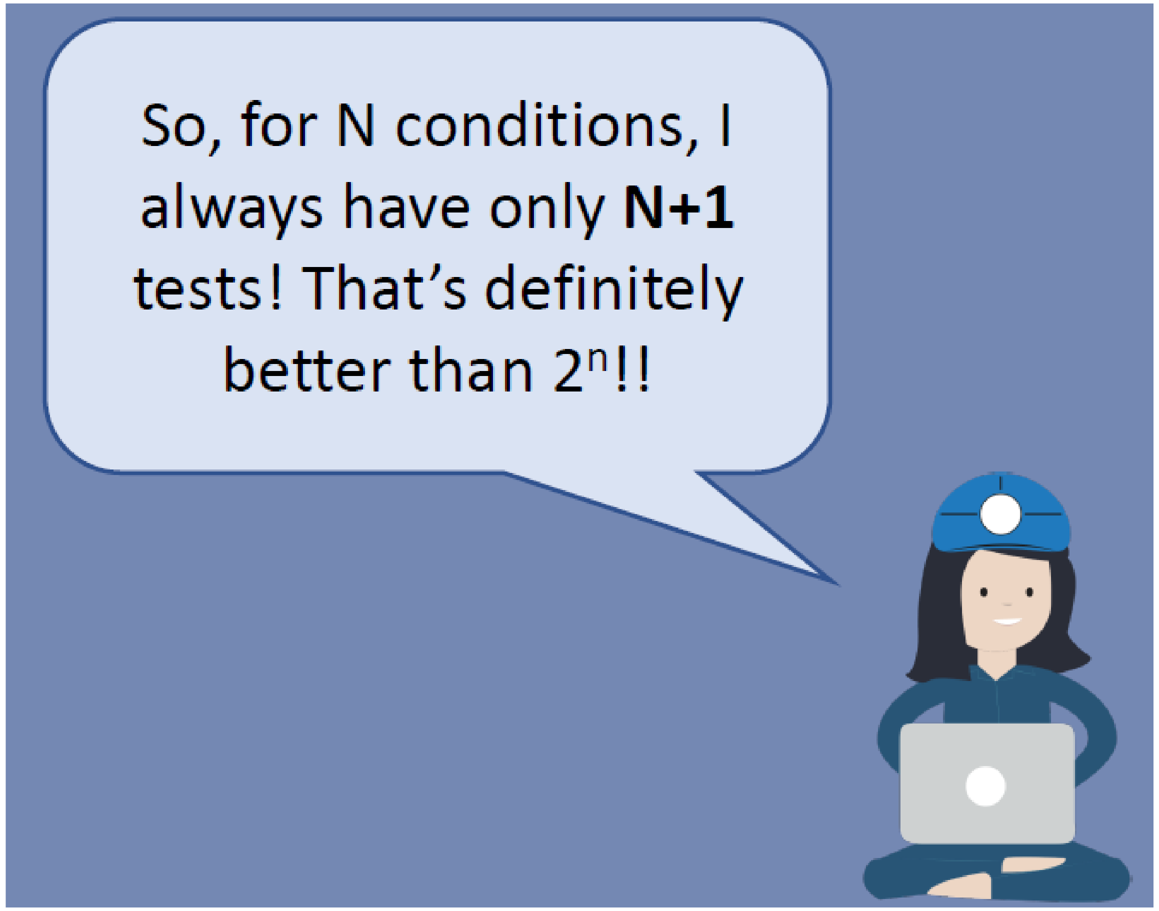
Tests	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

A = {1, 5}, **{2, 6}**, {3, 7}
B = **{2, 4}**
C = {3, 4}
T = {2, 3, 4, 6}

we are selecting a pair for C

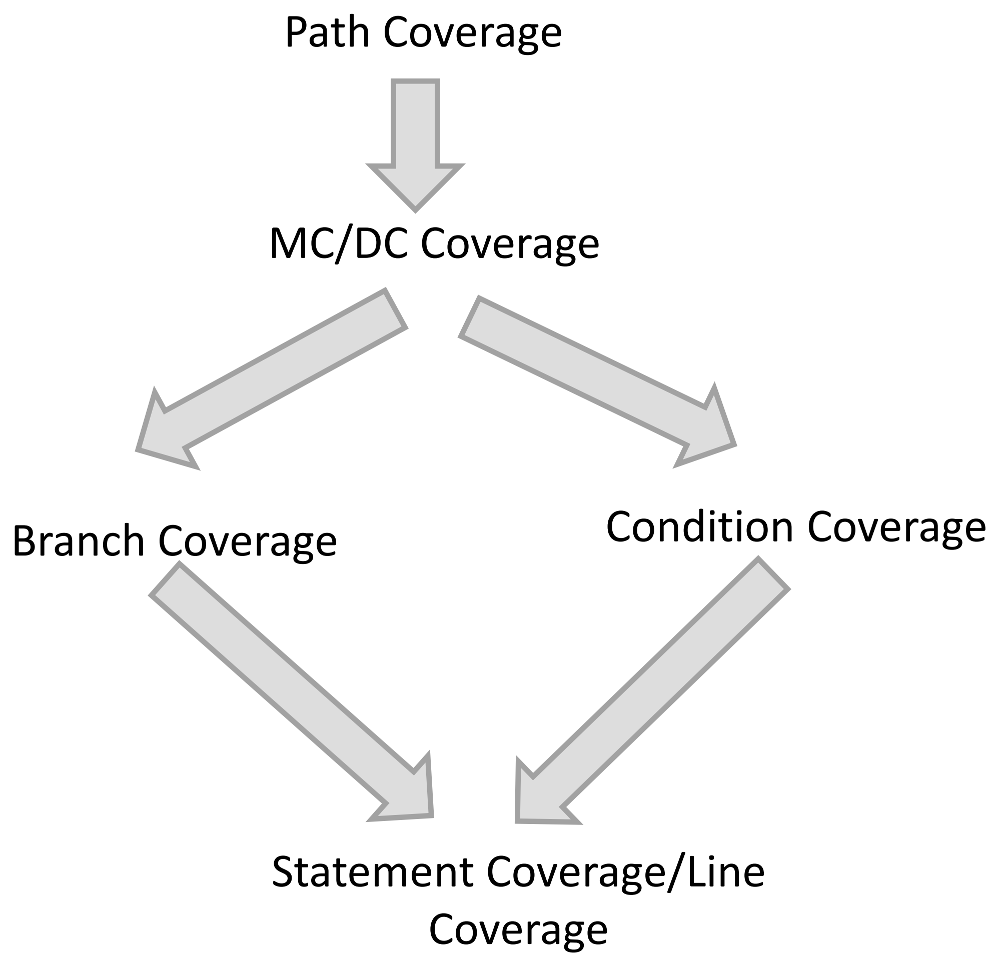
MC/DC Coverage

- In the example, our N was 3 (after all, A, B, and C), and thus, the number of tests is 4!
- This is definitely better than 2 to the power of N!
- MC/DC gives us a manageable number of tests to perform when the number of combinations is just too high!


So, for N conditions, I always have only **N+1** tests! That's definitely better than 2^n!!

Criteria subsumption

- If you think about all the different criteria we studied, we can draw some relations about them.
- For example:
- If you achieve 100% decision coverage, you also achieved 100% line coverage.
- If you achieve 100% path coverage, you also achieved 100% decision coverage and so on
- We call this relations: strategy subsumption.
- More formally: Strategy X subsumes strategy Y if all elements that Y exercises are also exercised by X.


Path Coverage
MC/DC Coverage
Branch Coverage
Condition Coverage
Statement Coverage/Line Coverage