SENG275 – Lab 1
Due Wednesday Jan 27, 5:00 pm

Technical note: You now have another repository in your gitlab.csc.uvic.ca account, named Labs. Lab 1 -> readme.txt is the file you'll use to submit your short-answer question responses.

Welcome to Lab 1. 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:

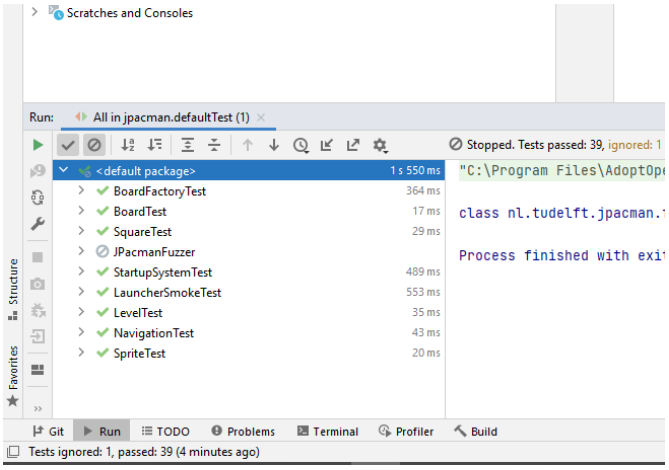
1. Complete the unit tests in the board.DirectionTest class in JPacman.
2. Complete the unit tests in the board.OccupantTest class in JPacman.
3. Push the completed tests back to your repository.
4. Clone your Labs repository from gitlab.
5. Complete the questions in Lab 1 -> readme.txt. (100 words max for each question) You must work independently on the questions in the readme.txt file – do not share your screen with readme.txt displayed.
6. Push the completed questions back to your repository.

You're done!

Running Tests

IntelliJ can run entire test suites or individual tests due its close integration with Gradle.

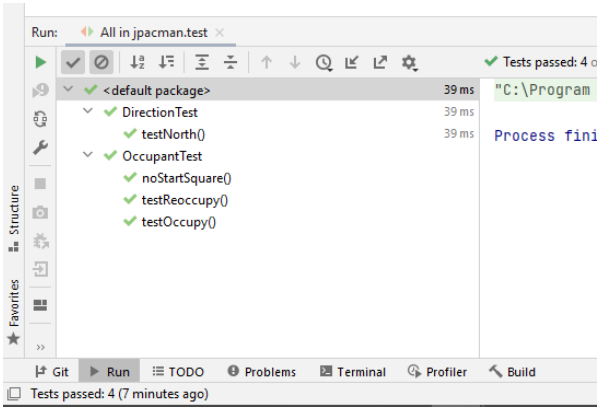
Several dozen tests have been written for you already – they are in the src -> default-test directory. To run these tests, right-click on the default-test directory, and choose "Run 'All Tests'" from the dropdown menu. The results of the tests will appear at the bottom left of the screen – 39 have passed, and one test has been disabled.



From this display, we can see each of the nine test classes in the built-in suite, some of which have multiple test methods.

Let's look at the set of tests we'll be writing during our work in the course.

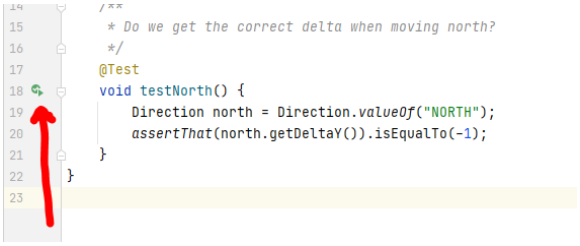
Run the four tests in the "Test" directory the same way you did above.

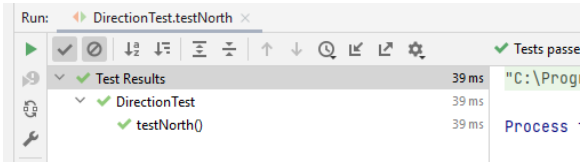


Again, the tests passed. Let's look at one particular test – double-click on the testNorth() method (or find the DirectionTest class somewhere below the test directory).

It is a very desireable feature of test that they run very quickly. That said, sometimes we would prefer to run an individual test, rather than the whole suite. Do so now for the testNorth()

function, by clocking on the green right-arrow-and-circle icon in the line number gutter on line 18.





Technical aside: Due to a problem with Gradle, some of you may be able to run the entire test suite, but unable to run individual tests. If this happens, you can switch from Gradle to IntelliJ by going to File -> Settings (or Preferences) -> Build, Execution, Deployment -> Build Tools -> Gradle. In the centre of the dialog that opens up, you should see two boxes – "Build and run using", and "Run tests using". Switch both to "IntelliJ IDEA" and you should be fine.

