OBJECTS AND GRAPHICS

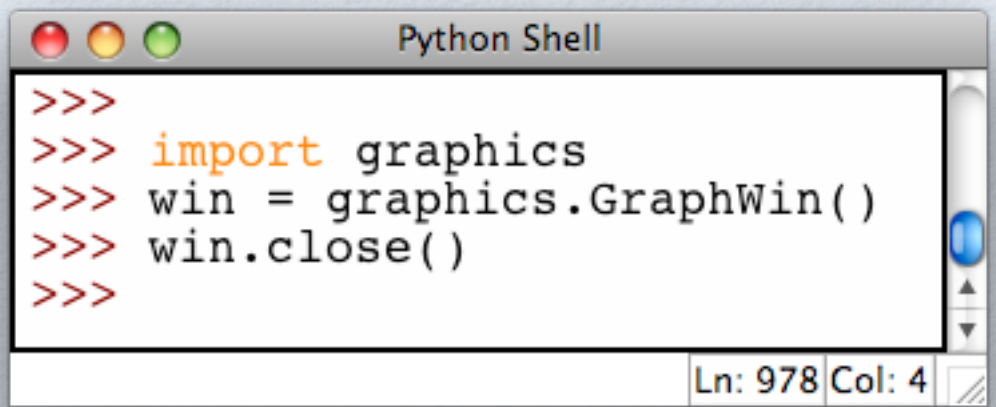
Photo from RCReptiles.com

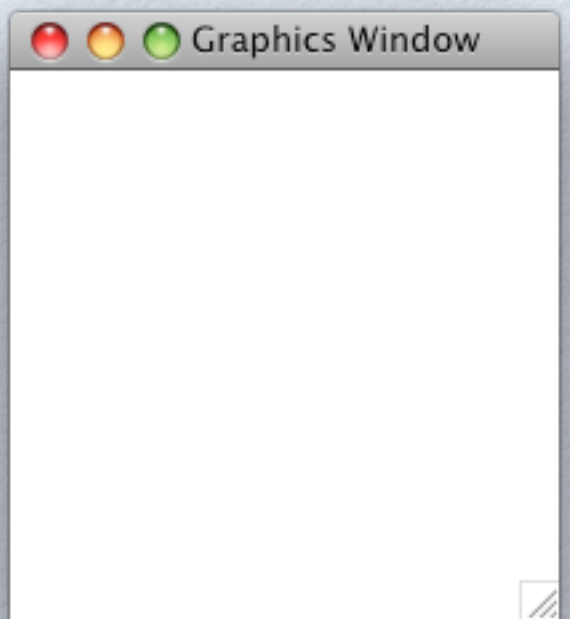
GRAPHICS

- So far, all of the input and output to our programs have been done using text.
- However, most computer programs nowadays use graphical output.
  - e.g. Windows, Firefox, even Microsoft Word
- In this session, we will learn how to make some simple graphics.

Our First Graphics Window

- The distribution of Python that we have given you (assuming you downloaded it from the course web page) has a module called `graphics.py`.
- This is our graphics module.





Using from with import

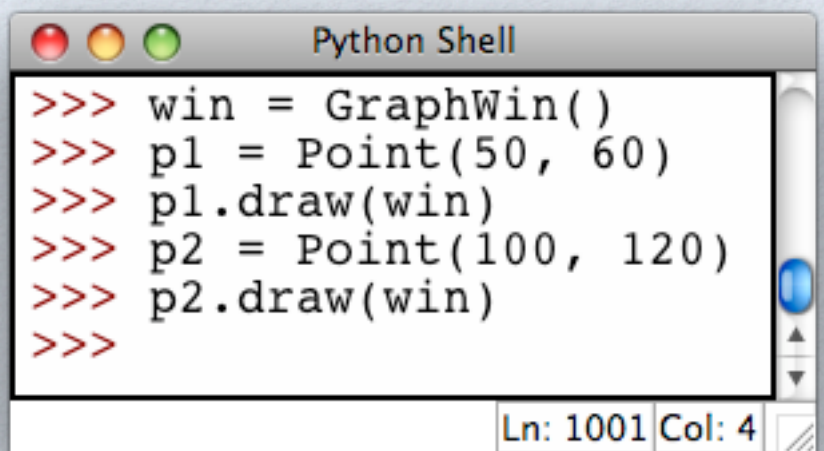
- When we type `graphics.GraphWin()`, we are telling Python: graphics module, call `GraphWin()`
- The graphics module can draw other shapes, such as points, circles, lines, etc...
- The way that we're importing the module right now, we would have to say `graphics.Point()`, `graphics.Circle()`, etc every time.
- Python provides an alternative form of import that can make this less tedious.

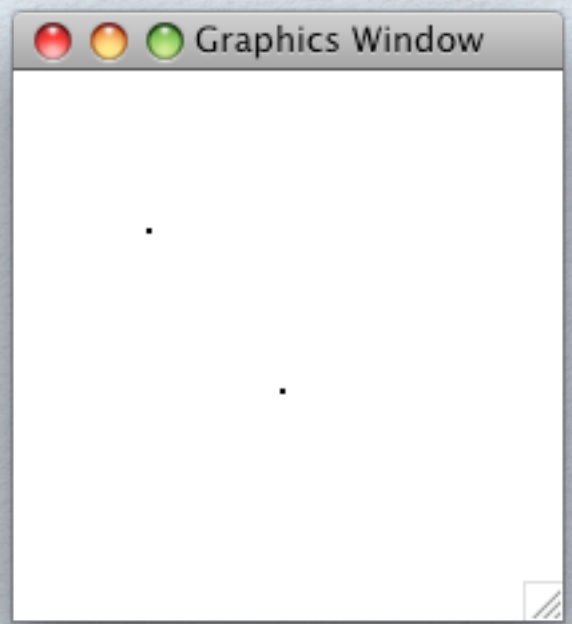
```
from graphics import *
```

- This allows us to just say `GraphWin()`, `Circle()`, etc

Inside the Graphics Window

- The graphics window is actually made out of a grid of tiny points called **pixels** (for "picture elements").
  - The "default" graphics window is 200x200 pixels.
- We draw things by controlling the color of each pixel.
- For example, we can make Points that correspond to individual pixels.
- Each point has an (x,y) position.
- (0,0) is at the *top* left hand corner of the window.





Drawing other shapes

- We could theoretically draw all shapes and even print text by drawing each individual pixel (and that's what you'll learn if you take computer graphics), but that would be quite tedious.
- The graphics module contains commands for drawing regular shapes and printing text.
- Try each of the commands on the next slide and write down what they do, in your own words.

