CS 1110, LAB 9: ENCAPSULATION AND DATA

Name: ____________________ **Net-ID**: __________

There is an online version of these instructions at

http://www.cs.cornell.edu/courses/cs1110/2012fa/labs/lab9

You may wish to use that version of the instructions.

his lab gives you some experience with encapsulation and abstraction. It also contains a recursion problem, to give you some more practice for the upcoming exam. The concepts of this lab were, believe it or not, part of assignments A1 and A3 in previous semesters of this course.

The lab this week is a little longer than the last one. But you should definitely try to finish it before next week because it provides some good practice for the exam (particularly the recursion question).

Requirements For This Lab. The very first thing that you should do in this lab is to download the file `lab9.py` from the course web page:

http://www.cs.cornell.edu/courses/cs1110/2012fa/labs/lab9/lab9.py

You will note that this file ontains a class `Person` as a well as a function `test`. The `test` function is essentially a unit test for this lab. In many cases, it is not necessary to have a separate file for a unit test; you can just put the test in the same file as everything else. To run the unit test on this class, simply type

```
python lab9.py
```

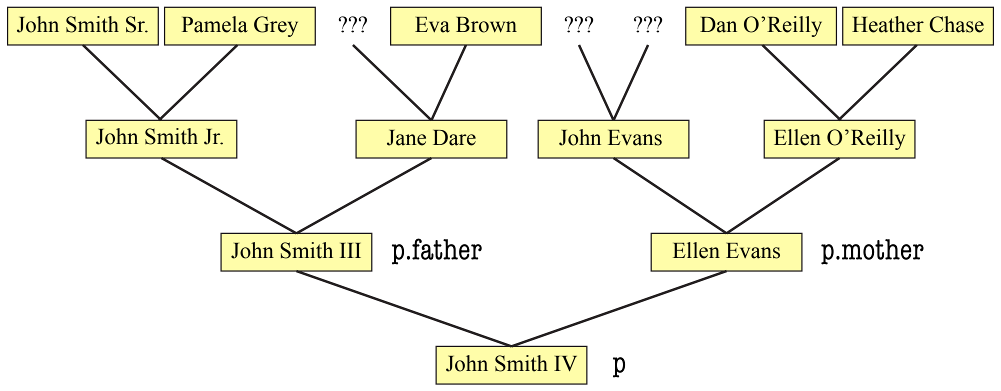
to run the module as an application. If you run it right now, you will see that it fails right away, because `Person` is incomplete.

In addition, we are using simple assert statements for our unit test, not `assert_equals` from `cunittest`. This is also okay; it cuts down on the number of files we give you. Outside of this class, in the real world, this would be an acceptable way to construct a unit test.

In this lab, you will modify the class `Person` in `lab9.py`, You do not need to modify the unit tests, just the class. When you are done, simply show these two files your lab instructor. As always, you should try to finish the lab during your section. However, if you do not finish during section, you have **until the beginning of lab next week to finish it**. You should always do your best to finish during lab hours; remember that labs are graded on effort, not correctness.

1. Lab Instructions

The class `Person` is meant to represent an entry into a geneological database. The object stores the name of the person, as well the mother and father, and the children for that person. The mother and father are additional Person objects stored in fields (much like the Worker object could store another Worker object in its `boss` fields). The `children` attribute is a list of Person objects. When you hook Person objects together, you will get a family tree much like the picture shown below.



If you look at the class `Person`, you will see hidden fields for all of these attributes, as well as properties that provide getters (and sometimes setters) for each of them. Most of the properties are immutable, except for `mother` and `father`. And the setters for those are not yet implemented.

Task 1: Complete the setters for `mother` and `father`.

Note that while `mother` and `father` are mutable, children is not. That is because we need these values to be consistent with one another. If object `q` is in the `mother` field of object `p`, then object `p` should be in the list of children of object `q`.

