
Due Date:	By 11:55pm November 30, 2018
Evaluation:	6% of final mark (see marking rubric at the end of handout)
Late Submission:	none accepted
Purpose:	The purpose of this assignment is to practice defining new types/classes and then using them in your program.
CEAB/CIPS Attributes:	Design/Problem analysis/Communication Skills

General Guidelines When Writing Programs:

See previous assignments.

```

-----****-----****-----****-----****-----****-----
                Welcome to Crazy Nancy's Garden Game!
-----****-----****-----****-----****-----****-----
  
```

For this assignment you will program Crazy Nancy's garden Game. The winner of the game is the player (called gardener from now on) who fills up his/her garden with flowers and trees the first. There are no ties in this game. This game requires luck with the dice and a bit of strategy.

Advice on how to tackle this assignment:

- Plan your solution before starting to program. This means be sure you understand how the game works before your program. Write as detailed an algorithm as you can!!!!!!
- Refer to the 2nd document in the .zip files which contains 2 sample runs to help you understand how the game works.
- Start by writing the 1st three classes (Dice, Garden & Player) and testing them to make sure they work as you expect them to. You don't want to be debugging your classes and your driver class all at the same time.
- When implementing the driver (LetsPlay) do it in chunks. Do not implement the entire game all at once. It will be harder to debug and will be frustrating.
- You will learn a lot from this assignment. Do not waste your time looking for solutions online. It is a made up game.
- Happy programming and happy gardening!



Rules of the game:

1. Number of players: 2 to 10
2. Garden size: N x N, where N is at least 3
3. Each player has a garden board which is empty when they start the game. A player can plant either a flower which takes up one square or a tree which takes up 4 squares (2 x 2).



4. The game works as follows:

- a) Determine who goes 1st: each player rolls 2 dice. The player with the highest roll goes first. However if any player rolls the same total, this process starts over again.
Here is a sample output to illustrate the above explanation:

Let's see who goes first ... a rolled a 7 b rolled a 12 c rolled a 7 We will start over as 7 was rolled by a as well. a rolled a 8 b rolled a 9 c rolled a 7 d rolled a 8 We will start over as 8 was rolled by a as well. a rolled a 10 b rolled a 12 c rolled a 7 d rolled a 4 b goes first.
Figure 1 – Who goes first?

- b) Each player has a turn until there is a winner. During his/her turn a player:
- Rolls the 2 dice.
 - Based on the outcome gets to plant a pre-set number of trees and/or flowers.

Total of Roll	Action
3	Plant a tree (2x2) and a flower (1x1)
6	Plant 2 flowers (2 times 1x1)
12	Plant 2 trees (2 times 2x2)
5 or 10	The rabbit that lives in your garden will eat something that you have planted - might be a flower or part of a tree(1x1) This is determined randomly
Any other EVEN number (2, 4, or 8)	Plant a tree (2x2)
Any other ODD number (7, 9, or 11)	Plant a flower (1x1)

Figure 2 – Action for each roll of 2 dice

- c. It is possible that a player does not have enough room left in his/her garden to plant a tree in which case the player loses a turn. It is also possible that a player fills his/her garden before finishing his/her turn when they have 2 items to plant. If this happens then they are declared the winner.

Implementation of the game:

You will create 4 classes: *Dice*, *Garden*, *Player*, and *LetsPlay*.

Write the declaration of these classes each in a file of their own based on the following specifications. You cannot omit any of the attributes or methods given in these classes. You can add some if you like.

1. Class *Dice*:

- a) A *Dice* object has 2 attributes: Two integers which record the value of each die.
- b) Default constructor which sets the value of each die to zero.
- c) Get (accessor) methods for each attribute.
- d) A *rollDice()* method which randomly assigns a number between 1 and 6 to each die and return the sum of the two dice.
- e) A *toString()* method that returns the content of each die as a String.



2. Class *Garden*:

- a) A *Garden* object has 1 attributes: *garden* a 2-D character array representing the garden. Each location will contain one of the following characters:
 - i. 'f': if a flower occupies that location.
 - ii. 't': if a part of a tree occupies that location.
 - iii. '-': if that location is empty.
- b) A default constructor which creates the array *garden* as 3x3 array (the default garden size) and initializes each space to '-', the symbol used to represent an empty space using the method *initializeGarden()* described in d) below.
- c) A constructor which takes an integer representing the *size* of the garden and creates the *garden* array as a *size* x *size* array and initializes each space to '-' the symbol used to represent an empty space using the method *initializeGarden()* described in d) below.
- d) A **private** *initializeGarden()* method which initializes each space in the array *garden* to '-'.
- e) A *getInLocation(int r, int c)* method which returns the character stored in location [r][c] of *garden*.
- f) A *plantFlower(int r, int c)* method which stores the character 'f' in the location [r][c] of *garden*.
- g) A *plantTree(int r, int c)* method which stores the character 't' in the locations [r][c], [r+1][c], [r][c+1] and [r+1][c+1] of *garden*.
- h) A *removeFlower(int r, int c)* method which stores the character '-' in the location [r][c] of *garden*.
- i) A *countPossibleTrees()* method which returns the number of places a tree can be planted in the *garden*. A tree takes 4 spots as a 2x2 block.