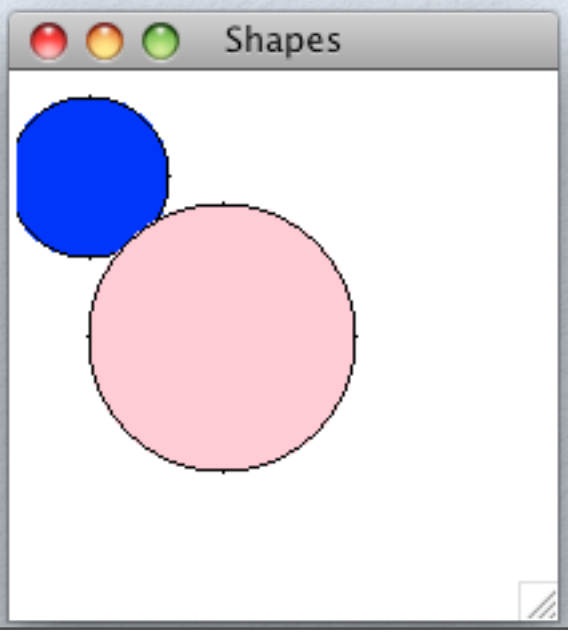
Command	Description
`win = GraphWin("Shapes")`	
`center = Point(100,100)`	
`circle = Circle(center, 30)`	
`circle.setFill("red")`	
`circle.draw(win)`	
`label = Text(center, "Red Circle")`	
`label.draw(win)`	
`rect = Rectangle(Point(30,30), Point(70,70))`	
`rect.draw(win)`	
`line = Line(Point(20,30), Point(180, 165))`	
`line.draw(win)`	
`oval = Oval(Point(20,150), Point(180. 199))`	
`oval.draw(win)`	

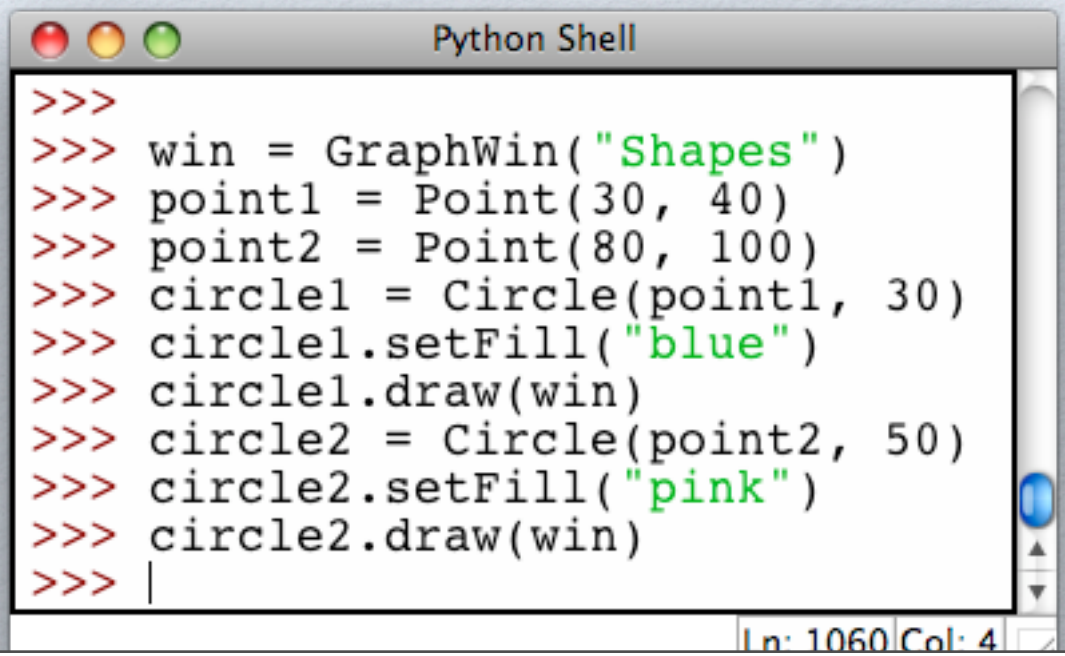
OBJECTS

- Notice the weird way in which we have been modifying the characteristics of the circles, lines and ovals:
  - `circle.setFill("Red")`, `rect.draw(win)`, etc
  - We have seen this before with strings (recall: `s.split()`, where `s` is a string)
- We're giving a command to the `circle` to perform `setFill()`, and use `"Red"` as its parameter.
- `circle` is an **object**. So are `line`, `rect`, `label` and `oval`.

OBJECTS

- Consider the example picture. We have four "things" in this picture:
- `point1` is a `Point` at (30, 40), and `point2` is a `Point` at (80, 100)
- `circle1` has its center at `point1`, and a radius of 30, and `circle2` has its center at `point2` and has a radius of 50.
- Each of these "things" is an **object**.





Objects and Classes

- From the previous slide:
  - `point1` and `point2` are both `Points`, though they are at different (x,y) coordinates.
  - `circle1` and `circle2` are both `Circles`, at different locations and with different radii.
- We say that `point1` and `point2` are both objects of the `Point` class.
- Similarly, `circle1` and `circle2` are both objects of the `Circle` class.

Objects and Classes

- A **class**, in programming, is like a **category**.
  - Things that belong to the same class have the same **characteristics**
    - All `Points` have a (x,y) coordinate.
    - All `Circles` are centered at a `Point` and have a radius.
  - Objects belonging to the same class may have different **values** for these characteristics, but they will still have the same characteristics.
- It is also like a **blueprint** for constructing objects.

Making Objects

- The program statement

```
point1 = Point(30, 40)
```

  makes a new Point object, with the (x,y) coordinates at (30, 40).

- Similarly, the program statement

```
circle1 = Circle(point1, 30)
```

  makes a new Circle object with center at the previously constructed point1 object, with a radius of 30.

