Testing Code Coverage in Eclipse

Eclipse can check your code coverage when it runs your JUnit testing class. This means that it can show you what statements were executed in at least one test case and what ones weren't.

For an if-statement, it will tell you whether there was a test case for the condition to be false and another for it to be true.

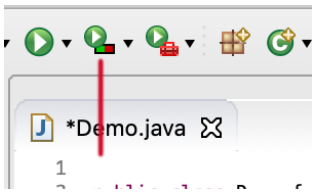
For a boolean expression with &&'s and ||'s, it will tell you whether there were adequate test cases for all possible ways of evaluating the expression.

When you do JUnit testing of any program, check your code coverage as explained below.

```
public class Demo {
   public static boolean m(int p) {
      if (p < 0) {
         p= 1;
         return p > 1;
      }
      return p == 1 || p == 9
}

   public static boolean mm() {
      return false;
   }
}
```

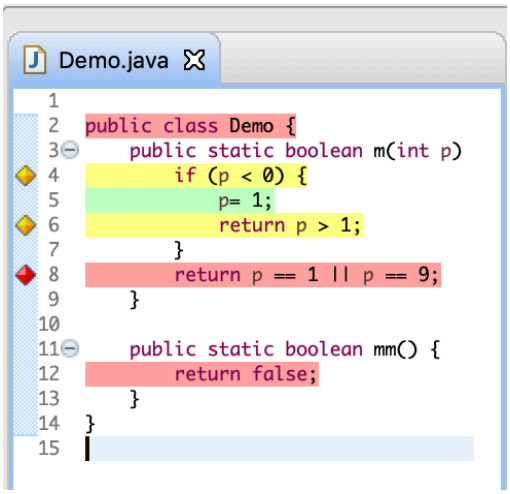
Running the JUnit testing class



To the far right is a JUnit testing class. To run this testing class and produce code-coverage information, look in the horizontal bar of icons for the one that is shown above the vertical red line in the image to the right. Hover your mouse over it, and a popup window will contain *Launch DemoTest* (*1*), since DemoTest is the JUnit testing class to be run.

```
class DemoTest {
   @Test
   public void test() {
      assertEquals(false, Demo.m(-1));
   }
}
```

Click that button, and class Demo will be changed to look as it does to the right. Note that the one test case, Demo.m(-1) was called. We explain the reasons for the highlighting.



- Line 4 is yellow, which indicates a problem. Hover your mouse over the yellow square to the left of line number 4; a window pops up with this in it: *1 of 2 branches missed*. Looking at the test case, we see that an evaluation of p < 0 gave the value true. But no test case found p < 0 to be false. This means code coverage is not complete.
- Line 5 is green, which indicates that it was executed in at least one test case. Code coverage is good.
- Line 6 is yellow. This is the same problem as in line 4. From the code, we know that p > 1 evaluated to false, and we need a test case in which it is true. But is that possible? We may have a logical error or a typo here.
- Line 8 is red. It hasn't been executed in a test case. Hover your mouse over the red square. A window pops up with the message: *All 4 branches missed*. We'll see what that means on the next page.
- Line 12 is red, but there is no red square to its left. All this means is that line 12 was not executed in a test case. We know why. No test case called method mm.
- When you are finished studying your code coverage in this way, you will want to get rid of all the color highlighting. To do that, just make any change in the file, for example, type a space character somewhere, and the highlighting will disappear.

Thus, Eclipse's *code coverage* feature helps us determine whether white-box testing was adequately done. Use this feature whenever you are testing a program. If it shows that your testing is inadequate, add more test cases.

On the next page, we have one more example to illustrate code coverage for boolean expressions.

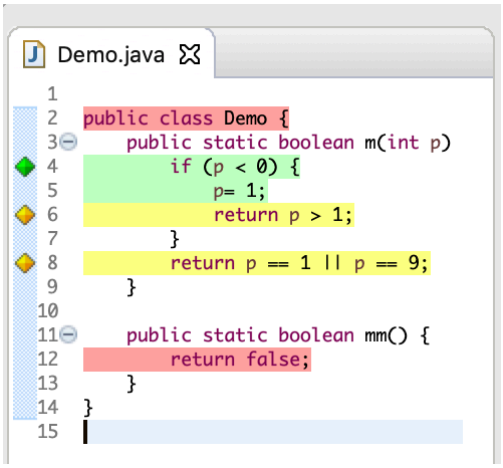
Testing Code Coverage in Eclipse

Code coverage for boolean expressions

In class DemoTest shown to the right, we added the call Demo.m(1). This will cause the if-condition on line 4 of class Demo to be evaluated once to true and once to false, so that line should end up green. This also means that the return statement on line 8 will be executed.

```
class DemoTest {
   @Test
   public void test() {
      assertEquals(false, Demo.m(-1));
      assertEquals(false, Demo.m(1));
   }
}
```

Now, running the testing class using the same button produces highlighting in class Demo as shown to the right below. We discuss it.



- Line 4 is green. The if-condition p < 0 was adequately covered.
- Line 5 is still green and line 6 still shows that the boolean expression p > 1 was not adequately covered. We expected that.
- Line 8 is yellow. Code coverage is not complete. Hovering the mouse over the yellow rectangle before the line number 8 brings up a window with this in it: *3 of 4 branches missed*. We missed three "branches"? Why? Well, there are *four* cases to consider, and our testing covered only one of them:

  p == 1 and p == 9
  p == 1 and p != 9
  p != 1 and p == 9
  p != 1 and p != 9

So, we have to put in more test cases.

You may decide that one case can't happen. For example, short circuit evaluation of x == 0 || 100/x = 2 never results in x being 0 and 100/x = 2 being evaluated and throwing an exception. So we *can't* write a test case to cover that. Obviously, Eclipse's code coverage feature doesn't take short circuit evaluation into account.

A word of caution

Having complete code coverage does not ensure that your program does not have errors! Your test cases might not have been judiciously chosen, and some execution sequence could still lead to an error. Corner or extreme cases might have been missed, for example. But you can have far more trust in a program that has been tested with complete code coverage than a program that has not.

About code coverage implementation

The software that is producing the code coverage information is looking not at the Java source program but at its machine language version —call the byte code. It has no information about types, for example. Therefore, it may not always do what you might expect, if you looked at the Java program itself.