Twenty Questions - Compsci 201

The code for this assignment is available through Snarf as well as on the course webpage.

Ambient: https://www.cs.duke.edu/csed/ambient/

INTRODUCTION:

In this assignment you will be doing several things including:

1. Read a file storing Twenty Questions game tree. The tree represents the questions and answers known to the system.
2. Keep track of the state of the game (where in the tree are you) and update it as the player answers your questions.
3. Update the tree as appropriate (i.e. when the system is unable to guess the correct answer).
4. Keep track of what you need for working communication with the view. There is more on this later in this file.
5. Write the updated game tree to a file such that it can then be used to "restore" the game knowledge later.

Background :

Twenty Questions is a classic guessing game. It goes like this: one player has a secret thing (usually a noun). The other player gets to ask yes-or-no questions in an effort to figure it out:

```
Alice: I'm ready to go.
Bob: Is it an animal?
Alice: Yes.
Bob: Is it smaller than a loaf of bread?
Alice: No.
Bob: Can you play games with it?
Alice: No!
Bob: Does it come in different colors?
Alice: No.
Bob: Is it native to Africa?
Alice: No.
Bob: Is it native to Asia?
Alice: No.
Bob: Is it green?
Alice: No.
Bob: Is it small?
```

```
Alice: No.
Bob: does it stand on two legs?
Alice: Sometimes? But not really. Let's say no.
Bob: Is it a herbivore?
Alice: No.
Bob: Can it jump?
Alice: No.
Bob: Does it make noise?
Alice: Yes.
Bob: Is it gray?
Alice: No.
Bob: Can it float?
Alice: No.
Bob: Does it have legs at all?
Alice: Yes.
Bob: Does it live in a forest?
Alice: Yes.
Bob: Does it have scales?
Alice: No.
Bob: Is it flexible?
Alice: No.
Bob: Is it worth a lot of money?
Alice: Yes.
Bob: Is it a bald eagle?
Alice: Yes!
```

As mentioned in the bizarrely exhaustive Wikipedia article, Twenty Questions is an example of a binary tree: each node represents a question, and each child an answer, either “yes” or “no”. 20q.net and other variations on the game are more-than-binary trees: they permit “maybe”, “Don’t know”, and other kinds of answers. We’ll be sticking to the simple yes-or-no version of the game.

In this assignment, you will implement a learning version of twenty questions: one that not only plays the game, but learns new objects when it loses. You’ll be able to save your learned tree to disk, so that you can exchange it with a friend, or start up again where you left off.

NOTE:
For historical reasons, the code calls the game “Animal” instead of Twenty Questions. You aren’t limited to animals! It’ll do whatever you want.

How To

Step 0:
Snarf the code. Run GameMain; observe that it doesn’t do very much yet. Read through IAnimalModel.java, which is the interface you’ll be implementing by completing AnimalGameModel.java. Also, take a look at AnimalNode.java, which implements a single node in our Twenty Questions game tree.

Step 1: File I/O

As provided, the game doesn't do very much. The first thing you'll need to implement is reading game trees from a file. When you run the game and select File -> Open File or File -> Open URL, the `initialize` method of `AnimalGameModel` is called. The Java Scanner type (what's passed as an argument) provides a `nextLine` method that will iteratively pull lines from whatever file you've opened.

To get file reading working, you'll need to do (at least) these things:

1. Add (to `AnimalGameModel`) the instance variable (or variables) necessary to store your game tree. *We recommend two AnimalNodes: one called myRoot that always points to the root of your game tree, and one called myCurrent that points to the current question. These aren't the only things you'll need to store, but they are a start.*
2. Write a recursive helper method for reading in the game tree. Note that the tree was written out using a pre-order traversal, with interior nodes marked as discussed in class. One extra thing: lines that begin with "//" should be ignored completely.
3. Have `initialize` call your recursive helper with the appropriate parameters, and set up your instance variables.

