

## CSCI 2110 Data Structures and Algorithms

### Assignment 1

Release Date: Sept 15 2020  
Due: Sept 29  
23h55 (5 minutes to midnight)

Welcome to your first assignment. The problem statement for the assignment is long, but the requirements are specified in detail. Develop your solution class-by-class, module-by-module, testing at every stage. Remember to test your code frequently. Follow all class, method, and command input directions exactly as they are given in this document. Before you start coding, consider drawing out a plan for how your classes will interact and what data structures you will use.

**Marking Scheme:** Throughout this course, we will be focusing on the complexity of algorithms and consequently, the efficiency of programs. In past CS courses your focus was on creating runnable code to specifications. This course builds on previous knowledge, but also focuses on efficiency. This will involve reducing unnecessary coding and designing or choosing good algorithms.

- Working code, Outputs included, Efficient, Good basic comments included: **40/40**
- Required classes missing or not functioning as specified: **subtract up to 14 points**
- Comments and documentation missing: **subtract up to 3 points**
- Code is not efficient: **subtract up to 7 points**
- Required test cases not given in text file: **subtract up to 14 points**
- Assignment submission and test case text files are not well labeled and organized: **subtract up to 2 points.**

For this assignment, you will write a series of programs to simulate movement in a simple board game. Most board games have a variety of pieces that move across the game board according to a set of rules about how far and in which direction they may move (think chess). After defining a series of pieces, you will set up the infrastructure for a board game and demonstrate its operation.

There will be a total of 7 required Java classes created for this assignment: *Piece.java*, *SlowPiece.java*, *FastPiece.java*, *SlowFlexible.java*, *FastFlexible.java*, *Board.java* and *GameDemo.java*. You are free to create extra classes if needed.

Start by defining a **Piece** class with the following fields:

- Name: A String (eg. 'Jedi', 'Droid', etc.)
- Colour: A String (eg. 'Black', 'Red', etc.)
- Position: A point described by a pair of integers (eg. [2, 4], [1, 1], etc.).

Assume that the game board is an 8 X 8 grid. The position is a coordinate ([x, y]) on the board, with the values of x and y each taking a value between 0 and 7. The top left corner of the board is taken to be the origin ([0, 0]). You can define Position in a separate class or use classes from the Java standard library if you wish.

Implement the following methods:

- Constructor: A constructor that creates a Piece object with the given name, colour and position.
- Getters and Setters: Appropriate methods to set, change and access fields.
- toString(): A method to display the Piece Object's fields.

Next define a **SlowPiece** class – A Piece that moves left or right exactly one step per move. Your SlowPiece should extend the Piece class, implementing a constructor and redefining the inherited toString method via overriding.

In addition, implement the following method:

- move (String direction): A method to move the piece left or right by one space. This method will accept a direction parameter. If a move is requested that would take the Piece off the edge of the board, simply return without doing anything.

Define a **FastPiece** class – A Piece that moves left or right a specified, arbitrary number of steps per move. Your FastPiece should extend the Piece class, implementing a constructor and redefining the inherited toString method via overriding.

In addition, implement the following method:

- move (String direction, int n): A method to move the piece left or right by a specified number of spaces. This method will accept a direction parameter and a number of spaces. If a move is requested that would take the Piece off the edge of the board, simply return without doing anything.

A flexible Piece is one that can move up or down as well as right or left. Extend the class hierarchy developed so far to include these two new kinds of Pieces. Define a **SlowFlexible** class that extends the SlowPiece class and a **FastFlexible** class that extends the FastPiece class. Define methods in these classes as needed.

Next define a **Board** class – The Board class will hold the 8 X 8 game board. Each position of the board either contains a Piece or is empty. Since the pieces move around on the game board, any location may contain any type of Piece. Consider what data structure you wish to use to represent the board.

Implement methods to do the following in the Board class:

- Add a new Piece to the game board.

**Note: a new Piece can be added to a location only if none exist at that location; display an error message if the Piece cannot be added.**

- Move a Piece at a given location in a given direction by a given number of spaces. Display an error message if the Piece cannot be moved.
- Display the game board, showing the name, color and type of each Piece on the game board at its current location. You can use a simple text-based display to show the board and the Pieces on it. For example, your display might look like this:

-	-	-	JediBlueS	CloneRedF	-	-	-
DroidBlueFF-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	CloneBlueSF	-	-	-
-	-	-	-	-	-	-	-

```

-      -      -      -      -      -      -      -
-      -      -      -      -      -      -      -
TroigRedF

```

In the above display, there is a Jedi (blue colour, slow piece) and a Clone (red colour, fast piece) in the first row. There is a Droid (blue colour, fast flexible piece) in the second row... and so on.

Finally define a **GameDemo** class – The GameDemo class should accept commands from a user to create Pieces and move them on an 8 X 8 game board, and display the game board.

You should use the following command syntax, with each command appearing on a new line. Words in plain, Roman script should appear verbatim in the command. Words in *Italic* script take one of several possibilities that are explained. Square brackets imply optional parameters.