For example if the status of the *garden* is

```
| 0  1  2  3
0 | -  -  -  -
1 | -  -  -  -
```

```

2 | t  t  t  t
3 | t  t  t  t

```

Figure 3 – Sample of a *garden* object

There are 3 possible places to plant a tree.

	0	1	2	3
0	-	-	-	-
1	-	-	-	-
2	t	t	t	t
3	t	t	t	t

	0	1	2	3
0	-	-	-	-
1	-	-	-	-
2	t	t	t	t
3	t	t	t	t

	0	1	2	3
0	-	-	-	-
1	-	-	-	-
2	t	t	t	t
3	t	t	t	t

Figure 4 – Locations where a tree will fit

- j) A *countPossibleFlowers()* method which returns the number of places a flower can be planted in the *garden*.
- k) A *gardenFull()* method which returns true if the garden is full and false otherwise.
- l) A *toString()* method that returns as a *String* the content of a garden object as an N x N square (see figure 3 above).

3. Class *Player* :

1. A *Player* object has two attributes: a *String name* that stores a gardener's name, and a *Garden* object *garden* which is this player's board game.
2. A constructor which takes two parameters, a *String* for the player's name and an integer for the size of the *garden*.
3. An Accessor/get method for the *name* attribute.
4. The following methods are all methods that will have the *garden* attribute call the corresponding method from the *Garden* class. This is necessary since *garden* is a private attribute of the class *Player*.
 - i. *howManyFlowersPossible()* which will return the result of a call to the method *countPossibleFlowers()* by the attribute *garden*.
 - ii. *howManyTreesPossible()* which will return the result of a call to the method *countPossibleTrees()* by the attribute *garden*.
 - iii. *whatIsPlanted(int r, int c)* which will return the result of a call to the method *getLocation(r, c)* by the attribute *garden*.
 - iv. *plantTreeInGarden(int r, int c)* which will call the method *plantTree()* to plant a flower in *garden* in location *[r][c]*.
 - v. *plantFlowerInGarden(int r, int c)* which will call the method *plantFlower()* to plant a tree in *garden* in location *[r][c]*, *[r+1][c]*, *[r][c+1]* and *[r+1][c+1]*.
 - vi. *eatHere(int r, int c)* which will call the method *removeFlower()* to remove a flower in *garden* in location *[r][c]*, which means assigning a '-' to the location



- [r][c]. Note the same method *removeFlower()* will be used to remove a portion of a tree.
- vii. *isGardenFull()* which will return the result of a call to the method *gardenFull()* by the attribute *garden*.
 - viii. *showGarden()* which displays the content of the *garden* object as in figure 1.

4. Class LetsPlay:

The *LetsPlay* class is the driver class and where all the action happens. Your players must be stored in an array of type *Player*. There must not be any *Garden* objects declared in the driver. Each *Player* object has a *Garden* object as one of its attributes. Here are the general steps of the algorithm.

1. Display a welcome banner.
2. Display the general rules of the game. Your text can differ from the sample provided, but the rules must be the same.
3. Ask the user the size of the board: Do they accept the default 3x3 grid or do they want to choose another size. Make sure the size is at least 3.
4. Ask the user for the number of players. Make sure number entered is between 2 and 10 inclusive.
5. Create the Player array. Be sure to include the player's names.
6. Decide who goes 1st. Each player throws the dice (using the DICE class). The one with the highest total goes first. If two players throw the same total, you need to start over. See rule 4 a).
7. As long as there is no winner
 - a. The next player throws the dice.
 - b. Based on the total rolled, either the player can plant something or has the rabbit eating a flower or part of a tree. If the player can plant, prompt the user for the coordinates of the flower(s) and or trees(s).
 - You are always to refer to a player by his/her name, show them the total of their roll, as well as the value of each die.
 - Tell them what they are to plant, show them the status of their garden, tell them in how many places in their garden they can plant the tree or flower. If there is no room for the item in the garden, player loses a turn. If there is room prompt them for the coordinates.

Here is a sample to illustrate the expected behaviour.

Tom you rolled 4 (Die 1: 3 Die2: 1)
You must plant a tree (2x2)

	0	1	2	3
0	f	f	t	t
1	f	-	t	t
2	t	t	t	t
3	t	t	t	t

and have 0 places to do this.

** Sorry no room left to plant a tree - You miss a turn

Figure 5 – No room to plant a tree

Tom you rolled 4 (Die 1: 3 Die2: 1)
You must plant a tree (2x2)

	0	1	2	3
0	f	-	-	-
1	f	-	-	-
2	t	t	-	-
3	t	t	-	-

and have 4 places to do this.

Enter coordinates as row column: 2 2

Figure 6 – 4 places to plant a tree so asking player where to plant

- Once the coordinates are entered check that they are valid. This means that they are within the grid and that the location is empty. If the item to be planted is a tree, make sure the tree will fit in the location specified and that all 4 locations are free.