A few things to remember:

1. Lines in the file corresponding to interior nodes start with "#Q:". The text read into nodes should contain only the question and not the "#Q:" at the beginning.
2. Your helper method will return an AnimalNode. I suggest you write it by completing this code:

```
private AnimalNode readHelper(Scanner s) {
  String line = s.nextLine();
  if (...line is a leaf...) {
    // Construct a leaf AnimalNode from line, and return it.
  }
  // Make a recursive call to read the left subtree.
  // Make a recursive call to read the right subtree.
  // Construct the resulting AnimalNode and return it.
}
```

To write your game tree to a file, you'll need to implement the `write` method of `AnimalGameModel`. This method is called by the view whenever you choose File -> Save. Dealing with creating a new file is taken care of for you; all you need to do is write your tree to the provided `FileWriter` object. `FileWriters` have a `write` method that takes a `String` as input. To start a new line, write the string "\n"; this is Java's syntax for the newline character. For example:

```
// Without a new line; produces
// PotatoTomato
writer.write("Potato");
writer.write("Tomato");
// With a new line; produces
// Potato
// Tomato writer.write("Potato\n");
```

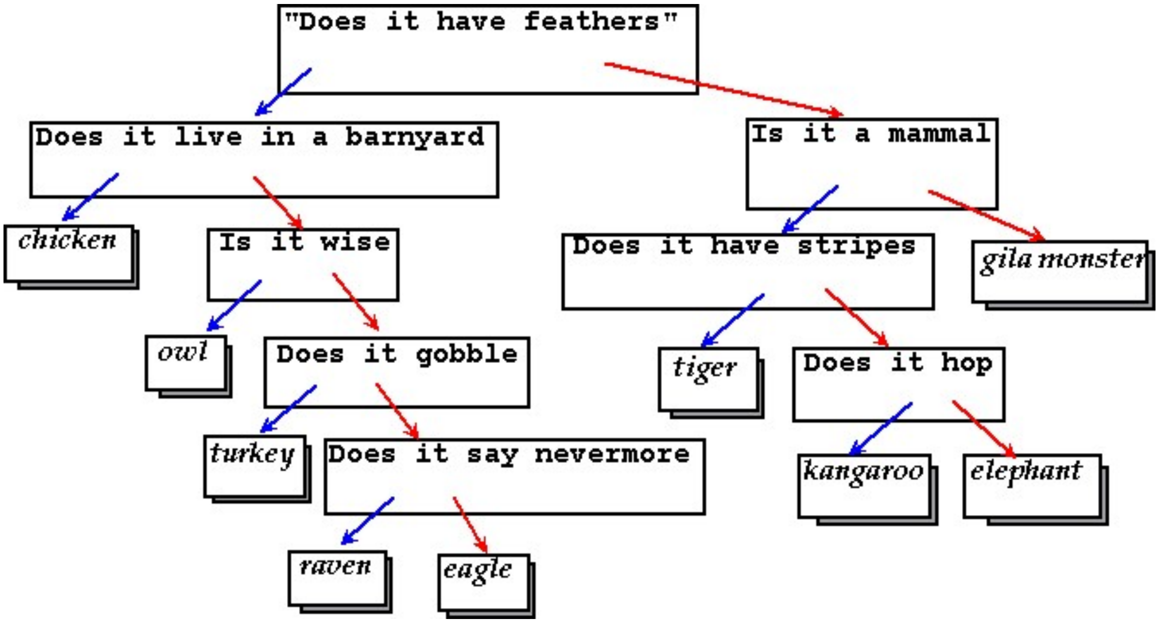
```
writer.write("Tomato");
```

Write your tree out using a pre-order traversal (to match your read-in code). You should test your write-out and read-in code by reading in the provided animals.txt, writing it out to a different file, and then comparing the two files. The files should be exactly the same.