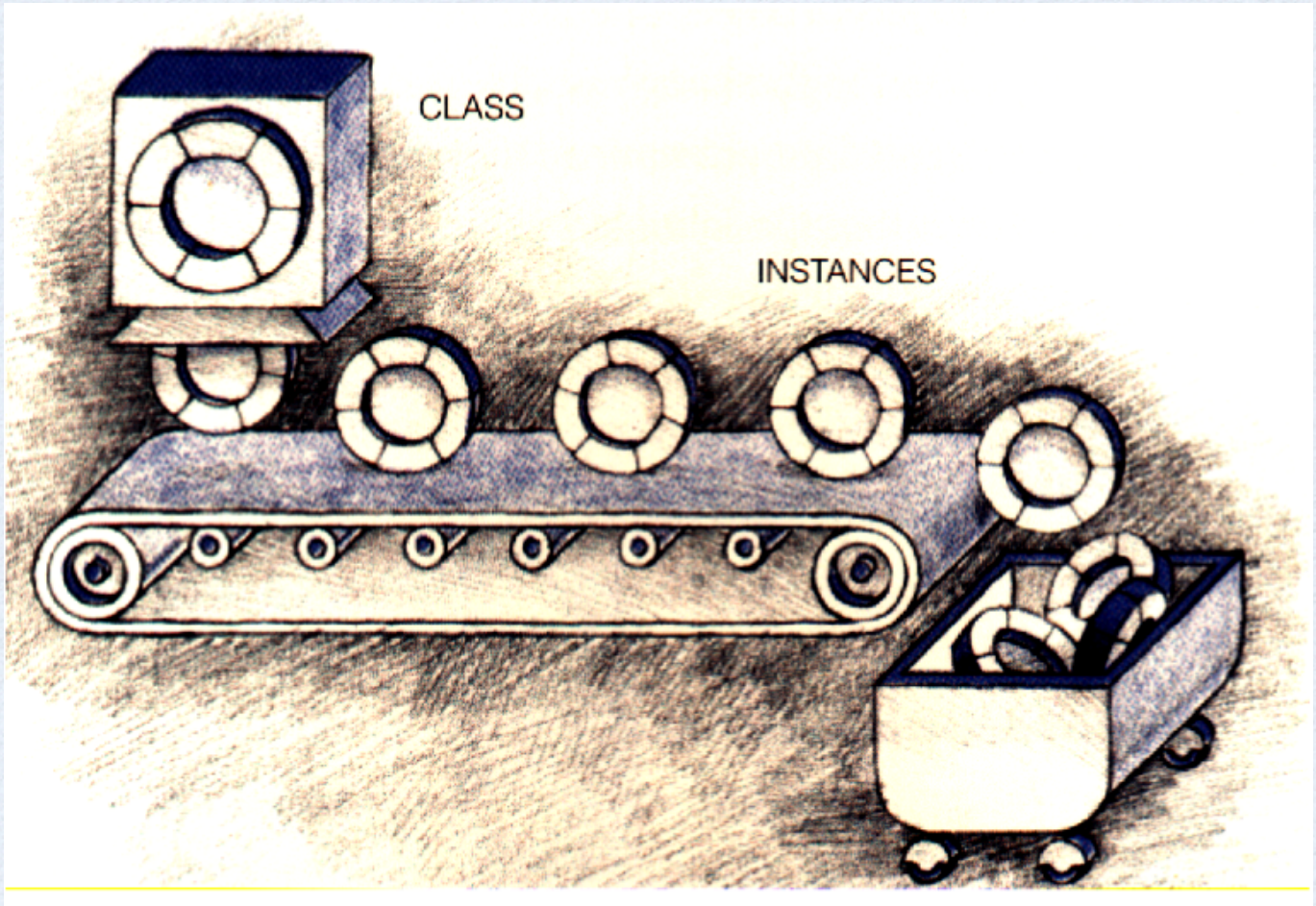
- This is called constructing an object, or **instantiating** an object.

- The result of instantiation is an object (obviously), or sometimes we also say that it's an **instance**, or **object instance**.

Object Instantiation

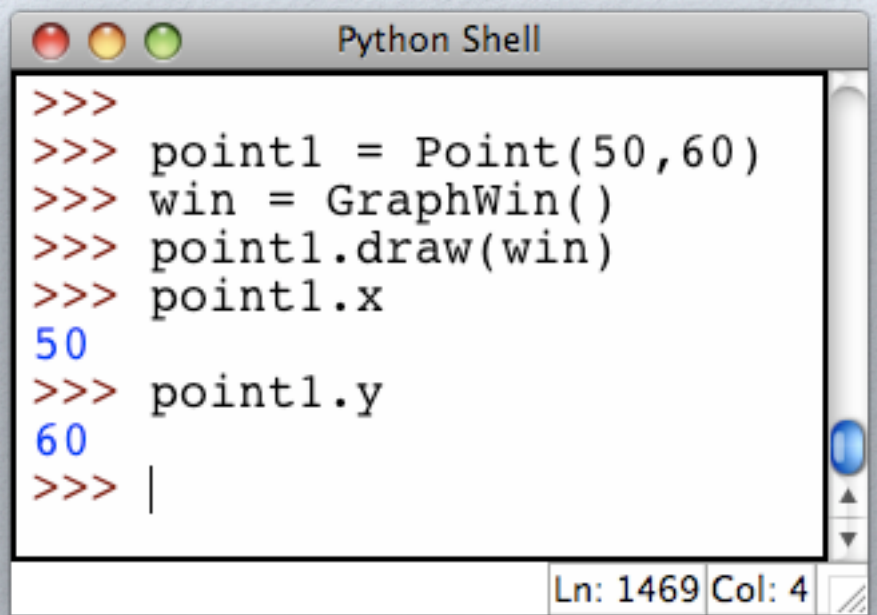
- When we instantiate an object, we have to tell the computer:
  - What class we want to instantiate the object from
  - What values we want the object instance's characteristics to take on.
- The Python syntax for object instantiation is to write the class name, followed by a pair of parentheses, with the values inside the parentheses.
  - `Point(50, 60)`
  - `GraphWin()`
  - `Circle(point1, 30)`
- All objects in Python are instantiated in the same way. The only exception is with strings, which do not need to be instantiated (but are still objects).