Here are 2 samples to illustrate the expected behavior or invalid coordinates

	0	1	2	3
0	f	-	-	-
1	f	-	-	-
2	t	t	t	t
3	t	t	t	t

Enter coordinates of 2nd flower as row column: 0 0
** Sorry that location is already taken up by a f
Please enter a new set of coordinates:
0 1

Figure 7 – 1st set of coordinates entered where not free

You must plant a tree (2x2)

		0	1	2	3
0		-	-	-	-
1		-	-	-	-
2		-	-	-	-
3		-	-	-	-

and have 9 places to do this.
Enter coordinates as row column: 3 2
** Sorry either the row or column is not in the range of 0 to 3
or your tree will be off the grid. Try again
2 2

Figure 8 – 1st set of coordinates would have tree off the grid


- If a player is to plant more than one item, they may have room only for one item. Tell the player and request the coordinates for the item they can plant.
- If a player has to plant two items but has a full garden after the first item is planted, then declare this player the winner.
- If after the player is planting only one item and the garden is full after planting, then declare this player the winner.

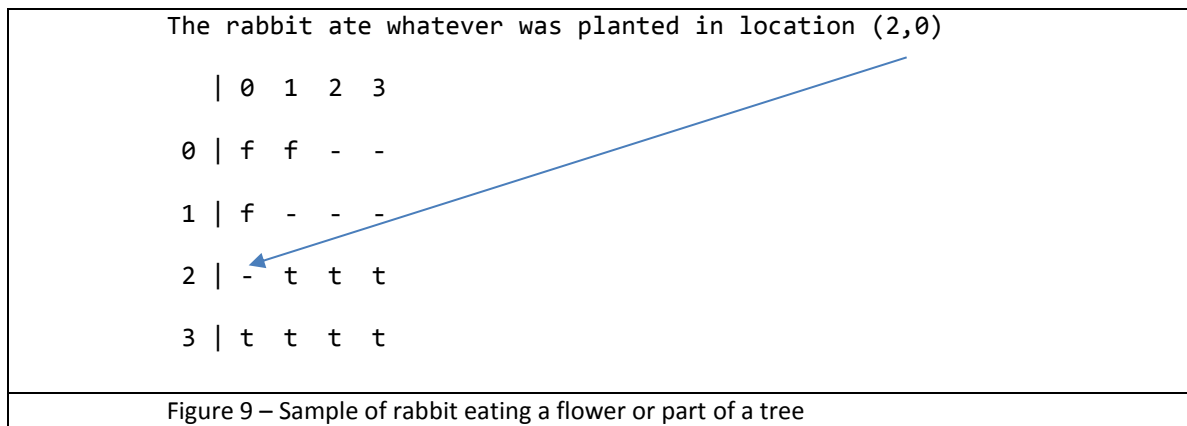
If the rabbit is eating from the garden, randomly generate coordinates until you land on a location that is not free and assign ' - ' . Show the player the status of his/her garden after the rabbit has feasted.

Here is a sample to illustrate the expected behavior of the rabbit feasting.

Nancy you rolled 10 (Die 1: 6 Die2: 4)

		0	1	2	3
0		f	f	-	-
1		f	-	-	-
2		t	t	t	t
3		t	t	t	t





8. Once you have a winner, show all players gardens, and declare the winner by specifying the number of turns it took to win.
Here is a sample to illustrate the expected behaviour.

FINAL RESULTS

Here are the gardens after 15 rounds.

Nancy's garden

		0	1	2	3
0		f	f	t	t
1		f	f	t	t
2		f	t	t	t
3		t	t	t	t

Tom's garden

		0	1	2	3
0		f	f	t	t
1		f	-	t	t
2		t	t	t	t
3		t	t	t	t

And the winner is Nancy!!!!

What a beautiful garden you have.

Hope you had fun!!!!

Figure 10 – Sample of Final Results output

Final Comments:

- You can change the messages as long as the same information is there.
- You can use, and in fact are encouraged, to use static methods in your driver class.
- You must have 4 classes, one of which is the driver class. You must have and use the methods given in the specifications of the classes. You can add other methods to the classes.
- It is not advisable to declare a Scanner object in more than one method. I recommend you create one in the driver class and pass it to any method that requires it.



Submitting Assignment 4

Please check your course Moodle webpage on how to submit the assignment.

Evaluation Criteria for Assignment 4 (20 points)

Source Code	
Comments & Programming Style (3 pts.)	
Description of the program (authors, date, purpose)	0.5 pts.
Description of variables and constants	0.5 pts.
Description of the algorithm	0.5 pts.
Use of significant names for identifiers	0.5 pts.
Indentation and readability	1 pt.
Implementation of classes (6 points)	
Dice class	1 pt.
Player class	2 pt.
Garden class	3 pt.
LetsPlay class (Driver) (11 points)	
Display welcome banner & game rules	0.5 pts.
Determine size of board	0.5 pts.
Get number and name of players & create array of Players	1 pt.
Who goes first?	1 pt.
Validate coordinates entered by player	1 pt.
Planting correct items based on dice roll	4 pts.
Format of output – easy to follow	2 pts.
Detecting and displaying winners	1 pt.
TOTAL	20 pts.