Last Name: ______________________ First Name: ____________________ Cornell NetID, all caps: ______

CS 1110 Final Exam May 15th, 2014

This 150-minute exam has 6 questions worth a total of 60 points. When permitted to begin, scan the whole test before starting. Budget your time wisely. Use the back of the pages if you need more space. You may tear the pages apart; we have a stapler at the front of the room.

If a question does not explicitly ask for you to write an invariant, you don't have to for that problem. However, we strongly recommend that you provide comments explaining the meaning of your variables if you think they might be unclear to the graders.

The second page of this exam gives you the specifications for some useful functions.

It is a violation of the Academic Integrity Code to look at any exam other than your own, to look at any other reference material, or to otherwise give or receive unauthorized help.
We also ask that you not discuss this exam with students who are scheduled to take a later makeup.
Academic Integrity is expected of all students of Cornell University at all times, whether in the presence or absence of members of the faculty. Understanding this, I declare I shall not give, use or receive unauthorized aid in this examination.

Signature: ____ Date ____________

Last Name: ________________ First Name: ____________ Cornell NetID: ______

For reference:

`str.find(substr)`	Returns: index of first occurrence of string `substr` in string `str` (-1 if not found)
`str.find(substr, i)`	Returns: index of first occurrence of string `substr` in string `str` that occurs at or after index `i` (-1 if not found)
`s.split(sep)`	Returns: a list of the words in string `s`, using `sep` as the delimiter (whitespace if `sep` not given)
`s.join(slist)`	Returns: a string that is the concatenation of the strings in list `slist` separated by string `s`
`s.lower()`	Returns: a copy of `s` with all letters in it converted to lowercase
`s.upper()`	Returns: a copy of `s` with all letters in it converted to uppercase

`range(n)`	Returns: the list `[0, 1, 2, ..., n-1]`
`lt.append(item)`	Adds `item` to the end of list `lt`
`lt.remove(obj)`	Remove the object `obj` from list `lt`. Does not return a value
`lt.index(item)`	Returns: index of first occurrence of `item` in list `lt`; raises an error if `item` is not found. (There's no "find" for lists.)
`lt[i:j]`	Returns: A new list `[lt[i], lt[i+1], ..., lt[j-1]]` under ordinary circumstances. Returns `[]` if `i` $\geq$ `len(lt)`

Question	Points	Score
1	2	
2	4	
3	10	
4	12	
5	22	
6	10	
Total:	60	

Last Name: ____________________ First Name: ________________ Cornell NetID: _______

The Important First Question:

1. [2 points] When allowed to begin, write your last name, first name, and Cornell NetID at the top of each page.

2. [4 points] **Objects.** Consider the following code (docstrings omitted for exam brevity, line numbers added for reference).

```
 1  class Prof(object):
 2       def __init__(self, n):
 3           self.lname = n
 4
 5  ljl2 = Prof("Schlee")
 6  srm2 = Prof("Schmarschner")
 7
 8  lecturingprof = srm2
 9  lecturingprof.wise = True
10  lecturingprof = ljl2
11  print "Is Prof " + srm2.lname + " wise? " + str(srm2.wise)
12  print "Is Prof " + ljl2.lname + " wise? " + str(ljl2.wise)
```

List all output and/or errors that would be generated by running this code, in the order they are produced. For each thing you write, indicate what line number produces it.

In the case of errors, it suffices to explain what the problem is — you don't know have to know precisely what Python would call the error or print out.

Hint: line 9 does *not* cause an error. It would be wise (ha!) to understand why before proceeding; what does Python always do when asked to assign to a variable that doesn't exist?

3. [10 points] **String processing, loops.** We say that a string is a *sentence* if it consists of "words" (non-empty sequences of non-space characters) separated by single spaces, with no spaces at the beginning or end of the string. A sentence is *chunked* by *delimiter* `dl` if an even number of its words are `dl`, and no two delimiters are consecutive. Here's an example of a sentence that is chunked by "!".

   "The ! Big Red Barn ! was ! blue !"

   The *interesting spans* of a chunked sentence are the sequences of words that appear between each *odd* occurrence of `dl` and the next occurrence of `dl`. So, "Big Red Barn" *is* an interesting span because it occurs between the 1st and 2nd "!". "was" is *not* an interesting span because it occurs after the 2nd "!" (and before the 3rd one).

