

APS 105 – Computer Fundamentals

Lab #4: Arrays

Fall 2000

You must submit this lab by 11:59pm Wednesday, October 18. Yes, this is a 2 week lab!

1 Objective

To gain experience with the use and manipulation of arrays. You will also use more methods and practise more thorough testing. This is a longer program than your previous labs, so be sure to start it early.

2 The Game of Checkers

For this lab, you are to write a program which permits two players to play a *simplified* game of *checkers*. The game will be played using *red* and *white* game pieces on an 8 by 8 square checkerboard, initially set up according to the following diagram.

```
Moves: 0
Score: Red 12 White 12

  0 1 2 3 4 5 6 7
B: 0 r r r r r 0
B: 1 r r r r r 1
B: 2 r r r r r 2
B: 3 . . . . . 3
B: 4 . . . . . 4
B: 5 w w w w w 5
B: 6 w w w w w 6
B: 7 w w w w w 7
  0 1 2 3 4 5 6 7
```

The squares on the board are identified using **row,column** coordinates, with the **row indicated first** and then followed by the column. There are coordinates printed along all sides of the board to help you determine the location of a game piece. The game pieces at 2,6, 1,7, 0,0 are all red. The piece at 7,1 is white. The TOP LEFT corner is at location 0,0 and the BOTTOM RIGHT corner is at 7,7.

The squares contain a single character to indicate the state of the game board:

- 'r' red player piece, or 'R' for a king
- 'w' white player piece, or 'W' for a king

1

- '.' an empty dark square where pieces can be placed

- '.' (a space character) an empty light square, no pieces can ever be placed here

Notice the dark squares are always on *even* values of `row + column`. This always gives you a dark square on the player's own **right** corner. Game pieces are only played on the dark squares.

It is important that you print out your checker board in *exactly* the same fashion shown above. The "B : " must be placed at the beginning of the line, and the row, and column numbers must be printed in the same way. Be sure to get the extra spacing right between the columns too (a single space character). At the top of every board is a move count, which goes up by one after every player move. As well, a score is printed, showing how many red and white pieces *remain* on the board (kings count as 2, see below).

The red player begins the game by sliding a red piece one position, to an empty dark square (marked by a period '.'). The moving of the game piece uncovers a new empty dark square in the old position. No game piece should ever move onto a white square (marked by a space character ' '). After that, players alternate turns, moving their pieces across the board. When entering a move, four values will always be required. Check the validity of the coordinates (and of the move) *after* accepting all four values. If the move is not valid, ask the user to try again after printing the following string:

```
"ERROR invalid move"
```

In order to keep this lab *simple* and markable, your program must enforce the following rules (and no more!):

1. A move is indicated by a beginning and ending coordinate pair.
2. Regular pieces can only make two types of moves:
 - (a) A piece can *slide* to a diagonally adjacent, empty dark square.
 - (b) A piece can *jump* along the diagonal, over one opponent's piece, into the adjacent empty dark square.
3. A player can only move his own pieces, and can only jump over an opponent's piece (not his own).
4. A piece which has been jumped over is removed from the board.
5. No multiple jumps.
6. No forced jumps. However, if you are ambitious, your program may attempt to look for forced jumps and print the message, "There is a forced jump" for the proper player before each turn.
7. Moves can only be made in the forward direction, towards the side opposite the player's starting position.
8. If a piece makes it to the most distant row, it becomes a *king*. Indicate that it is a king by a capital letter, 'R' or 'W'.
9. King pieces cannot move. Leave them on the board in their resting place.
10. A player's score is computed as 1 point per regular piece and 2 points per king piece (of their own colour).
11. The game is over when:

2

- (a) Any player establishes 4 kings. The winner is the one with the highest score (not necessarily with the most kings).
- (b) A player cannot make any moves. Your program should not try to detect this case. Instead, let the user enter a move consisting of four -1 values. Again, the winner is the one with the highest score.

