The University of Melbourne
School of Computing and Information Systems

COMP90041

Programming and Software Development

Second Semester, 2019

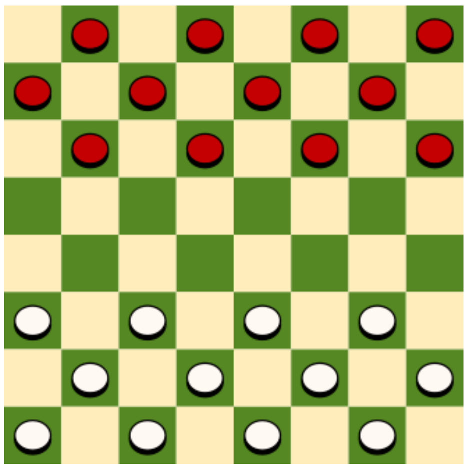
Third Assessed Exercise (lab3)

Submission due Friday, 13 September 2019, 5:00PM

These exercises are to be assessed, and so **must be done by you alone**. Sophisticated similarity checking software will be used to look for students whose submissions are similar to one another.

For this project, we will turn the tables. Instead of you writing code to implement a specification and me testing it, I will give you an implementation, and ask you to test it. You will be assessed on how thoroughly you test that the implementation meets its specification.

The class you will be given, `Checker`, represents a single checker (from the board game *checkers*, also known as *draughts*). Each checker is either red or white, and has a row and column (both between 1 and 8) on the checker board. See the illustration below.



The rows and columns of the board are numbered from 1 to 8 (left to right and bottom to top). Note that all checkers are on the dark squares, that is, the odd numbered squares on odd numbered rows and even numbered squares on even numbered rows. Since checkers can only move diagonally, this is holds throughout the game.

The `Checker` Class

The `Checker` class will have the following public methods:

`Checker(boolean isRed)` (constructor) creates a checker at row 1 column 1. If `isRed` is true, the checker is red, otherwise it's white.

`Checker(boolean isRed, int row, int column)`
(constructor) creates a checker at the specified row and column. If `isRed` is true, the checker is red, otherwise it's white.

`void move(int rows, int columns)` adds `rows` to the checker's row and `columns` to its column.

`boolean isRed()` returns `true` if the checker is red, and `false` otherwise.

`int getRow()` returns the checker's current row.

`int getColumn()` returns the checker's current column.

Furthermore, all methods must obey the rules of checkers. In particular, the row or column of a checker must always be in the range of 1 to 8 inclusive, and must remain so after a move. Also, checkers may only ever be placed on odd numbered columns in odd numbered rows, and only on even numbered columns on even numbered rows.

For the `move` method, the specified numbers of rows and columns to move should each be either 1 or -1, and the specified number of rows to move must be negative for red checkers and positive for white ones. If the specified move violates these requirements, the method should do nothing at all (later we will learn the proper way to handle invalid messages, but for this project, we just ignore them). If a constructor would place the checker in an invalid square, it should be placed in row 1, column 1 instead.

As mentioned above, you do not need to implement the `Checker` class; an implementation will be provided.

The `CheckerChecker` Class

You must implement a class called `CheckerChecker` with a `main` method that will thoroughly test that the `Checker` class implementation is correct according to the above specification. If the conclusion of your testing is that the `Checker` class works flawlessly, your program should simply print `CORRECT` as a single line; if you testing detects any bugs, it should print `BUG` as a single line. Do not print anything else.

I will test your code with a correct version of the `Checker` class, as well as with `Checker` classes exhibiting ten different bugs. Your score on this project will be the percentage of the bugs your testing program detects. However, if your program reports a bug for the correct version of the `Checker` class, you will receive zero for that submission. The `Checker` class I will give you is correct, so this should not be difficult to avoid.

Your program will need to create some checkers, manipulate them in various ways, and use the accessor methods to determine if the `Checker` class behaves correctly.

You will be supplied with a `Checker.class` file, but no `Checker.java` file. Put the `Checker.class` file in the same directory as your `CheckerChecker.java` file, compile with `javac CheckerChecker.java`, and run with `java CheckerChecker`.