   The *highlighted* version of a chunked sentence is one where the delimiters have been removed, every word in an interesting span has been capitalized, and every word not in an interesting span is in lowercase. For example, the highlighted version of the chunked sentence above is

   "the BIG RED BARN was BLUE"

   Implement the function below so it fulfills its specification.

   Hints (not requirements): Use `split` and/or `join` (see reference on page 2). Use a loop, but do *not* use a nested loop. Keep track of whether you're inside or outside an interesting span.

```
def highlight(input, dl):
    """Return: the highlighted version of the input.
    Pre: input: a sentence chunked by dl.  dl: non-empty string without spaces."""
```

4. [12 points] **Recursion.** We say that an input `input` is *well-formatted* with respect to a list `labels` if (a) `input` is a list, and (b) `input` has length at least two, and (c) `input`'s first item is in the list `labels`, and (d) each of the remaining items in `input` is either a string or a well-formatted list. Here are some examples of well-formatted and non-well-formatted inputs:

`input`	`labels`	well-formatted?
`['VP', ['V', 'eat']]`	`['VP', 'V']`	True
`['NP', ['N', 'a', 'or', 'b'], 'c']`	`['NP', 'V', 'N']`	True
`[1, [2, 'oui', [1, 'no']], 'no']`	[1,2]	True
`['VP', ['V', 'eat']]`	`['VP']`	False: 'V' not in `labels`
`['VP', ['V']]`	`['VP', 'V']`	False: list `['V']` too short
`'VP'`	`['VP', 'V']`	False: `input` is not a list

Implement the following function recursively according to its specification.

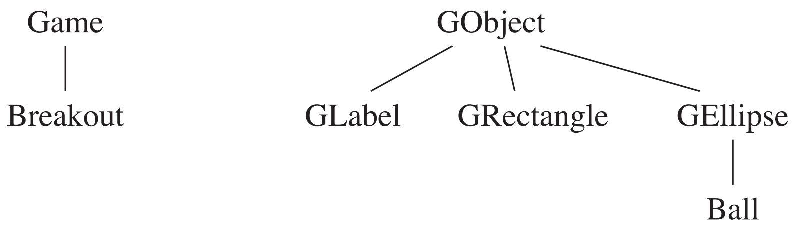
```
def wellformatted(input, labels):
    """Returns: True if <input> is well-formatted with respect to <labels>,
    False otherwise.

    Pre: labels is a (possibly empty) list.
    """
```

Last Name: ______________ First Name: ______________ Cornell NetID: ________

5. **Subclasses.**

Consider the accompanying code, which shows part of a solution to A5 that has a curious way of handling the flow of states in the game. Many details are omitted, but all the code related to adding and removing objects from the game is preserved.

(a) [3 points] Complete the class hierarchy below showing the relationships between all the classes in this code.



(b) [3 points] Remember that when an object is created, Python calls the method `__init__` on that object automatically. The same name resolution process is followed as with any method call.

Now, to the question: line 45 executes only once. *During the execution of that line*, line 79 gets executed. At that point in the execution (i.e. when line 79 is executed),

(i) What is the class of the object referenced by `self`?

(ii) Where does the variable named `self.TEXT` reside? We want to know in which specific class, instance, frame, or module it is found—that is, where it would be drawn when diagramming the execution.

(iii) At what line was the variable created?

(c) [4 points] Remember that the "call stack" is the set of frames that exist at a particular time. For instance, during a game when the player loses, line 201 executes exactly once, and at the start of executing that line, the call stack is:

```
...
Breakout.update: 51
Ball.update: 201
```

Here we are including just the first line from each frame, indicating which method is executing and which line it is at. The frames appear in the order they were created. (Note that since there are multiple functions with the same name, it's important to include the class they are defined in.)

The variable `Brick.num_bricks` is mentioned in the code only at lines 150, 154, 165, and 166. During a game that is won, it is assigned the value 3 exactly twice. At the first time it gets that value, what is the call stack? Use a format like the example above, and only mention functions that are defined in `breakout.py`.

