PIC 16: Homework 7 (due 6/2 at 10pm)

How should your answers be submitted?

- Download `hw7.py`. Replace `mjandr` by your name.
- Answer the questions, make sure your file runs correctly, and submit `hw7.py` to CCLE. Submitting something that runs with less code is better than something that doesn't run with more code.

This assignment consists of two parts.

- In the first part you will use mouse events, QPainter, signals and slots to create a GUI with a square that can be dragged around the window. Double-clicking the square will bring up a color picker dialog box that allows you to change the square's color.
- In the second part, you will use QTimer to help create an animation of a ball bouncing around inside a window.
- `hw7.mov` demonstrates what happens when I run my code.

  The black circle is *not* from my Python code; it is QuickTime's Screen Recording showing you when my mouse is pressed.

1. - Create a window 600px wide by 400px tall.
     Give it a white background. On your operating system this may be the default. However, use QPainter to paint a white rectangle of the correct size in case it's not the grader's default.
     Give it a 50px red square in a randomly selected position on the screen.
     I suggest that you store the position of the square's top left corner with instance variables `x` and `y`. I also suggest you store `50` using an instance variable `l` (that's a lower case `L`) so that you don't have to keep typing `50` later on in the assignment.
   - Write code that enables the user to click and drag the square. That is...
     - If the user presses the mouse within the bounds of the square, then the square is being dragged and should follow the mouse around the screen.
     - When the user releases the mouse button, the square is no longer being dragged and should not move with the mouse.
     - If the user presses the mouse outside the bounds of the square, the square is not being dragged and should not move with the mouse.

     To accomplish this, you will have to override `mousePressEvent` and `mouseMoveEvent`.
     - In `mouseMoveEvent`, you're going to need to use `e.x()` and `e.y()` to update the instance variables `x` and `y`, and use `update` to redraw the square.
       Introducing two new instance variables in `mousePressEvent` was helpful for me.
     - Dealing with the coordinates is tricky. The most common mistake will be that as soon as you start dragging the square, the top left corner (possibly the center, depending on your code) will jump to the point of the mouse. This is not what we want; the square should move smoothly with the mouse and should not move relative to the mouse while being dragged.
     - For the sake of learning the basics of GUIs, please do not try to use `QDrag`. The square should just be a rectangle painted on a "canvas" `QWidget`, and you should use regular mouse press events and mouse move events.
   - Write code so that a color selection dialog box is displayed when the user double-clicks within the square. There is a `QWidget` for this purpose; I've commented relevant links in `hw7.py` and more help is provided on page 3. The color of the square should only change when the user clicks "OK".

Grading (all or nothing for each of the following)

- Do you draw a red square at a random position completely inside the window? 1 point.
- Can the user click and drag or not drag the square correctly? 3 points.
- Does the square follow the mouse smoothly, without jumping relative to the cursor when clicked or when the mouse starts to move? 3 points.
  (It's okay for it to lag a tiny bit behind the cursor; that's normal. I want you to avoid a particular bug where the square jumps as soon as it is clicked or starts being dragged.)
- Does your code show the color selection dialog box when the user double-clicks in the square? 1 point.
- Does clicking okay or cancel in the color selection dialog box behave correctly? 2 points.

2. - Create a window 600px wide by 400px tall. Give it a white background and a 30px red circle at the top left corner of the window. I suggest that you store the position of the circle's top left corner with instance variables `x` and `y`. I also suggest you store `30` using an instance variable `d` (for diameter) so that you don't have to keep typing `30` later on in the assignment. Hint: a circle is an ellipse.
   - If you followed my suggestions, you have instance variables for the diameter, `x` coordinate, and `y` coordinate of the ball. Create two additional instance variables to represent the `x` and `y` components of the ball's velocity (`vx` and `vy` seem sensible names) and set them to 1 (i.e. we will make the ball move one pixel in the `x` and `y` direction each time we update the scene).
   - Write a method `animate` which increments the `x` and `y` position of the ball by the corresponding velocity components and calls the `update` method inherited from `QWidget`. This method requests that `paintEvent` be called. You should *not* call `paintEvent` yourself.
   - If you were to run your code now, the ball would stay put because the `animate` method is never called. Create a `QTimer` that calls your `animate` method every 25ms. Remember, you have to keep a reference to the timer, otherwise the timer can be garbage-collected. It is up to you to decide where to put the `QTimer` timer and how to keep a reference to it. I used `QThread` to `run` the timer in a new thread; doing this is optional.
   - If you have been successful up to this point, running your code now will show the ball moving in your window. However, it will not bounce when it reaches the edge. Instead, it will move beyond the edge. It is still there. You just can't see it without resizing your window.

     In your `animate` method (you decide exactly where), call a method `checkCollision` that determines whether a collision between the edge of the ball and the edge of the screen has occurred, and if so, changes the ball's velocity accordingly. Make sure that your method works even if the user resizes the window. Hint: check the widget's `width` and `height`.
   - When your code is working, feel free to tweak the velocity and `QTimer` period to see how these parameters affect the apparent speed and smoothness of the animation. If you used `QThread`, you'll probably be happier! Change the parameters back to `(1,1)` and `25` before you submit.

Grading (all or nothing except the last part)

- Does your code draw a red ball starting in the very top left corner? 1 point.
- Does your white background adapt to the shape of the window? 1 point.
- Does your code animate the ball moving across the screen? 2 points.
- Does the ball bounce around the window? 2 points.
- Does the ball bounce *exactly* when the perimeter touches the boundary? 1 point.
- Does *everything* continue to work even when the window is resized?

  1 point if it correctly adapts to a resize that doesn't move over the ball.

  3 points if it handles all nasty resizes from the bottom-right corner correctly: see the last bit of the demo video. (You don't need to worry about other resizes. If you do care, you may want to rethink the instance variables names `x` and `y`.)

`QColorDialogBox` help…

- Create a `QColorDialogBox` when you initialize your widget.
  Its constructor needs a color (red), and a parent (your widget).
- Connect the `colorSelected` signal to a method of your naming.
- Have this method update the color of your square and call the `update` method.
- Since `QColorDialogBox` inherits from `QWidget` it has a `show` method.
  This is what `mouseDoubleClickEvent` should call.

Notice that the first three steps follow the way in which we have made `QPushButtons`.

Notice in the demonstrational video that my color dialog box remembers colors even if I click "cancel". This is because I only ever have ONE `QColorDialogBox` object. I do NOT create one each time I double-click.