In Lab 2, you will complete a program for playing the game of Die. Die is a game involving a six-sided object that your instructor has invented and will likely become very popular. The goal is to guess the value of the die in as few guesses as possible. Before each guess, your program will get two bits of evidence which are probabilistically related to the value of the die. Naturally then, to properly play this game, your program will need to create and use Bayesian networks.

The initial program is diegame.zip, which you can download from the course web site. This code includes the variable elimination method for making inferences from a Bayesian network (see Section 6.4.1). Your part of the program is to put Bayesian networks together for different game situations. You can play this game by running DieGame.java. It is expecting you to enter how many games you want to play, so you will need to enter 1 (or some other number) first.

**Bayesian Networks for the Game of Die**

The Die game is played in the following way: First, your program receives two bits (see the variables evidence1 and evidence2 in DieGameSolver.java). Then, your program guesses the value of the die (see the variable guess in DieGameSolver.java). The game ends if the guess is correct. Else your program receives two more bits, your program makes another guess, and so on.

The way the two bits are generated correspond to the following Bayesian network.

![Bayesian Network Diagram](attachment://Bayesian_Network_Diagram.png)

Each round the program gets two additional bits of evidence. This should result in a bigger Bayesian network. For example, the following Bayesian network corresponds to round two.
A third round should be modeled with an additional switcher variable and two more evidence variables.

**Probability Variables and Tables**

The die has six possible values from one to six, equally likely. All the other variables have two possible values, zero or one. One of each pair of evidence variables is zero if the die is even, else one. The other evidence variable of each pair is probabilistic; it is more likely to be zero if the die is low. Which evidence variable is which depends on the corresponding switcher variable. The switcher variable is equally likely to be zero or one. If the switcher is zero, then the first evidence variable indicates an even/odd die, but if the switcher is one, the second evidence variable indicates even/odd.

The die corresponds to this probability table. Note that the values in DieGameSolver.java are often off by one because of zero-indexing.

<table>
<thead>
<tr>
<th></th>
<th>P(Die)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6</td>
</tr>
<tr>
<td>1/6</td>
<td>1/6 1/6 1/6 1/6 1/6 1/6</td>
</tr>
</tbody>
</table>

A switcher variable corresponds to this probability table:

<table>
<thead>
<tr>
<th></th>
<th>P(Switcher)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

The first evidence variable of each pair corresponds to this probability table:
The second evidence variable of each pair corresponds to this probability table:

| Die | Switcher | P(Evidence1 | Die, Switcher) |
|-----|----------|-------------------|
| 1   | 0        | 0.0   1.0         |
| 2   | 0        | 1.0   0.0         |
| 3   | 0        | 0.0   1.0         |
| 4   | 0        | 1.0   0.0         |
| 5   | 0        | 0.0   1.0         |
| 6   | 0        | 1.0   0.0         |
| 1   | 1        | 1.0   0.0         |
| 2   | 1        | 0.8   0.2         |
| 3   | 1        | 0.6   0.4         |
| 4   | 1        | 0.4   0.6         |
| 5   | 1        | 0.2   0.8         |
| 6   | 1        | 0.0   1.0         |

The Programming in Lab 2

Although you students would undoubtedly learn more if you had to implement Bayesian networks and variable elimination from scratch, your instructor has decided to try to simplify the creation and operation of Bayesian networks for you. The disadvantage is trying to reverse-engineer the instructor’s design while trying to implement the missing pieces. Hopefully, this explanation and the comments in the code will help.
Your Task

Your task is to finish the `dieNetwork` method in `DieGameSolver.java` so it creates a Bayesian network as specified above. More variables and factors need to be added to fully specify the network structure and probabilities. Also, the `dieNetwork` method needs to tell the Bayesian network what evidence has been observed.

The file `BayesianNetworkTest.java` shows an example based on a Bayesian network from the textbook. One method in this file shows several examples of running a Bayesian network (that is, calling the `eliminateVariables` method). However, the `dieNetwork` method should only call `eliminateVariables` once.

The Components of `diegame.zip`

- `Interact.java` runs `DieGame.java` (the “environment”) at the same time as `DieGameSolver.java` (the “agent”).

- The first thing `DieGameSolver.java` prints is the number of games to play. `DieGame.java` then proceeds to officiate multiple games of Die.

