CS 61A Week 7

Topic: Object-oriented programming

Reading:

Read "Object-Oriented Programming—Above-the-line view" (in course reader).

Homework:

Note: To use the OOP language you must first

```
(load "~cs61a/lib/obj.scm")
```

before using `define-class`, etc.

1. For a statistical project you need to compute lots of random numbers in various ranges. (Recall that `(random 10)` returns a random number between 0 and 9.) Also, you need to keep track of *how many* random numbers are computed in each range. You decide to use object-oriented programming. Objects of the class `random-generator` will accept two messages. The message `number` means "give me a random number in your range" while `count` means "how many `number` requests have you had?" The class has an instantiation argument that specifies the range of random numbers for this object, so

```
(define r10 (instantiate random-generator 10))
```

will create an object such that `(ask r10 'number)` will return a random number between 0 and 9, while `(ask r10 'count)` will return the number of random numbers `r10` has created.

2. Define the class `coke-machine`. The instantiation arguments for a `coke-machine` are the number of Cokes that can fit in the machine and the price (in cents) of a Coke:

```
(define my-machine (instantiate coke-machine 80 70))
```

creates a machine that can hold 80 Cokes and will sell them for 70 cents each. The machine is initially empty. `Coke-machine` objects must accept the following messages:

Continued on next page.

Week 7 continued...

`(ask my-machine 'deposit 25)` means deposit 25 cents. You can deposit several coins and the machine should remember the total.

`(ask my-machine 'coke)` means push the button for a Coke. This either gives a `Not enough money` or `Machine empty` error message or returns the amount of change you get.

`(ask my-machine 'fill 60)` means add 60 Cokes to the machine.

Here's an example:

```
(ask my-machine 'fill 60)
(ask my-machine 'deposit 25)
(ask my-machine 'coke)
NOT ENOUGH MONEY
(ask my-machine 'deposit 25)      ;; Now there's 50 cents in there.
(ask my-machine 'deposit 25)      ;; Now there's 75 cents.
(ask my-machine 'coke)
5                                 ;; return val is 5 cents change.
```

You may assume that the machine has an infinite supply of change.

3. We are going to use objects to represent decks of cards. You are given the list `ordered-deck` containing 52 cards in standard order:

```
(define ordered-deck '(AH 2H 3H ... QH KH AS 2S ... QC KC))
```

You are also given a function to shuffle the elements of a list:

```
(define (shuffle deck)
  (if (null? deck)
      '()
      (let ((card (nth (random (length deck)) deck)))
        (cons card (shuffle (remove card deck))) )))
```

A deck object responds to two messages: `deal` and `empty?`. It responds to `deal` by returning the top card of the deck, after removing that card from the deck; if the deck is empty, it responds to `deal` by returning `()`. It responds to `empty?` by returning `#t` or `#f`, according to whether all cards have been dealt.

Write a class definition for `deck`. When instantiated, a deck object should contain a shuffled deck of 52 cards.

Continued on next page.

Week 7 continued...

4. We want to promote politeness among our objects. Write a class `miss-manners` that takes an object as its instantiation argument. The new `miss-manners` object should accept only one message, namely `please`. The arguments to the `please` message should be, first, a message understood by the original object, and second, an argument to that message. **(Assume that all messages to the original object require exactly one additional argument.)** Here is an example using the `person` class from the upcoming adventure game project:

```
> (define BH (instantiate person 'Brian BH-office))

> (ask BH 'go 'down)
BRIAN MOVED FROM BH-OFFICE TO SODA

> (define fussy-BH (instantiate miss-manners BH))

> (ask fussy-BH 'go 'east)
ERROR: NO METHOD GO

> (ask fussy-BH 'please 'go 'east)
BRIAN MOVED FROM SODA TO PSL
```

Extra for experts:

The technique of multiple inheritance is described on pages 9 and 10 of "Object-Oriented Programming – Above-the-line view". That section discusses the problem of resolving ambiguous patterns of inheritance, and mentions in particular that it might be better to choose a method inherited directly from a second-choice parent over one inherited from a first-choice grandparent.

Devise an example of such a situation. Describe the inheritance hierarchy of your example, listing the methods that each class provides. Also describe why it would be more appropriate in this example for an object to inherit a given method from its second-choice parent rather than its first-choice grandparent.

Unix feature of the week: `|` (pipes in the shell)

Emacs feature of the week: `M-x spell-buffer`

Note: The second midterm exam is next week.