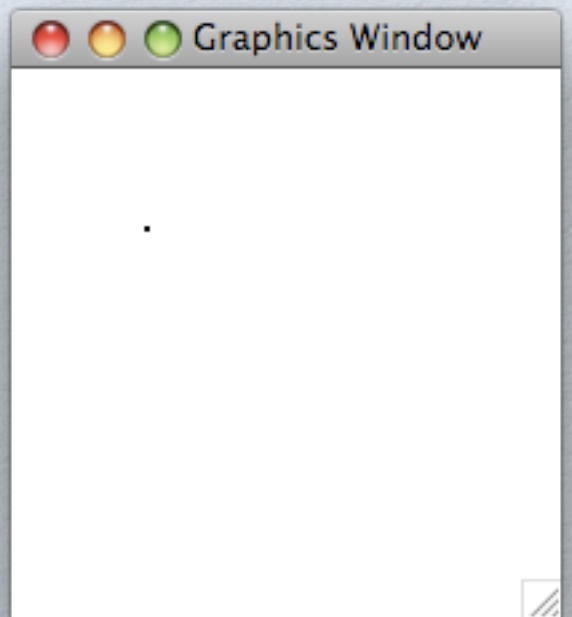
CLASSES AND OBJECTS



INSIDE AN OBJECT

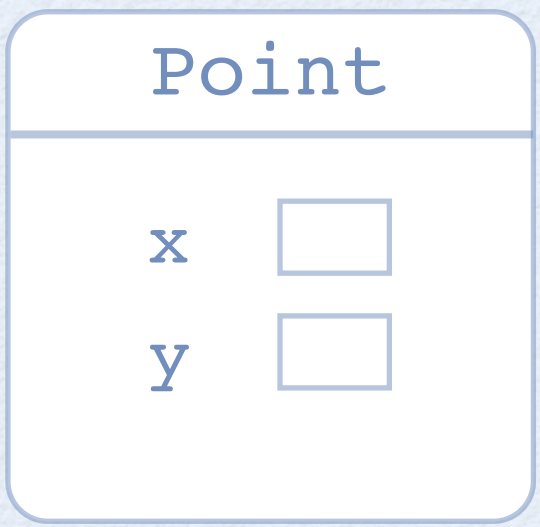
- After we instantiate the `Point` object instance, `drawing` the `Point` will put a point in the `GraphWin` window at location (50, 60).
- We can also retrieve the (x,y) coordinates and print them out.
- So obviously an object instance must remember the values for its characteristics.





INSTANCE VARIABLES

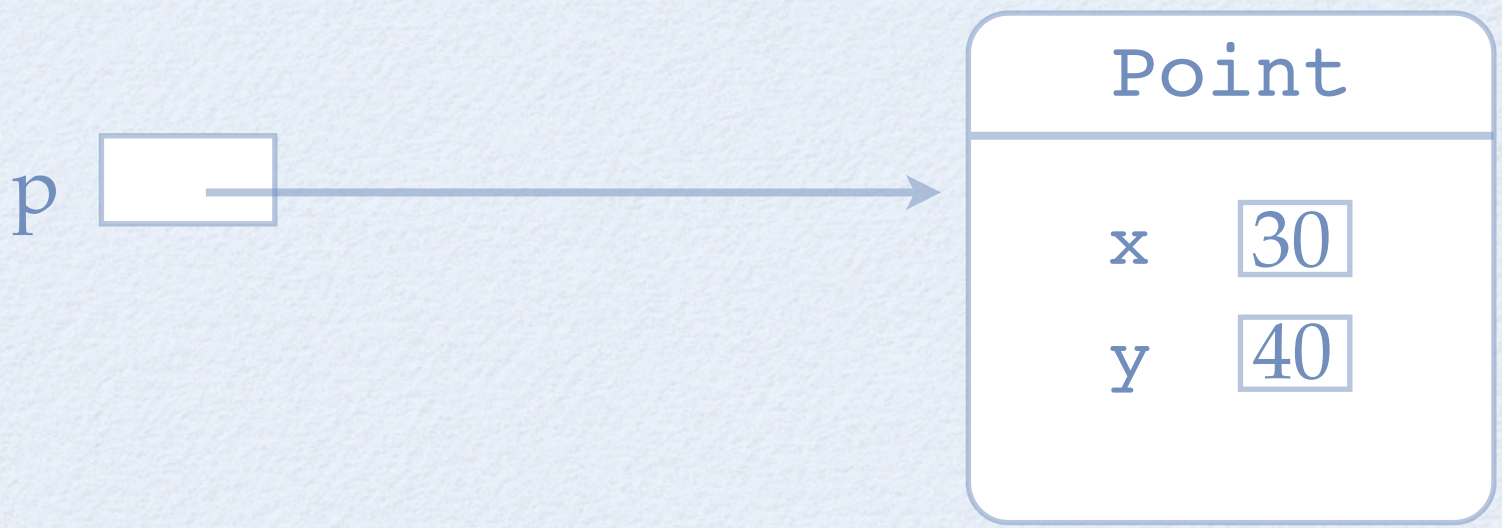
- Every Point object instance has two characteristics: the (x,y) coordinates.
- Recall: Storage places for information in the computer's memory are called **variables**.
- Variables inside an object instance are called **instance variables**, because they store data that "belongs" to a particular object instance.



Object Instantiation

```
point1 = Point(30, 40)
```

- Two things happen:
  - Python makes a Point object instance, and sets its instance variables (x,y) to (30, 40).
  - Python makes a variable called `point1`, and gets it to **refer** to the `Point` object instance.



REFERENCES AGAIN

- References are often hard for students to understand.
- The best analogy is this:
  - You are an object.
  - Your parents gave you a name.
  - Your name **refers** to you -- hence your name is a **reference** to you.
- Mapping to programming, we have:
  - A circle is an object.
  - The program has a variable that it uses to refer to the object.
  - The **reference variable** refers to the circle object.

Using Reference Variables

- A reference variable refers to an object in the same way as your name refers to you.
- In the school database, your details can be **retrieved** using your name (actually your studentID, which is another reference to you).
- Similarly, an object's data (the value of its instance variables) can be retrieved using the reference variable.
- To retrieve an object's data, we use the reference variable, plus a ".", and then the name of the data, or the name of a method that is used to get the data.

More References Examples

```
p = Point(30, 40) # p refers to the point.
p.x   # value of the x coordinate inside the point (30)
p.y   # value of the y coordinate (40)
p.getX()  # another way to get the value of x
p.getY()  # ditto for instance variable y

win = GraphWin() # win refers to the window.
win1 = GraphWin("New Window", 640, 320)  # another window

p.draw(win)  # displays (30, 40) in the first window
p.draw(win1) # displays (30, 40) in the second window.
```

YET MORE EXAMPLES

```
p = Point(30, 40) # p refers to the point.
p.getX()  # another way to get the value of x
p.getY()  # ditto for instance variable y

win = GraphWin() # win refers to the window.
p.draw(win)  # displays (30, 40) in the first window

p.getX() # What is the output?
p.getY() # What is the output?
```

Reference and Normal Variables

- Compare

```
x = 3     # x really stores the value 3
```

and

```
circle1 = Circle(Point(80, 50), 5)
# circle1 stores a reference to the Circle object.
```

- In Python, variables can only be used to store one number at a time.
  - Most of the variables that we have been using so far are normal variables.
- All other data (like Circles, strings, etc) are objects and the variable refers to them instead of actually storing their value.