Why are the test classes named the way they are?

How many test methods should there be in each class?

Writing Tests

Look at the DirectionTest.java class, and note the package declaration at the top.

Let's look at the testNorth() test method.

```
12   */
13   public class DirectionTest {
14       /**
15        * Do we get the correct delta when moving north?
16        */
17       @Test
18       void testNorth() {
19           Direction north = Direction.valueOf("NORTH");
20           assertThat(north.getDeltaY()).isEqualTo(-1);
21       }
22   }
23
```

We're using the Junit testing framework, which recognizes test methods and allows them to run tests and see their results as shown on the previous page. To use this framework, we import from **org.junit.jupiter.api.Test**. In order to mark a method as a test method, we write **@Test** before the return type. The return type of any Junit test MUST be void.

The textbook (in "Software testing automation") also discusses the "**triple-A**" test structure – *Arrange, Action, Assert*.

Which line number above corresponds to each part of the "Triple-A" structure?

The Direction Class

We need to understand the Direction class before we can finish writing tests for it. We could go looking for it in the src directory, but an easier way is to click on Direction on line 19, right-click, and choose Go To -> Implementation.

Pac-Man and the ghosts ('Units', as JPacman calls them), are constrained to move in four directions only, on a 2-dimensional grid. These four directions are North, West, East and South. The designers of JPacman decided to abstract the movement rules somewhat, by creating the Direction enum. The idea was that instead of having to think in terms of x and y coordinates, a user of the enum (for example, someone writing a ghost AI) could just create a direction variable (say Direction.NORTH), and the implementation detail that NORTH is (0, -1) could be hidden.

Is the testNorth() test complete as-is? Is it well-named? Should the getter functions be tested?

Exercise 1. Complete the DirectionTest class. Your solution must test the functionality of the class; use the current testNorth() function as a guide.

Assertions

An assertion is a way to express "Some x must be true. If it isn't, end the program immediately". Confusingly, in Java, they come in (at least) three different types.

Built-in

The first is the basic, built-in assert statement.

For example, the statement:

```
assert x == y;
```

is just shorthand for writing:

```
if (x != y) {
      throw new RuntimeException();
}
```

Assert statements like these are DISABLED by default – they have no effect at all. They must be enabled by passing the -ea parameter to the java command on the command line in order to have effect. They throw unchecked AssertionError exceptions, so do not need to be declared with the **throws** statement, and should never be caught or handled in any way – their only role is to end the program. Note the lack of parentheses; assert is a statement, not a function (although putting in parentheses won't hurt).

This form of assert is not used to test software – it is used by the developers of software to enforce invariants. Find one of the assert statements in the Board class of the JPacman game itself (in main).

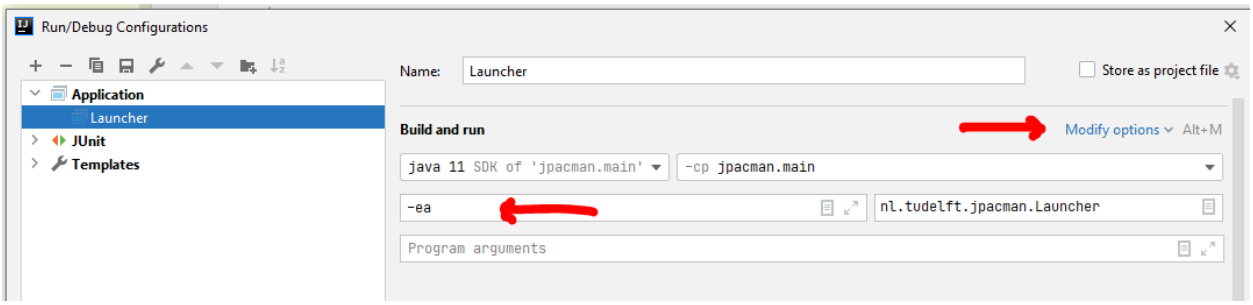
How is the assert statement on line 25 being used to enforce an invariant?

What is the invariant that the assert statement on line 76 enforcing?

There are assertions, exceptions, and invariants – three related, but different concepts. Explain in your own words the differences between these concepts, and how they are related.

Try modifying the Board class of JPacman to trigger an assertion. You'll need to add or modify a statement in the class so that it will always be false. Because assert statements are disabled by default, you'll also need to enable them, by going to Run -> Edit configurations. On the right, choose "Modify Options", then click "Add VM Options". In the new box that pops up on the left, add the switch "-ea" (without the quotes). Click OK and re-run the program. The game will immediately quit with an exception.

