A7: 20 Questions

due Thursday, August 11 11:59pm

This assignment will assess your mastery of the following objectives:

- Implement a well-designed Java class to meet a given specification.
- Implement, manipulate, and travese a binary tree.
- Follow prescribed conventions for code quality, documentation, and readability.

Overview: The Game of 20 Questions

Twenty Questions is a guessing game in which one player chooses a secret object and the other player asks yes/no questions to try to idenfity the chosen object. In our version, the human will be the chooser and begin a round by choosing an object. The computer will be the guesser and attempt to guess that object by asking a series of yes/no questions until it thinks it knows the answer. Then, the computer makes a guess; if its guess is correct, the computer wins, and otherwise the human player wins. If the computer loses, it will add the chosen object to its knowledge base so it will be able to guess it the next time it plays.

Program Behavior

In this assessment, you will create a class named `QuestionsGame` to represent the computer's tree of yes/no questions and answers for playing games of 20 Questions. You will also create a inner-class named `QuestionNode` to represent the nodes of the tree. You are provided with a client `QuestionMain.java` that handles user interaction and calls your methods from `QuestionsGame` to play the game.

QuestionNode

The contents of the `QuestionNode` class are up to you. Though we have studied trees of `int`s, your nodes should be specific to solving this problem. Your `QuestionNode` class should have at least one constructor used by your tree. Don't include constructors that are not actually used in your program. Your `QuestionNode` class must be a `private static` inner class within `QuestionsGame`. Your node's fields *must* be public. `QuestionNode` should not contain *any actual game logic*. It should only represent a single node of the tree. For reference, you can look at the `IntTreeNode` class from lecture.

QuestionsGame

This class represents a game of 20 Questions. It stores a binary tree whose nodes represent questions and answers. (Every node's data is a string representing the text of the question or answer.) Note that even though the name of the game is "20 questions", the computer will *not* be limited to only *twenty*; the tree may have a larger height.

You should not limit the size or shape of the tree in any way!

The *leaves* of the tree represent possible answers (guesses) that the computer might make. All the other nodes represent questions that the computer will ask to narrow the possibilities. The left branch indicates the next question the computer asks if the answer to the current question is *yes*, and the right branch is the next question if the answer is *no*. The game is played by starting at the root and asking the questions at each node, travelling down the the tree based on the user's answer. Once a leaf node is reached, the computer will ask if that answer is the correct one. Page 3 walks through a full example of a game.

In addition to adding questions to the tree as games are played, your class will also be able to read a pre-existing tree from a text file, or write the current tree out to a text file to save for later. These files will have a specific format that you must follow for both reading and writing (see below).

Your `QuestionsGame` class should have the following constructor:

```
public QuestionsGame()
```

This constructor should initialize a new `QuestionsGame` object with a *single leaf node* representing the object `"computer"`.

Your `QuestionsGame` class should also implement the following public methods:

```
public void read(Scanner input)
```

This method will be called if the client wants to replace the current tree by reading another tree from a file. Your method will be passed a `Scanner` that is linked to the file and should replace the current tree with a new tree using the information in the file. Assume the file is legal and in standard format (see below). Make sure to read entire lines of input using the `nextLine` method.

```
public void write(PrintStream output)
```

This method should store the current questions tree to an output file represented by the given `PrintStream`. This method can be used to later play another game with the computer using questions from this one. The file should be written using the standard format (see below).

```
public void askQuestions()
```

This method should use the current question tree to play one complete guessing game with the user, asking yes/no questions until reaching an answer object to guess. A game begins with the root node of the tree and ends upon reaching an answer leaf node.

If the computer wins the game, this method should print a message saying so.

Otherwise, this method should ask the user for the following:

- what object they were thinking of,
- a question to distinguish that object from the player's guess, and
- whether the player's object is the yes or no answer for that question.

Your method output should exactly match the format shown in the examples in the spec.

```
private boolean yesTo(String prompt)
```

This method asks the given question until the user types "y" or "n". Returns true if "y", false if "n".

This method is provided for you and should not be modified!

Implementation Guidelines

User Input: Yes and No

At various points in this assignment, you will need to get a yes or no answer from the user. You must construct a **single console `Scanner` attached to `System.in` that you store in a field called `console` and use throughout your class**. All input read from this `Scanner` should use the `nextLine` method.

To help with asking these questions, you are provided a method called `yesTo`. This method assumes there is a field called `console` that has been initialized with a Scanner as decribed above. You should include this method *without modification* in your `QuestionsGame` class and use it whenever you ask the user to answer a yes/no question. **The code for `yesTo` is included in the file `yesTo.txt`** - you should copy this method into your code.

Question Tree File Format

The `read` and `write` methods will be interacting with text files containing questions and answers from 20 questions games. These files will follow a standard format.

