CSci 1133
Lab Exercise 12

Simple Inheritance and Polymorphism

Graphical programs are natural applications for object-oriented programming. In this lab exercise, you will construct a number of classes to manage objects on a graphical display. We'll begin by extending our working knowledge of the Turtle Graphics interface. Recall that you can find a complete description of the entire module on the "official" Python reference online:

```
http://docs.python.org/3.2/library/turtle.html
```

Warm-up

You already know how to draw lines and turn the turtle, but a number of other objects can be drawn using turtle methods in one step. Circles are one example, complex polygons are another. For example, to draw a circle at the location 40,40:

```
>>> myt.penup()
>>> myt.goto(40,40)
>>> myt.circle(100)
>>> myt.pendown()
>>> myt.circle(100)
```

Voila! We have a circle at 40,40. But it isn't very colorful… In fact, it's *transparent*. The only thing that shows is the outline of the circle. In order to have a colored circle we have to do a few more things, including filling it with a color. Creating a "filled" color object, requires four steps in turtle graphics:

1. Set the fill color to whatever you want ("red", "blue", "yellow", etc.)
2. Indicate the start of a filled shape by invoking the `.begin_fill` method
3. Perform a *sequence* of drawing operations that will create an "enclosed" figure
4. Indicate the end of the filled shape, and fill the object with the fill color by invoking the `.end_fill` method

It's easier to do than describe! Let's redraw our circle filled with red:

```
>>> myt.fillcolor("red")
>>> myt.begin_fill()
>>> myt.circle(100)
>>> myt.end_fill()
```

We now have a nice red circle. Note that the `.fillcolor()` method takes a string. Let's do it again with a rectangle. There are turtle methods for drawing polygons that you might like to explore on your own. For now, we'll just use the line drawing methods we've used in the past:

```
>>> myt.penup()
>>> myt.goto(-50,-50)
>>> myt.pendown()
>>> myt.fillcolor("blue")
>>> myt.begin_fill()
>>> for i in range(4):
        myt.forward(100)
```

```
        myt.right(90)
>>> myt.end_fill()
```

Functions are Objects!

Yes, it's true. Function *names* in Python are object references (just like variables!) and can be used in lists, dictionaries and even as arguments in other functions. In order to obtain graphical style (i.e. "mouse") input in our programs, we need to exploit this capability by providing a function that will be called when a mouse "click" occurs.

Mouse tracking and input ("clicking" a mouse button) are handled *asynchronously* to your program's operation. When the user "clicks" a mouse button, Python calls a function that you supply and passes the *x* and *y* coordinates of the mouse location as they were when the button was pressed. Turtle graphics "knows" what function to call because you "tell" it by giving the name of the function to a screen method, `.onclick(`*function_name*`)`. Again, it's probably easier to demonstrate than describe. First, write a short function definition:

```
>>> def mouseInput(x,y):
        print(x,',',y)
```

Our `mouseInput` function will be called anytime the mouse button is "clicked" by the user. Note that it *must* have two arguments for the *x* and *y* coordinates of the mouse location corresponding to the button press (NOTE: if this is a class *member* function, you will also need to supply the "self" argument as always). In this case, our "button handler" function simply prints out the *x* and *y* coordinate values on the terminal.

Next, we have to instruct the turtle Screen object that the `mouseInput()` function is the one that should be called when the mouse button is clicked. We do that by registering the function using the *screen* method: `.onclick(`*function_name*`)`

```
>>> scr = myt.getscreen()   #get the Screen object for myt
>>> scr.onclick(mouseInput) #register the button handler function
```

Note that the single argument to the `.onclick()` method is the *name* of the function that we've written. `.onclick` just "sets up" the mechanism... we have to do one more thing to actually make Python start processing button presses. It involves one more *screen* method that takes no arguments:

```
>>> scr.listen()
```

Position the turtle window and the python shell so that you can see both. Now move the mouse around in the turtle window and press the left button on the mouse and watch the output in the python shell display. You should see *x* and *y* values printed each time you press the button.

You are now ready to construct more sophisticated graphics programs. Let's use objects and inheritance to do something fun.