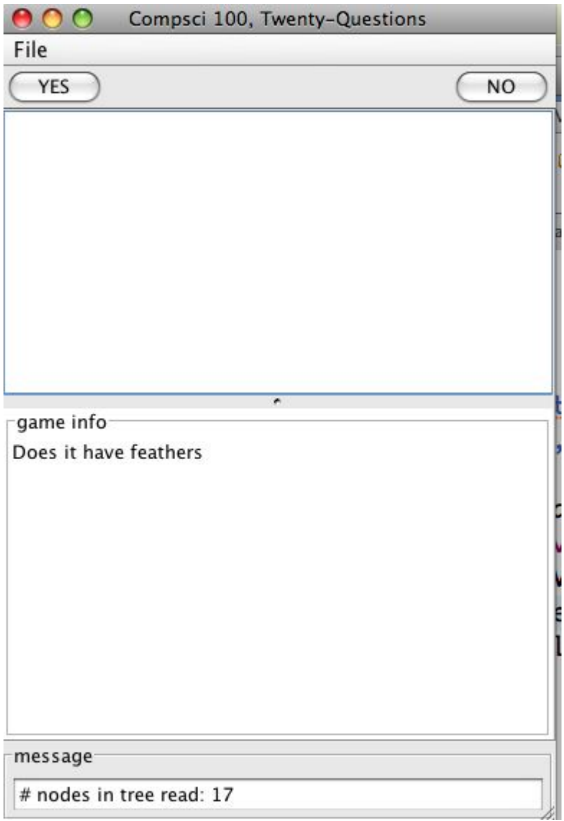
To enable the buttons on the GUI, you should call `myView.setEnabled(true)` right after you've finished loading your game tree.

Step 2: Playing Twenty Questions

Here's an example of a game tree:



This is also the tree represented by the provided animal.txt. The following is an example of somebody playing the game with this tree; in this case, the player is thinking of "chicken".