- `BayesianNetwork.java` defines a Bayesian network as a set of variables and a set of factors. Some error checking is done. You will need to use the `addVariable` and `addFactor` and `observe` methods. Calling the other methods as needed is already coded. `BayesianNetworkTest.java` shows how examples in the textbook are implemented.

- `Variable.java` defines variables for Bayesian networks. All a variable does is keep track of a name (String) and a set of possible values (an array of Strings). You will need to create `Variable` objects in the `dieNetwork` method. There are two things to keep in mind when using `Variable` objects in this program.
  - For each Bayesian network, you should not create duplicate names for variables.
  - Generally, a value is not referred to by name elsewhere in the program. Instead, an index into the array of values is used to refer to a particular value. For example in `DieGameSolver.DIE_VALUES`, "one" is index 0, "two" is index 1, and so on.

- `Factor.java` defines factors as a set of variables and an assignment of a `double` to each combination of values of the variables. Note that each probability table maps to a factor, and that each variable in the Bayesian network corresponds to a table/factor (the probabilities for that variable based on its parents). See `FactorTest.java` and `BayesianNetwork.java` for examples from the textbook.

Here is an example of assigning a `double` to a combination of values. For example, using `D`, `S` and `E1` to denote the names of the variables for the die, a switcher variable, and the first evidence variable, then one value in the probability table is:
\[ P(E_1 = 0 \mid D = \text{five}, S = 1) = 0.2 \]

and could be formally stated in a factor \( f \) as:

\[ f(D = \text{five}, S = 1, E_1 = 0) = 0.2 \]

In `Factor.java`, these assignments are represented as indexes.

\[ f(4, 1, 0) = 0.2 \]

This is because "five" is at index 4 of `DieGameSolver.DIE_VALUES`, and values for the switcher and evidence directly correspond to the indexes.

The code to correspond to the above assignment in `DieGameSolver.java` should be:

```java
factorForEvidence1.set(0.2, 4, 1, 0);
```

where `factorForEvidence1` is a `Factor` created earlier. Note that this factor requires 24 assignments. Also note that the probability comes first followed by the indexes. This is because of how Java’s variable-length arguments feature works.

### Turning in Your Lab

Submit the folder containing all your Java files to Blackboard. Your program should run under the following conditions: after all the .class files are deleted, then:

```
javac Interact.java
java Interact
```

should solve the problems.

The score of your lab will primarily depend on the performance of the program. Over 1000 games, the initial download will average about 3.5 guesses per game. With Bayesian networks, the average should be about 1.8.

Shared Extra Credit (100 pts., equivalent to one lab, no due date). Prove that, with best play, the expected number of guesses is 1.80432.
Testing Your Program

A correct implementation will produce the following output (shown in two columns) when it is run on 10 games (using 13 as the value of the seed field in DieGame.java).

```java
class DieGameSolver 10
class DieGame 1 0
class DieGameSolver 1
class DieGame wrong try again
class DieGame 0 1
class DieGameSolver 6
class DieGame right in 2 tries
class DieGameSolver 1
class DieGame wrong try again
class DieGame 1 0
class DieGame right in 2 tries
class DieGameSolver 6
class DieGame right in 2 tries
class DieGame 0 1
class DieGameSolver 1
class DieGame wrong try again
class DieGame 1 0
class DieGame 0 1
class DieGameSolver 6
class DieGame right in 2 tries
class DieGameSolver 1
class DieGame wrong try again
class DieGame 1 0
class DieGame 0 1
class DieGameSolver 6
class DieGame right in 2 tries
class DieGame 0 1
class DieGameSolver 1
class DieGame wrong try again
class DieGame 1 0
class DieGame 0 1
class DieGameSolver 6
class DieGame right in 2 tries
class DieGameSolver 1
class DieGame wrong try again
class DieGame 1 0
class DieGame 0 1
class DieGameSolver 6
class DieGame right in 2 tries
class DieGame 0 1
class DieGameSolver 1
class DieGame 1 1
class DieGame right in 3 tries
class DieGame 0 1
class DieGameSolver 1
class DieGame 1 1
class DieGame right in 1 tries
class DieGame 0 1
class DieGame right in 2 tries
class DieGameSolver 1
class DieGame 1 1
class DieGame right in 3 tries
class DieGame 0 0
class DieGameSolver 2
class DieGame 1 1
class DieGame quit with average 2.200000
class DieGameSolver quit
```