CSCI 141 Computational Problem Solving
Project 3

Due Sunday, 12 April 2020 at 2359

In project 2, you drew rectangles, squares, triangles, and hexagons using characters in the terminal. Next, we explore how to draw shapes graphically in your favorite windowing system. To do this, we will employ the object-oriented principles discussed in lecture including classes, methods, attributes, inheritance, and polymorphism.

Your program should display a 500x500 window on the screen at launch. When the user clicks anywhere in the window, a random shape should appear at that location. A naïve implementation would attempt to contain everything in `paintEvent`, but this approach quickly becomes unmanageable because of the information needed to draw each shape. A well-structured program will consider individual objects and how they interact. This is the fundamental principle of object-oriented design. What we outline below is a suggested implementation approach.

To begin, define a base class `Shape` that stores the shape's length, position, and (randomly generated) color. From there, you can inherit these basic properties and further specify additional shapes. At a minimum, your program must support squares, rectangles, triangles, circles, and ellipses. Each shape should define its own class in an inheritance hierarchy based on Shape. Recall that inheritance often spans more than one level, so consider and implement an appropriate inheritance scheme. The classes that define the various shapes should further specify the `Shape` class by adding additional needed attributes, appropriately invoking parent constructors during initialization, and including a method to draw the shape using a `QPainter` instance provided as a parameter. Notice that the shape classes fully encapsulate the shape concepts, including the graphical drawing code. By providing an already-in-use `QPainter` instance to the shapes' draw methods, all of the shape classes should be implemented in a single file called `Shapes.py`.

Rather than combine the shape implementations with the graphical event-handling code, we can further compartmentalize our solution by using two files. The advantage to this is that the `Shapes.py` file contains only the code pertinent to creating, maintaining, and drawing the shapes. Everything event-related, such as mouse clicks, belongs in the `ShapeDrawer.py` control file. We have provided a skeleton file as a starting point for `ShapeDrawer.py`.

Suggested Approach

When the user clicks somewhere in the window (indicated by a call to `mousePressEvent`), create a new `Shape` object at random to be located at the click coordinates and add that `Shape` to the list of items to draw, which should be an attribute of your class. The Shape object's constructor (in cooperation with its parent constructors) should set all its attributes, including coordinates, color, and size. Feel free to apply reasonable restrictions to shape sizes. Every time the window is drawn

(because something triggered a call to `paintEvent`), each of the shapes created so far should be drawn. Because they are in a list (see below), you can iterate over them using a loop and draw each shape. The beauty of polymorphism is that you can treat them all as basic `Shapes` and ask them to draw themselves, because they each implement a `draw` method that takes a `QPainter` instance (perhaps the one from `paintEvent` in `ShapeDrawer.py`) as a parameter. Note that you must already have activated the painter object by calling its `begin` method before passing it to a shape's `draw` method.

The `list` Data Type

We have seen that for loops iterate over sequential items like strings of characters or elements of a range. Python's list type, which stores a sequence of any objects, generalizes our ability to iterate. A list object can store a collection of items, and a for loop will visit each item in the list. To create a new list, we can (among other ways) use the list constructor, such as `my_list = list()`. To insert items into the list, we can use the append method, such as `my_list.append(shiny_object)`. Finally, to iterate over the items in the list,

```
for item in my_list:
   #do something with item
```

Functional Expectations and Point Values

- **2 points:** When the user clicks anywhere in the window, a shape appears somewhere in the window.
- **1 point:** The upper left corner of shape that appears (or its smallest enclosing rectangle) is located at the click coordinates, **or**
  1.5 points: The center of the shape that appears is located at the click coordinates.
- **1 point:** The shape that appears is randomly selected among Rectangle, Square, Triangle, Ellipse, and Circle.
- **1 point:** Shapes are of random size and orientation, **or**
  1.5 points: Shapes are of random size and orientation, but stay within the 500x500 window. Feel free to restrict the maximum size of shapes as you deep appropriate.
- **2 points:** The shapes are defined in individual classes that are fully self-contained (including graphical code). Only the code relevant to each shape is present in the shape classes. Shape classes are imported in `ShapeDrawer.py` to facilitate drawing. Polymorphism is employed so that the `paintEvent` never cares what shape it is drawing.
- **2 points:** Quality of your writeup, including its explanation of how classes and inheritance factor into your implementation, references to

items you consulted, and a detailed explanation of exactly what happens when a user clicks in the window.

- **1 point:** Dazzle us with something else that you can do with your program that personalizes it without changing the basic functionality described above. Some examples might include shapes taking random walks after they appear, shapes falling downward or floating upward slowly after they appear, shapes disappearing after some time on the screen, more complex and difficult shapes, or events occurring when shapes touch. Your extensions should illustrate your creativity and understanding of object-oriented programming. Some video game aficionados may notice that the classic Atari® game *Asteroids* is not too far away from this project. Though fully reaching that goal is still a bit difficult, you are closer than you might realize.

SUBMISSION EXPECTATIONS

`writeup.pdf`: A document that outlines how your program works, including a walkthrough of exactly what happens when the user clicks in your window. Your writeup must also explain what you implemented for the project's dazzle point and why it is worth of 10% of the project's value. Also include references to any sources you consulted, regardless of format.

`ShapeDrawer.py`: Your main program. We will launch this program and expect a window to appear in which we can click to create shapes. The window should be 500x500. We provide a skeleton for the file.

`Shapes.py`: Your shape implementations including their corresponding drawing methods according to a well-reasoned inheritance hierarchy.