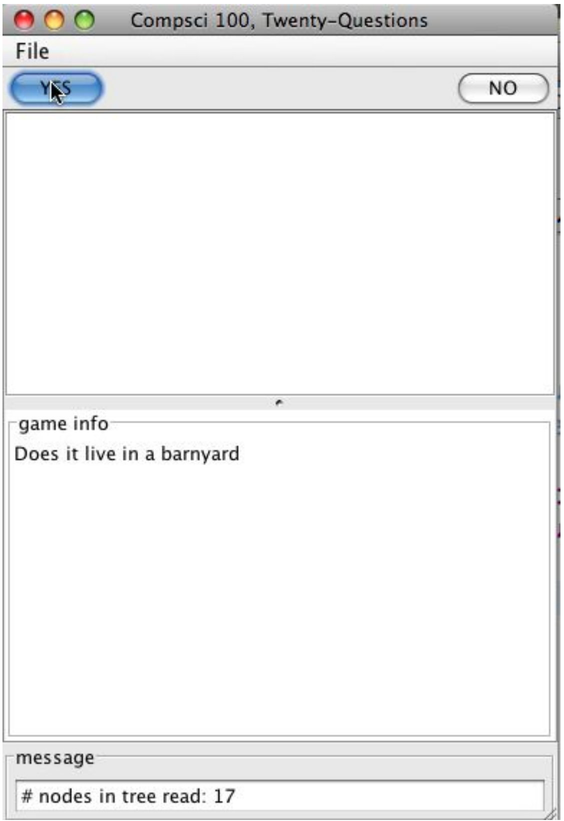


game info

Does it have feathers

message

nodes in tree read: 17

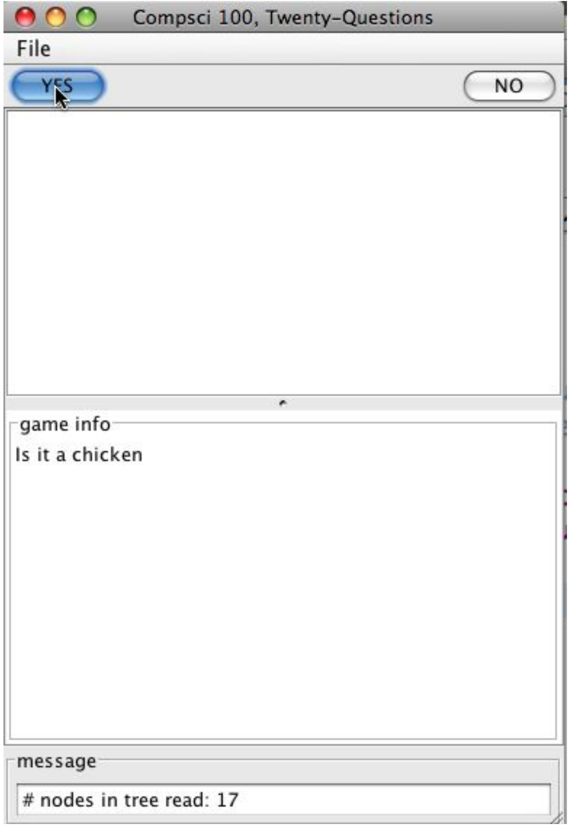


game info

Does it live in a barnyard

message

nodes in tree read: 17

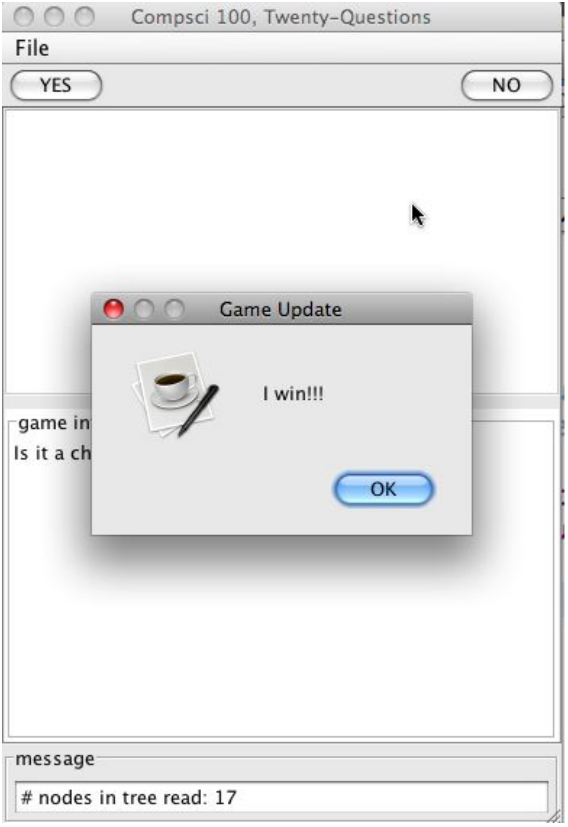


game info

Is it a chicken

message

nodes in tree read: 17



game in

Is it a ch

I win!!!

message

nodes in tree read: 17

Victory to the computer!

Understanding how the model (code you write) and the view (code we provide) interact is vital to getting this right. This is documented in IAnimalGameModel.java, and goes like this:

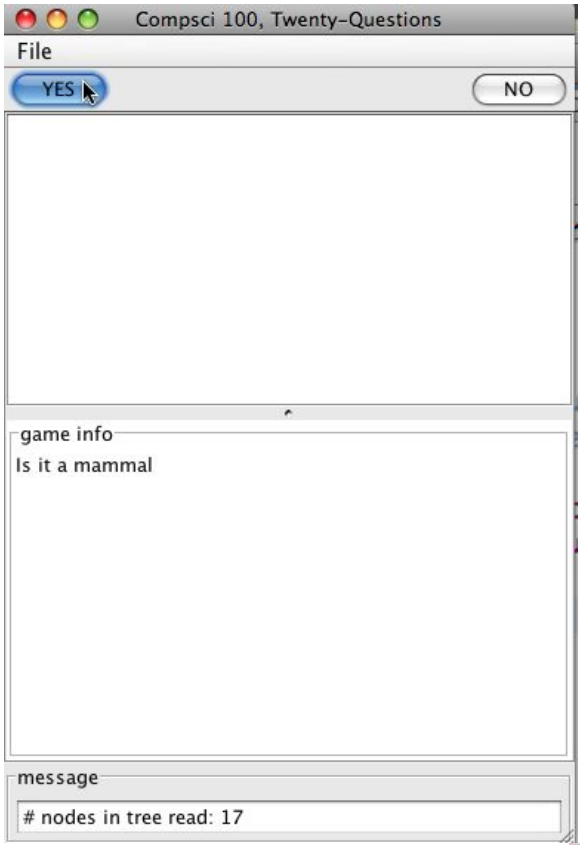
1. The player opens a file using the GUI. This calls `initialize` on the model, and is what you dealt with in part 1.
2. The player starts a new game. They can do this using File -> New Game. *Important note: loading a file should automatically start a game!* This is done by calling the model's `newGame` method, either from the view calls (player input) or automatically (as part of `initialize`). The `newGame` method will need to set up whatever instance variables the model needs to keep track of the state of the current game, and set them to initial values.
3. The model asks a question. This is done by sending a String to the view's update method, like this:

```
StringBuilder sb = new StringBuffer();
sb.append("Is it a walrus?\n");
//Note the use of "\n" for a newline!
myView.update(sb.toString());
```

