Software Testing - Code Coverage & Tools

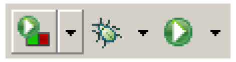
Dr. John H Robb
UTA Computer Science and Engineering

JaCoCo

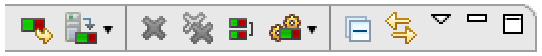
- Java code coverage (JaCoCo) is a free ECL Emma based tool for Eclipse or NetBeans
  - It provides direct code coverage analysis while in the workbench:
  - **Fast develop/test cycle:** Launches from within the workbench like JUnit -- test runs can directly be analyzed for code coverage.
  - **Rich coverage analysis:** Coverage results are immediately summarized and highlighted in the Java source code editors.
  - **Non-invasive:** EclEmma does not require modifying your projects or performing any other setup.
- The tool provides support of the individual developer in an interactive way.
- Eclipse installation: http://www.eclemma.org/installation.html

JaCoCo Source Code Annotations

- Eclipse allows running JaCoCo directly from the workbench. In addition to the Run and Debug, it provides a Run JaCoCo icon from the toolbar



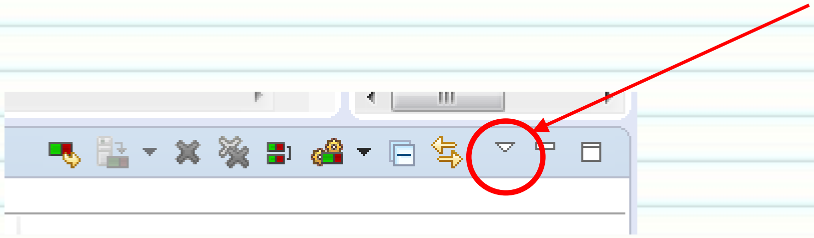
- The JaCoCo toolbar has several helpful features that you may need for your project:

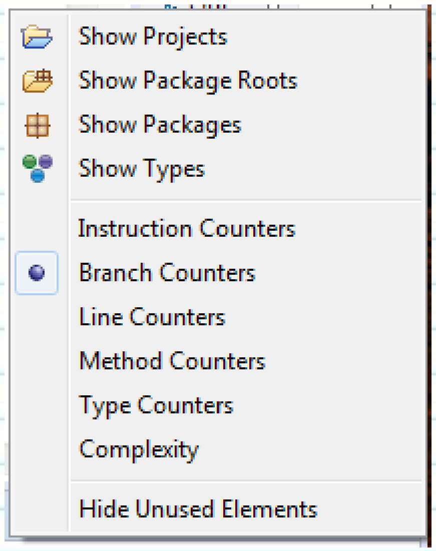


  - **Coverage Last Launched**: Re-run the currently selected coverage session.
  - **Dump Execution Data**: Dump execution data from a running process and create a new session from the data. Only active when at least one process is running in *Coverage* mode.
  - **Remove Active Session**: Remove the currently selected coverage session.
  - **Remove All Sessions**: Remove all coverage sessions.
  - **Merge Sessions**: Merges multiple sessions into a single one.
  - **Select Session**: Select session from the drop down-menu and make it the active session.
  - **Collapse All**: Collapse all expanded tree nodes.
  - **Link with Current Selection**: If this toggle is checked the coverage view automatically reveals the Java element currently selected in other views or editors.

JaCoCo Coverage Counters

- You will want to set the coverage counter to count Branches covered
- From the JaCoCo toolbar select the View Menu pulldown





- Make sure to select Branch Counters
- This is what we will use for our homework and project

JaCoCo Source Code Annotations

- Branch coverage of the active coverage session is directly displayed in the Java source editors. This works for Java source files contained in the project as well as source code attached to binary libraries.

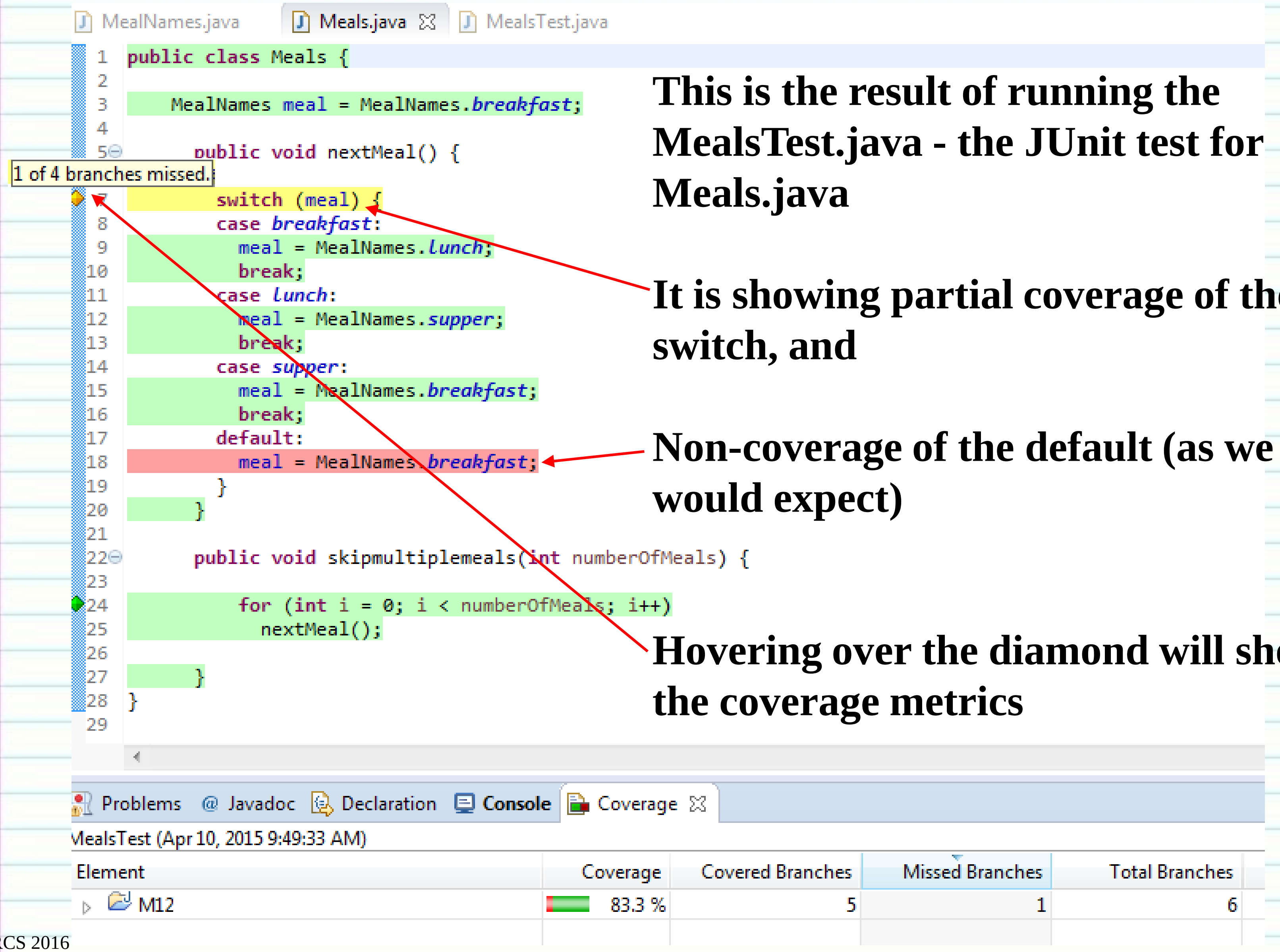
- Source lines containing executable code get the following color code:
  - green for fully covered lines,
  - yellow for partly covered lines (branches missed)
  - red for lines that have not been executed at all

- In addition colored diamonds are shown at the left for lines containing <u>decision branches</u>. The colors for the diamonds have a similar approach:
  - green for fully covered branches,
  - yellow for partly covered branches and
  - red when no branches in the particular line have been executed.

- The default colors can be modified in the Preferences dialog - <u>do not do this please</u>

- The source annotations automatically disappear when you start editing a source file or delete the coverage session.

JaCoCo Coverage Example



MealNames.java | Meals.java | MealsTest.java

```java
public class Meals {

    MealNames meal = MealNames.breakfast;

    public void nextMeal() {
        switch (meal) {
        case breakfast:
            meal = MealNames.lunch;
            break;
        case lunch:
            meal = MealNames.supper;
            break;
        case supper:
            meal = MealNames.breakfast;
            break;
        default:
            meal = MealNames.breakfast;
        }
    }

    public void skipmultiplemeals(int numberOfMeals) {

        for (int i = 0; i < numberOfMeals; i++)
            nextMeal();

    }
}
```

1 of 4 branches missed.

Problems | Javadoc | Declaration | Console | Coverage

MealsTest (Apr 10, 2015 9:49:33 AM)

Element	Coverage	Covered Branches	Missed Branches	Total Branches
M12	83.3 %	5	1	6

This is the result of running the MealsTest.java - the JUnit test for Meals.java

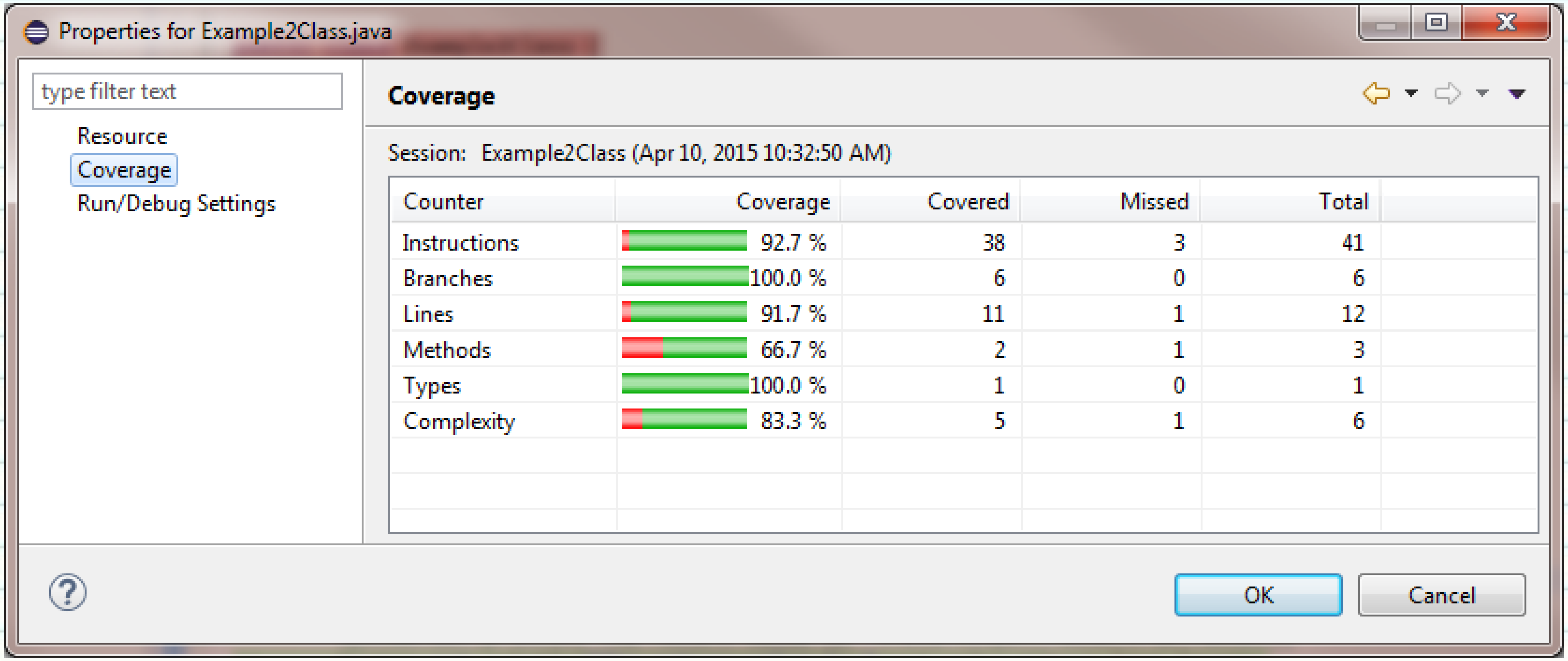
It is showing partial coverage of the switch, and

