Assignment 7

Hunt the Wumpus

One of the first computer games in the 1970's was an adventure game called "Hunt the Wumpus". In this game, you are forced to wander around a maze of caves, hunting the dreaded wumpus (or possible more than one wumpi). You are armed only with hand grenades, which you can throw at a wumpus to kill it. The problem is that wumpi are both fierce and fast. If you ever wander into a cave that contains a wumpus, then that wumpus will attack and kill you before you even have a chance to throw a grenade. Your only hope is to throw the grenade into the cave from an adjacent cave. Fortunately, wumpi are rather odorous creatures, so you will be able to smell a wumpus when you are in an adjacent cave. Of course, you won't know which cave the wumpus is in (every cave is connected to three other caves), so you will have to guess when you throw your grenade.

The object of the game is to kill all of the wumpi before you run out of grenades (and without getting killed). As if this weren't hard enough, you don't know exactly how many wumpi there are, or how many grenades you have (you will have 4 grenades for every wumpus). Plus, there are other things to watch out for. Somewhere in the maze is a group of giant bats, which will pick you up and fly you to some random cave (but at least they will only drop you in an empty cave). There is also a bottomless pit which you must avoid. Luckily, there is some warning about these hazards: you will be able to hear the flapping wings of the bats and feel a draft coming from the pit when you are in an adjacent cave. Oh, and there are the Lost Caverns of the Wyrm, which are very difficult to get out of. Below is a sample execution of the game.

```
> wumpus
HUNT THE WUMPUS:  Your mission is to explore the maze of caves
and destroy all of the wumpi (without getting yourself killed).
To move to an adjacent cave, enter 'M' and the tunnel number.
To toss a grenade into a cave, enter 'T' and the tunnel number.

You are currently in The Fountainhead
    (1) unknown
    (2) unknown
    (3) unknown

What do you want to do? m 2

You are currently in The Silver Mirror
    (1) The Fountainhead
    (2) unknown
    (3) unknown

What do you want to do? m 3

You are currently in Shelob's Lair
    (1) The Silver Mirror
    (2) unknown
    (3) unknown

What do you want to do? m 3

You are currently in The Lost Caverns of the Wyrm
    (1) unknown
    (2) unknown
    (3) unknown
You smell a WUMPUS!

What do you want to do? t 1
```

```
  Missed, dagnabit!
DANGER: Any nearby wumpi are on the move.
A wumpus is coming toward you with big, gnarly teeth... CHOMP CHOMP CHOMP
GAME OVER
```

Note: This game in no way intends to promote or condone violence. If you object to the violent nature of this game, alter the rules accordingly. Perhaps a wumpus is a poor soul who has been placed under a spell, and instead of grenades you are tossing bottles of potion that release the person from that spell. Use your imagination.

PART 1: In order to write a program for playing Hunt the Wumpus, you will first need to complete the implementation of the `Maze` class, whose definition is attached to the end of this assignment. This class stores all of the information about a maze of caves, as well as state information about the game (location of the player and wumpi, number of grenades left, etc.). It also provides member functions needed in order to write a game playing program.

Your program should assume the existence of a data file called "caves.dat", which stores the configuration of the cave maze. The first line of this file should specify the number of caves in the maze, and then each subsequent line will contain the following information:

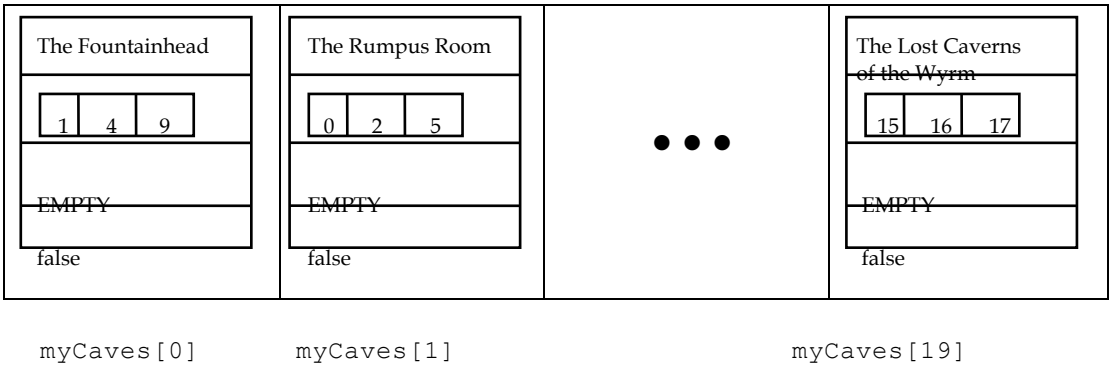
```
<cave number> <adjacent1> <adjacent2> <adjacent3> <cave name>
```