Working with Objects

- To "use" an object, we need to access it via a reference variable.
- The reference variable can be used to get the object's information.

```
p.x     # Value of the x coordinate
```

- It can also be used to instruct the object to "do things".
- This tells the `Point` referred to by `p` to give us the value of its x coordinate.

```
p.getX() # Returns the x coordinate
```

- This tells the `Circle` referred to by `c` to set its fill color to red.

```
c.setFill("red") # change fill color to red
```

Methods

- Methods are "actions" that the objects can do.
  - `getX(), setFill(), move(),...`
- If objects are like nouns (data), methods are like verbs (actions)
  - Data: the (x,y) coordinates, the radius, the color
  - Actions: retrieving the value of the x coordinate, moving the position, setting the color
- We ask a particular object to perform a method by using its reference variable, then writing a ".", then the name of the method.
  - Similar syntax to retrieving the information inside an object.

Different kinds of methods

- **Constructor** methods make new objects:
  - `Point()`, `Circle()`, `Line()`, `Label()`, `Oval()`, `Rectangle()`
- **Accessor** methods retrieve the value of an object's instance variables:
  - `getX()`, `getY()`
- **Mutator** methods change information (the state) regarding an object:
  - `move()`, `setFill()`
- Other methods just... well, do other stuff.
  - `draw()`

More complex objects

- The `Point` object was a simple object.
  - Only has two instance variables: `x` and `y`
- `Circle` is more complicated:
  - It has a `center`, with an `x` and `y`, and a `radius`.
- To make a `Circle`, we can do this:

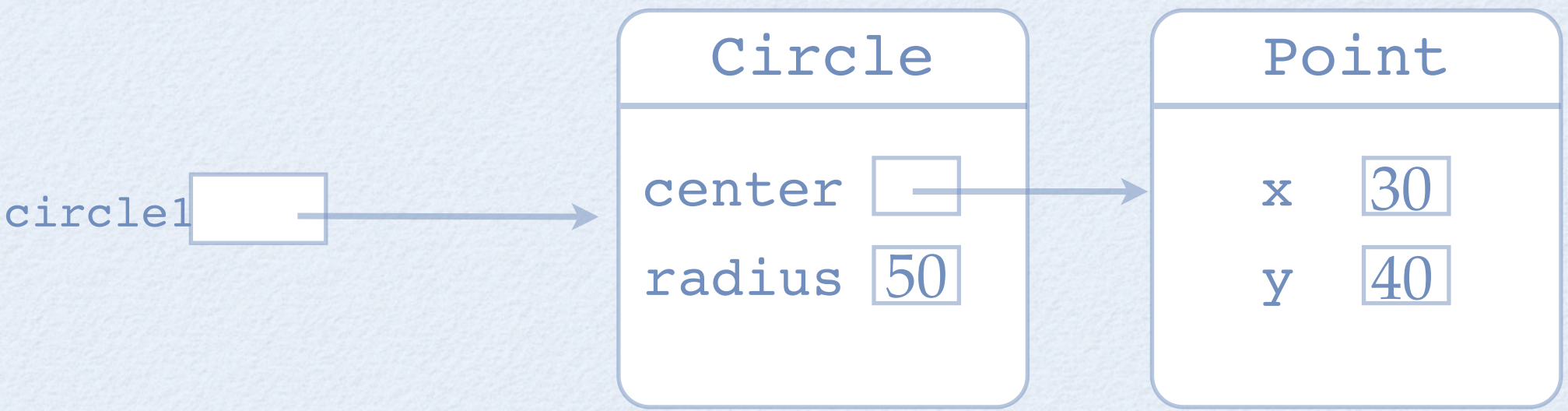
```
point1 = Point(30, 40)
circle1 = Circle(point1, 50)
```

- Or this:

```
circle1 = Circle(Point(30, 40), 50) # what's the difference?
```

INSIDE A CIRCLE

- A `Circle` has a `center` and also a `radius`.
- `center` is a reference variable, which refers to a `Point`, whose (x,y) coordinates are where the circle is centered.
- It is also an instance variable of the `Circle` object.



More Circle Methods

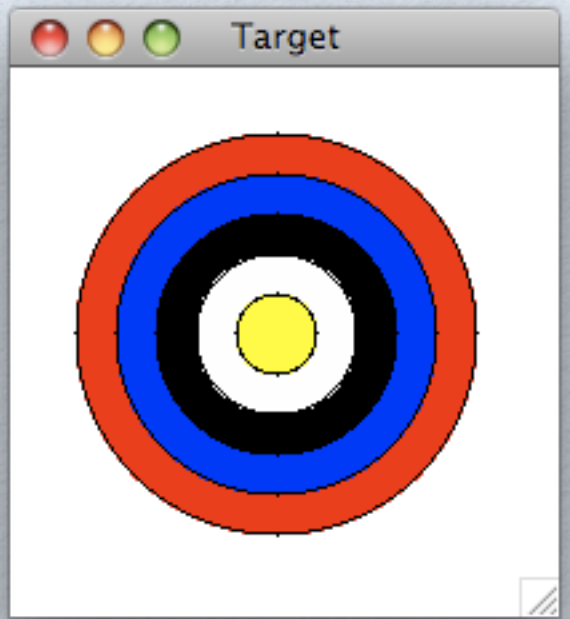
- Things that we can do with `Circles`:
  - Constructor: `Circle()`
  - Accessor: `getCenter()`, `getRadius()`
    - (Try this: How do you get the (x,y) coordinates of the center?)
  - Mutator: `move()`, `setFill()`, `setOutline()`

Why are objects useful?

- Objects and classes can seem very hard to understand at first.
- But they can be very useful.
- Consider drawing a circle in the graphics window.
- If we had to do it pixel by pixel, it would be very difficult indeed.
- But in our case, we can just make a `Circle` object, tell it to `draw()` itself in the appropriate `GraphWin` object, and it'll do all the work on its own.
- We will learn how to make our own classes later. For now, we'll just use classes that other people have written.

EXERCISE

- Try to create the following picture in the IDLE shell:
  - Center is at (100, 100)
  - Yellow circle has radius of 15
  - Each successive circle's radius increases by 15 units.