Non-coverage of the default (as we would expect)

Hovering over the diamond will show the coverage metrics

Coverage Properties

- Summary metrics can be provided by right clicking on the file, package, or project Package Explorer view, selecting Properties and then selecting the Coverage filter as shown below
- The file name is shown in both spots indicated.
- I prefer the Branches coverage measure

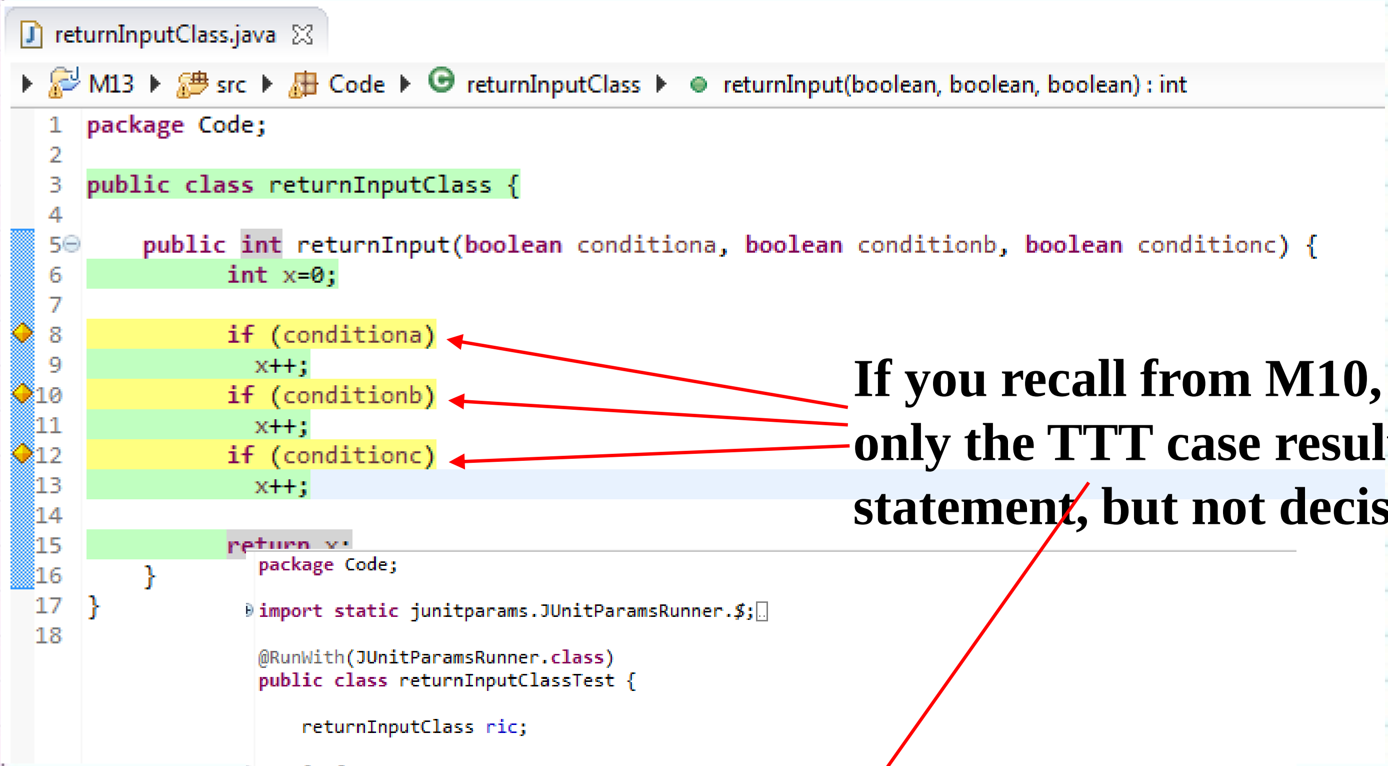


Coverage

Session: Example2Class (Apr 10, 2015 10:32:50 AM)

Counter	Coverage	Covered	Missed	Total
Instructions	92.7 %	38	3	41
Branches	100.0 %	6	0	6
Lines	91.7 %	11	1	12
Methods	66.7 %	2	1	3
Types	100.0 %	1	0	1
Complexity	83.3 %	5	1	6

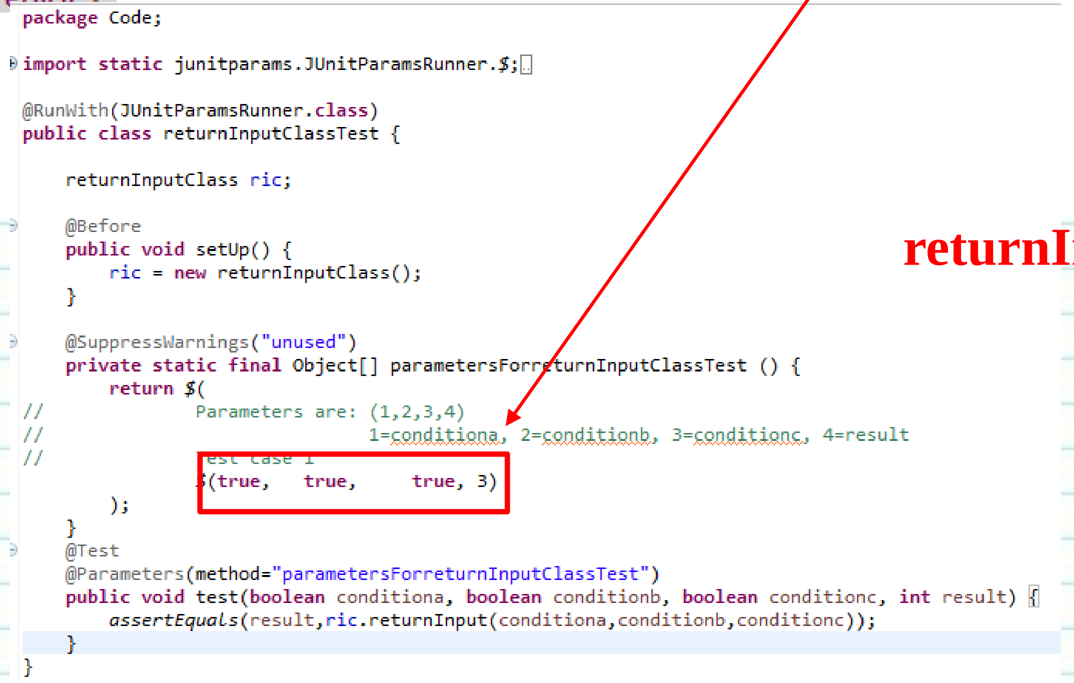
JaCoCo Coverage Analysis



returnInputClass.java

M13 ▸ src ▸ Code ▸ returnInputClass ▸ returnInput(boolean, boolean, boolean) : int

```java
package Code;

public class returnInputClass {

    public int returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {
        int x=0;

        if (conditiona)
            x++;
        if (conditionb)
            x++;
        if (conditionc)
            x++;

        return x;
    }
}
```

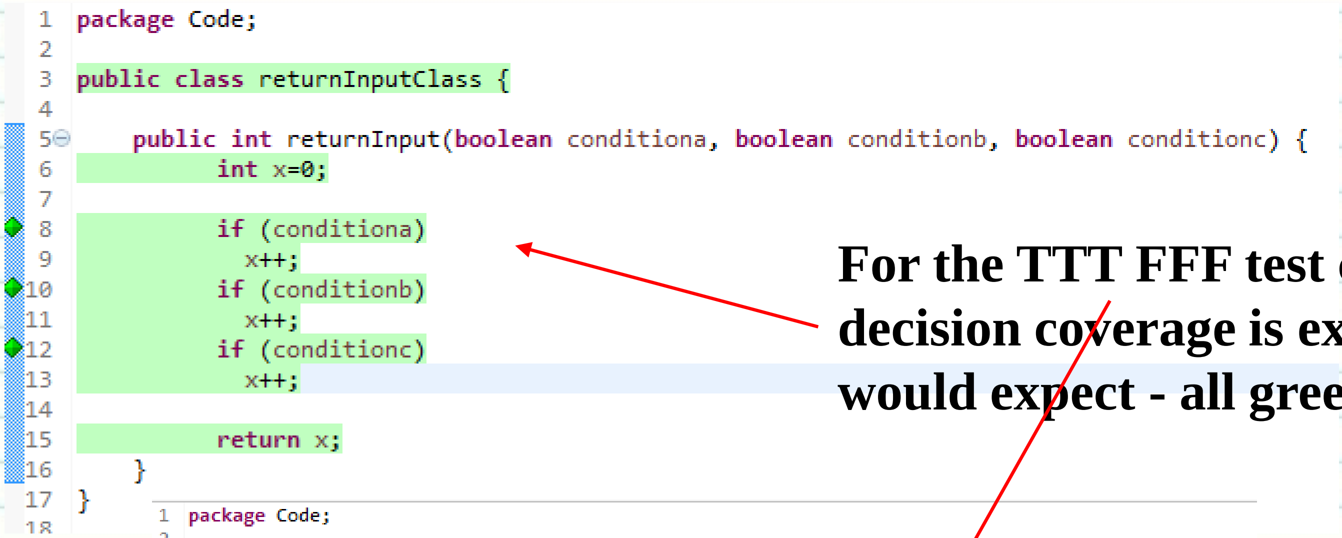


```java
package Code;

import static junitparams.JUnitParamsRunner.$;

@RunWith(JUnitParamsRunner.class)
public class returnInputClassTest {

    returnInputClass ric;

    @Before
    public void setUp() {
        ric = new returnInputClass();
    }

    @SuppressWarnings("unused")
    private static final Object[] parametersForreturnInputClassTest () {
        return $(
//              Parameters are: (1,2,3,4)
//                              1=conditiona, 2=conditionb, 3=conditionc, 4=result
//              Test case 1
                $(true,   true,     true, 3)
        );
    }
    @Test
    @Parameters(method="parametersForreturnInputClassTest")
    public void test(boolean conditiona, boolean conditionb, boolean conditionc, int result) {
        assertEquals(result,ric.returnInput(conditiona,conditionb,conditionc));
    }
}
```

If you recall from M10, execution of only the TTT case resulted in statement, but not decision coverage