4. The player clicks YES or NO. The view calls the model's `processYesNo` with the appropriate value. The model updates its internal game state (`myCurrentNode`, plus anything else you add) according to what the player said.
5. Steps 3 and 4 repeat until the computer guesses correctly, or until the computer runs out of game tree without guessing correctly. If the computer won, it should send a victory message to the view using the `showDialog` method. If it lost, things are more complicated: see below.
6. Your life will be easier if you get step 2 working before you do step 3.

Step 3: Player Victory.

What if the computer loses the game? Here's another sequence, in which I'm thinking of a lion. We'll start on the second question, after I've already said "no" to feathers.

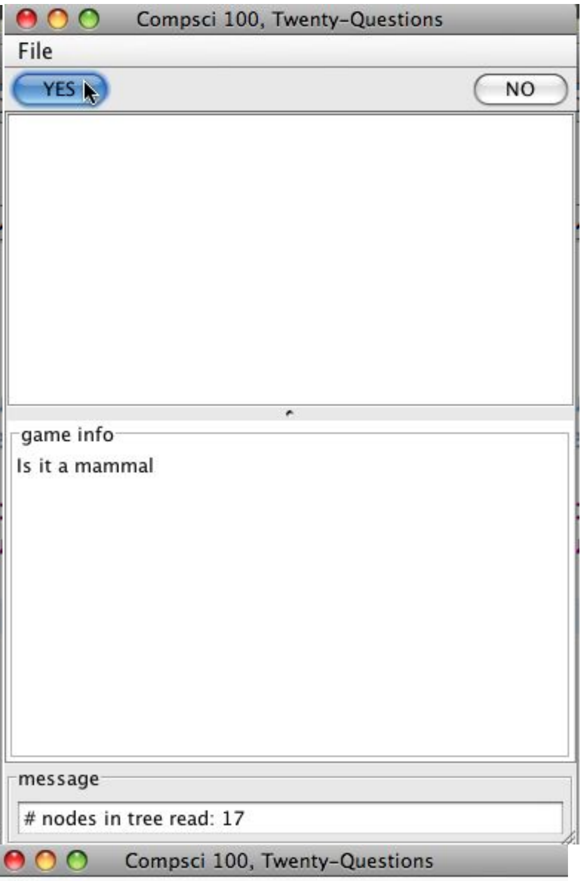


game info

Is it a mammal

message

nodes in tree read: 17

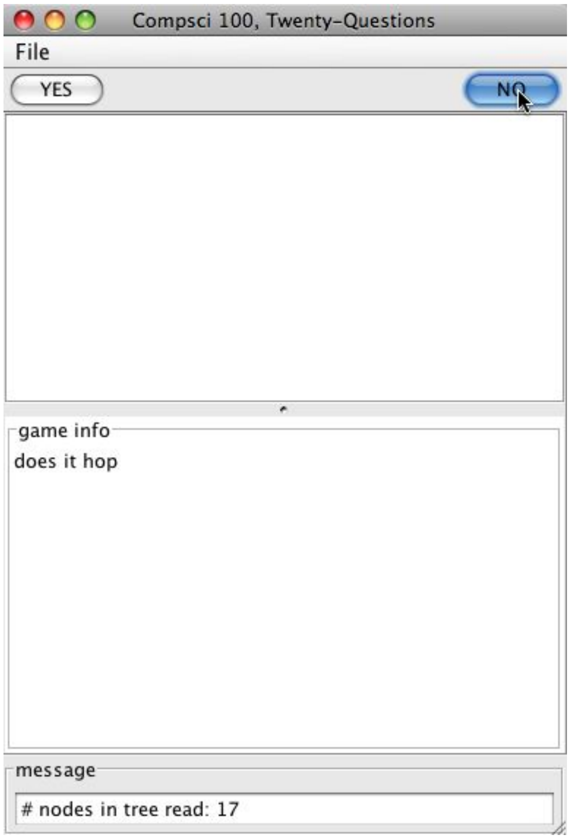


game info

Is it a mammal

message

nodes in tree read: 17

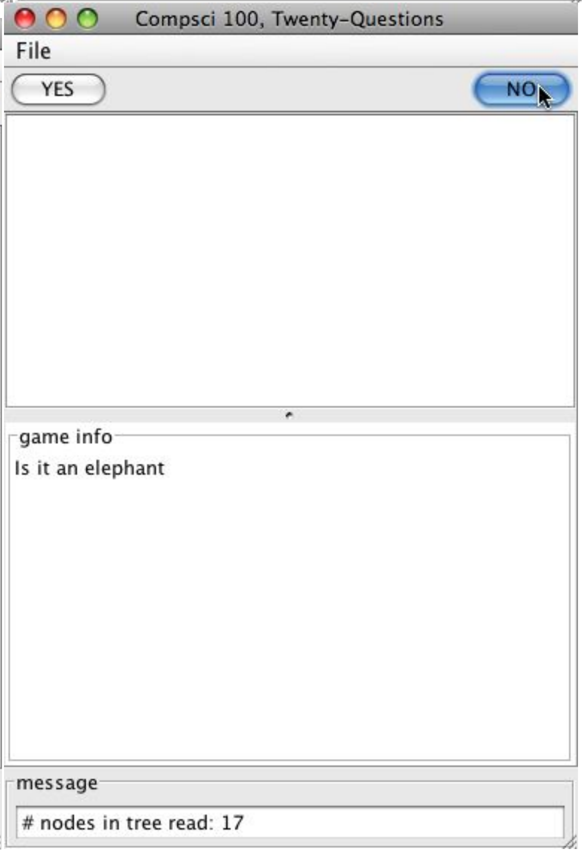


game info

does it hop

message

nodes in tree read: 17



game info

Is it an elephant

message

nodes in tree read: 17

Not an elephant!

Now something more complicated has to happen: we need to add a node to our game tree. To add a node, we need two things: the *the thing the player was thinking of* (in this case, a lion), and *a question that disambiguates that thing from the computer's guess*. Is this case, we'll use "Does it have tusks?" as elephants do, and lions don't.