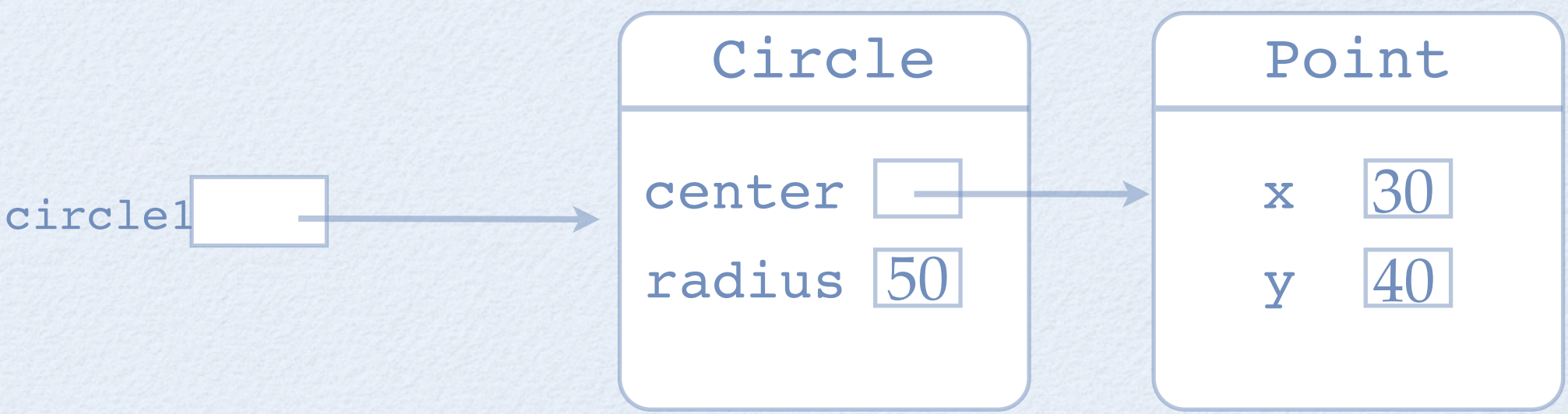
- Hint: Later objects are drawn *on top* of earlier objects.

Objects and Classes Recap

- We have learned a lot about objects and classes so far:
  - Classes are a *category* or **blueprint** for creating objects.
  - Objects are "things", such as circles, rectangles, etc.
  - We can use variables to **refer** to objects.
  - Objects can be "commanded" to perform **methods**.

Reference Variables Trickiness

- There is one tricky thing about reference variables.
- Consider the program and the illustration below.
- The reference variable `circle1` refers to the Circle object.
- It is not the Circle object!

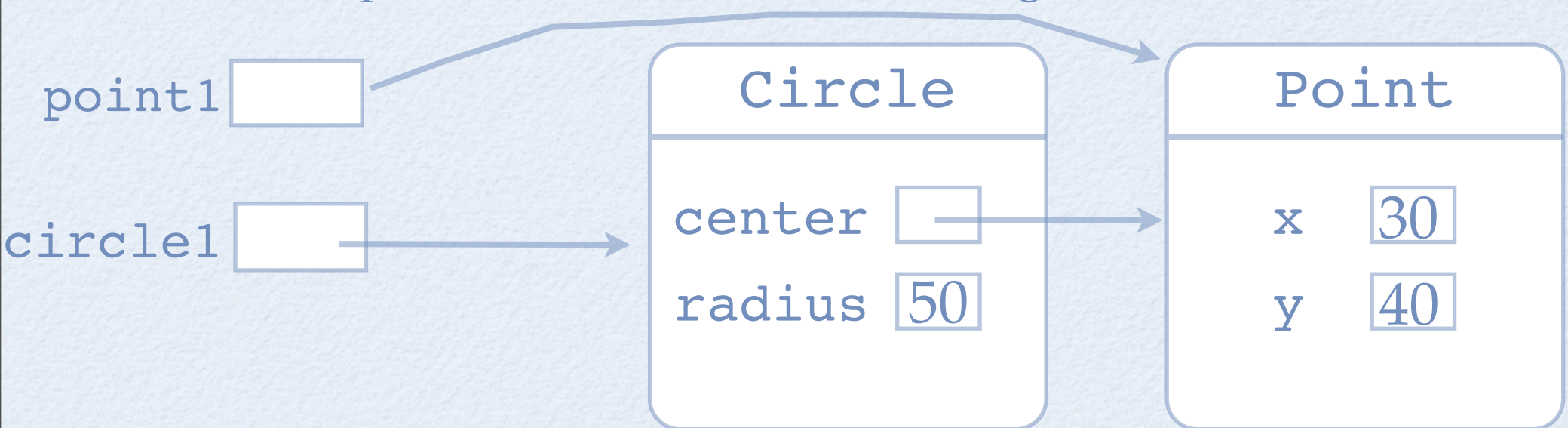


Reference Variables Trickiness

- In fact, it is possible for two different variables to refer to the same object!
- Consider the program statements:

```
point1 = Point(30, 40)
circle1 = Circle(point1, 50)
```

- The full picture would look something like this:



REFERENCE VARIABLES

- Draw the objects that would result from the following program statements:

```
circle1 = Circle(Point(30, 40), 50)
circle2 = circle1
```

My drawing:

REFERENCE VARIABLES

- What would happen if we now did:

```
circle2.move(50, 20)
```

?

My Answer:

Objects and Reference Variables

- When we assign a variable to an object, the variable *refers* to the object. It is not the object.
- If we make a new variable and assign it to equals the current variable, what we get is two variables referring to the same object, not two objects!
- Therefore if we want to make a new Circle (or any new object, for that matter), we need to instantiate a new one.
- In other words, if we want a second red Circle, we need to instantiate a new `Circle`, `setFill()` to red, and then `draw()` it, all over again.

Reference Variable Trickiness

- The relationship between reference variables and the objects they refer to is similar to the relationship between your name, or your HKID, etc, and you.
- Your name and HKID are references to you.
- When you entered PolyU, you got a studentID, which was a new reference to you.
- But there's still only one object -- you!