When the game is over, you must print the score and final board and determine the winner. Be sure to print the name of the winner, using one of the following strings:

```
"Red player wins"
"White player wins"
"Both players win"
```

You might find it fun to continue and add more of the basic checkers rules (multiple jumps, kings moving backwards, forced jumps, and so on), but you must not submit a program with these extra rules. Doing so will certainly result in a poor mark, despite your extra efforts. At a later time, such as during your project, you may be interested in rewriting this game as an applet to use graphics or even include a computer-based player. At that point, you might be able to use the extra rules you experiment with now.

3 Program Skeleton

To assist you in structuring your program, you **must** use the following file as a starting point:

<http://www.cerf.uconn.edu/aps105w/labs/4/Lab4.java>

(also: /share/copy/aps105/Lab4.java)

This file contains a set of *stub* methods that you can complete to obtain a solution to the lab. They are called stubs because they contain the smallest piece of dummy code possible to get the program to compile without syntax errors. You will replace the stubs with real code later. Work on one method at a time.

If you desire, you may add additional methods to the class. However, **do not** modify, add, or delete any parameters to the given methods. As well, do not delete any of the given methods.

The code we are giving you is actually a code specification that you must adhere to. This specification is often called the **API**, or Application Programming Interface, to your program.

By writing a clear API specification, we can write our own test program and actually call the methods in your program. We can send specific input into your methods, and then check the result it gives back. This type of testing is often called black box testing, because your code is being treated as a magical "black box" that we cannot look inside. Instead, the black box tester sends input and examines the output.

4 Easier Way to Run Tests

```
Moves: 0
Score: Red 12 White 12

  0 1 2 3 4 5 6 7
B: 0 r r r r r 0
B: 1 r r r r r 1
B: 2 r r r r r 2
B: 3 . . . . . 3
B: 4 . . . . . 4
```

```
B: 5 w w w w w 5
B: 6 w w w w w 6
B: 7 w w w w w 7
  0 1 2 3 4 5 6 7
```

When testing your program, you will find it useful to exploit the operating system's *pipe* facility. A pipeline connects the output of one program to become the input of another. By using the `cat` command, you can print the contents of a file to the screen. Using a pipe, represented with the symbol `|`, you can send the output of `cat` into your Java application instead, as if you had typed it at the keyboard yourself. For example, suppose you create the file `moves.Long` (using your editor) with the following contents:

```
2
6
3
7
-1
-1
-1
-1
```

You can then use this file to play the game with the command:

```
computer.ecf% cat moves.Long | java Checkers
```

Note that the methods of the `StdIn` class read a complete line every time they are called, therefore requiring each individual input to be on a separate line. The example file contains two different moves: the first move source 2,6, the destination 3,7; and the game ending move source -1,-1 and destination -1,-1. The final board would look like this:

```
Moves: 1
Score: Red 12 White 12

  0 1 2 3 4 5 6 7
B: 0 r r r r r 0
B: 1 r r r r r 1
B: 2 r r r r r 2
B: 3 . . . . . 3
B: 4 . . . . . 4
B: 5 w w w w w 5
B: 6 w w w w w 6
B: 7 w w w w w 7
  0 1 2 3 4 5 6 7
```

This input format can be a tedious and lengthy file which is difficult to edit, especially when you fall asleep at the keyboard and accidentally insert one extra number, misaligning everything. It is much better to create your test moves in an easier-to-read file first, say `moves.easy`, then convert it to the longer file `moves.Long` using the command given below. The `moves.easy` file, and the game board after these moves, are shown below:

```

0 0 0 0 This is an invalid move. I used this move to
0 0 0 0 demonstrate what comments look like in this file.
0 0 0 0 Comments begin after the fourth 'space' on the line,
0 0 0 0 and go to the end of a line.
0 0 0 0
2 6 3 7 First move (red player always moves first)
3 7 2 6 white move error -- cannot move red piece (also a backward move!)
3 7 4 6 white move error -- cannot move red piece forward
5 7 4 6 white move
2 6 3 7 red move error -- no red piece there
2 4 3 5 red move
4 6 2 4 white jumps over red, capture!
1 7 2 6 red move, does not do the "forced jump" move (1,3 to 3,5).
-1 -1 -1 -1 all done!

