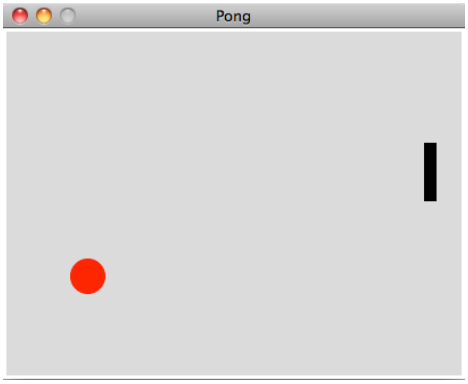
1. In this lab you will use object-oriented programming to create a single-player version of the classic Pong video game. The game consists of a ball and a paddle, as shown below. The ball will bounce around the window and the player must move the paddle vertically to try to contact the ball and keep it in play. If the ball ever reaches the edge of the right side of the window, the game ends. In order to make the game a little more challenging, after every fifth time the player hits the ball, its speed will increase slightly.



   We will create the program incrementally by first defining and testing a class for representing the ball, and then defining a class for handling the game itself. Download and unzip **lab12_files.zip** to get started.

2. The Ball class has been started for you in **pong.py**, which for now contains just the `__init__` constructor method. A short test program called `balltest` is also defined for you. Notice that the Ball class *inherits from* the Circle class, meaning that the Ball class extends the Circle class. We say that the Circle class is the *superclass* of the Ball class, and Ball is the *subclass*. To put it another way, Ball objects *are* Circle objects, with some additional functionality. For example, Ball objects have a `getCenter()` and a `getRadius()` method, just like Circle objects. Notice also that the Ball constructor defines the instance variables `self.dx` and `self.dy`. These will control how fast the ball moves in the horizontal and vertical direction, and are initially set to random values that will make the ball move at a reasonable starting speed. Larger values will make the ball move faster. Study the code, and then run `balltest()` to see what it does.

3. We will need some helper methods that return the coordinate of a ball's left edge, right edge, top edge, or bottom edge. The edge coordinate is just the ball's center coordinate, plus or minus its radius. For example, the left edge of the ball is its center *x* coordinate minus the ball's radius. The bottom edge is the center *y* coordinate plus the radius. Add definitions for the following methods to your Ball class:

   - **leftEdge(*self*)** returns the ball's center *x* coordinate minus the ball's radius
   - **rightEdge(*self*)** returns the ball's center *x* coordinate plus the ball's radius
   - **topEdge(*self*)** returns the ball's center *y* coordinate minus the ball's radius
   - **bottomEdge(*self*)** returns the ball's center *y* coordinate plus the ball's radius

   Since the Ball class inherits from the Circle class, the above Ball methods can just call `self.getCenter()` to retrieve the Ball's center point. To test your new methods, add the code below to the end of the `balltest` function and run it. Make sure the resulting values are 85.0 for the left edge, 115.0 for the right edge, 105.0 for the top edge, and 135.0 for the bottom edge, since the red ball's center point is located at (100.0, 120.0):

```
def balltest():
    ...
    print("red ball left edge is", redBall.leftEdge())
    print("red ball right edge is", redBall.rightEdge())
    print("red ball top edge is", redBall.topEdge())
    print("red ball bottom edge is", redBall.bottomEdge())
```

4. Now we need to make the balls move. To do this, add the method **move(*self*)** shown below to the Ball class, which will move the ball by the amount `self.dx` and `self.dy`. Since the Ball class inherits from the Circle superclass, this method just calls the Circle class's `move` method to move the ball:

```
def move(self):
    super().move(self.dx, self.dy)
```

The `balltest` program will use a loop to repeatedly call each ball's `move` method, with a short pause in-between updates. The loop will continue to run as long as no mouse click is detected in the window by the window's `checkMouse` method. Add the code below to the end of `balltest`, and then try it out:

```
def balltest():
    ...
    while win.checkMouse() == None:
        redBall.move()
        blueBall.move()
        greenBall.move()
        time.sleep(0.01)
```

5. The next task is to keep the balls within the window boundaries. To do this, add some code to the Ball class's `move` method that checks if the ball's left edge is ≤ 0 or its right edge is $\geq$ the width of the window (use your helper methods `leftEdge` and `rightEdge` for this). If so, multiply the ball's `self.dx` variable by -1 to reverse the direction of horizontal movement. Add similar code to reverse the direction of vertical movement when the ball's top edge is ≤ 0 or its bottom edge is $\geq$ the height of the window. (You may want to review your bounce() program from Lab 7, which used similar logic.) Rerun `balltest()` to make sure that all of the balls stay within the window.