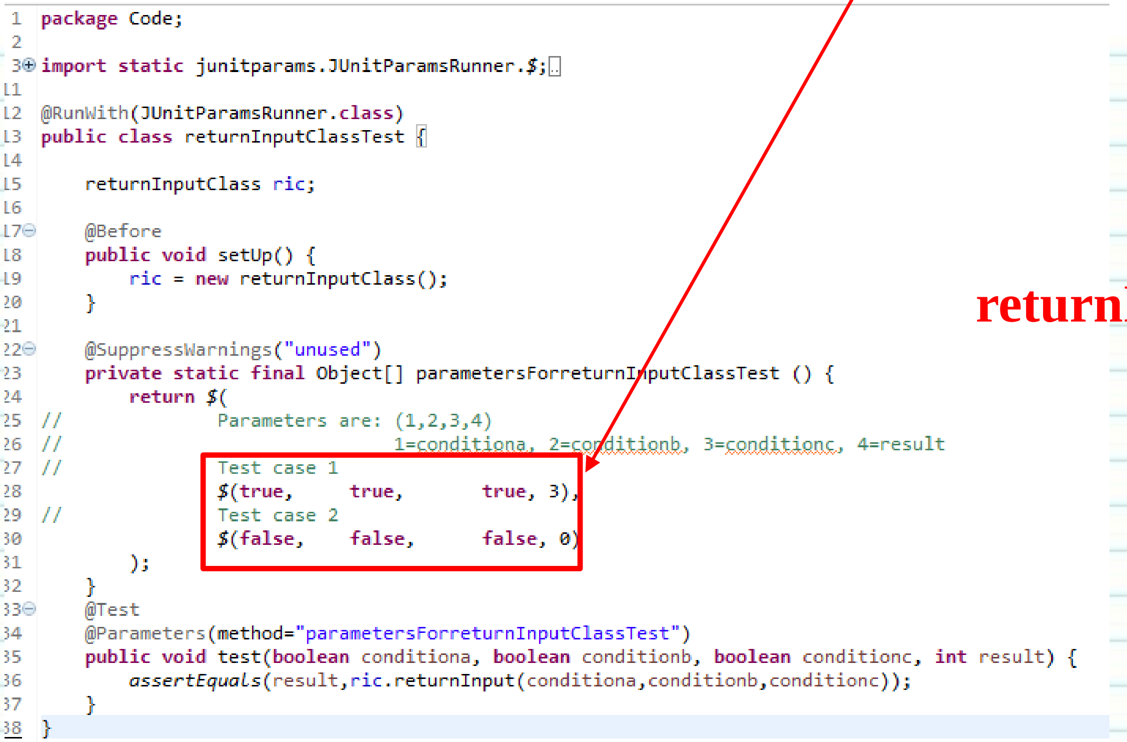
returnInputClass.java

JaCoCo Coverage Analysis (cont.)



```java
package Code;

public class returnInputClass {

    public int returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {
        int x=0;

        if (conditiona)
          x++;
        if (conditionb)
          x++;
        if (conditionc)
          x++;

        return x;
    }
}
```

For the TTT FFF test cases the decision coverage is exactly what we would expect - all green

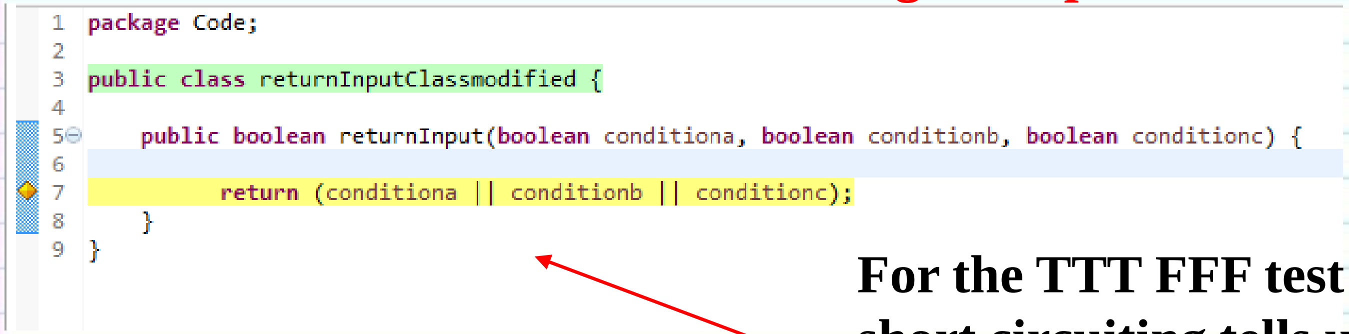


```java
package Code;

import static junitparams.JUnitParamsRunner.$;

@RunWith(JUnitParamsRunner.class)
public class returnInputClassTest {

    returnInputClass ric;

    @Before
    public void setUp() {
        ric = new returnInputClass();
    }

    @SuppressWarnings("unused")
    private static final Object[] parametersForreturnInputClassTest () {
        return $(
//              Parameters are: (1,2,3,4)
//                              1=conditiona, 2=conditionb, 3=conditionc, 4=result
//              Test case 1
                $(true,     true,       true, 3),
//              Test case 2
                $(false,    false,      false, 0)
        );
    }
    @Test
    @Parameters(method="parametersForreturnInputClassTest")
    public void test(boolean conditiona, boolean conditionb, boolean conditionc, int result) {
        assertEquals(result,ric.returnInput(conditiona,conditionb,conditionc));
    }
}
```

returnInputClass.java

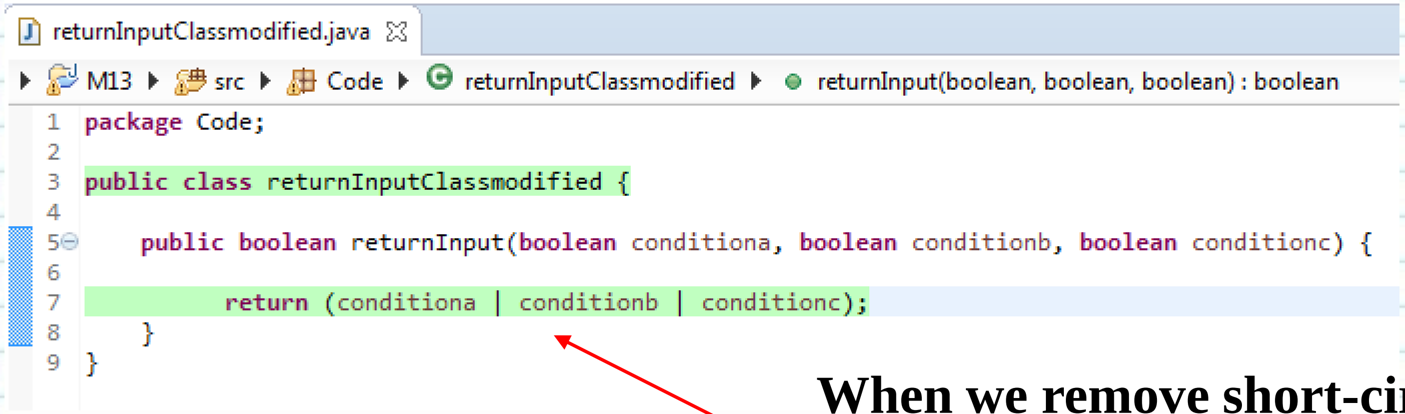
JaCoCo Coverage Analysis (cont.)

logicalExpressionClass2.java



```
package Code;

public class returnInputClassmodified {

    public boolean returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {

        return (conditiona || conditionb || conditionc);
    }
}
```

For the TTT FFF test cases here, short circuiting tells us that the underlying Java byte code has not been covered



returnInputClassmodified.java

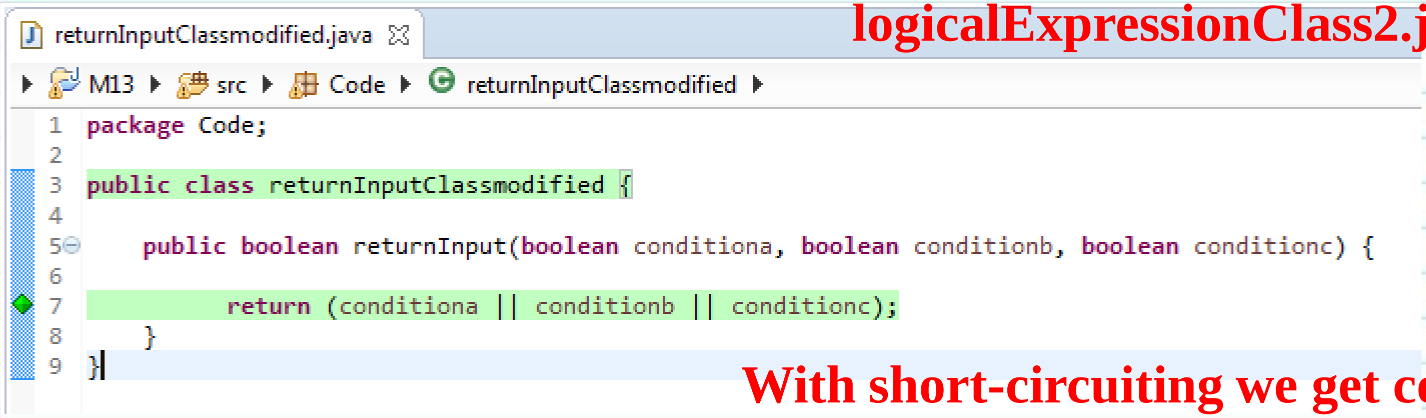
M13 ▸ src ▸ Code ▸ returnInputClassmodified ▸ returnInput(boolean, boolean, boolean) : boolean

```
package Code;

public class returnInputClassmodified {

    public boolean returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {

        return (conditiona | conditionb | conditionc);
    }
}
```

When we remove short-circuiting, and use the test cases that give us c/d coverage JaCoCo is all green

