SENG275 – Lab 4
Due Sunday Jun 4, 11:59 pm

Technical note: You should have created another repository on gitlab, named lab04

Welcome to Lab 4. Throughout this document, some sections will be in Red, and others will be in **bold**. The sections in Red will be marked. The questions in **bold** will not be graded. However, you should give these questions some thought, as these are the sorts of questions that could appear on a quiz or midterm.

Quick summary of what you need to do:

1. Test the three methods provided in the lab04 code on Brightspace.
2. Run the code coverage tool and achieve 100% line coverage.
3. Enable tracing and achieve 100% branch coverage
4. Generate a PITest report, and see if you can kill any mutants and further improve your tests.
5. Push the code in the lab04 repository on Gitlab.

You're done!

Motivations

Today we'll be doing coverage and mutation testing. Both of these techniques provide us with *metrics* – numerical values that hopefully correspond to the thoroughness and quality of our tests.

Not all metrics are useful. We could, for example, measure the quality of our tests by the number of tests that pass. There are problems with this approach - for example, the following test certainly passes:

```
AssertTrue(true);
```

Coverage tools allow us to address a big problem with testing as we've done it so far – it's really easy to write a poor or meaningless test that passes! Instead we'd like to evaluate how thoroughly our tests *cover* the code under test – do we manage to test every method? Every line? What about conditional statements – do we manage to test all the possible states of the machine after a conditional?

Load up your lab04 repo and take a look at the indexOf() method in the ArrayUtils class.

```java
public static int indexOf(final int[] array, final int valueToFind, int startIndex) {
   if (array == null) {
      return INDEX_NOT_FOUND;
   }
   if (startIndex < 0) {
      startIndex = 0;
   }
   for (int i = startIndex; i < array.length; i++) {
      if (valueToFind == array[i]) {
         return i;
      }
   }
   return INDEX_NOT_FOUND;
}
```

Given an array of integers and a target value, this method tells us the index of the first appearance of the target value, or -1 if it's not found.
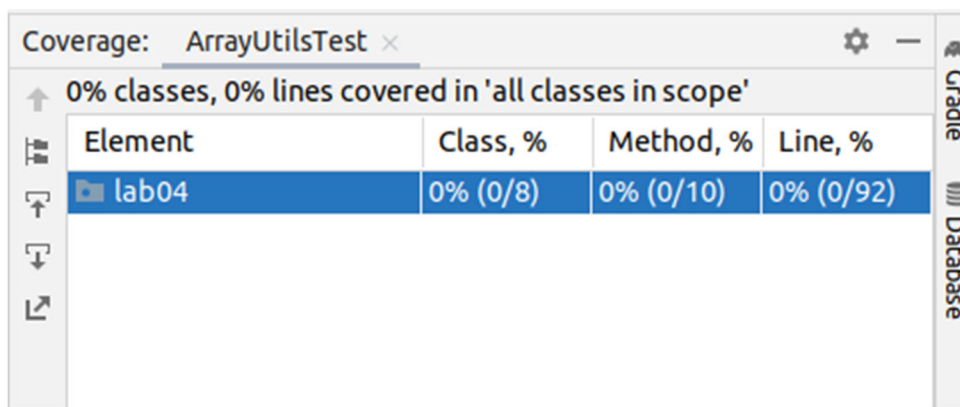
Run the tests in ArrayUtilsTest, and note all 13 tests pass. However, you may have noticed something's wrong – the actual test method in ArrayUtilsTest isn't actually calling the method indexOf(), or asserting the return value – both those lines have been commented out! Any test method that reaches its closing brace without throwing an assertion will pass. How could we have caught this problem?

Coverage testing is the solution. We need to be able to see how many classes and methods under test actually got executed when our tests were run, and which lines actually ran. We can do that with the Run... with Coverage feature of IntelliJ – click on the icon in the upper right

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

This pops out our basic coverage report, which is normally hidden in a tab along the right side
of the IDE. Now we can see something's gone wrong – no classes, methods or lines have been
executed.