(d) [12 points] Complete the subclass Countdown of Message that shows the message "Get ready!" and then, 90 frames after it was created and added to the game, removes itself from the game and "serves" by adding a new Ball instance to the game. The constants `TEXT` and `DELAY` should determine the message and the time delay before serving the ball. Be sure your code adheres to the provided class invariant. (To save time on the exam, there's no need to write specifications.)

```
class Countdown(Message):
    """See spec in code handout, line 106"""

    TEXT = 'Get ready!'
    DELAY = 90
```

6. [10 points] **Invariants.** Suppose we are given the task of rearranging a string so that certain characters are moved to the beginning and the other characters are moved to the end. Implement this method to the specification given below, following the comments in the code.

```
def partition_string1(s1, s2):
    """Return: a string that has the same characters as s1, only reordered
    so that all the characters that appear in s2 are at the beginning, and
    all the characters that do not appear in s2 are at the end.  The ordering
    of the characters within each of the two segments is not important.

    Examples:
        s1            ; s2    ; some correct results
        -------------------------------------------------------
        'abracadabra' ; 'abc' ; 'abacaabardr', 'aaaabbcrdr', ...
        'foobar'      ; 'ob'  ; 'boofar', 'oboarf', ...
        'foobar'      ; ''    ; 'foobar', 'oofarb', ...
        'a'           ; 'b'   ; 'a'
    """

    # This function works by converting the input to a list and rearranging
    # the list in place using swaps, following the invariant below.  Your
    # code must agree with the invariant and postcondition for full credit.

    # convert to a list b


    # initialize counters



    # inv: b[0..i-1] in s2; b[j+1..len(b)-1] not in s2

    while






    # post: b[0..j] in s2; b[j+1..len(b)-1] not in s2
    # convert b back to a string and return
```