NOTE:
The Yes answer to your question should always be the existing animal. In the example the question is 'Does it have tusks?' because the existing animal is Elephant and Lion is new. Elephant (the existing animal) is the Yes answer and Lion (the new animal) is the No answer.

You should follow this guideline, otherwise the UTAs will run into massive difficulty grading your assignment and you *will* lose points.

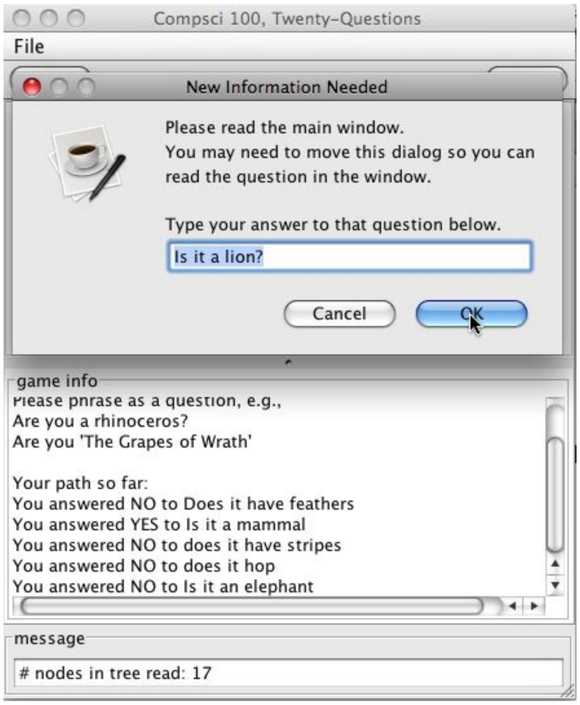
For the game to do the right thing in this case requires a fairly complex model-view interaction. First, consider how the model knows that it needs new information: it has landed at a leaf node in the tree, and still been told no. Now it needs to do several things:

1. Display the *path down the tree* to the player in the GUI. This is so that the player doesn't use a question that contradicts something that's already been seen higher up the tree. To display the path, construct a `String` storing it, and then call `myView.update` with that `String`. For example:

   ```
   StringBuilder sb = new StringBuffer();
   sb.append("You said no to feathers.\n");
   // "\n" again...
   sb.append("You said yes to mammal.");
   myView.update(sb.toString());
   ```

   Displaying the tree is going to require storing some instance variables; just knowing the root of your tree, and the current leaf, is not enough to reconstruct the path; you'll need to store it.