Coverage: ArrayUtilsTest ×			
0% classes, 0% lines covered in 'all classes in scope'			
Element	Class, %	Method, %	Line, %
lab04	0% (0/8)	0% (0/10)	0% (0/92)

Uncomment the two lines in the test method and try the tests with coverage again.
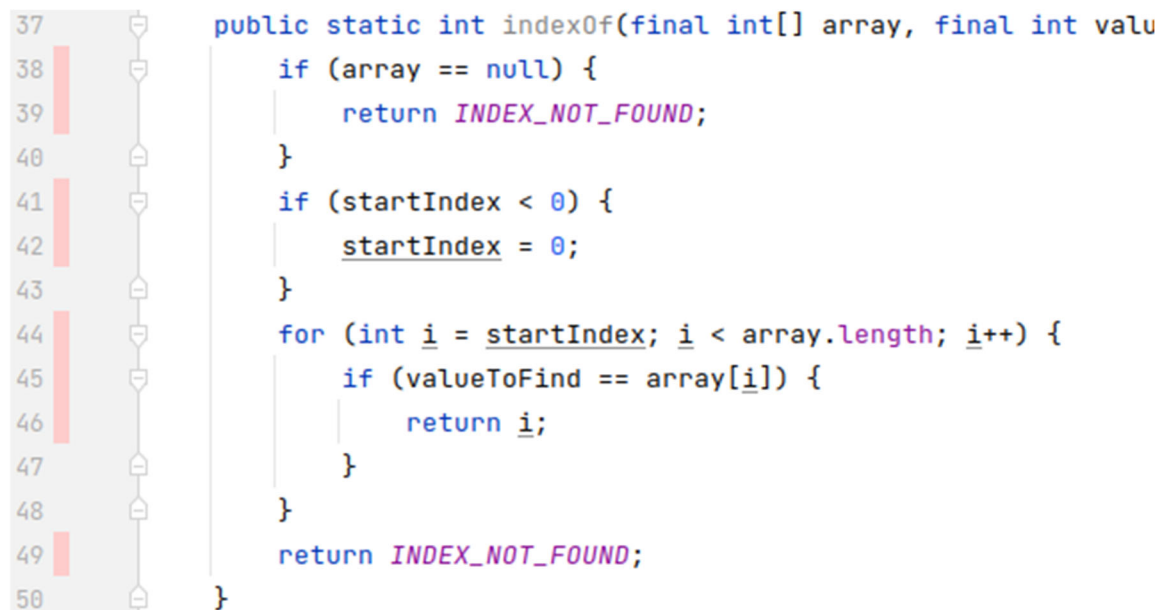
Class and Method coverage

For our purposes, Class and Method coverage aren't very useful beyond these basic warnings –
we're not testing that many classes or methods, so it's not easy to miss one. In a much larger
project, we might want to test everything, and it might be easier to detect something we
carelessly missed by looking at the numbers. By default, this display mixes test classes and
application classes – that's why the numbers of classes and methods are so high.

Line Coverage

After we Run with Coverage, we can see what lines were executed during the test – take a look
at ArrayUtils.java again. The gutter (the area to the left of the code next to the line numbers)
has now been annotated with coloured blocks – red means the line was not executed when the
last test batch was run, green means it was, and yellow stands for 'partially' executed – perhaps
a multi-conditional decision statement was short-circuited. (Yellow results can also result from

bugs in the coverage tool though – line coverage analysis works on the bytecode, so there isn't always a one-to-one correspondance between displayed and executed lines).

For example, before the comments were removed, the coverage report annotated the code like this:

```
37          public static int indexOf(final int[] array, final int valu
38              if (array == null) {
39                  return INDEX_NOT_FOUND;
40              }
41              if (startIndex < 0) {
42                  startIndex = 0;
43              }
44              for (int i = startIndex; i < array.length; i++) {
45                  if (valueToFind == array[i]) {
46                      return i;
47                  }
48              }
49              return INDEX_NOT_FOUND;
50          }
```

Once the comments are removed, the red annotations turn green, indicating the problem has been fixed. These annotations also appear by default in your test code as well.

Branch Coverage

Simple line coverage often isn't sufficient. If a line of code contains multiple conditionals or possible exit points (function calls that might throw exceptions are an example), then the fact that the line itself got executed doesn't tell you whether each possible *branch* of code that might follow that line were in turn executed. To get information about the number of branches that our code tests, we need enable branch coverage tracing in IntelliJ.