A single `QuestionNode` will be represented as a non-empty sequence of line pairs. The first line of the pair will contain either "Q:" or "A:" to differentiate between questions (branches) and answers (leaves). The second line of the pair should contain the text for that node (the *actual* question or answer). You may assume that the lines containing "Q:" or "A:" will contain exactly that text (case-sensitive) and no other text. You may also assume the file contains an even number of lines, exactly following this format. The nodes of the tree will appear in the file following a *pre-order* traversal (i.e. the overall root of the tree will be the first node in the file).

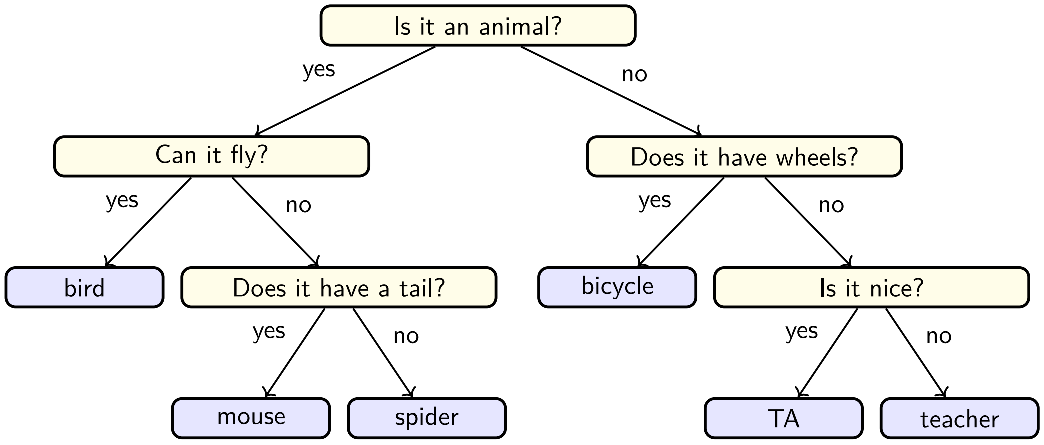
You should both assume that any files passed to your `read` method **AND** ensure that any files you create in the `write` method follow this format. The `readTree` and `writeTree` methods from section will be *very helpful* in writing your `read` and `write` methods for this assessment.

Sample Walk-Throughs

Here is an example question file, the associated tree, and a sample game played with this tree:

Notice that the file represents a *pre-order* traversal of the tree.

questions.txt
```
Q:
Is it an animal?
Q:
Can it fly?
A:
bird
Q:
Does it have a tail?
A:
mouse
A:
spider
Q:
Does it have wheels?
A:
bicycle
Q:
Is it nice?
A:
TA
A:
teacher
```



Sample game (computer wins)
```
Welcome to the cse143 question program.

Do you want to read in the previous tree? (y/n)? y

Please think of an object for me to guess.
Is it an animal? (y/n)? n
Does it have wheels? (y/n)? y
Would your object happen to be bicycle? (y/n)? y
Great, I got it right!

Do you want to go again? (y/n)? n
```

Initially, the computer is not very good at the game, but it improves each time it loses. If the computer guesses incorrectly, it asks you to give it a new question to help in future games. For example, suppose in the preceding log that the player was thinking of a car instead. That game might look like this:

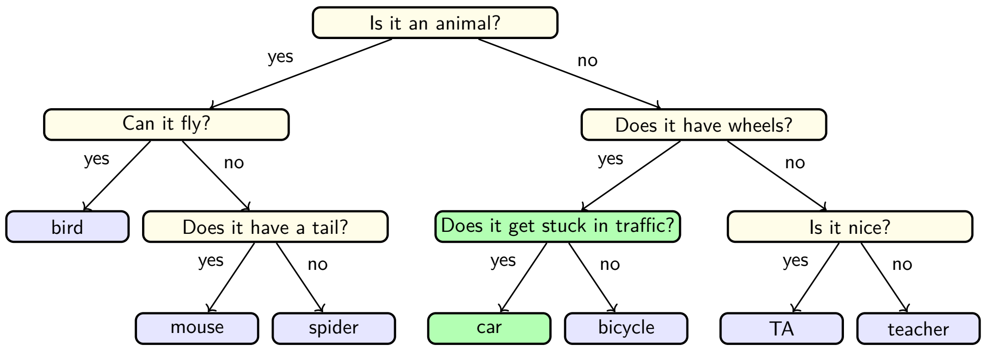
Sample game (computer loses)

```
Welcome to the cse143 question program.

Do you want to read in the previous tree? (y/n)? y

Please think of an object for me to guess.
Is it an animal? (y/n)? n
Does it have wheels? (y/n)? y
Would your object happen to be bicycle? (y/n)? n
What is the name of your object? car
Please give me a yes/no question that
distinguishes between your object
and mine--> Does it get stuck in traffic?
And what is the answer for your object? (y/n)? y

Do you want to go again? (y/n)? n
```