where the cave number is a unique number between 0 and the number of caves minus 1, the next three numbers are the numbers of the caves adjacent to this one (all caves are adjacent to three other caves), and the rest of the line is the name of the cave. For example, the standard data file for this game is given below:

```
20
 0  1  4  9The Fountainhead
 1  0  2  5The Rumpus Room
 2  1  3  6Buford's Folly
 3  2  4  7The Hall of Kings
 4  0  3 14The Silver Mirror
 5  1  9 11The Gallimaufry
 6  2  7 12The Den of Iniquity
 7  3  6  8The Findledelve
 8  7  3 13The Page of the Deniers
 9  0  5 10The Final Tally
10  9 11 14Ess four
11  5 10 12The Trillion
12  6 11 13The Scrofula
13  8 12 18Ephemeron
14  4 10 15Shelob's Lair
15 15 16 19The Lost Caverns of the Wyrm
16 15 17 19The Lost Caverns of the Wyrm
17 16 17 18The Lost Caverns of the Wyrm
18 13 17 19The Lost Caverns of the Wyrm
19 15 16 17The Lost Caverns of the Wyrm
```

However, it should be possible to customize the game simply by replacing this file with a different cave configuration. The data structure used to store the maze information is a Vector of structs, with each struct containing the name of the cave, the numbers of adjacent caves, the contents of the cave (e.g., wumpus, bats, ...), and whether or not that cave has been previously visited. The constructor for the Maze class, which is provided for you, reads in the information

from the "caves.dat" file and initializes the Vector. For example, the contents of the data file above would be stored as shown below:



The constructor determines how many wumpi there are going to be (a random number between 1 and the number of caves divided by four), initializes the number of grenades (four grenades per wumpus), and randomly places the wumpi, bats and pit in the maze.

Complete the definition of the Maze class by implementing the following member functions:

- `Toss`: This function is called with a tunnel number (1 - 3) as argument. It simulates throwing a grenade from the current location into the cave corresponding to that tunnel. Note that given the current location, the corresponding cave number can be determined by the expression

If the player has no grenades left, then the function should inform him/her of that fact. Otherwise it should decrement the number of grenades and display the result of the explosion. If a wumpus is in the specified cave, then that wumpus is killed and the wumpus count is decremented. Any wumpi in adjacent caves, however, are alerted by the explosion. For each adjacent wumpus, the function should pick an empty cave adjacent to the wumpus' location (if one exists), and move the wumpus there. If it so happens that the wumpus moves into the cave containing the player, then it devours the player.

- `Move`: This function is called with a tunnel number (1 - 3) as argument. It simulates moving from the current location into the cave corresponding to that tunnel. If that cave contains a wumpus, then the player is devoured. If the cave contains a pit, then the player falls into the pit and is killed. If the cave contains bats, then the player is moved to a random, empty cave in the maze. Note: the `FindEmpty` private member function should prove handy here.

- `Feel`: This function returns true if a cave adjacent to the current location contains a pit.

- `Hear`: This function returns true if a cave adjacent to the current location contains bats.

- `Smell`: This function returns true if a cave adjacent to the current location contains a wumpus.

- `CurrentName`: This function returns the name of the cave where currently located.

- `AdjacentName`: This function is called with a tunnel number (1 - 3) as argument. If the cave corresponding to that tunnel has been previously visited, then the function returns the cave name. If not, it returns "unknown". In this way, caves that have been previously explored will be recognized when they are next encountered.

- `StillAlive`: This function returns true if the player is still alive.

- `StillWumpi`: This function returns true if at least one wumpus is still alive in the maze.

You should write a driver program to test your code. For easier testing, consider modifying the constructor temporarily so that you know exactly where things are. As always, interleave the coding and testing phases -- test each function independently as you complete it.

PART 2: Using your `Maze` class, write a program which allows the user to play Hunt the Wumpus. You may find that your program will be quite short, since most of the work is done in the `Maze` member functions .