Stretch

1). **Shape Class**

Construct a base class named `Shape` that will maintain common "shape" information such as location, fill color, etc. The shape class should have instance variables for the x and y location of the shape (type `int`), the fill color (type `str`) and a `boolean` indicating if the shape should be filled or not. The location should default to 0,0. The fill color should default to a Null string, and filled should default to `False`. Provide the following `Shape` methods:

- `setFillcolor(str)` Mutator to set the fill color to the value of the string argument
- `setFilled(bool)` Mutator to set the filled boolean to the value of the boolean argument
- `isFilled()` Accessor that will return the value of the filled instance variable

2). **Circle Class**

Now, create a new class named `Circle` that inherits the `Shape` base class. The constructor for the `Circle` class should have arguments for the x and y location (default to zero) and the radius of the circle (default to 1). Be sure to properly call the super-class constructor! Provide the following `Circle` methods:

- `draw(turtle)` An accessor method that will draw a circle using turtle graphics. `draw()` will take a single turtle object as an argument and draw the circle at its specified x,y location. If the circle is "filled", be sure to implement the proper sequence of operations to draw a filled circle of the appropriate color. Otherwise draw an unfilled circle.

Test your two class definitions by creating and drawing several circles

3). **Graphics Display Object**

Create another class named `Display` that will maintain a graphics environment involving shapes and mouse input. The constructor for the `Display` class will instantiate a number of instance variables and set up the turtle environment to enable the input of mouse button selection events. Be sure to import the turtle module in your class definition file.

The Display constructor should do the following:

- Initialize a `Turtle` instance variable using the `Turtle()` method
- Initialize a `Screen` object member using the `Turtle .getscreen()` method
- Initialize an instance variable named `elements` (a null list)
- Set the speed and delay of the turtle and screen to zero and hide the turtle

Create a `Display` *method* named `mouseEvent(x,y)` that will process mouse button presses. For this version of the program, the `mouseEvent()` method will do the following:

- Create and draw a `circle` object at the x,y location indicated by the mouse press (passed as arguments to the function). The `circle` should be filled and have a random radius between 10 and 100. It should also have a random color. (consider using a list of color strings that you can use the `random.shuffle()` method on, then simply pick the first from the shuffled list!)

You will also need to add statements to the `Display` constructor to register the `mouseEvent()` method as the "on-click" routine (don't forget to enable it using the `listen()` screen method!)

If you've done this correctly, you should be able to simply instantiate a `Display` object, then use your cursor to roam about the window. Each time you press the mouse button, a randomly sized/colored circle should appear at the mouse location.

Workout

1). **Adding and Removing Shapes**

The `Display` class has an instance variable named `elements` that was initialized as a null list. The `elements` list will be used to keep track of shapes that are currently being displayed. Add the following mutator methods to the `Display` class:

> `add(shape)` : Takes a shape object as an argument and adds it to the end of the `elements` list. After the shape has been added, it should be drawn on the display using the object's `draw` method.
>
> `remove(shape)` : Takes a shape object as an argument and removes it from the `elements` list. After the shape has been removed from the list, clear the display (using the appropriate Turtle method), then redraw each shape in the `elements` list. The shapes should be drawn in order of appearance in the list.

Be sure to test each of these new methods using Python in interactive mode. Add a few circles, then remove them.

2). **Rectangle Class**

Add another geometric shape class to draw rectangles. The `Rectangle` class should also be derived from the `Shape` base class. The constructor should have arguments for the x and y location, plus the width and height of the rectangle object. Also include a `draw` method that will draw a rectangle based on its x, y, width, height, filled and fillColor attributes (refer to the `Circle` class).

Test your new class by using the `add` and `remove` methods in the Display object to display and remove a few rectangle objects.

3). **Selecting Objects**

Add a method to both the `circle` and `rectangle` classes named `isIN` that will take x,y coordinate values and return `True` if the point x,y is *within* the boundaries of the object.

[Hint: The "location" coordinates of the *object* reflect where it was drawn. For a circle, this is the bottom-center point of the circle]

4). **Displaying and Removing Shapes**

Modify the `mouseEvent` handler function in the `Display` class to remove a shape if the mouse button is pressed while the pointer is "inside" a currently displayed shape. If it's not inside any displayed object, construct and display either a randomly sized/colored circle or a randomly sized/colored rectangle per the instructions in Stretch problem 3.

Test your program by creating and removing objects from the display using the mouse.

Challenge

Add a `Button` class that is a child of a rectangle and adds both a label (string) and a button handler (function reference). Either bind the handler function to the object using the constructor or a mutator method (or both!)

Now create a button-handler function to toggle a state variable (in the Display class) that will direct the construction of rectangles or circles. In other words, choose which type of shape to draw based on the state variable rather than a "flip of the coin".

Modify the `mouseEvent` handler in the `Display` class to recognize a button-press and call the bound handler function. Finally, construct and add a Button to the display.

Congratulations! You've created a fairly sophisticated graphics program using the simplest of all graphics interfaces: turtle graphics.

Use your imagination and modify the program to do something creative and interesting!