Sit by a partner with whom you will pair program.

Assistants: Determine who is working with whom and set up the repositories accordingly.

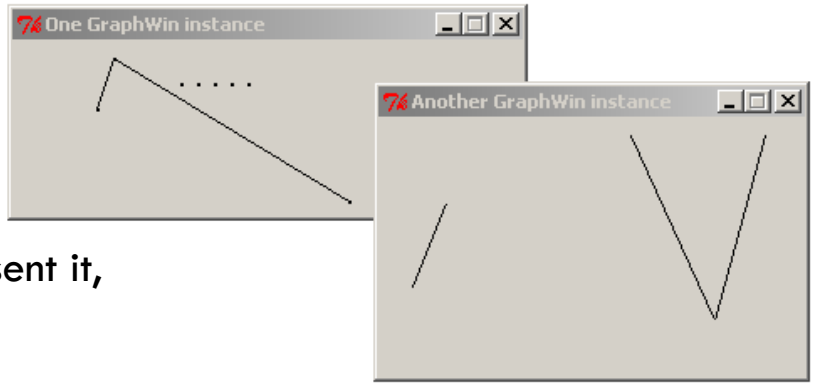
MORE ON FUNCTIONS, MORE ON OBJECTS

Outline

- Objects
  - What is a ***class***? An ***object***?
  - What do objects have: ***data attributes*** and ***methods***
  - How to ***construct*** an object
  - How to ***use an object's methods*** and data attributes
- Functions
  - Passing parameters
    - How it works, are actual arguments changed? Mutated?
    - Optional parameters
  - Returning more than one value from a function

Classes and Objects – what are they?

- A ***class*** is a *type* of things
  - Examples: Point, Line, GraphWin, Circle, Rectangle, Button, Student
  - Style: class names (and only class names) begin with a capital letter
- An ***object*** is an ***instance*** of a class
  - Example: you might have two GraphWin instances, each with many Point and Line instances on it, as shown on the picture below
- Objects have:
  - ***data attributes*** – what the object knows
    - Example: a Point object knows its x and y coordinates, the size of the dot used to represent it, the color used to draw it, and more
  - ***methods*** – what the object can do
    - Example: a Line can get its center Point, move itself by a given delta, draw itself on a given GraphWin, and more
      - Methods are like functions, but associated with its object (and may be based on its object's data)



Q1-3

How to construct an object

- The ***constructor*** for an object has the same name as the object's class.
  - For example, to construct a GraphWin:

```
win = GraphWin()
```

  - Constructors take arguments, just like functions do.
    - Some constructs allow optional arguments (i.e., arguments with default values), e.g.

```
win = GraphWin()
win = GraphWin("I love Lucy")
win = GraphWin("xxx", 800, 600)
win = GraphWin(width=500)
```

  - Constructors allocate space for the object and initialize it, setting its data attributes appropriately.
    - The object lives until there is no longer anything that refers to it, at which time the ***garbage collector*** can reclaim the object's space

How to use an object's methods and data attributes

- You ask an object to do something using the "**who . what with-what**" notation:

```
win = GraphWin()
p = Point(100, 50)
p.draw(win)
```

who **what** **with-what**

p draws itself onto the given GraphWin

Note that *p* is important – its coordinates determine where the point is drawn

- *p* is called the *implied parameter*
- *win* is called the *explicit parameter*

A similar notation lets us refer to an object's data attributes, e.g. `p.x` to refer to *p*'s x coordinate.

- In PyDev (in Eclipse) you can find out what an object can do by constructing it, then typing the variable name followed by a dot, and then pausing for a couple of seconds
  - Watch me demo this; it is easier shown (and done) than explained.
    - Sometimes you have to backspace over the dot and retype it (quirky!)
  - Alternatively, you can execute **help(*Blah*)** to see documentation about *Blah*

Q7-8

The getMouse() method of GraphWin

- Causes the program to pause,
  waiting for the user to press the mouse somewhere in the window
- Returns the Point where the mouse was pressed
  - So to find out where the mouse was pressed,
    simply assign the returned value to a variable, e.g.

```
p = win.getMouse()
```

Recall that in the default coordinate system, the y axis goes *down* from the *top* – so (0, 0) is the *upper*-left corner of the window