Open the Edit Configurations panel under the Run menu. Select the test configuration you're interested in – if you've been running multiple tests on different sections of the code, there may be several configurations. Click the Modify Options label on the right-hand side, and

select Use Tracing from the Code Coverage part of the drop-down menu. Click Ok and run your tests again. (This menu can also be used to exclude certain classes from your code coverage – useful if you're annoyed by the presence of test classes in your code coverage statistics.
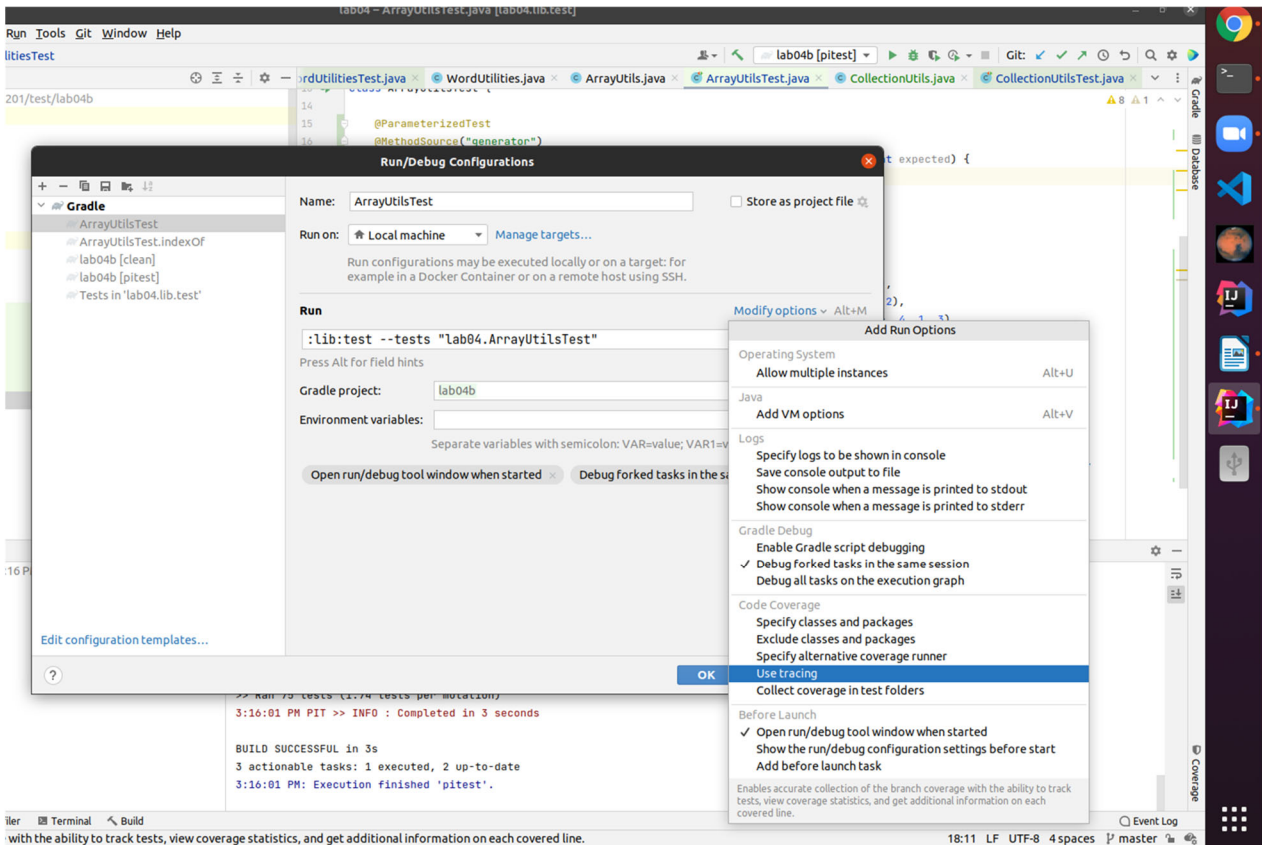
Now when you run a test suite with coverage, you'll see branch statistics in the coverage window as well.

overage: ArrayUtilsTest ×

25% classes, 17% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %	Branch, %
lab04	25% (2/8)	20% (2/10)	17% (16/92)	18% (8/44)

Now if you click on the coloured annotation square in the gutter beside a line of code that contains multiple conditionals, you'll be able to see how many tests exercised each conditional statement.


```
21        public static boolean canBalance(int[] nums)
22            if (nums == null || nums.length <= 1)
```

↑ ↓ ⬛ ⬛ 🔧 Hide coverage

```
Hits: 12
    nums == null
        true hits: 1
        false hits: 11
    nums == null || nums.length <= 1
        true hits: 3                          0; i++) {
        false hits: 8
```

Condition Coverage & MC/DC

To achieve full coverage of our code, we should make sure that our conditions are fully covered by applying MC/DC. Recall what we need to ensure:

1. Each entry and exit point is invoked

2. Each decision takes every possible outcome

3. Each condition in a decision takes every possible outcome

4. Each condition in a decision is shown to independently affect the outcome of the decision.

Line coverage can tell us whether we've achieved #1 and #2, and branch coverage can help us with #3, but full MC/DC analysis can only be done by tracing code manually and reasoning carefully about its behaviour.