When grading this program, special attention will be paid to style and robustness. As always, your program should be modular and should follow the commenting, indentation, and naming conventions presented earlier in the semester. Robustness refers to your program's ability to behave correctly in all cases, even those that may be unexpected or rare. As you test your program, you should be very careful to think about any such cases that might arise, and test for them. For example, a wumpus that is startled by a nearby explosion is supposed to run to an adjacent empty cave. What if there are no empty caves adjacent to a wumpus? Although this is unlikely, especially with a small number of wumpi, it is possible. In such a case, the wumpus should stay where it is. Special cases such as these may affect your design of the wumpus program, and may force some modifications to code in the Maze member functions.

```
import java.io.*;

/**
 * WumpusMaze.java -
 *   An object that represents the maze in which the
 *   "Hunt the Wumpus" game is played.  This maze
 *   is constructed from an interconnected collection
 *   of WumpusCave objects.
 *
 * @author Grant William Braught
 * @author Dickinson College
 * @version 4/5/2000
 */
class WumpusMaze {

    // Instance data for the maze.
```

```
private WumpusCave [] theCaves;   // caves that make up the maze.
private int [][] theTunnels;      // indicates how caves are connected.

private int numCaves;             // number of caves in the maze.
private int numWumpi;             // number of live wumpi.

private int numGrenades;          // number of remaining grenades.
private int currentCave;          // index of current cave
private boolean stillAlive;       // is the player still alive?

// Instance data that will be used by the methods that
// read the data from the input file.
private FileReader fr;
private StreamTokenizer inputFile;

// Private methods that can only be called from within
// this class.  These methods are used to help with
// reading the data file.

/**
 * Read the next integer from the cave.dat file.
 *
 * @return the next integer in the inputFile.
 */
private int caveFileReadInt() {
  try {
      inputFile.nextToken();
  }
  catch (final IOException e) {
      System.out.println("Error reading integer from caves.dat");
      System.exit(-1);
  }

  return (int)inputFile.nval;
}

/**
 * Read the characters from the current point to
 * the end of the line into a String.
 *
 * @return a string containing the remainder of the
 *         current line of the input file.
 */
private String caveFileReadLine() {
  String ret ="";
  try {
      inputFile.eolIsSignificant(true);
      inputFile.nextToken();
      while(inputFile.ttype != StreamTokenizer.TT_EOL) {
        ret = ret + inputFile.sval + " ";
        inputFile.nextToken();
      }
      inputFile.eolIsSignificant(false);
  }
  catch (final IOException e) {
      System.out.println("Error reading line from caves.dat");
      System.exit(-1);
```

```
    }

    return ret.trim();
  }

  // Public methods below...

  /**
   * Construct a new WumpusMaze according to the data contained
   * in the "caves.dat" file. This includes creating the
   * WumpusCave objects and indicating how these caves are
   * connected.  This constructor also randomly places wumpi,
   * bats and pits in the maze and picks a random starting cave
   * for the player.  The player will always be started in an
   * empty cave.
   */
  public WumpusMaze() {

    // The following lines setup the input file so that
    // the caveFileReadInt and caveFileReadLine will work.
    // They do not need to be modified in any way!
    try {
        fr = new FileReader("caves.dat");
        inputFile = new StreamTokenizer(fr);
    }
    catch (final IOException e) {
        System.out.println("Error opening caves.dat");
        return;
    }


    // Your code to construct and initialize the maze
    // goes here!
  }


  /**
   * Generate a String representation of this WumpusMaze. This
   * is useful in debugging the constructor and instance methods.
   * This String may simply be all of the instance
   * data of the WumpusMaze class in an easy to read format.
   *
   * @return a String representation of this WumupsMaze.
   */
  public String toString() {
    return "No WumupsMaze here.";
  }
}
```

HINTS

Reading the `caves.dat` file

The `WumpusMaze.java` file you downloaded contains some code that will be helpful for reading the data in the `caves.dat` file. You are not expected to understand how this code works but you will need to know how to use the two private methods:

`private int caveFileReadInt()`
This method will read and return the next integer from the `caves.dat` file.

As an example of how you might use `caveFileReadInt()` in the `WumpusMaze` constructor, consider the following code:

```
int numCaves = caveFileReadInt();    // Read in the 20
int caveNum = caveFileReadInt();     // Read in the 0
int adj1 = caveFileReadInt();        // Read in the 1
int adj2 = caveFileReadInt();        // Read in the 4
int adj3 = caveFileReadInt();          // Read in the 9
```

Placing this code in the constructor of `WumpusMaze` will read the first 5 integers from the `caves.dat` file and place them into the variables. The next item in the file is a string, to read it see the `caveFileReadLine()` method below.

`private String caveFileReadLine()`
This method will read and return the remainder of the current line of the `caves.dat` file.

As an example of how you might use the `caveFileReadLine()` method in the `WumpusMaze` constructor, consider the following line of code:

```
String name = caveFileReadLine();  // Reads in "The Fountainhead"
```

If you were to place this line of code in the constructor of `WumpusMaze` (following the 5 calls to `caveFileReadInt()`) it will read the name "The Fountainhead" and place it in to the variable `name`.