The computer takes the new information from a lost game and uses it to replace the old incorrect answer node with a new question node that has the old incorrect answer and new correct answer as its children. After the preceding game, the computer's game tree would be the following:

questions.txt

```
Q:
Is it an animal?
Q:
Can it fly?
A:
bird
Q:
Does it have a tail?
A:
mouse
A:
spider
Q:
Does it have wheels?
Q:
Does it get stuck in traffic?
A:
car
A:
bicycle
Q:
Is it nice?
A:
TA
A:
teacher
```



Note that QuestionMain will always read and write to a file named questions.txt. If you want to start with the tree from big-questions.txt, then you should copy the contents of those files to a file named questions.txt. Be careful, since the program will write the tree to this file every time.

Development Strategy

We suggest that you develop the program in the following stages:

(1) Before you begin, you should write "stubs" for the required methods so that you will be able to test using `QuestionMain`. Write "dummy" methods that do essentially nothing as placeholders so `QuestionMain` will be able to compile and run.

(2) First, you should decide what fields belong in the `QuestionNode` and `QuestionsGame` classes. Once you've chosen the fields, you should implement the full `QuestionNode` class and the constructor for `QuestionsGame`.

(3) Next, you should implement `write` (outputting a tree is easier than reading one). Make sure to look back at `writeTree` from section!

(4) Then, you should implement `read` which reads in a question tree. Make sure to look back at `readTree` from section!

(5) Finally, you should implement `askQuestions`. At this point, you'll be able to play the game. When you play, you can add questions one by one and play with your game to check if it's working.

Code Quality Guidelines

In addition to producing the behavior described above, your code should be well-written and meet all expectations described in the General Style Deductions, Style Guide, and Commenting Guide. For this assessment, pay particular attention to the following elements:

`x = change(x)`

An important concept introduced in lecture was called `x = change(x)`. This idea is related to proper design of recursive methods that manipulate the structure of a binary tree. You should follow this pattern where necessary when modifying your trees.

For example, at the end of a game lost by the computer, you might be tempted to "morph" what used to be an answer node of the tree into a question node by directly modifying its fields. This is considered bad style because question nodes and answer nodes are fundamentally different kinds of data. You can rearrange where nodes appear in the tree, but you shouldn't turn a answer node into a question node just to simplify the programming you need to perform. Instead, you should create or rearrange nodes as needed. (This is similar to `AssassinManager`, where you were also rearranging rather than "morphing" nodes.)

Recursion

For this assessment, you **MUST** implement your algorithms recursively, and you must *not* use any loops. Don't create special cases in your recursive code if they are not necessary. Avoid repeated logic as much as possible.

Avoid Redundancy

Create "helper" method(s) to capture repeated code. As long as all extra methods you create are `private` (so outside code cannot call them), you can have additional methods in your class beyond those specified here. If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code.

Data Fields

Properly encapsulate your objects by making data fields in your `QuestionGame` class `private`. (Fields in your `QuestionNode` class should be `public` following the pattern from class.) Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place. Fields should always be initialized inside a constructor or method, never at declaration.

Commenting

Each method should have a header comment including all necessary information as described in the Commenting Guide. Comments should be written in your own words (i.e. not copied and pasted from this spec) and should not include implementation details.

Running and Submitting

While developing your code, you should work locally on your computer in jGRASP. To test your code, you should run the provided test files and copy the printed logs into the Output Comparison Tool to check against the expected output.

If you believe your behavior is correct, you can submit your work by uploading your files and clicking the "Mark" button in the Ed assessment. You will see the results of some automated tests along with tentative grades. **These grades are not final until you have received feedback from your TA**.

You should treat Ed as a final check for your correctness, not as a debugging tool. If you aren't passing the tests, you should go back to jGRASP to debug your work.

You may submit your work as often as you like until the deadline; we will always grade your most recent submission. If you submit a version that you later decide you do not want to have graded, **you must warn your TA not to grade that version and to wait for a later submission from you.**

Getting Help

If you find you are struggling with this assessment, make use of all the course resources that are available to you, such as:

- Reviewing relevant examples from class
- Reading the textbook
- Visiting the IPL or Taylor's office hours
- Posting a question on the Ed discussion board

Collaboration Policy

Remember that, while you are encouraged to use all resources at your disposal, including your classmates, **all work you submit must be entirely your own**. In particular, you should **NEVER** look at a solution to this assessment from another source (a classmate, a former student, an online repository, etc.). Please review the full policy in the syllabus for more details and ask the course staff if you are unclear on whether or not a resource is OK to use.