Nicholas Bowman, Sonja Johnson-Yu, Kylie Jue
CS106AP

Section #6
July 31, 2019

Section Handout #6: Classes and Interactive Graphical Programs

1. Mouse Circles

Write a graphical program that starts with a blank window. Whenever the mouse is clicked, your program should draw a circle centered at the location of the mouse click with a random color and random radius between `MIN_RADIUS` and `MAX_RADIUS`. The end result of your program after several mouse clicks should look like what is shown in Figure 1.

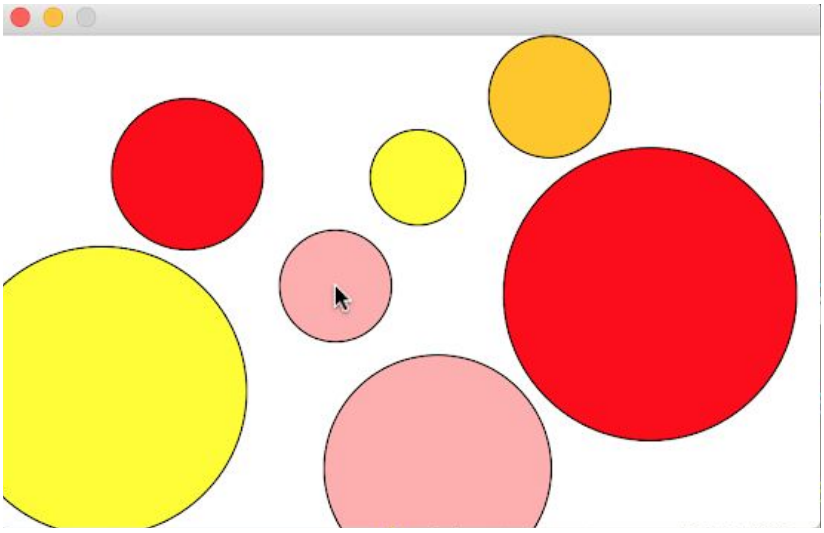

Figure 1: Result of clicking the mouse in the window seven times

Remember, we are working with mouse listeners, so you should consider breaking down this problem into 2 parts: a class definition that does most of the heavy lifting, and then a main function that creates an instance of the desired graphical window.

We suggest the following breakdown:

1. Write a method that takes in `(x, y)` coordinates and draws a circle centered at `(x, y)`. For now, use a constant radius and ignore coloring. Check that the method draws circles as you expect it to. Feel free to reuse and modify code from last week's section problems if relevant. Try calling this method directly from `main()` as a way of testing it.
2. Use campy mouse listeners to associate the above method with mouse click events. In doing so, you will need to make the expected parameters of your circle-drawing function match the format we've seen in class. Check that your program works correctly by removing the explicit calls to the previous method that exists in `main()` and then seeing if you can get circles to show up when clicking on the window.

3. Modify your original method to choose a random radius between **`MIN_RADIUS`** and **`MAX_RADIUS`** and random color. Once again, check that your program works as intended.

Optional extra challenge: make the above program do other cool things! Here's one idea: Have a mouse press draw each circle, and then drag that circle around on the screen until the mouse is released.

2. Rubber Band Graphics

Write a graphical program that allows the user to draw lines on the canvas. Clicking the mouse sets the starting point for the line. Moving the mouse moves the other endpoint of the line around, creating a visual effect where the line seems to follow the mouse around.. Clicking the mouse again fixes the line in its current position and then starts a new line. For example, suppose that you click the mouse somewhere on the screen and then move the mouse rightward an inch. What you'd like to see is what is displayed in Figure 2.

