Designing a Program.

In this class, we designed a program that played the game Chutes and Ladders.

There are two ways to design a complex program or piece of a program.

1) Top-down design

We start with the entire idea for the program and break it into the separate tasks.

We then break each task into sub-tasks.

We continue breaking down the sub-tasks further until we get to a stage where we finally see how to write code for it.

Example: to write the Chutes and Ladders program we must:

1) Create the game board

2) Create the players

3) Play the game

4) Deal with the winner (Reset to play again?)

For 1: how to create the board?

- create the spaces

- number the spaces

- create the chutes and ladders

We need to think of two "boards" in our program. The visual board displayed on the screen and the model of the board used inside our program.

These two might be completely different as the model should closely match the way the game is played but the visual is how we want the game to appear.

2) Bottom-up design

In this technique, we do not worry about the entire program at first. Instead, we start by figuring out what piece of the program we

can code first.

For example, I may not know how to code the game, but I do know how to generate a die roll. So, I write code for that, and then I figure

out what to do next.

In this technique, we will often need to change or throw away what we write first (because we do not know how all the pieces will fit together),

but it has the advantage that we can at least have something to write and start playing with as we try to grasp the full complexity of the desired

program.

Please see the Chutes and Ladders sample program.

We needed to make some initial decisions before coding:

1) How to represent a board?

2) How to represent a player?

In both cases, we should focus on the behavior we need, and ignore what it looks like.

For example, while the board game is a grid of squares, the actual game is played linearly, by running through the squares from 1 to 100, occassionally

jumping to a new square.

As a result, we will use a single dimensional array to represent the board. We will make the array size be 101 (so entry 56 corresponds to square 56).

This will make the code easier to read since we will not be constantly subtracting and adding 1 to match the array index with the square number, and it only costs

us a trivial amount of additional memory.

Also, it means we start all players at 0, instead of -1.

What type should be stored in each array entry?

For the players, all we need to record is which square the player is on. Since each square is an entry in an array, we can store the index for the player.

Thus, we will represent each player by an int, and so the set of players will be an array of int.

Finally for the board, all we care about is, if the player lands on a certain square, do they need to jump to a new square? So, we will represent the board as

an array of int. The value of the board[x] will be the square you need to jump to if you land on square x.

For all non-chute and ladder squares, we will just store board[y] = y to indicate the player should stay at this spot.

For playing the game, we used a bottom-up design.

First we created the die (we just used the class created earlier in the course)

Then we created a way to move a player.

Then, we created code to change players after a turn.

Finally, we put it together in a loop.

Because we did a bottom up design, things were not quite as nice as they could be.

For one, there are a lot of magic numbers in our code. This can be later corrected to improve the program.

Debugging Code

DrJava has a debugger as part of the program. (Almost all IDE's have a debugger, and all of them behave similarly.)

To use the debugger, you must have the Java SDK loaded on your system. The default compiler that comes with DrJava does not have all the features needed to run the debugger.

Key terms:

Breakpoint: This is a spot in your code where you want the execution to pause. You set this using the menu you get when you right click the mouse while on the code.

Then you can run your code by entering appropriate Java into the interactions pane. When the execution reaches the breakpoint, the execution will pause.

Stack Trace: This is a list of the methods on the call stack at the moment when the execution has paused.

Watch List: You can enter variables into the watch list, and the current values will be displayed.

You can also type in any expression (including variable names) into the interactions pane to see what the value of the expression is.

You can change the value of variables in the interactions pane, but the value listed in the watch list will not be immediately updated.

Debugger controls:

Step over: Execute the next line of the program as a single step, and then pause.

Step into: Execute the next line of the program. If the next line is a method call, go into that method and execute only its first line, and then pause.

Step out: Go out to the current line of the calling method.

Resume: Continue execution. The program will run until it reaches the next break point.