EXERCISES

- Describe what will happen in the following scenarios:

```
point1 = Point(40, 50)
circle1 = Circle(point1, 60)
circle1.setFill("red")
circle2 = circle1
circle2.setFill("yellow")
```

```
point1 = Point(40, 50)
circle1 = Circle(point1, 70)
circle1.setFill("red")
circle2 = Circle(point1, 70)
circle2.setFill("yellow")
```

Cloning a Shape

- Fortunately, the people who designed the graphics module gave us a easy way of making new shapes that are identical to ones that we already have.
- All the shapes in the module have a clone() method that instantiates a new, identical shape.

```
circle1 = Circle(Point(30, 40), 50)
circle1.setFill("red")
circle1.setOutline("purple")
circle1.draw(win)   # displays the circle
circle2 = circle1.clone()  # an identical copy
circle2.draw(win)   # displays the circle
circle2.move(30, 30)
```

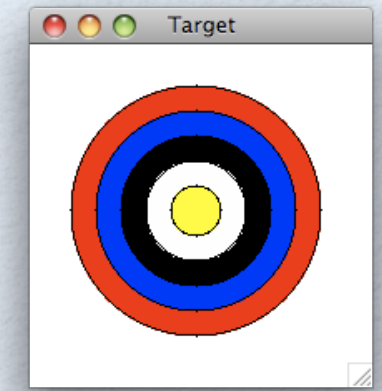
COORDINATE TRANSFORMATION

```
circle1 = Circle(Point(30, 40), 50)
```

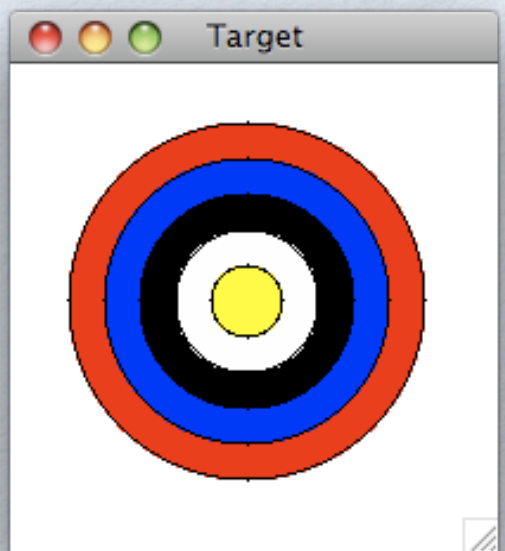
- So far, we have been using the pixel coordinates for the positions of our graphics pictures.
- This can get very tedious.
- Also, if we want to change the size of our window, then we need to calculate the pixel positions all over again.

COORDINATE TRANSFORMATION

window: 200x200 pixels
center at (100,100)
center circle's radius: 15
"width" of each circle: 15



window: 400x400 pixels
center at (200,200)
center circle's radius: ?
"width" of each circle: ?



COORDINATE TRANSFORMATION

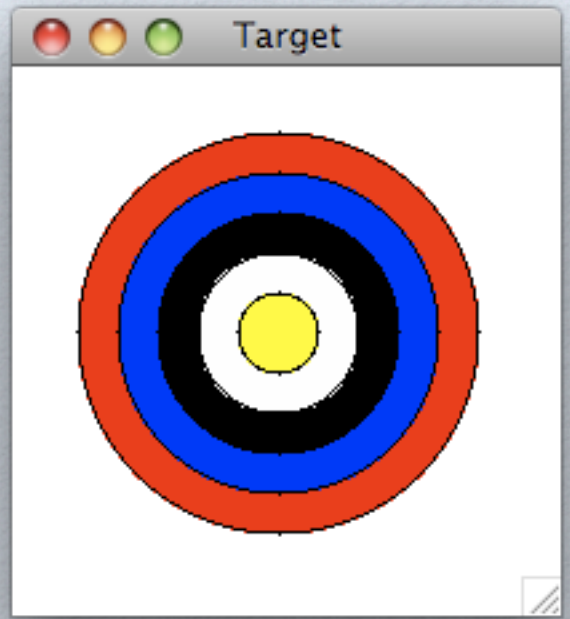
- In computer graphics, a process called *coordinate transformation* can map our desired coordinates to pixel positions.
- We can then specify the locations of our graphics objects in our coordinates.
- The computer will then calculate what these coordinates are in pixel positions.

Coordinate Transformation in the graphics.py Module

- The `GraphWin` object in the `graphics.py` module can do coordinate transformation for us.
- All we need to do is to set the coordinates of the bottom left and top right corners.
  - `win.setCoords(`*bottom-left-x*`,` *bottom-left-y*`,` *top-right-x*`,` *top-right-y*`)`
  - e.g. `win.setCoords(-1, -1, 1, 1)`
- Now, we can position our graphics objects using our coordinates and the `GraphWin` object will calculate the pixel positions for us automatically.

EXERCISE

- Using coordinate transformation, write a program to redraw the target picture from the earlier exercise:
  - Center is at (0, 0)
  - Yellow circle has radius of 1
  - Each successive circle's radius increases by 1 unit.



- Try changing the window size and watch how Python will automatically scale up/down the picture for you.

User interaction

- We now know how to give graphical output.
- We even know how to make *animated* graphical output.
- However, our input is still limited to text (keyboard).
- Could we also have non-graphical input as well?

Graphical User Interfaces

- GUIs (graphical user interfaces) are the "normal" mode of interaction nowadays.
  - e.g. Video games, Internet Explorer, Windows, even MS Word.
- Interaction with GUIs is done via an alternate programming paradigm called *event driven programming* (as opposed to *sequential programming* which we have been doing so far).
- Event-driven programming is outside of the scope of COMP 201, but we can still have interactive GUIs because the `graphics` module will do all the work for us.

Interactive Graphics in Python

- Let's learn by example.
- What do each of the following program statements do?

Python Shell

```
>>> for i in range(10):
        p = win.getMouse()
        print("You clicked (", p.getX(), ",", p.getY(), ")", sep="")


You clicked (149,99)
You clicked (70,88)
You clicked (77,138)
You clicked (131,47)
You clicked (43,53)
You clicked (72,129)
You clicked (74,159)
You clicked (127,102)
You clicked (60,100)
```

Ln: 1197 Col: 0

Mouse Clicks

- The `getMouse()` method of `GraphWin` waits for the user to click the mouse somewhere in the window.
- Each mouse click is turned into a ready-made `Point` and given to the program.
- We can use the `getX()` and `getY()` Point methods to get the location of the mouse click (in pixels).