This is very similar to the problem with farenheit and celsius from the `Temperature` class shown in class. You are used to invariants being limited to the type or range of a single attribute. But invariants can also specify the relationship between two attributes. In that case, the setter for each attribute should ensure that the relationship between the two attributes remains preserved.

In this example, when we change either the `mother` or the `father` attribute, we need to change the children *in the `mother` or `father` as well*. We have spelled out how to do this in the comments of these setters. The final code should five lines long; each comment corresponds to a line of code. The code for each setter will be identical except for `self._mother` being replaced by `self._father` and vice versa.

Even though the comments spell out how to do this, here are some more hints to make this go quickly.

Use `list` methods to modify the `children` attribute. You want to use the following to methods to add or remove children:

Method	Description
`children.append(p)`	Adds `p` to this list of `children`.
`children.remove(p)`	Removes the first occurence of `p` from this list.

In the case of `remove`, each child should appear at most once in the list of children. Therefore, a single `remove` will remove all occurrences.

Do not use `children`; use `_children`. You will notice that every time you call the getter for `children` **it makes a copy of the list**. So if you try to use

```
self.children(p)
```

it will add to the *copy* and not the original list of children. If you want to add to the actual list of children, use the field `_children` instead.

Why do we do this? Because we want `children` to be immutable. Even though there is no setter for `children`, you could still modify the *contents* of `children` since a list is itself mutable. This negates the whole purpose of making `children` immutable. So the getter gives a copy of `_children`, which can be modified without affecting this object. Modify the original, not the copy.

Remember to modify the children of the parents. You might be tempted to write the following:

```
self._children.append(self)
```

If you do this, you are claiming that the current object (`self`) has itself as its own child. While this is the subject of many interesting science fiction novels, it is not what we want here. Modify the `_children` fields *in the parents.*

The setter has to handle None. `None` is a valid value for a `mother` or `father`. `None` means that the appropriate parent is unknown. Even if we had an actual value for that parent at one time, we may reassign it to `None` later (perhaps on the revelation that this person was adopted). You have to be prepared for this, as you cannot access the `_children` field in `None`

Testing it Out. To test out your code, simply run `lab9.py` as an application. If you have done it correctly, you should see the following output:

```
Family Tree
-----------
Smith,John [0 kids]
   Evans,Ellen [1 kid]
      O'Reilly,Ellen [1 kid]
         Chase,Heather [1 kid]
         O'Reilly,Dan [1 kid]
      Evans,John [1 kid]
   Smith,John [1 kid]
      Dare,Jane [1 kid]
         Brown,Eva [1 kid]
      Smith,John [1 kid]
         Grey,Pamela [1 kid]
         Smith,John [1 kid]

Properties mother and father working correctly
```

The program should then fail immediately afterwards. That is okay; that error message is about the test for the next task.

Task 2: Complete the method `familyNamed`.

Recall the famiy tree created when we link objects together, which was illustrated on the previous page. Up until now, our use of recursion has been limited to sequences. But we can use recursion on this family tree as well. Note that "ancestor" has a recursion definition.

- The mother or father of a Person an ancestor
- The mother or father of an ancestor is an ancestor

Given this recursive definition, we can use recursion to define the following function.

```
def familyNamed(self,name):
    """Returns:  number of family with name as first name.

    The number of family members includes this person
    (e.g.  self) as well as all family members.  You should
    implement this method recursively.

    Precondition:  name is a string."""
```

In the family tree shown on the previous page, `p.familyNamed('John')` is 5 (remember to count `p` as well), while `p.familyNamed('Ellen')` is 2.

Write this recursive function. In order to figure out how to do this, you might find the method `fullstr` above it to be quite useful. This is a recurisve procedure that creates a string representing the entire family tree, indenting four spaces at each generation. In fact, you should recognize it, as it is what produced the output in the previous task.

When you are finished, run `lab9.py` again as an application. If you did it correctly, the output should end with the message

```
Method familyNamed working correctly
```

At this point show your `lab9.py` to your instructor.