- ***Exercise:*** implement the ***clickMe*** module in your ***Session08b-ObjectsAndGraphics*** project. Implement ***and test*** in small stages:
  - Stage 1: The window appears, waits for a mouse-press, then disappears
  - Stage 2: The window disappears after the 6th mouse-press
  - Stage 3: For the first 5 mouse-presses, the mouse-position is displayed on the console
  - Stage 4: Draws a small red-filled circle with blue outline at the mouse position each time

Functions – Review

Example below shows:
- parameters (in purple)
- invoking built-in function (*abs*, in black)
- invoking methods (*getX, getY, in green*)
- invoking function in math module (*math.sqrt*, in red)

- Functions can have multiple ***parameters***

def keyword begins the definition of a function. Then parameters in parentheses (order matters).

colon begins ***body*** of a function

body of function is indented. End of indenting means end of function definition.

```
def distance (p1, p2): # p1, p2 are Points
    xdist = abs(p1.getX()- p2.getX() )
    ydist = abs(p1.getY()- p2.getY() )
    return math.sqrt(xdist * xdist + ydist * ydist)
```

- ***Invoke (call)*** a function; must supply ***actual arguments***:

Capture ***returned value*** in a variable

```
d = distance(Point(-1,2), Point(2,6))
```

- Functions can ***return*** values (as in the *distance* function above)

Note: Everything that we have said and will say about *functions*, applies equally well to *methods*. So we can have "ordinary" functions and functions that are methods.

Passing parameters

- Formal parameters receive values of actual parameters
  - If we assign new values to formal parameters within a function or method, does this affect the actual parameters? No!
    - Example: see ***Session09-FunctionsAndObjects***, ***functionExamples*** module, ***first*** demonstration
  - But if we ***mutate*** a formal parameter, the actual argument is also mutated
    - Example: see ***second*** demonstration

Optional parameters

- A python function may have some parameters that are optional.

```
>>> int("37")
37
>>> int("37", 10)
37
>>> int("37", 8) # specify base 8
31
```

Also look at calls to GraphWin

We can declare a parameter to be optional by supplying a default value.

- Example: see ***Session09-FunctionsAndObjects***, ***functionExamples*** module, ***third*** demonstration

```
# Demonstrates how OPTIONAL parameters work.
def printDate(month, day, year=2007):
    print "%s %d, %d" % (month, day, year)

def testOptionalParameters():
    printDate("December", 17, 2008)
    printDate("January", 1)
```

Q13

Returning Multiple Values

- A function can return multiple values
  - **def powers(n):**
    return n2, n**3, n**4**
- What's the type of the value returned by this call?
  powers(4)
- Assign returned values individually, or to a tuple:

```
listOfPowers = powers(5)
p2, p3, p4 = powers(5)
```

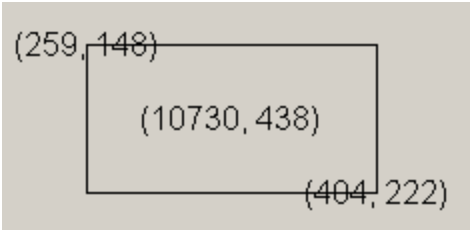
Summary

- Objects
  - What is a ***class***? An ***object***?
  - What do objects have: ***data attributes*** and ***methods***
  - How to ***construct*** an object
  - How to ***use an object's methods*** and data attributes
    - Dot and pause!
- Functions
  - Passing parameters
    - How it works, are actual arguments changed? Mutated?
    - Optional parameters
  - Returning more than one value from a function

Pair Programming: Three Rectangles

You do this project with pair programming, so there is a ***driver*** and a ***navigator***. (Switch roles for the next exercise.)

1. Driver: Checkout the **Session09-FunctionsAndObjects** project from the *partnership* SVN repository your instructor assigned to you.
   Navigator: watch Driver's screen closely and discuss problems/solutions throughout.
2. Run the **threeRectangles** module and see what it does.
3. Skim the code, talking with your partner about it, to see:
   - How the module works, so far
   - What you are to do (per the TODO: tags)
4. ***Using the Task tab***, navigate to and complete the TODO's ***in the order that they are numbered*** (1, 2, …)
   - The picture to the right shows an example of what should be displayed on each rectangle when you are done.



   - Remember the Do's of pair programming: Talk, Listen, Be patient, Be respectful.
5. When done, *Team → Commit* your project back to your partnership repository.
   - ☐ And then begin working on the rest of your homework, individually