2. Request *new information* from the player by calling the view's `getNewInfoLeaf` method. In this case, what the right question have been (i.e. the animal you were thinking of), seen below:



Please read the main window.
You may need to move this dialog so you can
read the question in the window.

Type your answer to that question below.

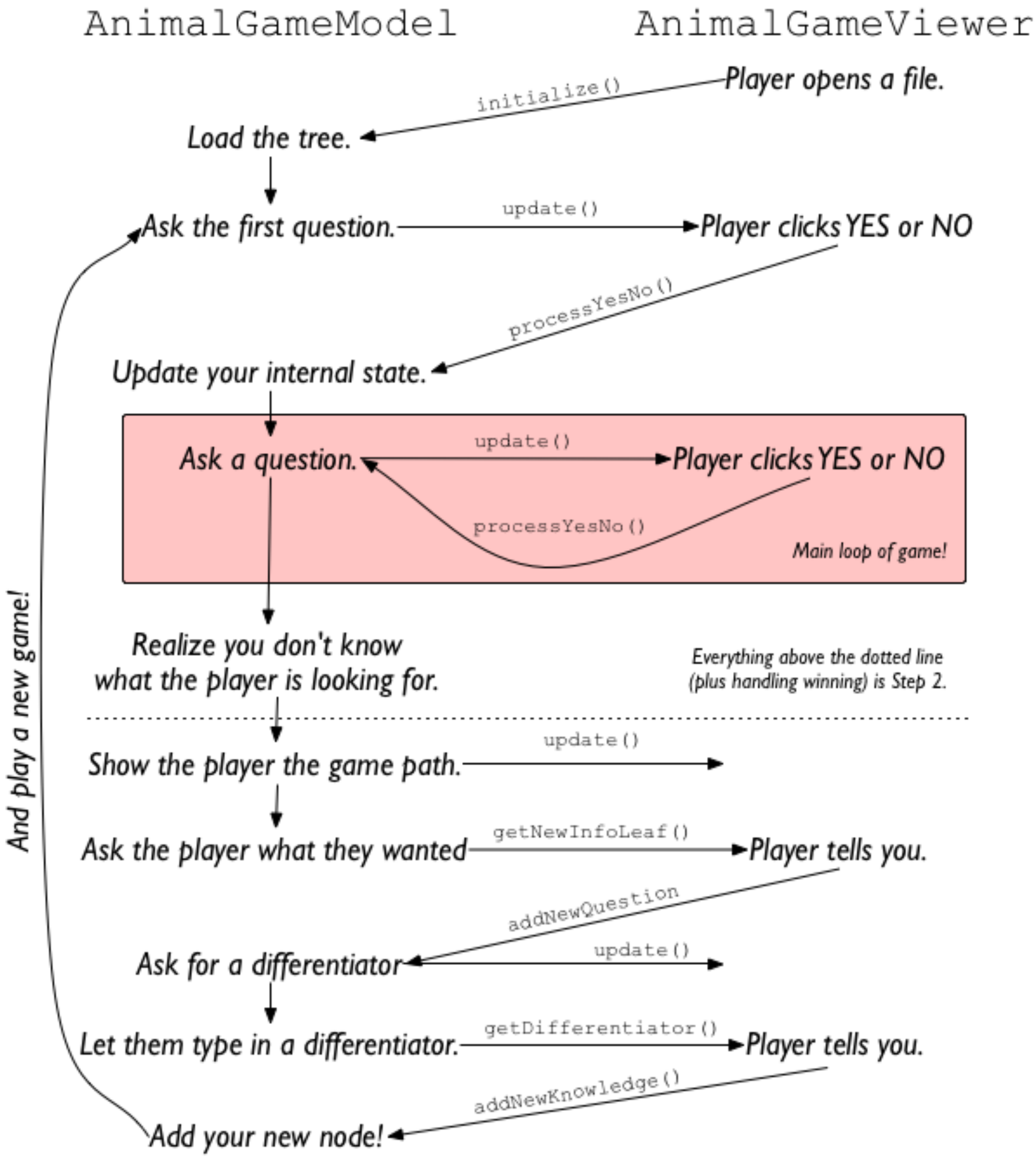
Is it a lion?

```
Please phrase as a question, e.g.,
Are you a rhinoceros?
Are you 'The Grapes of Wrath'

Your path so far:
You answered NO to Does it have feathers
You answered YES to Is it a mammal
You answered NO to does it have stripes
You answered NO to does it hop
You answered NO to Is it an elephant
```

```
# nodes in tree read: 17
```

