CS61A Week 8 | Where The Objects Roam (v1.0)

Paradigm Shift (or: The Rabbit Dug Another Hole)

And here we are, already ready to jump into yet another programming paradigm - object-oriented programming. To those used to Java, this will be a pretty straightforward section; to those not, OOP does take a bit of getting used to. However, I've always regarded OOP as one of the easiest parts of CS61A - and this is because OOP is often the most natural way to approach certain problems.

We will now model the world as many objects, each with its own properties and a set of things it can do. Recall that we started out with smart procedures; given some piece of argument, they know what to do and what to return. Then, we talked about data-directed programming, where we took the smarts away from the procedures and stored them, Big-Brother-like, into this gigantic table; in that case, both the procedures and the data are dumb. Now, we're going to place the smarts entirely within the data; the data themselves know how to do things, and we merely tell them what to do.

Being in a new programming paradigm requires some new syntax, and there's lots of that in the OOP section in your reader. Read over that carefully and familiarize yourself with your new friends. Those notes are all reasonably clear, so I'm not going to repeat them here.

One thing that often confuses people is the difference between classes and instantiations. A **class** is a set of methods and properties defined using `define-class`. But defining a class doesn't give you an **instance** of that class, however; after you define a `person` class, you still don't have a `person` until you *instantiate* one - through a call to `instantiate`. The important difference is that, whereas a class is just an abstract set of rules, an instance is something real that follows those defined rules.

Three Ways of Keeping State (or: Memento)

- **instantiation variables** - these are variables that you pass into the `instantiate` call; they're remain available to you throughout the life of your object.
- **instance variables** - these are variables associated with each instance of a class; instantiation variables are really just instance variables. You declare these with `(instance-vars (var val) (var val))`
- **class variables** - these are variables associated with a whole class, not just a single instance of the class. Sometimes it makes more sense to keep certain information in the class rather than in each instance (for example, it doesn't make much sense to have each `person` object keep track of how many persons have been created; the class should do that. You declare these with `(class-vars (var val) ...)`

Inheritance (or: We are all extensions of our parents)

A powerful idea in OOP comes from the observation that most things are alike in some way. In particular, some things are a *kind* of other things. For example, I am a student, which is a kind of human, which is a kind of animal. Such objects in the same "family" may all have similar abilities (all of them, say, eat), but may do them differently (an animal may chew, while I certainly do not).

More details, of course, are in the documentations in the reader; let's jump into some examples.

Midterm Fun (or: No, Seriously)

Suppose we want to simulate you taking a midterm (a rather suitable topic, I thought).

First, of course, we need an object that represents a question:

```
(define-class (question q a hint weight) ...)
```

Where `q` is a list that is the question, `a` the answer, `hint` a hint, and `weight` the number of points the question is worth. For example, here are the two questions for our midterm:

```
(define q1 (instantiate question '(what is 2+2?) '(5) '(a radiohead song) 10))
(define q2 (instantiate
            question '(how cool is 61a?) '(extremely) '(I like 61A) 90))
```

Your midterm will probably be a little bit harder.

Now, there are several things you can do with a question:

```
(ask q1 'read) ==> '(what is 2+2?) ;; read the question
(ask q1 'answer '(17)) ==> doesn't return anything ;; answer the question
(ask q1 'cur-answer) ==> '(17)
(ask q1 'grade) ==> 0 ;; earn no point for this question
(ask q1 'answer '(5))
(ask q1 'grade) ==> 10 ;; earn 10 points for this question
(ask q1 'hint 'some-password) ==> '(wrong password! I hope you are proud)
(ask q1 'hint 'redrum) ==> '(a radiohead song)
```

Note that you need a password to ask for a hint; therefore, this option is available only to proctors, not to students.

QUESTIONS:

1. Implement all of the above functionalities for the `question` class.

2. Now, there's a special kind of `question`, a `bonus-question`. It is designed to be so hard, illogical and obscure that it cannot possibly be solved and no answer earns you any point. Therefore, it really only needs to take in one instantiation argument: the question. It also gives no hints. For example, here's one:

```
(define q3 (instantiate bonus-question
    '(explain the popularity of britney spears)))

(ask q3 'hint 'redrum) ==> '(a bonus question gives no hints)
```

   Implement the `bonus-question` class to inherit from the `question` class, using minimal code.
   `(define-class (bonus-question q)`

3. We also have a `midterm` class; it's just a collection of questions:

```
(define-class (midterm q-ls)
    (method (get-q n)
        (if (> n (- (length q-ls) 1))
        '(you are done)
         (list-ref q-ls n)))
    (method (grade) ...))
```

   Where you can get the nth question of the midterm by using the `get-q` method. Implement the `grade` method for the `midterm` class that calculates the total grade of a midterm.

So you are now ready to make our midterm:

```
(define m (instantiate midterm (list q1 q2 q3)))
```

Of course, while you do so, there will be proctors walking around. Consider this `proctor` class:

```
(define-class (proctor name)
   (method (answer msg) (append (list name ':) msg))
   (method (get-time) (random 100))
   (method (how-much-time-left?)
        (ask self 'answer (list (ask self 'get-time))))
   (method (clarify q) ...))
```

So if we have a `proctor`,

```
(define jeff (instantiate proctor 'jeff))
```

You can either ask `jeff` how much time is left (in which case, of course, he picks a random answer from 0 to 100), or you can ask him to clarify a question (in which case he answers with a `hint` for the question).

QUESTIONS:

1. What would be returned from the call `(ask jeff 'how-much-time-left?)`?

2. Implement the `clarify` method. The password, as you saw, is "redrum".

3. Now, there's a different kind of `proctor`, of course - a `professor`. A `professor` exhibits the following behavior:

   ```
   (define brian (instantiate professor 'brian))
   (ask brian 'how-much-time-left?) ==> ;; ALWAYS answers 30
   (ask brian 'clarify q1)
        ==> ALWAYS answers (the question is perfect as written)
   ```

   Implement the `professor` class with minimal code.
   `(define-class (professor name)`

4. Another kind of `proctor` is a `ta`. A `ta` behaves just like a normal `proctor`, but with a temper. That is, ask him too many questions, and he'll start being rude. The `ta` takes in a `temper-limit` as an instantiation variable, and increments his temper by one every time he answers a question.

```
(define wei (instantiate ta 'wei 3))
(ask wei 'how-much-time-left?)
(ask wei 'how-much-time-left?)
(ask wei 'how-much-time-left?)
(ask wei 'how-much-time-left?) ==> answers (how the hell would I
know?)
```

Implement the `ta` class with minimal code.

```
(define-class (ta name temper-limit)
```

5. Lastly, we want a `lenient-ta` class. A `lenient-ta` also takes in two `proctor` objects upon instantiation, and when asked how much time is left, always gives the more generous answer of the two `proctor` objects.

   ```
   (define kevin (instantiate lenient-proctor 'kevin brian jeff))
   ```

   Implement the `lenient-proctor` class with minimal code.

   ```
   (define-class (lenient-proctor name p1 p2)
   ```