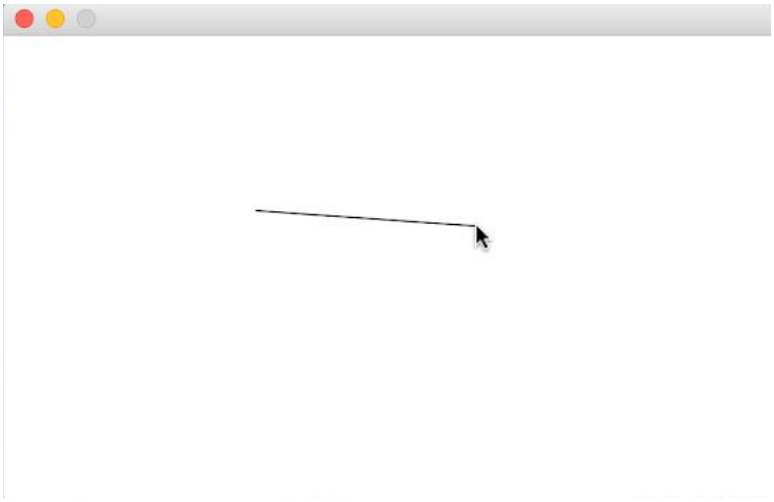

Figure 2: Program result when mouse is clicked on a location and then moved to the right

If you then move the mouse downward, the displayed line will track the mouse, as seen in Figure 3.

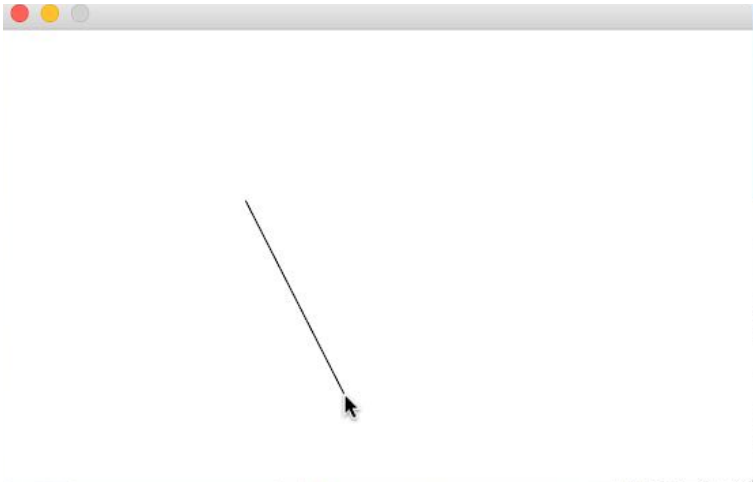

Figure 3: The line is seen following the mouse to the bottom of the screen

Because the original point and the mouse position appear to be joined by some elastic string, this technique is called rubber-banding. This program is quite powerful as it allows you to draw any figure on the screen that can be drawn with lines! An example creation is shown in Figure 4.

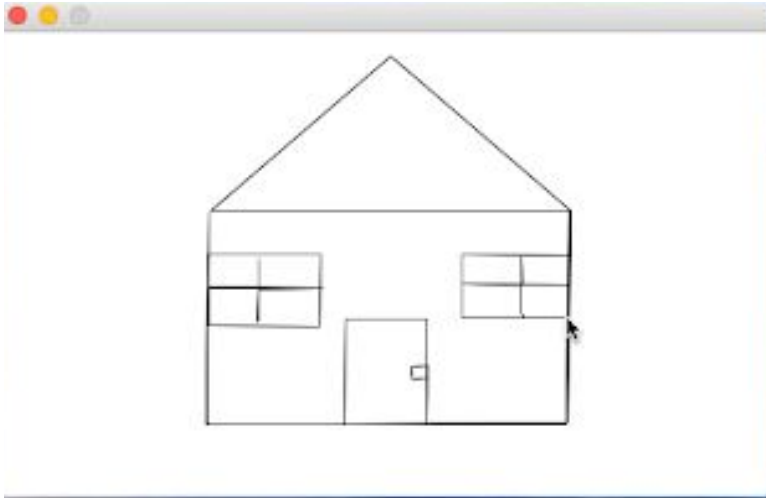

Figure 4: An example of what you can draw with the rubberband graphics program!