JaCoCo Coverage Analysis a + b + c

logicalExpressionClass2.java

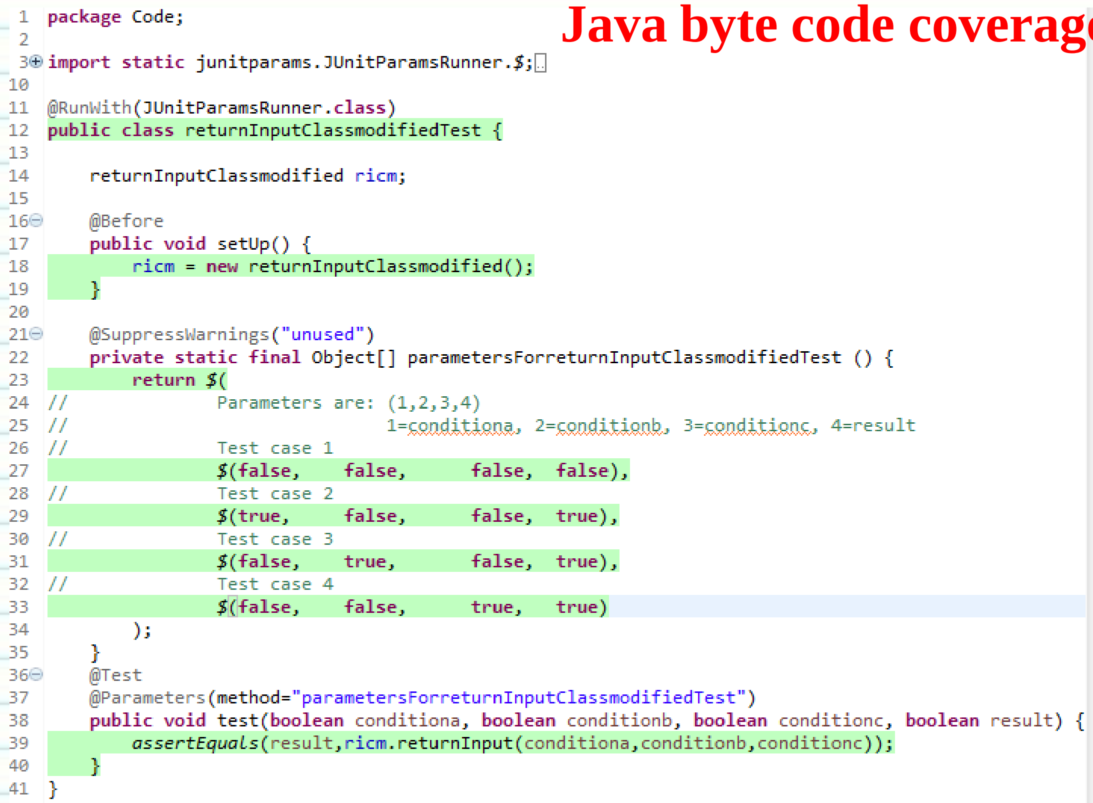


returnInputClassmodified.java

M13 ▸ src ▸ Code ▸ returnInputClassmodified ▸

```java
package Code;

public class returnInputClassmodified {

    public boolean returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {

            return (conditiona || conditionb || conditionc);
    }
}
```

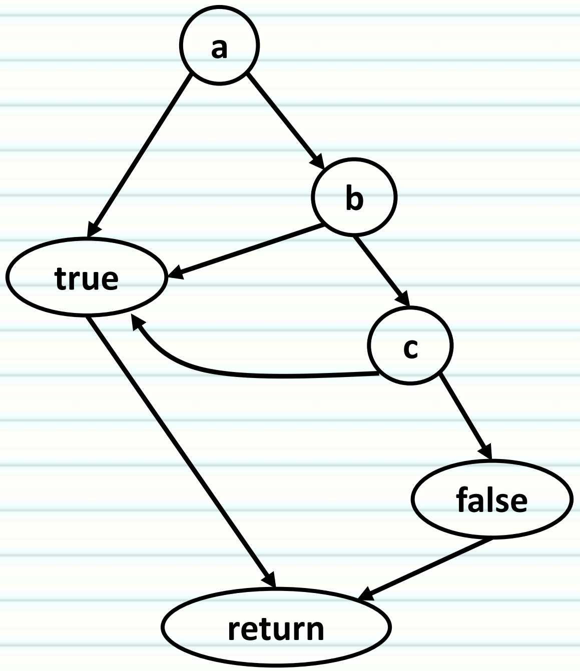
With short-circuiting we get complete Java byte code coverage with MCDC



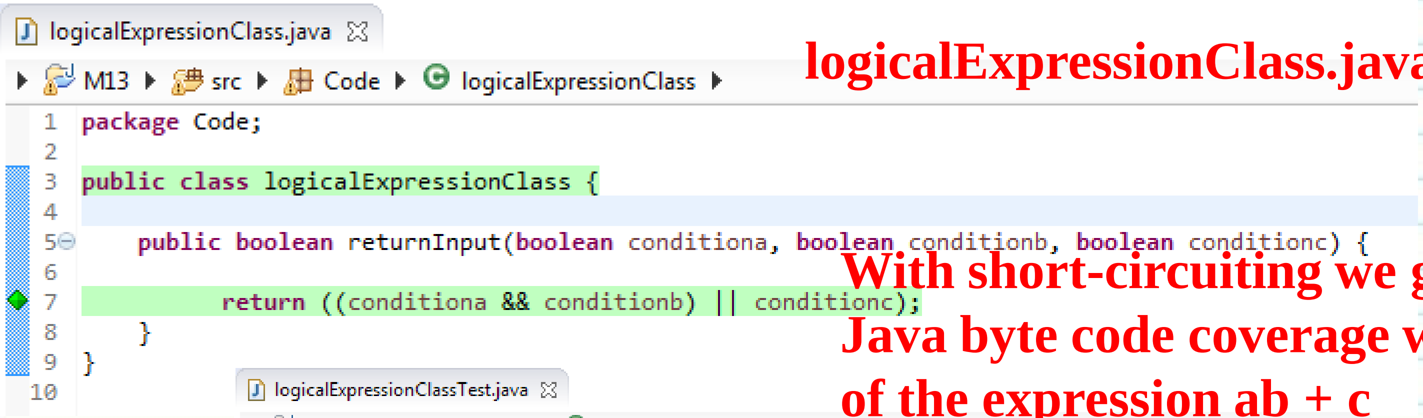
```java
package Code;

import static junitparams.JUnitParamsRunner.$;

@RunWith(JUnitParamsRunner.class)
public class returnInputClassmodifiedTest {

    returnInputClassmodified ricm;

    @Before
    public void setUp() {
        ricm = new returnInputClassmodified();
    }

    @SuppressWarnings("unused")
    private static final Object[] parametersForreturnInputClassmodifiedTest () {
        return $(
//              Parameters are: (1,2,3,4)
//                              1=conditiona, 2=conditionb, 3=conditionc, 4=result
//              Test case 1
                $(false,     false,      false,  false),
//              Test case 2
                $(true,      false,      false,  true),
//              Test case 3
                $(false,     true,       false,  true),
//              Test case 4
                $(false,     false,      true,   true)
        );
    }
    @Test
    @Parameters(method="parametersForreturnInputClassmodifiedTest")
    public void test(boolean conditiona, boolean conditionb, boolean conditionc, boolean result) {
        assertEquals(result,ricm.returnInput(conditiona,conditionb,conditionc));
    }
}
```

JaCoCo Coverage Analysis a + b + c (cont.)

- Here is the object code mapping of a + b + c
- We can see that MCDC will cover all four unique paths



JaCoCo Coverage Analysis ab + c



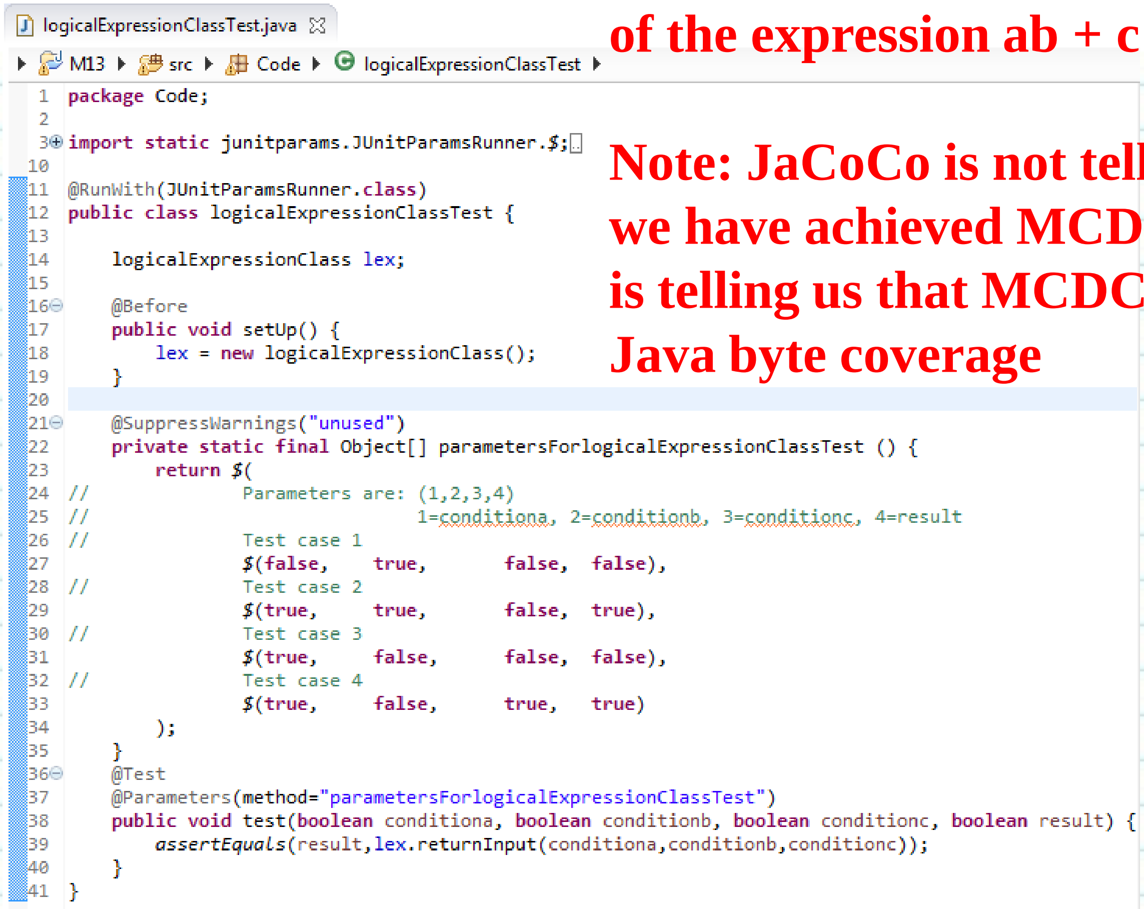
logicalExpressionClass.java

M13 ▸ src ▸ Code ▸ logicalExpressionClass ▸

```java
package Code;

public class logicalExpressionClass {

    public boolean returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {

        return ((conditiona && conditionb) || conditionc);
    }
}

```

logicalExpressionClass.java

With short-circuiting we get complete Java byte code coverage with MCDC of the expression ab + c



logicalExpressionClassTest.java

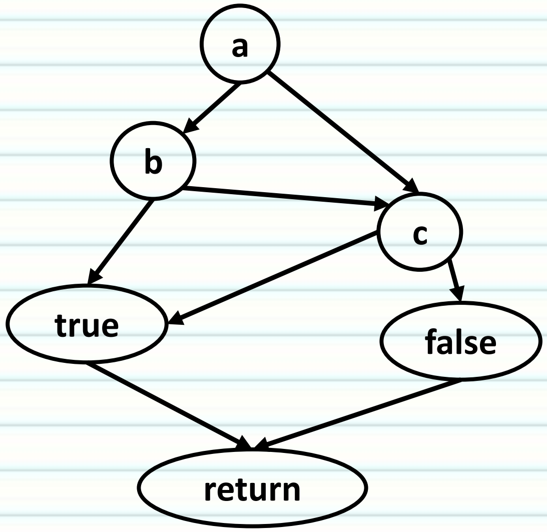
M13 ▸ src ▸ Code ▸ logicalExpressionClassTest ▸

```java
package Code;

import static junitparams.JUnitParamsRunner.$;

@RunWith(JUnitParamsRunner.class)
public class logicalExpressionClassTest {

    logicalExpressionClass lex;

    @Before
    public void setUp() {
        lex = new logicalExpressionClass();
    }

    @SuppressWarnings("unused")
    private static final Object[] parametersForlogicalExpressionClassTest () {
        return $(
//              Parameters are: (1,2,3,4)
//                      1=conditiona, 2=conditionb, 3=conditionc, 4=result
//              Test case 1
                $(false,    true,       false,  false),
//              Test case 2
                $(true,     true,       false,  true),
//              Test case 3
                $(true,     false,      false,  false),
//              Test case 4
                $(true,     false,      true,   true)
        );
    }
    @Test
    @Parameters(method="parametersForlogicalExpressionClassTest")
    public void test(boolean conditiona, boolean conditionb, boolean conditionc, boolean result) {
        assertEquals(result,lex.returnInput(conditiona,conditionb,conditionc));
    }
}
```