- create *location* [fast][flexible] – Create a Piece and place it in a given initial *location* of the form x y. By default a Piece is slow and nonflexible.

Examples:

- ‘create 1 1’ → would create a slow nonflexible Piece at location [1, 1]
- ‘create 1 1 fast’ → would create a fast nonflexible Piece at location [1, 1]
- ‘create 1 1 flexible’ → would create a slow flexible Piece at location [1, 1]
- ‘create 1 1 fast flexible’ → would create a fast flexible Piece at location [1, 1]

The create command should be followed by a prompt asking the user to ‘enter the name and the colour of the Piece’.

- move *location direction* [*spaces*] – Move a Piece from a given *location* of the form x y, in a *direction* given as one of four possible Strings: ‘left’, ‘right’, ‘up’ or ‘down’ (the last two only apply to flexible Pieces). The optional [*spaces*] parameter is only applicable to FastPieces, and specifies how many places to move a Piece.

Examples:

- ‘move 5 5 left 3’ → moves the Piece at (5,5) by 3 spaces to the left. In order for this to work, the piece at (5,5) must be a fast Piece.
- ‘move 5 5 up 2’ → moves the Piece at (5,5) two spaces up. In order for this to work, the piece at (5,5) must be a fast flexible Piece.
- ‘move 5 5 right’ → moves the Piece at (5,5) one space to the right. In order for this to work, the Piece can be a slow Piece or a slow flexible Piece or a fast Piece or a fast flexible Piece. If it is a fast Piece or a fast flexible Piece, you would invoke the method move(right, 1) indicating that it is to be moved 1 space to the right.
- ‘move 5 5 up’ → moves the Piece at (5,5) one space up. In order for this to work, the Piece must be a slow flexible Piece or a fast flexible Piece. If it is a fast flexible Piece, you would invoke the method move(up, 1) indicating that it is to be moved 1 space up.
- ‘move 5 5 left 1’ → moves the Piece at (5,5) one space to the left. In order for this to work, it can be a slow Piece or a slow flexible Piece or a fast Piece or a fast flexible Piece. If it is a slow Piece or a slow flexible Piece, you would invoke the method move(left) since the Piece can only move by one space.

- print – Display the board
- help – List all the commands that can be used.

- exit – Exit the game.

Your program must include proper error checking to ensure that the parameters passed to your methods are legal.

Note that the GameDemo class does not play a game. It is only a test harness allowing others to test your code.

Here's a sample portion of the run of the program (not all methods and commands are shown here):

Enter a command (type help for details):

help

Possible commands are as follows:

create *location* [fast][flexible]: Creates a new piece.

move *location direction* [*spaces*]: Moves a piece.

print: Displays the board.

help: Displays help.

exit: Exits the program.

Enter a command (type help for details):

print

```
- - - - -
- - - - -
- - - - -
- - - - -
- - - - -
- - - - -
- - - - -
- - - - -
```

Enter a command (type help for details):

create 3 3 fast flexible

Input a name for the new piece:

Jedi

Input a colour for the new piece:

Red

Enter a command (type help for details):

print

```
- - - - -
- - - - -
- - - - -
- - JediRedFF - - -
- - - - -
- - - - -
- - - - -
- - - - -
```

Enter a command (type help for details):

move 7 7

Error: no piece at (7,7)

Enter a command (type help for details):

move 3 3 up 2

Piece at (3,3) moved up by 2 spaces

Enter a command (type help for details):

print

```
- - - - - - - -
- - - JediRedFF - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
```

Enter a command (type help for details):

exit

Done.

Note: While your GameDemo class is designed to be interactive, prompting users for input and acting upon commands received via stdin, you must hew to the example above. Your GameDemo must accept commands **formatted according to the specification** (ie. create *location* [fast][flexible]... followed by name and colour parameters; move *location direction* [spaces]; print; help; exit). Your markers are again likely to test your code via I/O redirection.

### **Submission:**

All submissions are through Brightspace. Log on dal.ca/brightspace using your Dal NetId.

### **What to submit:**

Submit one ZIP file containing all **source code** (files with .java suffixes) and a **text document containing sample outputs.**

**Create your own test cases** for sample outputs as outlined below.

Sample outputs needed from stdout of the following cases:

1. Creating a Slow Piece and moving it left and right
2. Creating a Fast Piece and moving it left and right by 2 spaces at a time
3. Creating a Flexible Piece and moving it up/down left/right
4. Creating a Fast Flexible Piece and moving it up/down left/right by 2 spaces at a time
5. Show what happens when 2 pieces try to occupy the same location
6. Show what happens when a piece tries to move off the board
7. Show a demo of the help menu

Your final submission should include the following files: Piece.java, SlowPiece.java, FastPiece.java, SlowFlexible.java, FastFlexible.java, Board.java, GameDemo.java, TestCases.txt

**Follow class names exactly as they are given in the assignment.**

### **Your code must compile. Failure to compile may result in a mark of 0.**

Markers may test more cases than given above.

You MUST SUBMIT .java files that are readable by your TAs. If you submit files that are unreadable such as .class, you will lose points. Please additionally comment out package specifiers.