IF YOU DO THIS, REMEMBER TO CHANGE IT BACK BEFORE PUSHING YOUR CODE BACK.



AssertJ assertions

Let's return to DirectionTest.java. Note that the assertion on line 20 is an assertThat() function call, which looks like it returns an object that has an isEqualTo() method. Where did all this come from?

AssertJ is an extremely extensive and flexible library of assertions, and we'll only scratch the surface of it during this course. We imported one of its functions, assertThat(), up on line 3.

The basic structure of an AssertJ assertion looks like this:

assertThat(*some_object_or_value*).*has_some_relation_to*(*some_other_object_or_value*)

So the code on line 20 retrieves the DeltaY integer from 'north', then asserts that it is equal to negative one. If the assertion fails, assertThat throws an AssertionFailedError, with quite a lot of useful information:

The number of *has_some_relation_to* functions is immense; to get a feel for them, try typing 'assertThat("hello").' (note the period), and let the auto-complete function show you some of the relations available. Other assertJ assertions that will be useful in this course:

assertThat(*some_number*).isNotEqualTo(*some_other_number*);
assertThat(*some_object_reference*).isNotNull();
assertThat(*some_object*).isSameAs(*some_other_object_reference*);
assertThat(*some_condition*).isTrue();
assertThat(*some_condition*).isFalse();

Try modifying an assertJ assertion so that it fails, and note the output.

When thrown, what information does the assertJ assertion give us that the basic assert statement doesn't?

Junit Assertions

Junit provides an assertion class alongside its other testing framework capabilities. JPacman doesn't use it by default, but you are welcome to use it if you prefer; it is covered in the textbook.

To use these assertions, we'll need to add an import statement to the top of our DirectionTest.java file:

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

These assertions take the following forms:

assertEquals(*expected_value, actual_value*);
assertTrue(*some_expression*);
assertFalse(*some_expression*);
assertSame(*expected_object_reference, actual_object_reference*);

There are more; see the Junit 5 documentation for the full list. People have a lot of trouble remembering the "expected goes first, actual goes second" order, and getting them in the wrong order can produce some very confusing error messages. IntelliJ annotates the expected and actual arguments, so that helps, and Junit also provides useful information in its exception messages.

For example, the assertThat() statement on line 20 could be re-written using Junit as:

assertEquals(1, north.getDeltaY());

Choose whichever feels most comfortable for you – although AssertJ is more popular, both are in widespread use.

Is assertEquals() a good function to use to compare two strings to see if they have the same contents? Why or why not?

OccupantTest

The OccupantTest class in the test package has been prepared for you. The test functions noStartSquare(), testOccupy() and testReoccupy() need to be written. You may use any of the tools described in this lab to write the tests, and any Java you know. Remember that your tests should be as clear, and as simple as possible. Some hints before you proceed:

- Note that you have a Unit field in the class. The setup() function puts a BasicUnit into that field, and the @BeforeEach annotation guarentees that unit refers to a valid BasicUnit as each of your tests begins.
- What is a BasicUnit? Read the class; it's a Unit. What's a Unit? Again, check the Unit class, this time in the Board directory in main. What can Units do? It looks like they can occupy and leave Squares. What else can they do?
- What are Squares? Again, the code and documents will tell you. It looks like you can create them, and it looks like they can list their occupants.
- If a unit thinks it's occupied a square, then it stands to reason that the square should think that the unit is one of its occupants. Right?
- For testReoccupy, the idea is that if a unit enters a square, then it should know it's in the square, and the square should know it has the unit as an occupant. If the unit then leaves, and then re-enters the square, the same relations should hold.

Commit and Push back to Gitlab

Ensure that your tests pass, and then commit and push your JPacman repository back to the Gitlab remote. Choose Git -> Commit, and select the files in the default changelist that you've altered. Type a meaningful commit message in the "Commit Message" box, and click "Commit and Push". If you have not yet run git config, you may be asked for your name and e-mail address during this first commit.

Go to Gitlab and ensure that your changes are present in the remote repository.

We cannot mark your work if you do not push your changes back to gitlab.csc.uvic.ca.

Conceptual Questions

Clone your Labs repository from Gitlab. Inside the Lab 1 directory, you'll find a readme.txt file.

This file has three conceptual questions related to your textbook readings and lectures. Place your answers, in your own words, in the indicated spaces – use a maximum of 100 words for each answer. Save the file, and commit and push it back to gitlab.

If you're working at the command line, the following commands will help:

git commit -am "some meaningful message about this commit"

git push

For reference, the questions are as follows:

1. Why can't we exhaustively test our entire software project? What should we do instead?
2. What is the pesticide paradox about and what does it imply to software testers?
3. Why should we automate, as much as possible, the test execution?