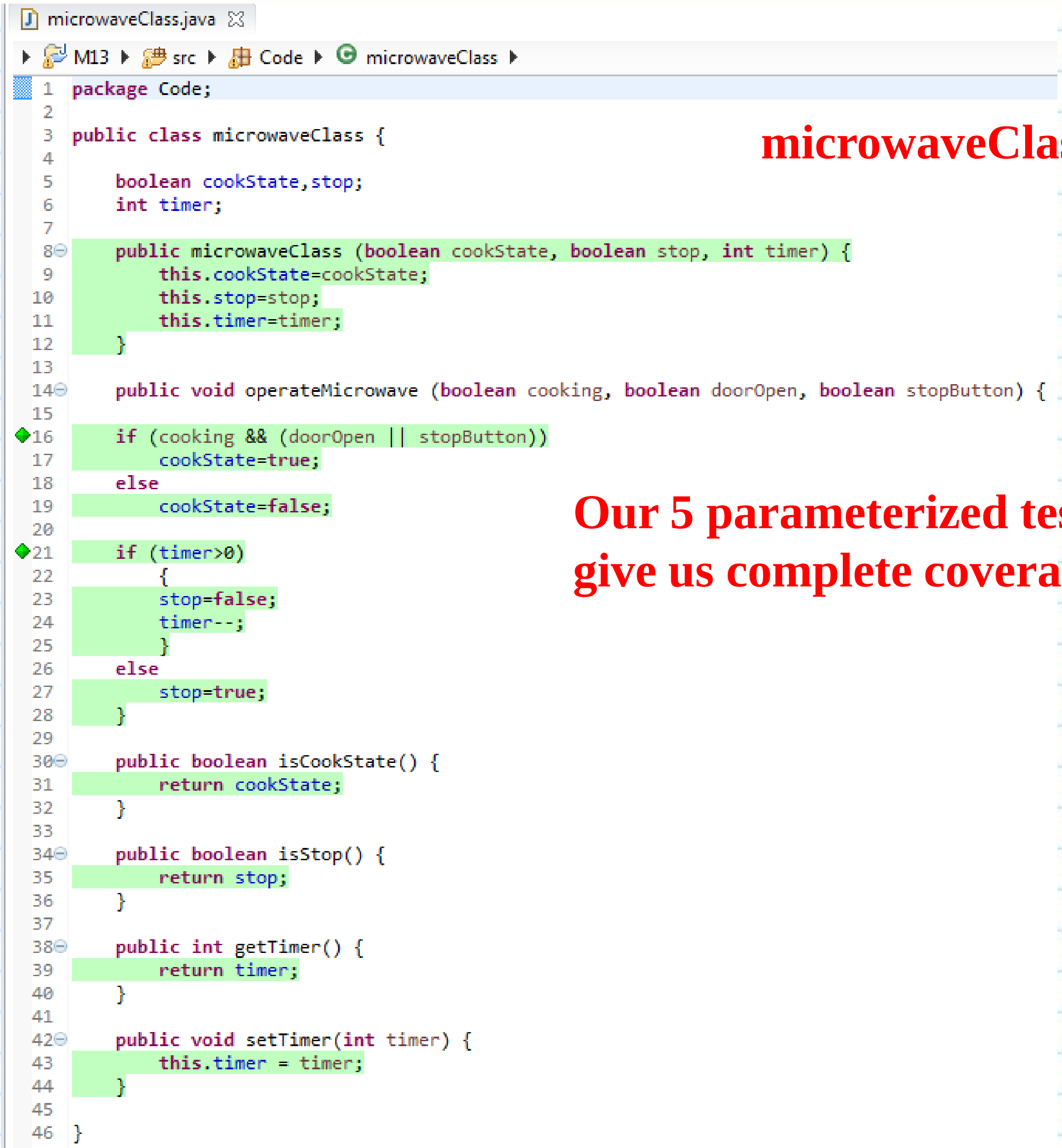
Note: JaCoCo is not telling us whether we have achieved MCDC coverage - it is telling us that MCDC achieves full Java byte coverage

JaCoCo Coverage Analysis ab + c (cont.)

- Here is the object code mapping of ab + c
- We can see that MCDC will cover all four unique paths



JaCoCo Coverage Analysis (M12 example)



```java
package Code;

public class microwaveClass {

    boolean cookState,stop;
    int timer;

    public microwaveClass (boolean cookState, boolean stop, int timer) {
        this.cookState=cookState;
        this.stop=stop;
        this.timer=timer;
    }

    public void operateMicrowave (boolean cooking, boolean doorOpen, boolean stopButton) {

    if (cooking && (doorOpen || stopButton))
        cookState=true;
    else
        cookState=false;

    if (timer>0)
        {
        stop=false;
        timer--;
        }
    else
        stop=true;
    }

    public boolean isCookState() {
        return cookState;
    }

    public boolean isStop() {
        return stop;
    }

    public int getTimer() {
        return timer;
    }

    public void setTimer(int timer) {
        this.timer = timer;
    }
}
```

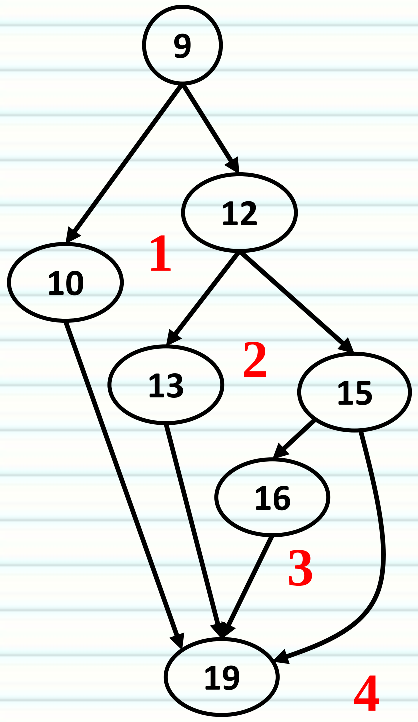
microwaveClass.java

Our 5 parameterized tests from M12 give us complete coverage

JaCoCo and Boundary Values

```
public void setAlerts (double fuel_level, Problem3Class prb3) {

        prb3.setRed_light(false);prb3.setYellow_light(false);prb3.setGreen_light(false);

        if (fuel_level<20.0)
                prb3.setGreen_light(true);
        else
                if (fuel_level<35.0)
                        prb3.setYellow_light(true);
                else
                        if (fuel_level<=75.0)
                                prb3.setRed_light(true);


        }
```

JaCoCo and Boundary Values (cont.)



Test Case	Inputs	Expected Outputs			
	fuel_level	alerts.red_light	alerts.yellow_light	alerts.green_light	Basis Path Tested
1	19.9	FALSE	FALSE	TRUE	9-10-19
2	34.9	FALSE	TRUE	FALSE	9-12-13-19
3	75.0	TRUE	FALSE	FALSE	9-12-15-16-19
4	75.1	FALSE	FALSE	FALSE	9-12-15-19
5	0.0	FALSE	FALSE	TRUE	-
6	20.0	FALSE	TRUE	FALSE	-
7	35.0	TRUE	FALSE	FALSE	-
8	100.0	FALSE	FALSE	FALSE	-

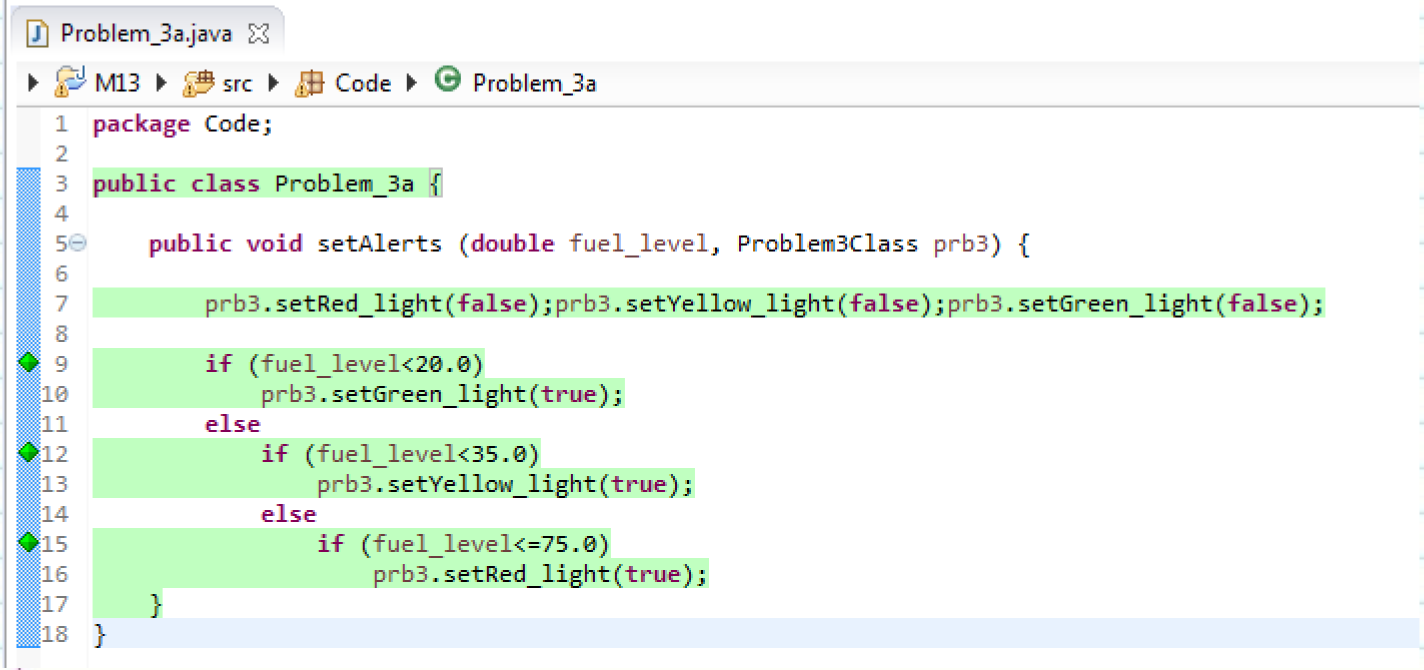
Problem_3a.java

Basis path provides coverage of the values in black

0.0 19.9	20.0 34.9	35.0 75.0	75.1 100.0

Let's just run the first 4 test cases (basis path)

JaCoCo and Boundary Values (cont.)