```

The final game board after these moves looks like this:

```

Moves: 5
Score: Red 11 White 12

0 1 2 3 4 5 6 7
B: 0 r r x r x 0
B: 1 r r x r . 1
B: 2 r r w r x 2
B: 3 . . . x 3
B: 4 . . . . 4
B: 5 w w w . 5
B: 6 w w w w 6
B: 7 w w w w 7

0 1 2 3 4 5 6 7

```

```

Final Score
Red player: 11
White player: 12

```

White player wins

Here are the commands to convert this format to the longer moves. long, and then use that file as input to your program:

```

computer.ecf% awk ' { print $1; print $2; print $3; print $4 } ' < moves.easy > moves.long
computer.ecf% cat moves.long | java Checkers | less

```

In the command above, we also show the command 'less' being used to slow up the output of your program, one screenful at a time. You can just hit the space bar to go to the next page of output, or press the 'u' key to scroll back one page. Pressing 'q' quits.

Also, note that Java does not display the input when it is received through a pipe. Instead, you may find it useful to add an extra `println` statement to print the user move before printing the updated game board.

5 Testing Requirements

As part of writing a larger program, you will find that thorough testing is a very important part of getting it to work. It is important that you test all corner cases in your code, because if they aren't tested there is likely a bug hidden in there.

There should probably be at least one (even minor) test case for every `if()`, `else`, `while()`, and `for()` word in your program. If you use `&&` or `||`, you are likely adding more test cases. Major test cases should be created to verify how these things fit together.

To ensure that you formalize your testing, you must submit two test cases named `test0.easy` and `test1.easy`:

- `test0.easy` must contain at least 5 corner case tests for invalid input, interspersed with actual game moves, and must terminate before completing an entire game. As shown in the example above, please comment which moves are invalid input and why they are invalid.
- `test1.easy` must contain a complete game ending by the creation of 4 kings. Comment all jumps and moves that create kings, clearly indicating who is jumping or being kinged.

In addition to these, you should write many more test cases to satisfy yourself that your program is working correctly.

To cultivate good programming habits, **for this lab only** we encourage you to share your test cases with your friends. This will give you more test cases than you can generate on your own, but it is not intended to replace your own careful thinking of good test cases. Although you are sharing them, **the test cases you submit must be your own unique work** — do not confuse your test cases with those of your friend. Please note that their test cases may not properly or adequately test your checker's program.

Again, as in all labs, everything you submit *must be your own work*, including: `test0.easy`, `test1.easy`, and `Checkers.java`.

6 Software Engineering

This section contains useful information about testing and how it relates to engineering as a profession, but it is not a specific component of your lab.

One reason that a large program like Windows crashes so often is because the programmers focus too much time on "adding new features" rather than extensively testing corner cases. This is not sound software engineering, and you must not begin to fall into the same chasm. You must test every piece of code you write. By doing incremental testing as you go along, you verify that the little pieces are working correctly. When you place them all together, you will have greater confidence that the program will work as a whole.

There is no money in fixing software bugs, only pride and marks!!!
 You are studying **engineering** now, and one principle of engineering is that you will be held accountable for mistakes in your work. You must also maintain high ethical standards, which goes beyond basic law to state that you must not knowingly, through your action or inaction, cause public harm or not protect the public interest. Between 1985 and 1987, four cancer patients died while undergoing treatment due to lethal overdoses of radiation caused by a software bug. Do not write buggy software! To read more about the dangerous consequences of some software bugs, see: <http://coverage.cnet.com/Content/Features/DiHe/Bugs/ss02.html>

The concept of accountability for your engineering work comes from the legal system of tort law, meaning somebody can sue you for negligence or breach of contract. Unfortunately, *software engineering* is not yet a formally recognized engineering discipline by the Professional Engineers of Ontario (PEO), the governing body of engineering in this province. However, you can only call yourself an engineer if you are reg-

istered as a professional engineer (PEng) with them. Here is their website: <http://www.peo.on.ca/>. You will find a number of links to your legal obligations as an engineer here <http://www.peo.on.ca/EngPractice/guidelines.htm>.