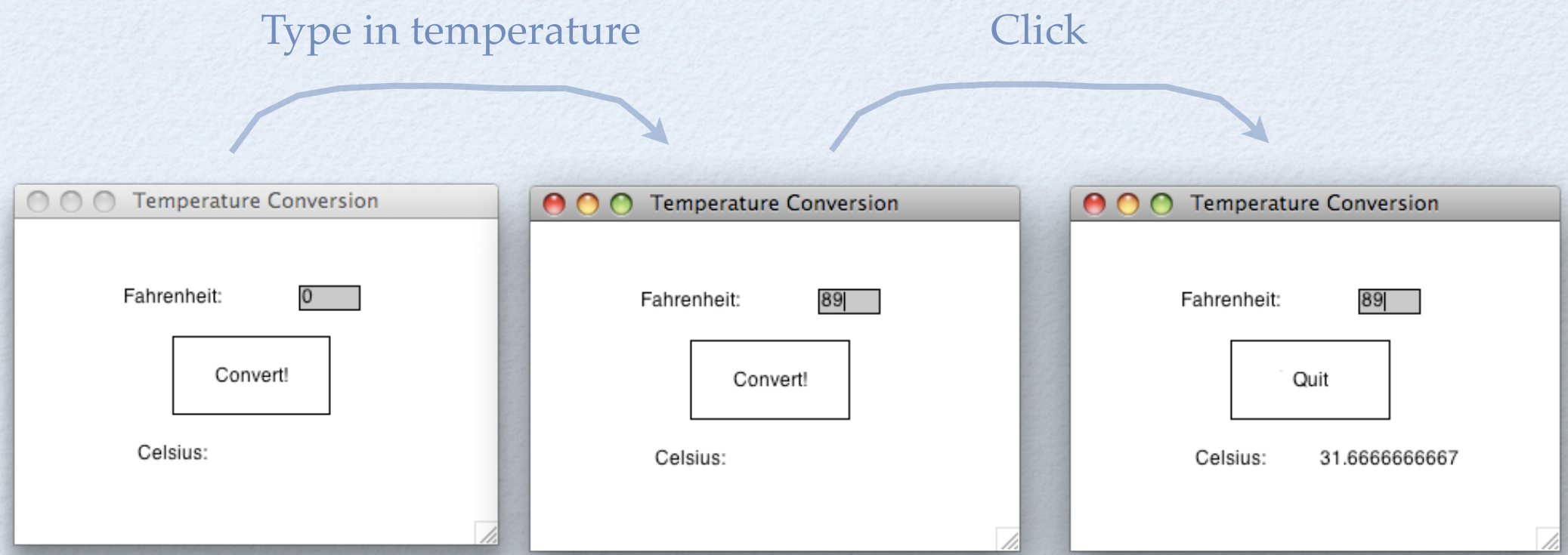
EXERCISE

- Write a program that will wait for three mouse clicks from the user and then use them to draw a triangle (the three mouse clicks are the vertices of the triangle).
- You may use the `Polygon()` object to draw your triangle:
  - `Polygon(point1, point2, point3, ... pointn)`
- Your program should not use any shell input/output and be called `DrawTriangle.py`
- You can use the Text object to give the user instructions in the graphical window:
  - `setText()`
- What is your algorithm?

MY ALGORITHM

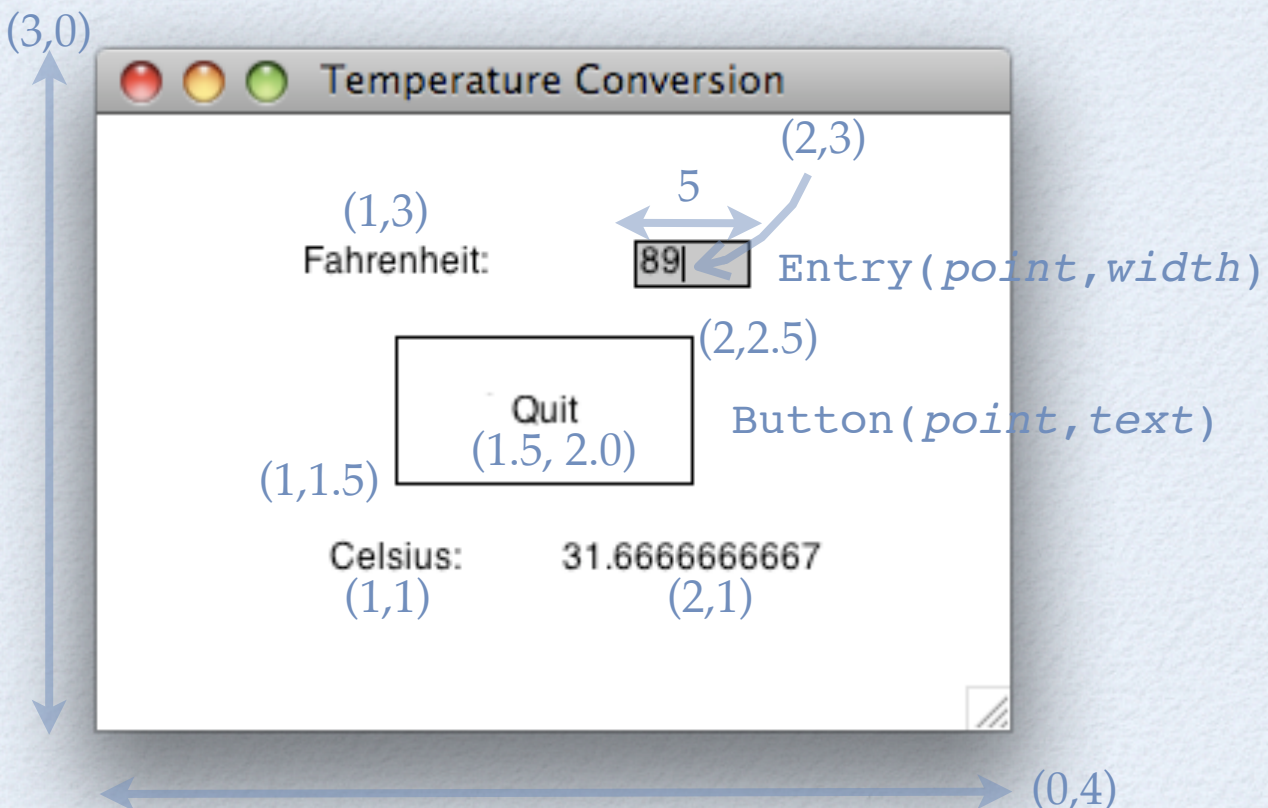
EXERCISE

- Write a program to generate the following interface:



DIMENSIONS

- Making graphical interfaces by hand, even with coordinate transformation, can get very tedious...
- In COMP 303 (Human-Computer Interaction), you will learn another language where you don't need to set these coordinates by hand.



MY ALGORITHM

Graphics Module

- The documentation for the `graphics` module can be found in Zelle, Chapter 5.8