CS61A Notes – Week 4a: Object-oriented programming

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 simpler 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)

As you saw in the reading, there are three kinds of variables:

1. **instantiation variables** – these are variables that you pass into the instantiate call; they're remain available to you throughout the life of your object.
2. **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))`.
3. **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) ...)`.

Taste the Rainbow (or: Dinner is not Ready)

Using objects to represent Skittles is kind of a classic (I'm really not sure why). But it's easy enough, so let's try one!

Consider this class `skittle` (the singular of "skittles", of course):

```
(define-class (skittle color))
```

This is a class with no methods (how many interesting things can a skittle do?) and a single property – its color. Now, we'd like to hold skittles in a bag, so let's define a bag class:

```
(define-class (bag)
   (instance-vars (skittles ‘()))
   (method (tag-line) ‘taste-the-rainbow)
   (method (add s) ...) ;; adds a skittle s to the bag
   (method (take) ...)) ;; takes a skittle from the bag
```

A bag object will be able to hold multiple skittles, and you can `add` or `take` skittles to or from the bag.

QUESTIONS

1. **Implement the `add` and `take` methods.**

2. **Implement a `(take-color color)` method that takes a skittle of the specified color from the bag. You can use `find` or `remove`.**

Directories and Files (Again)

Consider a `directory` class that can hold multiple directories and files:

```
(define-class (directory name)
   (instance-vars (content ‘()))
   (method (type) ‘directory) ...)
```

And a `file` class that holds a list of symbols as content:

```
(define-class (file name content)
   (method (type) ‘file) ...)
```

We’d like to be able to do these:

```
(define root (instantiate directory ‘root))
(define hw1 (instantiate file ‘hw1.scm ‘(I have no idea how to do this)))
(define hw2 (instantiate file ‘hw2.scm ‘(please have mercy on me)))
(define proj1-soln
   (instantiate file ‘proj1.scm ‘(my dad is going to kill me)))

(ask root ‘add hw1) ;; add the hw1 file to root
(ask root ‘add hw2) ;; add the hw2 file to root
(ask root ‘mkdir ‘proj1) ;; makes an empty directory in root named ‘proj1’
(define proj1 (ask root ‘cd ‘proj1)) ;; returns directory named ‘proj1’
(ask proj1 ‘add proj1-soln) ;; add file proj1-soln to directory proj1
(ask root ‘mv ‘hw1.scm ‘proj1) ;; moves file hw1.scm from root to proj1
(ask root ‘ls) ;; returns list ‘(proj1 hw2.scm)
(ask proj1 ‘ls) ;; returns list ‘(hw1.scm proj1.scm)
```

QUESTIONS

1. **Implement the `add`, `mkdir`, `cd`, `mv` and `ls` methods for the `directory` class.**

2. **Implement a `size` method for the `file` class; it returns the length of its content.**

3. **The size of a directory is defined as the sum of the sizes of all files in that directory and in all its subdirectories. Implement the `size` method for the `directory` class.**

Pairs Are Objects Too

Here's yet another way to model a cons pair:

```
(define-class (cons-pair-list the-car the-cdr) ...)
(define-class (the-null-list) ...)
```

We can make up lists using the `cons-pair-list` object. So to make the list `'(2 3)`, we can do:

```
(define l (instantiate cons-pair-list 2
   (instantiate cons-pair-list 3 (instantiate the-null-list))))
```

QUESTIONS: For questions 2-5, you cannot use `if` or `cond`. Assume a `cons-pair-list` object represents a valid, flat list for the following.

1.**Implement a `(make-oop-list ls)` procedure that takes in a normal list and returns a `cons-pair-list` constructed with instances of the above `cons-pair-list` and `the-null-list` classes. Assume `ls` is a flat list.**

2.**Implement the `listify` method for a `cons-pair-list` so that `(ask l 'listify)` returns the list `(2 3)`.**

3.**Implement the `length` method for a `cons-pair-list` so that `(ask l 'length)` returns `2`.**

4.**Implement the `(accumulate combiner init)` method for a `cons-pair-list`; for example, `(ask l 'accumulate + 0)` returns `5`.**

5.**Implement the `(map proc)` method for a `cons-pair-list` so that `(ask l ‘map square)` returns a `cons-pair-list` containing `4` and `9`, leaving `l` unchanged.**

6.**Implement the `(map! proc)` method for a `cons-pair-list` so that `(ask l ‘map! square)` changes `l` itself to an object containing `4` and `9` (instead of returning a new `cons-pair-list` containing `4` and `9`). More specifically, you cannot have any calls to `instantiate`. The return value is unimportant.**

Inheritance (or: Midterm Fun)

A powerful idea in OOP comes from the observation that most things are alike in some way. (Humor me for a minute.) 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 documentation in the reader; let's jump into an example.

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?) '(ice cold) '(cooler than being cool) 90))
```

Your second 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; this option is available only to proctors, not to students.

QUESTIONS

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

2. **Note that there’s a method called `hint` *and* an instantiation variable called `hint`. When you call `(ask q1 ‘hint)`, which will be used?**

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 Twilight)))

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

3. **Implement the `bonus-question` class to inherit from the `question` class, using minimal code.**

```
   (define-class (bonus-question q)
```

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.

4. **Implement the `grade` method for the `midterm` class that calculates the total grade.**

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 steven (instantiate proctor 'steven))
```

You can either ask steven 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).

5.**What would be returned from the call (ask steven 'how-much-time-left?)?**

6.**Implement the clarify method. The password, as you saw, is “redrum”.**

Now, there’s a different kind of proctor, of course – an instructor. An instructor exhibits the following behavior:

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

7. **Implement the instructor class with minimal code.**

```
(define-class (instructor name)
```

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

8. **Implement the `ta` class with minimal code.**

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

Lastly, we want a `lenient-proctor` class. A `lenient-proctor` 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 eric (instantiate lenient-proctor ‘eric jon george))
```

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

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