To sum it all up, although most software companies make a lot of money selling buggy programs, as an engineer you may find yourself financially responsible when someone begins legal action to sue you, or ethically bound to do the right thing. There is active work to try and include *software engineering* as a recognized part of the PEO and other professional engineering governing bodies.

7 FAQ

This section will be expanded as questions are asked on the newsgroup. The body of the lab will remain constant, unless there are grave errors to be fixed (and noted here).

1. Is the game piece at 2,6 really red?
Yes. We are not using (x,y) coordinates but (row,col). I have added a few more examples in the body of the text to clarify.
2. Do I redraw the board at the top of the screen every turn?
Just let it scroll. Nobody knows how to clear the screen in Java, and doing so would only make autotesting harder.
3. Is the red score calculated by pieces remaining, or the number of white pieces it jumped over?
You score 1 point per regular piece, and 2 points per king piece of your own colour remaining on the board.
4. When will the stub methods be up?
The stub methods are now online.
5. Why does the board look so funny?
Egads! I had the board backwards (dark and light). I had to update all the examples for the new board. Note that dark squares are now on even row+col sums.
6. What is a "valid position"?
A valid position, or isDarkValidSquare(), is any black square on the 8 x 8 checkerboard. The methods isMySquare() and isOpponentSquare() should not assume they are given valid positions. They will only return true if the square is occupied by the proper piece.
7. How do I print the checkerboard properly?
Print one row at a time. In a row, when you go from one square (one column) to the next, print one extra space. Since a light square also appears as a space, a total of 3 spaces will separate black squares. Do not print any blank lines between rows.
8. Do we have to add methods to the ones given in the lab?
I had quite a few more methods in my solution. You are not required to add more, but I'm sure it will be better if you do.
9. Do we include the header (the 0 - 7) in the newCheckerBoard or the printBoard method? or is it up to us?
No, store only the checkerboard squares in an 8x8 array. Please do not store the characters around it. Whether you print the header inside printBoard() or not is up to you, but it makes most sense here – that's what the moves parameter is for.

10. Can we remove/change the comments in the skeleton program?
Yes, you can replace them with your own.

11. How can makeMove() change the board, even though the return type is void?
Any method can modify the contents of an array passed as a parameter, and these modifications will persist even after leaving the method. Java does not pass a "copy of the array" to a method, it sends a "copy of the reference to an array", which is a different beast.

12. The methods isSlideMove() and isJumpMove() don't have the current player or board as parameters. How can they check if it's a valid move?
You should only check that the coordinate pairs can represent a slide or jump move on the checkerboard (independent of exactly where the current pieces are). Verifying that the source has the correct player piece on it, for example, can be done elsewhere.

13. What's an ArrayIndexOutOfBoundsException error?

This is a runtime error your program has made when accessing an array element. Specifically, the array index used is too small (i.e., negative) or too large (beyond the end of the array). The error message should give you an idea where the error occurred (a list of methods called, perhaps the line numbers). To debug, you can try printing the array index values before you access the array. This error isn't too hard to fix, once you figure out where the error is happening.

14. How do I use computeScore() because printBoard() doesn't have the player as a parameter!
You must call computeScore() twice, once for each player. No, don't print the score in computeScore(), just return the score.

15. Do I print Moves : 5 or Move : 5?
Use the first one.

16. How often is the move incremented?
Increment it once after every move (either red or white). After two moves, red and white have each had one turn.

17. How often do I print the board?
Print it after every move is made. Do not print it if an incorrect move is entered.

18. How do I print the board with the right spacing?
The printBoard() code was added to share/copy/ups105/Lab4.java for you.

19. Can we print out extra stuff?
Yes! If you do print extra stuff, place it on its own line. Our program will only be looking for the special lines we told you to print.

8 What to submit

You must submit your Checkers.java, StdIn.java, and your 2 testing files, test0.easy and test1.easy. **Please submit your easy-to-read test cases. We will convert them.** If you wrote any other classes, place them in their own .java file and submit them as well.

computer.ecf% submitaps105f 4 Checkers.java StdIn.java test0.easy test1.easy