After you have read a complete line from the `caves.dat` file the next call to `caveFileReadInt()` will read the first integer on the next line of the file. For example, executing a call to`caveFileReadInt()` after the above code will read the number 1, that appears at the start of the third line of the file.

Keeping Track of the Caves

As you read the data for each cave in the maze you will need to construct a new `WumpusCave` object for that cave. You will probably want to create an array of references to `WumpusCave`objects as instance data in `WumpusMaze` to keep track of the caves. This array should contain one element for each of the caves in the maze. I

would suggest using the reference in the 0'th location of the array to refer to the `WumpusCave` object for cave #0 and so forth. In this way the `WumpusCave` object for a given cave can easily be found by using its cave number as the index for the array.

To test what you have done so far, I suggest adding a `toString()` method to the `WumpusMaze` class that simply prints out the name of each cave. Keep in mind that if you implemented`WumpusCave` correctly `getName()` will return "unknown" until a cave has been visited. So for testing purposes, your `toString()` method could visit every cave and then print its name.

Keeping Track of the Tunnels

As you read in the `caves.dat` file and create the `WumpusCave` objects for each cave you will also need to keep track of how the caves are interconnected by the tunnels. To do this I suggest creating a 2 dimensional array of integers as instance data in the `WumpusMaze` class. This array will need to be created in the constructor to have one row and 3 columns for every cave.

The row number of the array will represent the cave number and the column number will represent the tunnel number. With this arrangement the integer in row `R` and column `C` will be the number of the cave that is reached if the tunnel `C` is followed from the cave `R`. The first few rows of the array for the given `caves.dat` file would look as follows:

```
              Column:
            0   1   2
          +---+---+---+
      0   | 1 | 4 | 9 |
          +---+---+---+
      1   | 0 | 2 | 5 |
Row:      +---+---+---+
      2   | 1 | 3 | 6 |
          +---+---+---+
      3   | 2 | 4 | 7 |
          +---+---+---+
      ...
```

From the table it can be seen that from cave 0 (row 0) if tunnel 1 (column 0) is followed the player will end up in cave 1. Similarly, if the player were in cave 3 (row 3) and followed tunnel 3 (column 2) they would end up in cave 7.

To test the addition of the array for keeping track of the tunnels, I would suggest adding to the `toString()` method in `WumpusMaze`. When `toString()` prints out the name of a cave have it also print out the caves to which it is connected by the tunnels. You can then compare this output to the `caves.dat` file to check if your program is working correctly.

Randomizing the Game

Once you have created the maze and recorded all of the tunnels you will need to place the Wumpi, bats and pits throughout the maze. These items must be placed randomly so that the game will be different every time it is played. The constructor for the `WumpusMaze` class should:

- Decide on a random number of wumpi between 1 and the number of caves divided by 4. So no more than 25% of the caves will have a wumpus in them.
- Randomly place the wumpi.
- Randomly place a bottomless pit in one of the caves.
- Randomly place a flock of bats in one of the caves.

When placing these items keep in mind that a cave may only contain one item. Because you will be required to find random empty caves at several places in your program You might find it useful to add a private method for this purpose. Your method might be named `findRandomCave()` and it could return an integer that corresponds to a randomly selected empty cave.

To be sure that you have correctly placed these items you should add code to `toString()` that prints out the contents of each cave. Being able to display this information will also be invaluable when it comes time to test methods like `smell()`, `listen()` and `feel()`.

Moving Around the Maze

The `move(int theTunnel)` method will handle everything associated with moving a player around the maze. It will update instance data that indicates which cave a player is currently in and if the player is still alive after the move. If the player moves into a cave containing a wumpus or pit then the the player will no longer be alive after that move. If the player moves into a cave containing bats then they should be transported to a random empty cave in the maze. This method should print informative and possibly funny messages when the player is killed or transported to another cave.

Tossing Grenades

The `toss(int theTunnel)` method will handle everything associated with tossing a grenade. It will update instance data that indicates the number of grenades remaining and decrease the number of live wumpi if one is killed. This method must also move any wumpi that are scared but not killed by a grenade. Finally, the method must handle the situation in which a scared wumpus flees into the same room as the player

and Chomps them to death! This method should print informative messages indicating if a wumpus has been killed or not and if the player has been Chomped.

Playing the Game

This is the easy part! Once your two classes are working writing the code to play the game is quite easy. The following pseudo code should give you a start:

```
create a maze.

while(the player is alive && Wumpi are alive) {
   display the player's current location and any smells, sounds or
feelings.
   read in the player's move for this turn.
   move the player or toss the grenade.
}
```