6. We also need a Ball method called **goFaster(*self*)** that will slightly increase the speed of the ball by making the ball's `self.dx` and `self.dy` variables slightly larger. One way to do this is to multiply each variable by a random floating-point number in the range 1.0 to 2.0:

```
def goFaster(self):
    self.dx = self.dx * random.uniform(1, 2)
    self.dy = self.dy * random.uniform(1, 2)
```

After adding this method to your Ball class, modify `balltest()` so that the green ball speeds up every 100 loop cycles. You can do this by introducing a counter variable that gets incremented by one on each loop cycle. Within the loop, check if the counter is divisible by 100, and if so, call the green ball's `goFaster()` method. You should see the green ball gradually speed up as it bounces around inside the window.

7. Once you are satisfied that your Ball class is working properly, you can move on to implementing the game itself, which will be represented by the Pong class. The Pong constructor method is already provided for you, which sets up the game window by drawing the paddle and placing a red ball at a random location. Run the `main()` program to see what the game window looks like.

8. The **play(*self*)** method currently doesn't do anything, but will eventually contain the main game loop. Here is a pseudocode outline of the loop:

```
while the game is not over:
    check if a key was pressed
    if a key was pressed:
        move the paddle up or down based on which key was pressed
    move the ball
    if the ball is in contact with the paddle:
        increment the number of hits
        if the number of hits is divisible by 5:
            increase the speed of the ball
    sleep briefly
report that the game is over
report the final score
```

As a first step, define a Pong method called **gameOver(*self*)** that returns True if the right edge of the ball is greater than or equal to the width of the Pong window. Otherwise the method should return False. Then add the following code to the `play` method to create a basic game loop:

```
while self.gameOver() == False:
    self.ball.move()
    time.sleep(0.01)
print("Game over!")
```

Test out the game by calling `main()`. The game should end when the ball hits the right edge of the window.

9. Now we need to be able to move the paddle up and down by pressing the Up or Down arrow keys. The `checkKey()` method of a GraphWin can be used to check if a key was pressed in the window. Add the following code to your game loop, just before the line that moves the ball:

```
key = self.win.checkKey()
if key == 'Up':
    self.paddle.move(0, -30)
elif key == 'Down':
    self.paddle.move(0, 30)
```

To test your code, it may help to temporarily change the while loop condition to True, rather than calling gameOver(), so that the game will not immediately terminate when the ball hits the right window edge.

10. The final task is to detect when the ball hits the paddle. To do this, we will define a new Ball method called **checkContact(*self*, *rectangle*)**, that takes a Rectangle object as an input parameter and checks whether the right edge of the ball overlaps the rectangle. The code is given below. Add this method to your Ball class:

```
def checkContact(self, rectangle):
    x1 = rectangle.getP1().getX()  # left edge of paddle
    y1 = rectangle.getP1().getY()  # top edge of paddle
    x2 = rectangle.getP2().getX()  # right edge of paddle
    y2 = rectangle.getP2().getY()  # bottom edge of paddle
    contact = False
    if (self.rightEdge() >= x1 and self.rightEdge() <= x2
        and self.topEdge() <= y2 and self.bottomEdge() >= y1):
        contact = True
        # reverse horizontal direction
        self.dx = -1 * self.dx
        # wait for right edge of ball to clear the paddle
        while self.rightEdge() >= x1:
            self.move()
    return contact
```

Next, in the `play` method's game loop, add the following code just after the line that moves the ball:

```
if self.ball.checkContact(self.paddle) == True:
    print("Contact detected!")
```

Now whenever the ball hits the paddle, it should reverse direction and print the message "Contact detected!" Once everything works, add some more code to keep track of the number of hits, and increase the speed of the ball every five hits. You may want to keep track of the number of times the speed increases, and report this as the player's final score once the game ends. Feel free to add any other enhancements that you wish.

11. The Pong class can be simplified by making it a *subclass* of GraphWin. That is, instead of keeping track of a separate GraphWin object in the variable `self.win`, the Pong object will actually *be* the GraphWin object, and the `self.win` variable can be removed. As a subclass of GraphWin, the Pong class will inherit all of the functionality of GraphWin, just like the Ball class inherits the functionality of Circle. For example, instead of a Pong object calling `self.win.getMouse()`, it can just call `self.getMouse()`. Rewrite your Pong class in this way, and test the new version to make sure it works exactly as it did before.