

WES-CS GROUP MEETING #1

Exercise 1: Drawing Pictures

A Java program involves creating and manipulating *objects*, each of which provides some *operations*. An operation can either perform a task (like printing something on the computer screen), or it can do a computation and tell you the answer. Some operations require that you provide values to be used in their task/computation.

For this exercise, we'll assume that we have an *Artist* object named *picasso* that provides the following operations:

`drawLineDown(int length)`

Draw a vertical line of the given length (in inches), starting from the current position and going straight down. The current position is changed to be at the bottom end of the line.

`drawLineUp(int length)`

Draw a vertical line of the given length, starting from the current position and going straight up. The current position is changed to be at the top end of the line.

`drawLineRight(int length)`

Draw a horizontal line of the given length, starting from the current position and going straight to the right. The current position is changed to be at the right end of the line.

`drawLineLeft(int length)`

Draw a horizontal line of the given length, starting from the current position and going straight to the left. The current position is changed to be at the left end of the line.

`moveRight(int d)`

Move the current position *d* inches to the right.

`moveLeft(int d)`

Move the current position *d* inches to the left.

`moveUp(int d)`

Move the current position *d* inches up.

`moveDown(int d)`

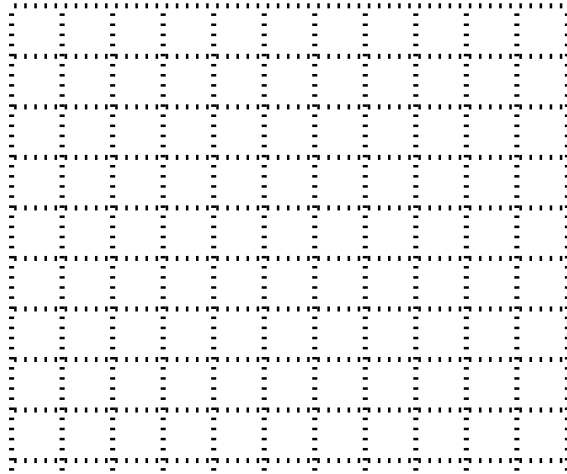
Move the current position *d* inches down.

Part (a). What is drawn when the following code executes? (Use the grid to do the drawing; assume that the current position starts in the top left corner and that the squares in the grid are 1 inch high and 1 inch wide.)

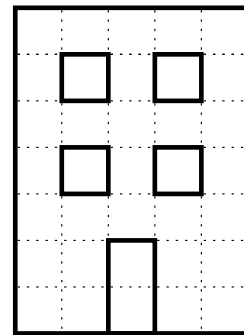
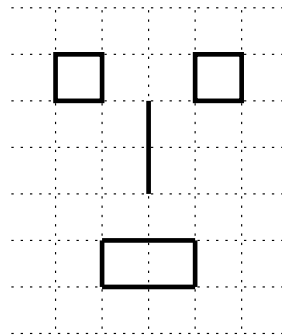
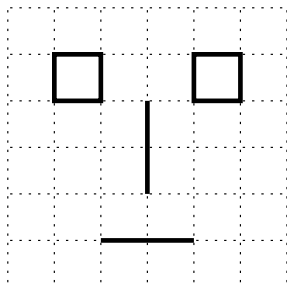
```

picasso.drawLineDown( 7 );
picasso.moveUp( 4 );
picasso.drawLineRight( 2 );
picasso.moveUp( 3 );
picasso.drawLineDown( 7 );
picasso.moveRight( 2 );
picasso.drawLineRight( 2 );
picasso.moveLeft( 1 );
picasso.drawLineUp( 7 );
picasso.moveLeft( 1 );
picasso.drawLineRight( 2 );

```



Part (b). Divide into groups of two. Each group choose one of the pictures shown below and write Java code that would make *picasso* draw the picture. (The dotted lines are not part of the pictures; they're there to show you how many inches you need to move or draw. Each dotted box is one inch on each side. Assume that the current position starts in the top left corner of the grid.)



Part (c). What other *Artist* methods would have made it easier to draw the pictures?

Part (d). Most programming languages, including Java, let you write *loops*. For example, in Java you can essentially say “repeat the following commands *n* times”, Can you simplify any of the code you wrote for part (b) by using loops?

Exercise 2: Logical Thinking (Sudoku)

One of the benefits many students find they get from taking Computer Science courses is that it helps develop their logical-thinking skills. We’ll work on that today by trying some Sudoku puzzles. Here’s a simplified example:

1			2
		3	2

The whole 4-by-4 thing is called a *grid*; each 2-by-2 piece is called a *box*; and the 15 individual parts are called *squares*.

The object is to fill in the empty squares with numbers from 1 to 4 so that the same number doesn’t appear twice in any **row**, or any **column**, or any **box**.

Part (a). Work with a partner to solve the puzzle given above. To solve the puzzle you could just guess, but that won’t work very well. Instead, see if you can find one square that’s *forced* to have a particular number in it. Wait until both you and your partner have found a square like that, then explain to each other which square it is, and why you think it needs to have the number you put in there. If you think your partner made a mistake, explain why (nicely!). Then work together to solve the whole puzzle.

Part (b). Now try solving a real Sudoku puzzle (9-by-9 instead of 4-by-4). Hint: start by looking at a row, column, or box with a lot of numbers already in it.

5		6						
	2			8		9	7	1
	8				4		3	
2			8			7		
			7		1			
		8			5			9
	6		1				9	
1	4	9		5			2	
						3		5

Part (c).

Before we try any more Sudoku puzzles, let's think more about a systematic way to solve them. If you wanted to write a Java program to solve Sudoku puzzles, you'd have to come up with some rules for how to fill in the squares. One way to do that is as follows:

- First, fill in every blank square with a list of all of the numbers that might go there: the numbers that don't appear in the same row, column, or box.
- If you filled in any of the squares with a single number, then that's the final answer for that square, and you can use it to eliminate that number from the lists in the squares in the same row, column, and box. That may cause some other list to turn into a single number, and then you can repeat the process over and over.

This technique works for a lot of Sudoku puzzles (though it's kind of tedious to do it by hand). First try it on our small puzzle:

1	
	2
	2
3	

Work with a partner to solve this puzzle using the rules given above. Then compare your solution with the other pairs in the class.

The technique you just used won't solve all puzzles. Try this one (note that this puzzle has more than one solution):

2			
4		2	
		4	

See if you can figure out any rules for eliminating numbers from some lists. If you can eliminate enough to make a list turn into a single number, then you can go back to the technique we've been trying. There are several interesting rules that you might be able to discover for this puzzle. Once you've done that, go ahead and try to solve some more full-sized puzzles (on the last two pages).

			1			7		2
	3		9	5				
		1			2			3
5	9					3		1
	2						7	
7		3					9	8
8			2			1		
				8	5		6	
6		5			9			

9	4					3	1
	8		9		6	2	
2		6				8	9
			1		3		
7							5
			8		2		
1		2				3	8
	7		3		1	5	
4	5					9	7