Assignment 7: Dem Bones, Dem Bones

Due Friday November 30, 2001

1 Problem Description

Your dog is a rather odd dog. She buries the bones you give her in peculiar patterns in the back yard. Specifically, she buries the bones in short lines — oriented either North–South or East–West (never diagonal), and always buries each of the bones that makes up a line of bones at one of the positions of a grid as in Figure 1:

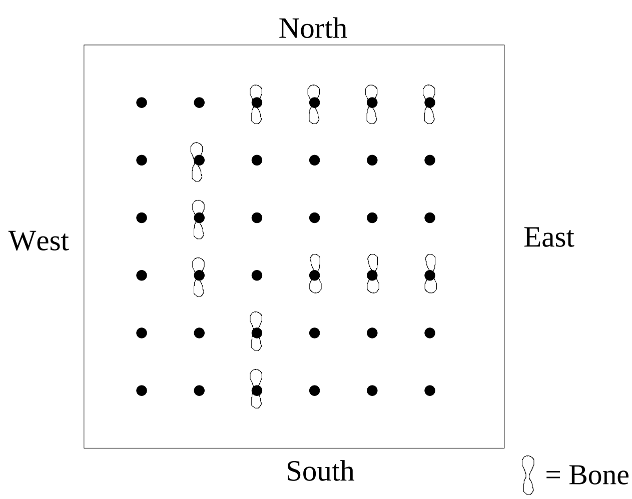

Figure 1: Your back yard after your dog has been burying bones.

Here she has buried four short lines of bones, one of length four, two of length three, and one of length two. (The length of a line of bones is the number of bones in the line). Notice that she **never** buries bones on top of one another; the short lines of bones can never cross each other or overlap. Therefore no grid position ever contains more than one bone, and no bone ever belongs to more than one line.

Now, you don't watch your dog bury the bones, so you don't know where they are buried. But you would like to dig them up from time to time so that your backyard doesn't become completely filled with bones. In order to find them, you'll have to guess grid positions until you've found them all.

The goal of this assignment is to simulate the foregoing by writing a Java program. That is, simulate your dog burying four lines of bones of the lengths given above, and allow the user to guess grid positions in order to find the bones. Your program will hide the bones so that the user doesn't know where they are, and after each guess by the user, your program will have to check to see if a bone is buried at that grid position. In addition, your program will print a 'map' showing the user which grid positions haven't been checked yet, which grid positions were checked and found to contain a bone, and which grid positions were checked and found to contain no bone.

If you've ever played the game BattleShip (TM Milton-Bradley) you may recognize the idea for this assignment, although the game described by this handout is somewhat simpler.

2 Playing The Game

The class which contains the `main` method for playing the game must be called **BoneDig**.

Your program must first hide the bones, and then ask the user to type in two integers, one for the row and one for the column of a grid position (have the user input the row first, and then the column). Your program must then read these two integers and use them to determine whether a bone is buried at that grid position. You can assume that the user will input an integer, but you must check to make sure that the integer which the user has typed is in the bounds of the grid. Your program **must** print the grid after each guess in the following manner: At each grid position which has been guessed so far, your program must print a character to indicate that a bone has been discovered at that position, or a character to indicate that no bone has been discovered at that grid position. At each grid position which hasn't been guessed, your program must print the character '.' to indicate that the position hasn't yet been guessed.

In addition you **must** print a version of the grid which shows where the bones were initially hidden. This will remain the same throughout the game, but must be printed after each guess. (This clearly ruins the game aspect since the user will be able to see where the bones are and therefore can easily 'guess' their positions, but we want you to do this so that our markers can test your program easily.)

Your program will continue to ask the user for guesses and print the two versions of the grid just described until all the bones have been discovered. Once all the bones have been discovered (and not before) the game will end.

There is no way to clear the screen in Java, so after each guess you'll be printing the two grids below the previous two grid images; as you keep doing this, the oldest grids will just 'scroll off the top' of your output window.

3 The Grid

The grid must be square, and it must have 10 rows and 10 columns. However, the size of the grid must be specified on one line of your program so that a programmer could change the grid size by changing that one line. Note that since your program <u>won't</u> change the size of the grid, the variable that contains this value should be declared as `final`. You **must** use a two (or higher) dimensional array to represent the grid.

As an example, the grid shown in Figure 1 is square with 6 rows and 6 columns.

4 Burying The Bones

Your program must choose where to bury each line of bones randomly. To do this, it will have to:

1. Decide (randomly) whether the line of bones should be North–South or East–West. The probability of each of these outcomes should be .5 (50%).

2. If horizontal, randomly choose the leftmost position of the line so that the line won't be too long for the grid

3. If vertical, randomly choose the topmost position of the line so that the line won't be too long for the grid

4. If one of the bones in this line of bones will occupy a grid position that already has a bone, then reject the position of this line, and generate a new position using steps 1,2, and 3.

5. If none of the bones in this line of bones will occupy a grid position that already has a bone, then accept the position of this line of bones.

6. Of course, once you accept the position of the line of bones, you'll have to store it somewhere. There are many ways to represent the line of bones. Choose one that is most convenient for you.

Your program must bury four lines of bones of lengths 4,3,3, and 2.