3. Now, the tricky part. The user types in their question (“Is it a lion?”), and clicks OK. When they do this, the view (which handled the mouse click) calls the model’s `addNewQuestion` method with the typed-in String as an argument. From the model’s point of view (that is, your point of view), this looks slightly strange: you call a view method from one place in your code (where you call `getNewInfoLeaf`), and you get the answer back somewhere else (as an argument to `addNewQuestion`). It looks like you’ve jumped from one method to another; that’s because the view has jumped you there! Handling this correctly means storing some extra state in your model just before you call `getNewInfoLeaf`, so that you can use that state in `addNewQuestion`.
4. You’re halfway there. At this point, you’ve queried the user for the correct answer to their question (“lion”). Now, you’ll need to add a *differentiator*: a question that tells lions and elephants apart. This process has the same flavor as the previous step. First, the model calls the view’s `update` to ask for a differentiator. Then the model calls `getDifferentiator`, which pops up a dialog box for the player to type into. The model then calls the view’s `addNewKnowledge` with whatever the player typed. As before, the model calls a view method one place, and gets the result back somewhere else. The figure below shows the flow back and forth between the model and view over the course of the game; it may help clear things up.
5. Finally, once you’ve added a new node, start a new game!



This flow chart follows exactly the path of calls your program should be making.

Submitting

Submit your code and README.txt using either the Ambient plugin, or Ambient web submit (https://www.cs.duke.edu/csed/websubmit/app/). You should submit to the 20Q project.

Incorrect submissions will incur a penalty.

Code Style

Fact: code is hard to read.

How you layout your code, and your program, make a big difference in how easy it is to understand. There are (at least!) four people who need to understand your code: you (while writing it), anybody you ask for help, your grader, and you (six months from now, when you look at it to figure out how you did something). While the following rules are not set in stone, they provide a good place to start:

- Think about how your code is going to be organized *before* you've written it. Good design matters!
- Give your classes, variables, and methods descriptive names. Naming your variables foo and bar doesn't say much; xPosition and yPosition are much better.
- Use Java's conventions on naming. ClassesAreNamedLikeThis: words are run together, and each word is capitalized. Similarly, methodsAreNamedLikeThis: words run together, and all but the first word are capitalized. Variables are named like methods.
- Indent your code properly. Roughly speaking, this means "everything inside a curly brace gets indented one extra level." See the code we provide for how this should look. To make this easy, Eclipse can do it for you: select your code with the mouse, and then do Source -> Correct Indentation. Using this is the best last thing to do before submitting.
- Don't let your lines get too long. Although Java doesn't care how long your lines are, long lines are *much* harder to read. The closest thing there is to a universal standard is 80 characters. Eclipse makes this easy, too: in Preferences -> Text Editors, turn on "Print Margin" and set it to 80. That will add a vertical line at the 80-character mark.

Remember: easy-to-understand code makes your grader happy. Happy graders are friendly graders!

Grading

These are some of the things that could be graded on this assignment.

- File I/O - 5 points.
    - Reading the file.
    - Writing the file.
    - Dealing with '#Q'.
- The Game - 8 points.
    - Tree Traversals.
    - Correct game messages.
    - Adding knowledge.
- Code Style - 2 points.
    - See above.