The feedback you will be provided when you submit will give one line for the correct `Checker` class, and one for each buggy version. At the end of testing, a summary will show what percentage of the bugs your suite was able to find, and how many points you have earned.

Hint

It is tempting to test by creating one checker and performing many tests on it. This makes it more difficult to maintain the testing code by adding or removing tests. In the long run, it is easier to create a fresh checker for each test.

Using the provided `Checker.class` file

To use the provided `Checker.class` file in your IDE, you need to tell it where to find your "additional class files". Begin by creating an empty folder called `classes` in some convenient location.

Then for Eclipse, right-click on your project, select *Build Path* → *Configure Build Path* → *Libraries*, then choose *Add Class Folder* and select the folder you just created.

For Netbeans, right-click on the *Libraries* item within your project and select *Add Library...*. Click the *Create* button in the dialog, give the library a name, such as `CheckerLibrary`, and click *OK*. Click the *Add Jar/Folder* button in the dialog that appears, select the folder where you put the `Checker.class` file, and click *OK*, and *OK* again, and then click *Add Library*.

Now you should be able to use the `Checker` class in the usual way in your code in that project.

Submission and Verification

You must submit your project from any one of the student unix servers. Make sure the version of your program source files you wish to submit is on these machines (your files are shared between all of them, so any one will do), then `cd` to the directory holding your source code and issue the command:

```
submit COMP90041 lab3 CheckerChecker.java
```

Important: you must wait a minute or two (or more if the servers are busy) after submitting, and then issue the command

```
verify COMP90041 lab3 | less
```

This will show you the test results and the marks from your submission, as well as the file(s) you submitted.

If your output is different from the expected (correct) output, when you verify your submission you will see the differences between your output and what was expected. This will be shown as some number of lines beginning with a minus sign (`-`) indicating the expected output and some number of lines beginning with a plus sign (`+`) presenting your actual output. There may also be some lines beginning with a single space showing lines you produced that were as expected. Carefully compare the expected and actual lines, and you should be able to find the error in your output. The actual and expected outputs will be aligned, making it easier to find the differences. Some differences are hard to spot visually, however, such as the difference between a capital `O` and a zero (`0`) or the difference between a small `l` and a capital `I` and a one (`1`). This depends on the font you are using. If the only difference between actual and expected output are in whitespace or capitalisation, you will receive partial credit; this is shown in your verification feedback.

Also note that the differences shown only reflect program *output*, not input. Therefore, if your program outputs a prompt, waits for input, and then outputs something else, the differences shown will not include the input, or even the newline the user types to end the input. In that case, the prompt would be shown immediately followed by your program's next output (which may be another prompt), on the same line. This is as expected.

If your program compiles on your computer but the verification output reports that your program does not compile on the server, you may have failed to submit all your files, or you may have named them incorrectly. It is also possible that your program contains a `package` declaration. This would appear near the top of your .java file. If you have such a declaration, your program will probably not compile, so you should delete any such declaration before submitting.

If the verification results show any problems, you may correct them and submit again, as often as you like; only your final submission will be assessed. If your submission involves multiple files, you must submit *all* the files every time you submit.

If you wish to (re-)submit after the project deadline, you may do so by adding "`.late`" to the end of the project name (*i.e.,* `lab3.late`) in the `submit` and `verify` commands. But note that a penalty, described below, will apply to late submissions, so you should weigh the points you will lose for a late submission against the points you expect to gain by revising your program and submitting again. **It is your responsibility to verify your submission.**

Late Penalties

Late submissions will incur a penalty of 1% of the possible value of that submission per hour late, including evening and weekend hours. This means that a perfect project that is a little more than 2 days late will lose half the marks. These lab exercises are frequent and of low point value, and your lowest lab mark will be dropped. Except in unusual circumstances, I will not grant extensions for lab submissions.

Academic Honesty

This lab submission is part of your final assessment, so cheating is not acceptable. Any form of material exchange between students, whether written, electronic or any other medium, is considered cheating, and so is the soliciting of help from electronic newsgroups. Providing undue assistance is considered as serious as receiving it, and in the case of similarities that indicate exchange of more than basic ideas, formal disciplinary action will be taken for all involved parties.