Problem_3a.java

M13 ▸ src ▸ Code ▸ Problem_3a

```java
package Code;

public class Problem_3a {

    public void setAlerts (double fuel_level, Problem3Class prb3) {

        prb3.setRed_light(false);prb3.setYellow_light(false);prb3.setGreen_light(false);

        if (fuel_level<20.0)
            prb3.setGreen_light(true);
        else
            if (fuel_level<35.0)
                prb3.setYellow_light(true);
            else
                if (fuel_level<=75.0)
                    prb3.setRed_light(true);
    }
}
```

What if we only run the basis path set - this should give us decision coverage.
BUT we use values of 10.0, 28.0, 50.0 and 80.0 for the basis path test inputs - these are not BVs - what does JaCoCo tell us?

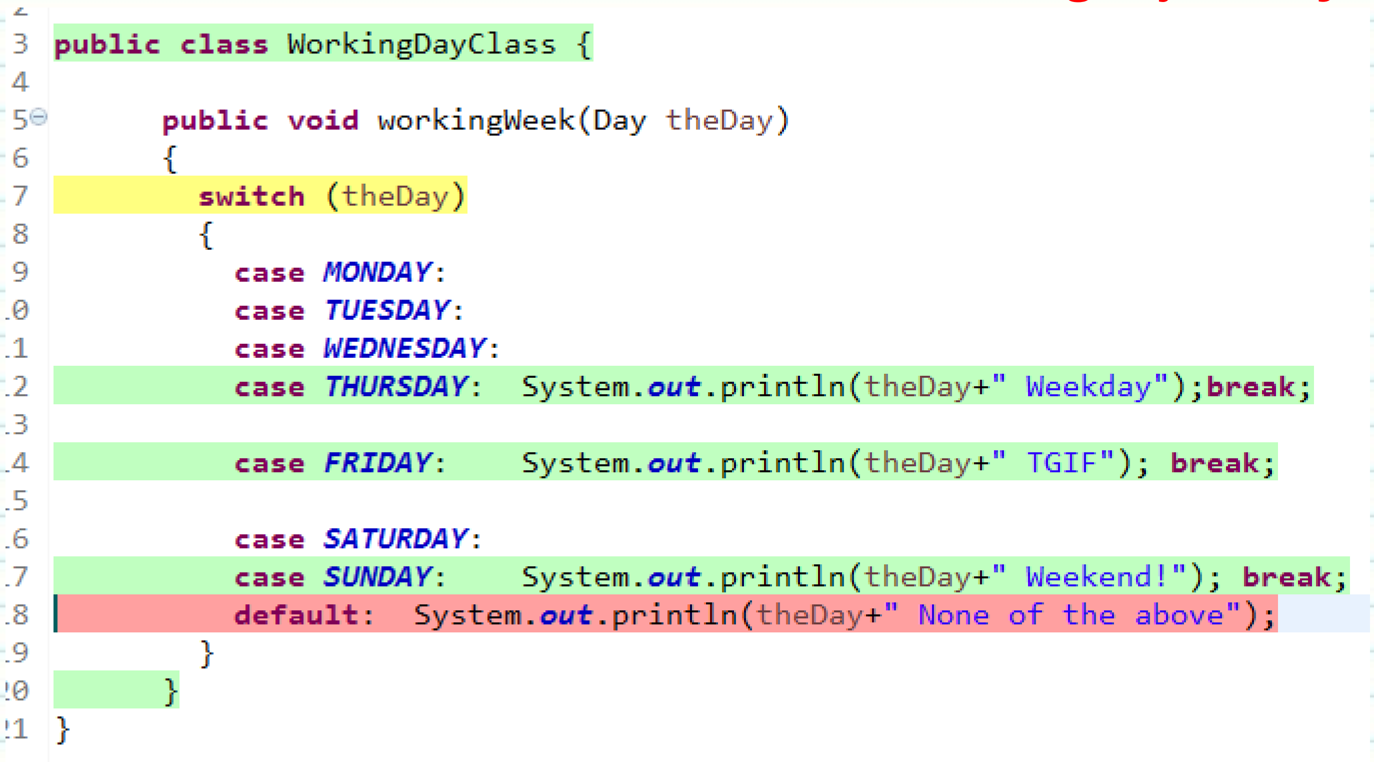
JaCoCo gives us full coverage but we haven't tested a single BV!

Why?

JaCoCo and Switches

- It is typical to get a missing "branch" with a switch when the default cannot be covered

WorkingDayClass.java

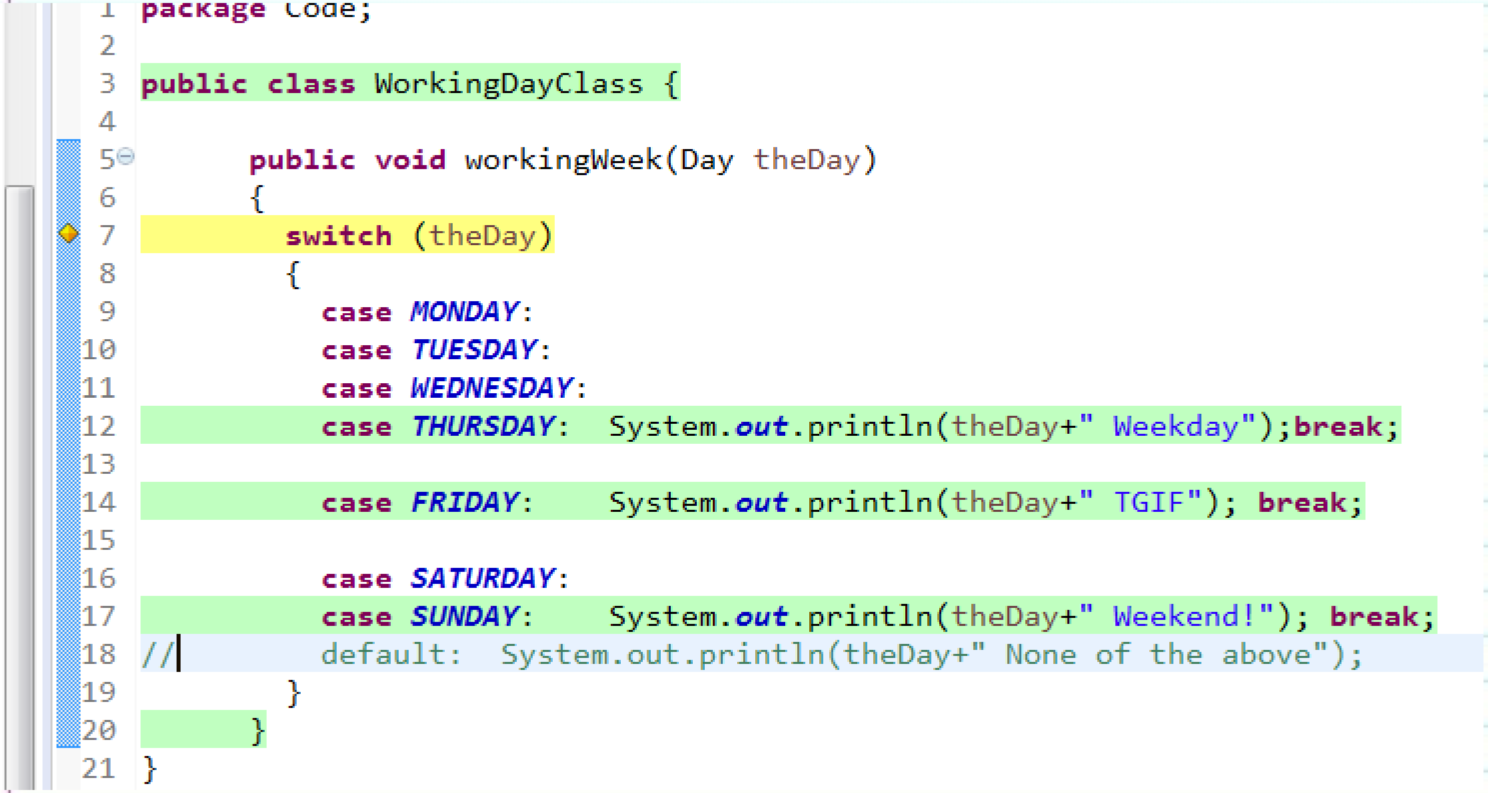


```java
public class WorkingDayClass {

        public void workingWeek(Day theDay)
        {
          switch (theDay)
          {
            case MONDAY:
            case TUESDAY:
            case WEDNESDAY:
            case THURSDAY:   System.out.println(theDay+" Weekday");break;

            case FRIDAY:     System.out.println(theDay+" TGIF"); break;

            case SATURDAY:
            case SUNDAY:     System.out.println(theDay+" Weekend!"); break;
            default:   System.out.println(theDay+" None of the above");
          }
        }
}
```

JaCoCo and Switches (cont.)

- Even without a default we still get a yellow - but every statement is covered

WorkingDayClass.java

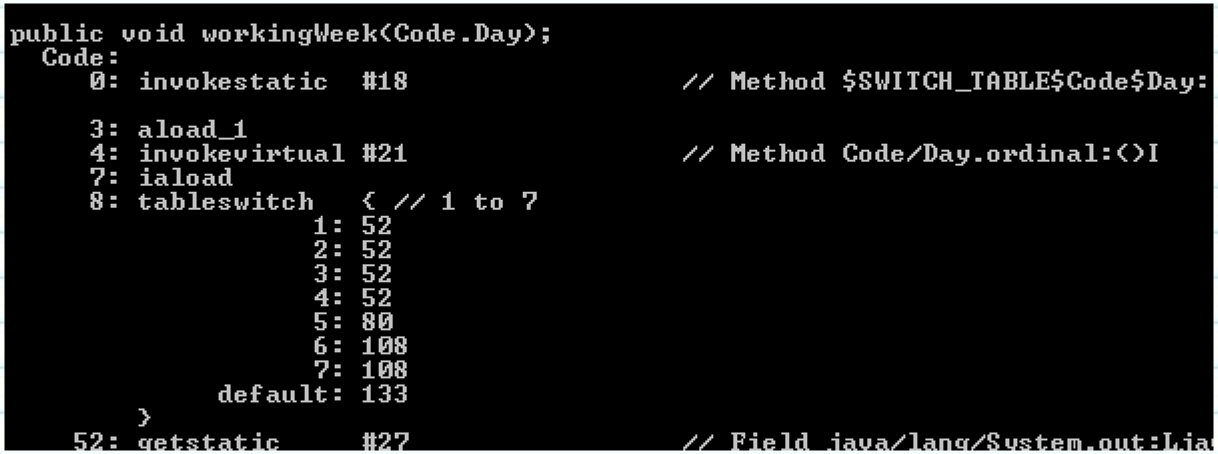


```
package code;

public class WorkingDayClass {

        public void workingWeek(Day theDay)
        {
          switch (theDay)
          {
            case MONDAY:
            case TUESDAY:
            case WEDNESDAY:
            case THURSDAY:   System.out.println(theDay+" Weekday");break;

            case FRIDAY:     System.out.println(theDay+" TGIF"); break;

            case SATURDAY:
            case SUNDAY:     System.out.println(theDay+" Weekend!"); break;
//          default:  System.out.println(theDay+" None of the above");
          }
        }
}
```

JaCoCo and Switches (cont.)