Did you write your name and netID on each page, and re-read all specs?
Then, have a great summer break!

```python
# breakout.py
# Steve Marschner (srm2)
# May 15, 2014
"""Breakout game for CS 1110 final."""
import colormodel
import random
import math
from game2d import *

# Window Size
GAME_WIDTH  = 512
GAME_HEIGHT = 512

class Breakout(Game):
    """The main class for an alternative design of the Breakout game.
    This program breaks the Model/View/Controller mold, organizing the
    whole program around a list of "game objects" that have update and
    draw methods that are called once per frame by the update and draw
    methods in this Breakout class.  All game sequencing is handled by
    manipulating the list of active objects: when some condition is
    detected that requires changing the state of the game, the update
    method that discovered this adds or removes game objects as
    appropriate so that the game will continue.

    Instance variables:
        view [GView]:         the view (inherited from Game)
        prev_touch [GPoint]: the value of view.touch in the previous frame
        lives [int]:          number of balls remaining

    Class variables:
        the_game [Breakout]: the (one and only) instance of Breakout
    """
    # Private attributes:
    #    _game_objs: list containing all the game objects that currently exist
    #    _next_objs: list of all game objects that will exist in the next frame

    the_game = None

    def init(self):
        """Initialize the program state."""
        Breakout.the_game = self
        self.prev_touch = None
        self.lives = 3
        self._game_objs = []
        self._next_objs = [StartMessage()]

    def update(self, dt):
        """Animate a single frame. Called every 1/60 of a second."""
        self._game_objs = self._next_objs[:]
        for obj in self._game_objs:
            obj.update(dt)
        self.prev_touch = self.view.touch

    def draw(self):
        """Draw all objects in the view."""
        for obj in self._game_objs:
            obj.draw(self.view)

    def add_game_object(self, new_obj):
        """Add a new object to the game; it will first exist during the
        next frame."""
        self._next_objs.append(new_obj)

    def remove_game_object(self, old_obj):
        """Remove a given object from the game; it still exists in the
        current frame but will be gone in the next frame."""
        self._next_objs.remove(old_obj)


class Message(GLabel):
    """A message that appears on the screen, always in the center.  The
    text of the message is controlled by the attribute TEXT.  By default
    the update method does nothing, but subclasses may want to override
    that method to provide some behavior.
    """

    def __init__(self):
        """A message with text given by the attribute TEXT."""
        GLabel.__init__(self, text=self.TEXT, halign='center', valign='middle',
                        x=GAME_WIDTH/2, y=GAME_HEIGHT/2)

    def update(self, dt):
        pass


class StartMessage(Message):
    """A message telling the user to click to start the game."""

    TEXT = 'Click to start'

    def update(self, dt):
        """When the player clicks, start the game by setting up the
        bricks and paddle and creating a Countdown message that will
        start the game after a short delay.
        """
        game = Breakout.the_game
        if game.view.touch is not None:
            for row in range(Brick.ROWS):
                for col in range(Brick.COLS):
                    game.add_game_object(Brick(row, col))
            game.add_game_object(Countdown())
            game.add_game_object(Paddle())
            game.remove_game_object(self)


class Countdown(Message):
    """A message that appears on the screen and will start the game after
    a certain delay.

    Instance variables:
        frames_left: number of frames remaining until the serve.
    """

    TEXT = 'Get ready!'
    DELAY = 90

    # ...


class LoseMessage(Message):
    """A message that stays forever telling the player that they lost."""

    TEXT = 'Game over!'


class WinMessage(Message):
    """A message that stays forever telling the player that they won."""

    TEXT = 'You win!'


class Brick(GRectangle):
    """A brick in the game, which is part of a grid of bricks and
    responds to collisions with the ball by disappearing.

    Class variables:
        num_bricks: the number of bricks that currently exist in the game
    """

    SEP_H    = 5
    SEP_V    = 4
    HEIGHT   = 8
    Y_OFFSET = 70
    COLS     = 8
    ROWS     = 12

    # ... more constants ...

    # The number of bricks that currently exist
    num_bricks = 0

    def __init__(self, row, col):
        # ... call to superclass initializer ...
        Brick.num_bricks += 1

    def update(self, dt):
        """Handle this brick's behavior for the current frame by
        checking if the ball collided with it and, if so, removing
        it. Removing the last brick results in winning the game.
        """
        # ... logic to get ahold of the ball ...
        if ball is not None and ball.collide(self):
            game = Breakout.the_game
            game.remove_game_object(self)
            Brick.num_bricks -= 1
            if Brick.num_bricks == 0:
                game.remove_game_object(ball)
                game.add_game_object(WinMessage())


class Ball(GEllipse):
    """The game ball.  It moves itself according to an (x,y) velocity,
    bounces off the walls and ceiling, and responds to collisions with
    any objects that call Ball.collide by bouncing off of them.  When
    the ball falls past the bottom of the window, a life is lost.

    Instance variables:
        vx, vy [float] -- the ball's velocity in pixels per frame

    [... other variables ...]
        """

    DIAMETER = 18

    def __init__(self):
        GEllipse.__init__(self, center_x=GAME_WIDTH/2, center_y=GAME_WIDTH/2,
                          width=Ball.DIAMETER, height=Ball.DIAMETER)
        # ... instance initialization ...

    def update(self, dt):
        # ... logic for moving and responding to collisions ...
        # if (ball falls off the bottom of the screen):
```

```
        if self.y < -Ball.DIAMETER:
            # ...
            game = Breakout.the_game
            game.remove_game_object(self)
            game.lives -= 1
            if game.lives > 0:
                game.add_game_object(Countdown())
            else:
                game.add_game_object(LoseMessage())

    def collide(self, other):
        """Check for a collision between the ball and another GObject.
        If there is a collision, record it so that update can make an
        appropriate response.  Return: True if there was a collision,
        otherwise False.
        """
        # ... logic to detect collisions ...


class Paddle(GRectangle):
    """The paddle that is controlled by the player.  It moves in response
    to the player's touch and collides with the ball.
    """

    WIDTH  = 58
    HEIGHT = 11
    Y_OFFSET = 30

    def __init__(self):
        """A new paddle in the center of the bottom of the window."""
        # ... call to superclass initializer ...

    def update(self, dt):
        """Move the paddle in response to player input, and call
        Ball.collide to detect collisions."""
        # ... logic for paddle movement ...


# Script Code
if __name__ == '__main__':
    Breakout(width=GAME_WIDTH,height=GAME_HEIGHT,fps=60.0).run()
```