Object Oriented Programming

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. Implement the add and take methods.

```
(define-class (bag)
  (instance-vars (skittles '()))
  (method (tag-line) 'taste-the-rainbow)
  (method (add s) (set! skittles (cons s skittles)))
  (method (take) (let ((first (car skittles)))
      (set! skittles (cdr skittles)) first)))
```

Suppose we want to simulate you taking a midterm. 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))
```

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 ;; answers the question
(ask q1 'cur-answer) ==> '(17)
(ask q1 'grade) ==> 0 ;; returns the grade
(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.

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

```
(define-class (question q a hint weight)
  (instance-vars (cur-answer '()))
  (method (read) q)
  (method (answer ans) (set! cur-answer ans))
  (method (grade) (if (equal? cur-answer a) weight 0))
  (method (hint pwd) ;; this overwrites the hint instantiation var.
    (if (equal? pwd 'redrum)
      hint
      '(Wrong password! I hope you are proud))))
```
Note that there's a method called hint and an instantiation variable called hint. When you call (ask q1 'hint), which will be used?
The method
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)

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

(define-class (bonus-question q)
  (parent (question q '() '(a bonus question gives no hints) 0)))

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) ...))

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.

  (method (grade)
    (accumulate (lambda(x y) (+ (ask x 'grade) y)) 0 q-ls)))

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 tyler (instantiate proctor 'tyler))

You can either ask tyler 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 tyler 'how-much-time-left?)? (tyler : 34) or some other random integer from 0 to 100

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

  (method (clarify q) (ask self 'answer (ask q 'hint 'redrum))))

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)

7. Implement the professor class with minimal code.
(define-class (professor name)
  (parent (proctor name))
  (method (get-time) 30) ;; why didn’t we overwrite how-much-time-left?
  (method (clarify q)
    (ask self ‘answer’ (the question is perfect as written))))

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 hell 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 chris (instantiate ta ‘chris 3))
(ask chris ’how-much-time-left?)
(ask chris ’how-much-time-left?)
(ask chris ’how-much-time-left?)
(ask chris ’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)
  (parent (proctor name))
  (method (answer msg)
    (set! temper-limit (- temper-limit 1))
    (if (< temper-limit 0)
      (usual ‘answer’ (how the hell would I know?))
      (usual ‘answer msg)))))

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 matloob (instantiate lenient-proctor ‘matloob brian tyler))

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

(define-class (lenient-proctor name p1 p2)
  (parent (proctor name))
  (method (get-time) ;; why not overwrite how-much-time-left?
    (max (ask p1 ’get-time) (ask p2 ’get-time))))