Mutation Testing

Last week we did boundary analysis – we looked at ranges of inputs and tried to find critical values that distinguished code behaviour with precision.

One semi-automated way of judging whether or not we've done a good job at this is mutation testing. Suppose we've got some code, and a test that properly tests that code:

```
boolean ourCode(int x) {
   if (x > 10) return true;
   return false;
}
```

```
assertFalse(ourCode(10));
```

This test passes. Suppose we then change, or *mutate* our code – say we change the operator:

```
if (x >= 10) return true;
```

We should expect the test to fail! In fact it does, since it's a test that checked a well-chosen boundary value.

Suppose our test value hadn't been well-chosen:

```
assertFalse(ourCode(5));
```

This poorly-written test passes *both* tests – we can change the code we're testing, and our test doesn't even notice!

We can sum up the philosophy behind mutation testing as the following:

> **If the code we're testing changes, our tests should fail!**
> **If they all still pass, we need better tests.**

A mutation testing framework like PITest applies multiple *mutators* to the bytecode produced by the java bytecode compiler. Some change operators, some alter return values, some skip arguments, etc.

Then our tests are run, producing *mutants* – variations of our code, each slightly incorrect. If our tests fail, we say the mutant was *killed* (which is good). If our tests ignored the change and still pass, we say the mutant *survived* (which is bad). Our goal is to kill all the mutants – that is, write enough well-written tests that any change to the code we're testing will be caught.

PITest scans can be run through Gradle – we choose **Tasks -> Verification -> pitest** from the Gradle tab on the right edge of the screen. PITest output appears in two places – in the console output section, and as a series of web pages buried in **lib -> build -> reports -> pitest -> (date_time_stamp) -> index.html**.

Check out a typical report:

```
21          *           non-empty
22          * @since 2.1
23          */
24         public static boolean containsAny(final Collection<?> coll1, final Collection<?> coll2) {
25 2          if (coll1.size() < coll2.size()) {
26              for (final Object aColl1 : coll1) {
27 1                if (coll2.contains(aColl1)) {
28 1                    return true;
29                  }
30              }
31          } else {
32              for (final Object aColl2 : coll2) {
33 1                if (coll1.contains(aColl2)) {
34 1                    return true;
35                  }
36              }
37          }
38 1        return false;
39      }
40  }
```

Mutations

```
25   1. changed conditional boundary → SURVIVED
     2. negated conditional → SURVIVED
27   1. negated conditional → KILLED
28   1. replaced boolean return with false for lab04/CollectionUtils::containsAny → KILLED
33   1. negated conditional → KILLED
34   1. replaced boolean return with false for lab04/CollectionUtils::containsAny → KILLED
38   1. replaced boolean return with true for lab04/CollectionUtils::containsAny → KILLED
```

On line 25, we see one line that was mutated, and we see that it was changed in two different ways – the conditional boundary was changed (ie, the < was changed to a <=) and the conditional was negated (the < was changed to a >=). Nonetheless, none of the tests run on this particular code caught those changes, those mutants survived. To improve this code, we need to figure out how we could write a test case that would pass with the original code, but fail with these mutations.

It's important to note that PITest only bothers to mutate code that is covered by tests. If you haven't acheived 100% test coverage, PITest will ignore the untested portion of your code, which may give you false confidence. Provide full test coverage BEFORE using mutation testing!

Interpreting these reports is not easy, and killing the mutants that are revealed is even more difficult. Not all mutants that survive are meaningful, and it may be impossible to kill some. Experience and familiarity with the mutator rules can help, but a 100% score on PITest is often impossible to achieve.

The mutator rules are found at https://pitest.org/quickstart/mutators/ - use this as a detailed reference.

Your task:

1. Test the three methods provided in the lab04 code on Brightspace. The tests for ArrayUtils.indexOf() are already written for you as a guide.
2. Run the code coverage tool and achieve 100% line coverage.
3. Enable tracing and achieve 100% branch coverage. Remember you will have to access the run configuration through the Run → Edit Configurations menu.
4. Generate a PITest report, and see if you can kill any mutants and further improve your tests. It is not expected that you will be able to achieve a 100% killed mutant metric – that may not even be possible given the coding of a particular test or method. Just give it a try.
5. Push the code in the lab04 repository on Gitlab.