3. Employee Tracker

Write a class definition for a class called **`Employee`**, which keeps track of an employee's **name**, **job title, years of service at the company,** and **annual salary**. The first two attributes should be set as part of the constructor, and it should not be possible for the client to change the employee name after that. For the job title, salary, and years of service, your class definition should provide getters and setters that manipulate those fields. Your class should also implement a **`promote()`** method that promotes an employee by adding "Senior" to the front of an employee's job title and doubling their salary.

Below, we've included a sample **`main()`** function that uses the **`Employee`** class to build a console program that might help an employer. The program reads in some information about a company's employees, promotes everyone that has worked there for 5 years or longer, and then displays the resulting employee information. A sample run of the console program is shown in Figure 5. Note that defining our own class makes it much easier to store information pertaining to each employee!

```
def main():
    employees = []
    while True:
        name = input('----\nName: ')
        if name == '':
            break
        title = input('Title: ')
        salary = int(input('Salary ($): '))
        years_of_service = int(input('Years of service: '))
```

```
        employee = Employee(name, title)
        employee.set_salary(salary)
        employee.set_years_of_service(years_of_service)
        employees.append(employee)

    # promote all employees that have worked here for more than 5 years
    for employee in employees:
        employee.promote(threshold=5)

    # print all employee information
    for employee in employees:
        print('--- ' + employee.get_name() + '(' + employee.get_job_title() +
') ---')
        print('Length of service (years): ' +
str(employee.get_years_of_service()))
        print('Salary: $' + str(employee.get_salary()))
```

```
👀 python3 employee_tracker.py
----
Name: Nick
Title: Head TA
Salary ($): 2000
Years of service: 1
----
Name: Kylie
Title: Lecturer
Salary ($): 4000
Years of service: 7
----
Name: Sonja
Title: Lecturer
Salary ($): 4200
Years of service: 5
----
Name:
--- Nick (Head TA) ---
Length of service (years): 1
Salary: $2000
--- Kylie (Senior Lecturer) ---
Length of service (years): 7
Salary: $8000
--- Sonja (Senior Lecturer) ---
Length of service (years): 5
Salary: $8400
```

Figure 5: A sample run of a console program used to track employee information.

4. Up, Up, and Away!

Write two classes, an **`Airport`** class and an **`Airplane`** class, which work together to create and dispatch airplanes. The **`Airport`** class should manage the manufacturing and dispatch of airplanes. It should be able to build **`Airplane`**s, keep track of **`Airplane`**s that have been built, and tell **`Airplane`**s to **`take_off()`**. The **`Airplane`** class should act as a new variable type that helps you define a new Airplane, keep track of whether the Airplane is airborne, and implement how an **`Airplane`** can **`take_off()`**. Each airplane should have a unique integer id associated with it that is given to it when it is first built, and which does not change over the course of the plane's lifetime. In order to actually build an airplane, you have to build the fuselage and then the wings. You can assume that the **`Airplane`** class already has two methods, **`build_fuselage()`** and **`build_wings()`**, which can be called with no parameters or return values in order to build the components of the airplane. You don't need to write the code for these two methods yourself– it has been provided for you in the starter code. Another question to consider: how can a user find out whether an **`Airplane`** is airborne?

Once you have defined these two classes, write a program using the provided **`main()`** function that creates an **`Airport`**, builds 3 **`Airplane`**s, dispatches 2 of them, builds one more, and then dispatches all airplanes that haven't been dispatched yet. Consider what data structure out of all the different ones we've learned so far might make sense for keeping track of airplanes inside the **`Airport`** class – there is no one right answer, but some might be simpler or more generalizable than others for the purposes of this task.

A note on scope and decomposition, as they relate to classes: the **`Airport`** class doesn't need to know how a single **`Airplane`** is built or its internal workings; all it cares about (and should be able to do) is making, monitoring, and dispatching **`Airplane`**s according to the instructions above. Similarly, the **`Airplane`** class doesn't need to know how many airplanes are built, what data structure(s) they may be stored in, or other information about how **`Airplane`**s are managed; all it cares about is the inner workings and status of a single **`Airplane`**. Maintaining this "wall of abstraction" is key to using classes properly!