- Let’s examine the Java byte code for the jump table
  $ javap -c WorkingDayClass.class

WorkingDayClass.java



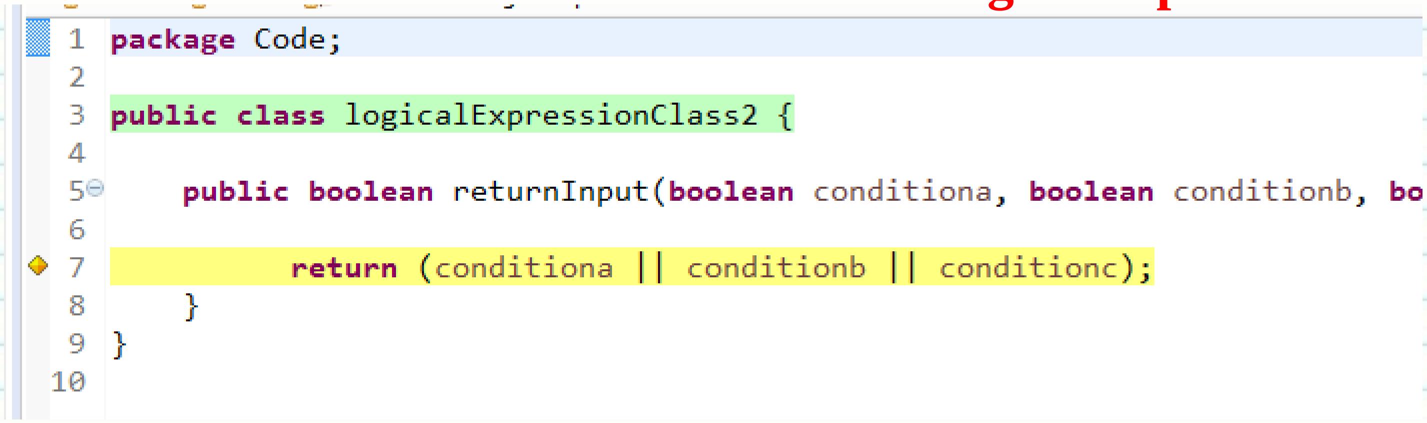
```
public void workingWeek(Code.Day);
  Code:
     0: invokestatic  #18                 // Method $SWITCH_TABLE$Code$Day:

     3: aload_1
     4: invokevirtual #21                 // Method Code/Day.ordinal:()I
     7: iaload
     8: tableswitch   { // 1 to 7
                   1: 52
                   2: 52
                   3: 52
                   4: 52
                   5: 80
                   6: 108
                   7: 108
             default: 133
        }
    52: getstatic     #27                 // Field java/lang/System.out:Lja
```

- Notice that the compiler is using the Ordinal values for the enumeration (1 to 7)
- Even though the source code doesn’t have a default statement the compiler still generates a default in the Java byte code

JaCoCo and Logical Expressions (cont.)

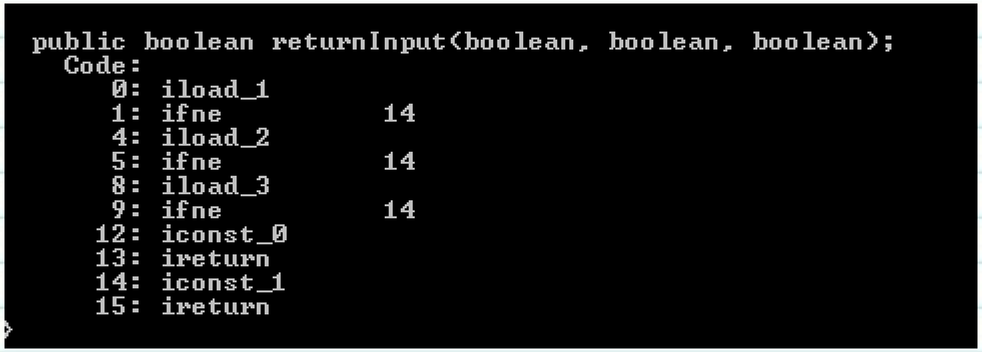
- If we just test the following expression with c/d coverage (TTT, FFF) we get the following from JaCoCo

logicalExpressionClass2.java



```
package Code;

public class logicalExpressionClass2 {

    public boolean returnInput(boolean conditiona, boolean conditionb, bo

        return (conditiona || conditionb || conditionc);
    }
}

```

- JBC is



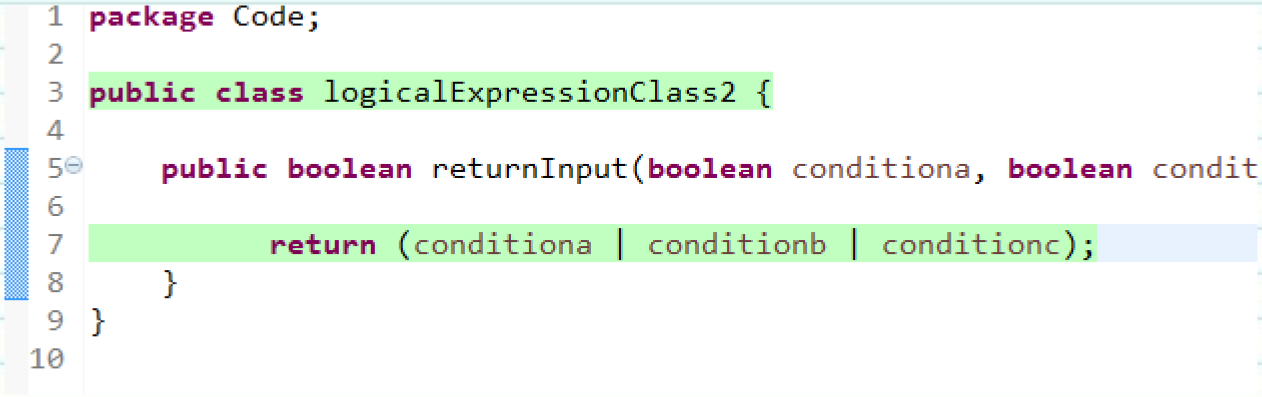
```
public boolean returnInput(boolean, boolean, boolean);
  Code:
     0: iload_1
     1: ifne          14
     4: iload_2
     5: ifne          14
     8: iload_3
     9: ifne          14
    12: iconst_0
    13: ireturn
    14: iconst_1
    15: ireturn
```

With short circuiting we have four paths in the JBC

JaCoCo and Logical Expressions

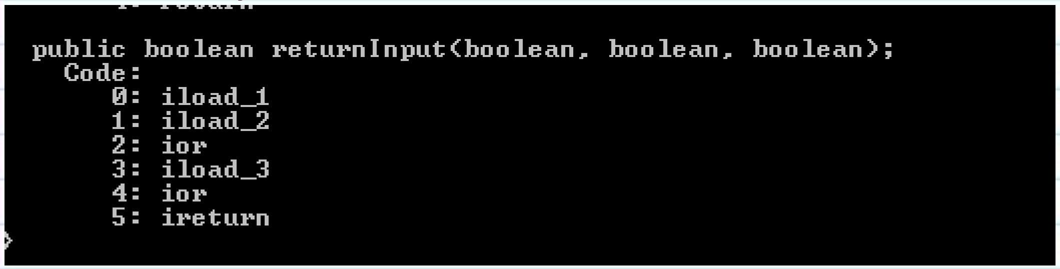
- Let’s use bit-wise operators with c/d coverage (TTT, FFF) we get the following from JaCoCo

logicalExpressionClass2.java



```
package Code;

public class logicalExpressionClass2 {

    public boolean returnInput(boolean conditiona, boolean condit

            return (conditiona | conditionb | conditionc);
    }
}

```

- Java byte code is



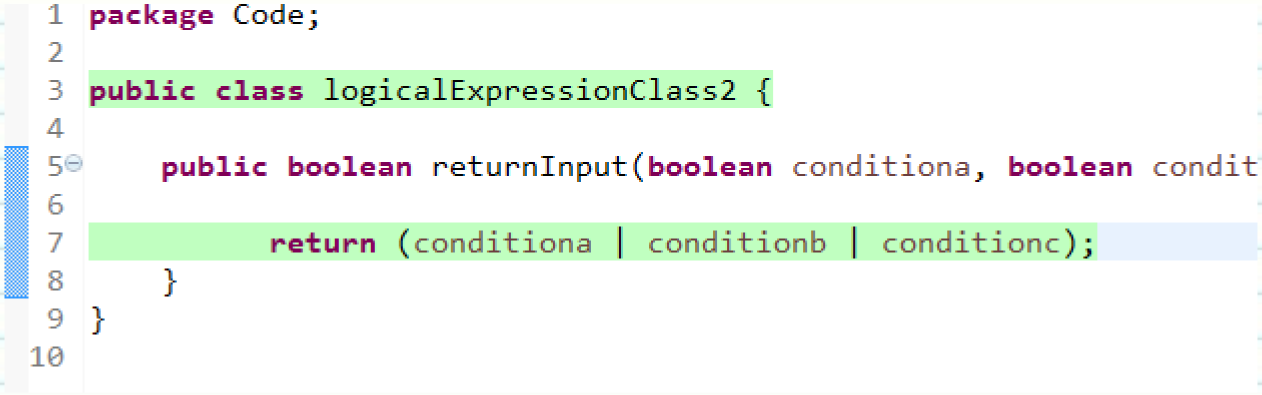
```
public boolean returnInput(boolean, boolean, boolean);
  Code:
     0: iload_1
     1: iload_2
     2: ior
     3: iload_3
     4: ior
     5: ireturn
```

No branches in the JBC

JaCoCo and Logical Expressions

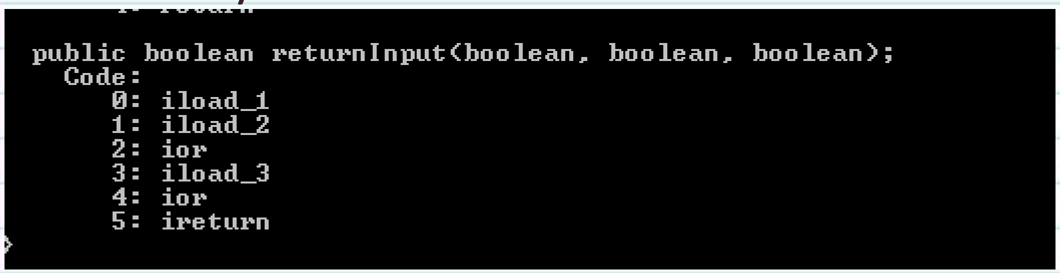
- Let’s use bit-wise operators with one test (TTT) we get the following from JaCoCo

logicalExpressionClass2.java



```
package Code;

public class logicalExpressionClass2 {

    public boolean returnInput(boolean conditiona, boolean condit

        return (conditiona | conditionb | conditionc);
    }
}

```

- Java byte code is



```
public boolean returnInput(boolean, boolean, boolean);
  Code:
     0: iload_1
     1: iload_2
     2: ior
     3: iload_3
     4: ior
     5: ireturn
```

No branches in the JBC

- Any test case will show full coverage even FFF

JaCoCo Coverage Analysis (cont.)

