Lab 8: Covering our Bases

Objectives

- More practice making test cases and test suites

Preliminary Setup

1. Create a new IntelliJ project.
2. Download the starter code from the Google Drive folder and put it in your project.

Have a look around

Now that you have a project and some code, explore to see what you have:

1. Skim the code in **TicTacToeBoard.java**. Notice that some methods are marked as private. What methods are public and which are private?
2. Take a look at **Main.java**. Notice the **Information Hiding**:
    - When **main** wants to create a tic tac toe board, it calls **TicTacToeBoard()**, which is a call to the constructor in the class **TicTacToeBoard**. So, **main** doesn't need to know how boards are stored; the constructor handles that.
    - When **main** wants to modify a board, it uses methods defined in the **TicTacToeBoard** class. So, **main** doesn't need to know how boards are stored; the methods in **TicTacToeBoard.java** handle that.
    - So, if we wanted to change the internal representation of boards, **main** would not need to change.

Look at some tests

Take a look at **TicTacToeBoardTests.java**, which uses the helper class **Tester** to test **TicTacToeBoard.getWinner()**. There are **three** tests in our *test suite* (i.e. set of tests), and for each test we have:

- A message indicating what we are testing.
- A call to **getWinner()** with a specific board as a parameter.
- The result we expect from **getWinner()** *assuming **getWinner()** is working correctly*

Notice that the board we pass to **getWinner()** is **hard-coded** in each of the tests. In regular code, you would not want to hard-code what could be computed, but in testing code, you want to hard-code, so that:

- You will know exactly what tests you ran.
- You will be able to re-run the exact tests if you ever make changes.
- For the specific hard-coded examples, you will **know the right answer** without having to write general code to figure it out. Remember, you are testing code that's supposed to figure it out in the general case; if you write new code to figure it out, you might make the same mistakes as were made in the code you're testing.

What makes a test suite *thorough*?

In the projects (and in some previous labs), you have tried to make your test suites *thorough* because you want confidence that your code that passes all of the tests in the test suite will also work for those situations you didn't test. One way to think about this is to consider each test case as somehow "covering" a set of scenarios:

- A test case tests a particular scenario:
    - Eg. A TicTacToeBoard initialized in a particular way, such as one where 'X' wins horizontally on the first row.
    - Eg. A particular parameter to a sine function, such as 90.
- There are several/many other scenarios in which the code behaves similarly to the way it does in the tested scenario:
    - Eg. Maybe other TicTacToeBoards where 'X' wins horizontally behave in a very similar way, follow the same logic in the code, and/or exercise the code in similar ways.
- The other cases where the code behaves similarly to the test case form an "equivalence" class:
    - If your code **fails any** of them, then it will also fail the test case.
    - If your code passes the test case, then it would also **pass all** of the others.
- We want to make our test suite such that all of the equivalence classes together represent the entire space of possible situations – then we have a *thorough* test suite. Because all of the space of possible situations is covered by the test suite, we would know that our code that passes the test suite would also pass for the un-tested scenarios.

Unfortunately, that's ***not generally possible*** to do. It's hard to know for sure that all the interesting scenarios are covered – i.e. it's hard to know for sure there isn't some situation that falls outside the union of all the equivalence classes defined by the test cases, in part because it's hard to know whether two situations cause the code to "behave similarly". But, even though this is hard to do, that's the goal. We want all of the situations covered.

Another approach

Often, you can use your *domain knowledge* (i.e. your knowledge of the real-world situations the program is modeling) to think of situations where the code might behave differently than it does for the tests already in the test suite. This is probably what you've done so far.

For this lab, we'll introduce another way to think about it. Note: this is **not better**. If you have domain knowledge, **you can and should still use it**. But sometimes, we have code we want to test and we don't have domain knowledge – perhaps we are testing code that someone else wrote. In such cases, we can still make thorough test suites by using an approach called **test coverage**. In this lab, we will use "line of code coverage": we say that a test "covers" a line of code if running that test causes that line of code to be executed. For a test suite, we say that the test suite covers a line of code if there's at least one test case in the test suite that covers the line of code.

So, in this lab, you will try to make a test suite that covers all of the lines of some particular methods in our code (note: you may use the tests I've given you or start from scratch – if you are going to start from scratch, be sure to remove my tests when you are done):

- Take a look at **getWinner()** and make sure you create tests so that all of the lines in **getWinner()** are covered.
- **getWinner()** calls the helper method **isWinner()**. See if you can create tests for **getWinner()** that cause all of the lines of **isWinner()** to be covered.

Try to make a test suite with **as few tests as possible**, but still covering all of the lines mentioned above. If your test suite has a lot more tests than needed, you will not earn full credit. **Extra credit** is available if your test suite is the minimum possible number of tests that meets our coverage criteria.

Note: this is just one of many possible **coverage metrics**, but it's a good start, especially if you don't have special domain-knowledge to be able to design test cases likely to uncover bugs. For this lab, I'd like you to pretend you don't really know tic tac toe, and use this coverage metric to design a "thorough" test suite.

How to turn in this lab

Before turning in any program in this class, remember this mantra:

> ***Just because it works doesn't mean it's good.***

Part of your grade will also come from things like how understandable and readable your code is. You will also be graded on the neatness, presentation, and style of your program code.

For all labs, turn in the lab via gradescope.

Ask for help if you're having problems!