5 Reading from Keyboard, Printing to Display

One principle of good software design is to create the software so that it can be modified easily in the future. In this assignment you will be using that same kind of statements that you have used in the past to do I/O, but we would like you to design your program so that it could be easily modified (hypothetically) at some later date. (This modification won't actually take place, but in order to motivate a good design, let's assume that it *might*)

Imagine that your program will be modified in the future so that instead of doing I/O as we learned in class, a Graphical User Interface (GUI) will be implemented. (**NOTE:** You shouldn't write a GUI for this assignment, unless you really want to. If you do, you won't receive any extra marks.) The GUI might display nice images of a backyard and bones, and let you click on grid positions with the mouse, for example.

The question is this: How do we best structure our program so that the program could be changed at a later date to use a GUI? The answer is that we should put all our I/O statements in a single class. That class will just do I/O and nothing else. In that way, a GUI could be implemented at a later date by just replacing that one class with a class that implements the GUI.

In this assignment you must put all statements that read from the keyboard or write to the display in a number of methods of your choosing that belong to a single class. Whenever you want to read from the keyboard or write to the display, you will have to call a method in this input/output class. There will be no I/O statements anywhere else in your program.

More specifically, when you want to print information to the screen, you will have to call a method in your input/output class and send that information as an argument to that method. That method will then print information to the screen. When you want to read something from the keyboard, you will have to call a method in the input/output class which reads from the keyboard and returns the information to the calling method as a return value.

The upshot is that all statements that read from the keyboard and write to the display must be in one class, and you shouldn't be writing a GUI for this assignment. (no bonus marks or penalty if you do)

The user of the program will have to specify a row number and a column number in order to specify a position on the grid. You must use two calls of the method **readLine()** (or any other suitable method) to input this information: one for the (integer) row number and one for the (integer) column number. You are free to read these as strings, or characters, or whatever you like. In the case of a string or a character, you will most likely want to convert the input value to an integer.

6 For Those Who Are Keen....

There are some things that you can do to make this programming assignment more interesting. You will **not** get extra marks for any of these suggestions, and **none** of them are required.

1. Print out a message saying how many bones were discovered after each guess

2. Print out a message every time a complete line of bones has been discovered

3. Write the program so that in addition to the user trying to guess where bones are hidden, the user *hides* bones in a (different) yard, and the *computer* guesses where the user has hidden the bones. Note that if you write the required version of this assignment in good Object Oriented style, then making it a two player game is fairly easy, especially if the strategy that the computer uses to guess is simple. (guessing randomly on the grid, for example)

4. Write an intelligent strategy for guessing so that the computer is a tough opponent.

5. Write the program so that instead of hiding 4 lines of bones of lengths 4,3,3,2, the program will hide an arbitrary number of lines of arbitrary length. Note that this is not possible: once the size of the grid has been specified that puts limitations on the number and lengths of bonelines. If you try this, you should restrict 'arbitrary' to sets of bonelines that will actually fit, or write an algorithm to determine whether the set of bonelines can possibly fit. Otherwise, your random placement algorithm might loop forever....

6. Write a GUI for the game.

7 Summary: Requirements

1. The class which contains the `main` method for playing the game must be called **BoneDig**.

2. The grid must be square, and it must have 10 rows and 10 columns. The size of the grid (10) must be assigned to a single variable on one line of your program in such a way that you could change 10 to 15 on that line, recompile, and be able to play the game properly on the grid of size 15, for example.

3. You must use a two (or higher) dimensional array to represent the grid.

4. Your program must bury 4 lines of bones of lengths 4,3,3, and 2.

5. Your program must randomly choose positions for the lines of bones. A line of bones must not 'go off the edge' of the grid: it must fit within the grid.

6. No grid position can have more than one bone, and no bone can belong to more than one line of bones. Therefore there must be exactly 12 (=4+3+3+2) grid positions that have bones immediately after the lines of bones have been placed on the grid.

7. You must read the guess from the user so that the user types an integer, presses 'return' and then types another integer and presses 'return'.

The row number should be input first and then the column number, but this order is not required.

8. You may assume that the user will type an integer in each case, but you may **not** assume that the integer is within the bounds of the grid. Therefore, you must check to see if the row and column are valid integers (within the grid size), and take appropriate action if they are not. Appropriate action means that an 'out of bounds' guess should not impede the playing of the game: you might ask the user to try again, for example, or you might just discard the user's guess which will in effect cause the user to lose that turn.

9. You must print a version of the grid which indicates the grid positions at which bones have been discovered, the grid positions at which guesses have been made but no bones discovered, and the grid positions at which no guesses have been made. You will indicate these three different possibilities by using a different printed character for each case. **This version of the grid will not show any bones that have not yet been discovered**, and it will change after every guess made by the user.

10. You must print a second version of the grid which shows how many bones are hidden at each grid position. This is the map of where the bones are after your program has randomly placed them, but before any guesses have been made, and remains the same throughout the game. It **must** be printed after the version of the grid described in the previous item. Therefore you will be printing this version of the grid after **every** guess by the user, as is the case for the first version of the grid. We are requiring this in order to make your program easier to mark; the marker will actually play the game, and we would like the marker to be able to choose grid positions that have bones or don't have bones without having to choose randomly.

11. All the input and output statements **must** be in a class that is separate from all other classes. This class will do nothing but input and output.

12. The game must end when all the bones have been discovered, but not before all bones have been discovered. A bone can be discovered only once. Thus you will have to know or be able to find out whether

undiscovered bones remain in the yard in order to determine whether the game should end or not.

Note that there are many correct ways of doing this assignment, and you are free to write the code as you like and use the methods that you like as long as the program conforms to the specified requirements. None of the suggestions in the section For Those Who Are Keen... are required, and extra marks will neither be given nor deducted if you decide to implement any of these suggestions.