- From the JaCoCo site:
  - *Why is this line yellow?*
  - *EclEmma annotates all lines which are partly covered in yellow. Partly covered means that not all instructions and branches associated with this line have been executed during the coverage session. In some cases it is not obvious why the Java compiler creates extra byte code for a particular line of source code.*

- Additional Resources for JaCoCo: http://www.eclemma.org/resources.html

JaCoCo Coverage Summary

- **JaCoCo** measures **Java byte code** - let's look at statements to see what Java byte code looks like
- ***If*** statement has only a true and false
  - decision coverage will get us all green
  - on assymetric ***If*** statements yellow is statement coverage if no red lines in the entire ***If*** statement
  - a red line anywhere in the entire ***If*** statement indicates no coverage
- ***switch*** statement each **unique case** (including default **even if not present**) has a unique path -
  - think of the Cyclomatic complexity of the switch
  - executing at least each unique case (including default **even if not present**) will get us a green
- **logical expression** statement **with short circuiting** - 98% of the time the number of paths is the # of MCDC test cases in the Java byte code
- **logical expression** statement **bit wise operators** - there is only one path in the Java byte code - any coverage but none will get green

JaCoCo Coverage Summary (cont.)

- When we have decision(s) with only one condition
  - basis path testing will give us all green diamonds and lines.

- When we have decisions with multiple conditions
  - we need to use MCDC on each multiple condition decision to achieve all green diamonds and lines
  - MCDC **satisfies** JaCoCo - JaCoCo **does NOT confirm** MCDC

- When we have boundary conditions (values) we will still achieve all green diamonds and lines even though we have only tested the basis path
  - This means that some BVs are not tested.
  - This means that the extreme ECP values are not tested.
  - when we add the test cases to cover all BVs and ECPs it does not get JaCoCo **greener**.

JaCoCo Coverage Summary (cont.)

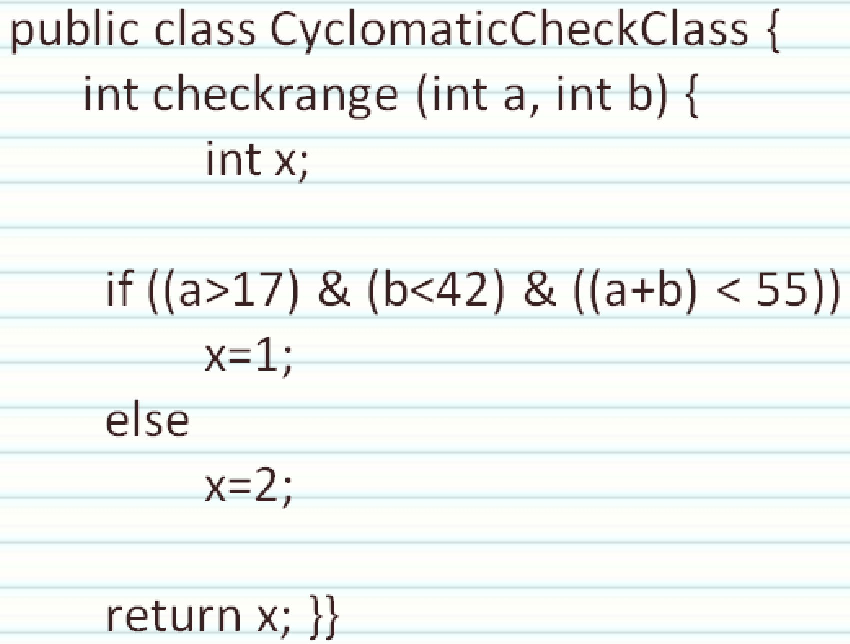
Source line color	Diamond color	Possible?	Caused by?
Green	None	Yes	Line covered and has no decisions (sequential code statement)
	Green	Yes	1) Single decision statement with both decisions covered 2) Multiple decision statement with complete Java byte code coverage
	Yellow	No	
	Red	No	
Yellow	None	No	
	Green	No	
	Yellow	Yes	1) Typical state for a switch statement OR 2) Single decision statement with only one decision covered OR 3) Multiple decision statement with incomplete Java byte code coverage (possibly not even decision coverage)
	Red	No	

JaCoCo Coverage Summary (cont.)

Source line color	**Diamond color**	**Possible?**	**Caused by?**
Red	None	Yes	Statement that has not been executed
	Green	No	
	Yellow	No	
	Red	Yes	Code with decisions (and/or multiple conditions) and no statements in source code have been executed

Always Double Check

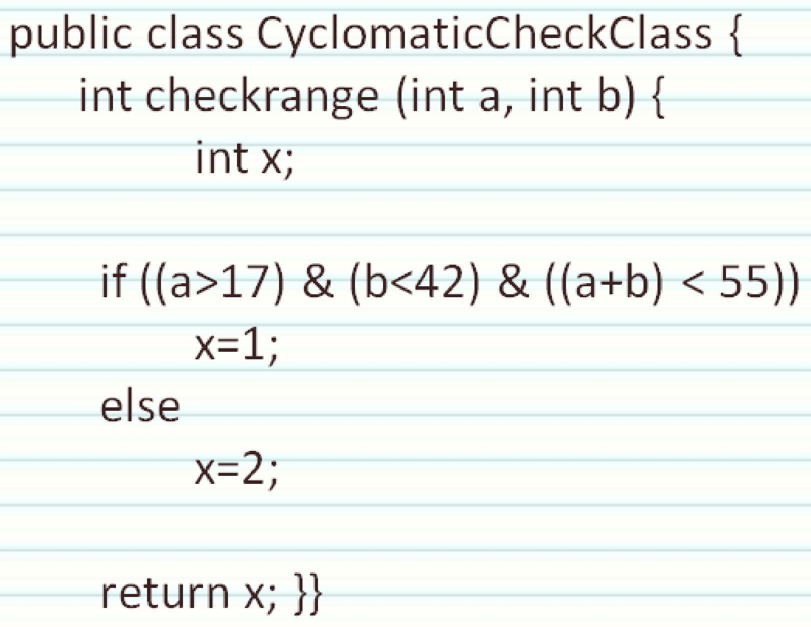
- For the following code:



```
public class CyclomaticCheckClass {
    int checkrange (int a, int b) {
            int x;

        if ((a>17) & (b<42) & ((a+b) < 55))
            x=1;
        else
            x=2;

        return x; }}
```

- Three different widely used tools reported different cyclomatic complexity values of 2, 4, and 5 - you need to double check all tools used.
- What answer is correct?
- This code does have a problem - what is it?

Infeasibility



```
public class CyclomaticCheckClass {
    int checkrange (int a, int b) {
            int x;

     if ((a>17) & (b<42) & ((a+b) < 55))
            x=1;
     else
            x=2;

     return x; }}
```

- MCDC test cases would be TTT, FTT, TFT, TTF
- I cannot cause the TFT to take place because it is infeasible - for example, a=18, b=42, 18+42=60 which cannot be <55
- Two solutions:
  1. Relax the test requirement and substitute TFF for TFT
  2. Re-write the algorithm as "if ((a>17) & ((a+b)<55)) ..."

- The latter is the best choice - it is testable and logically correct!

Testing tool classification

Requirements testing tools

Static analysis tools

Test design tools

Test data preparation tools

Test running tools - character-based, GUI

Comparison tools

Test harnesses and drivers

Performance test tools

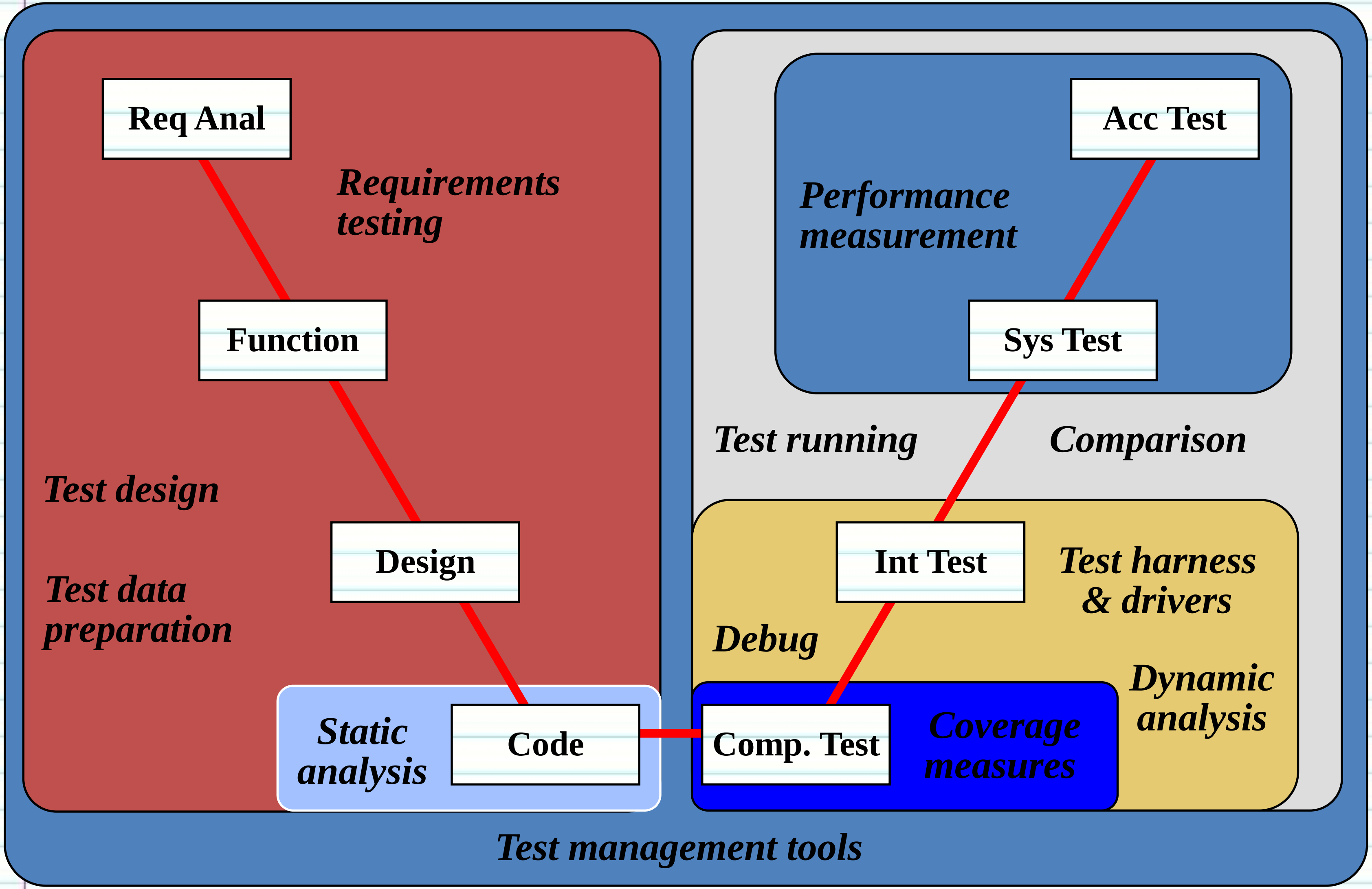
Dynamic analysis tools

Debugging tools

Test management tools

Coverage measurement

Where tools fit


Req Anal
Requirements testing
Function
Test design
Design
Test data preparation
Static analysis
Code
Acc Test
Performance measurement
Sys Test
Test running
Comparison
Int Test
Test harness & drivers
Debug
Dynamic analysis
Comp. Test
Coverage measures
Test management tools