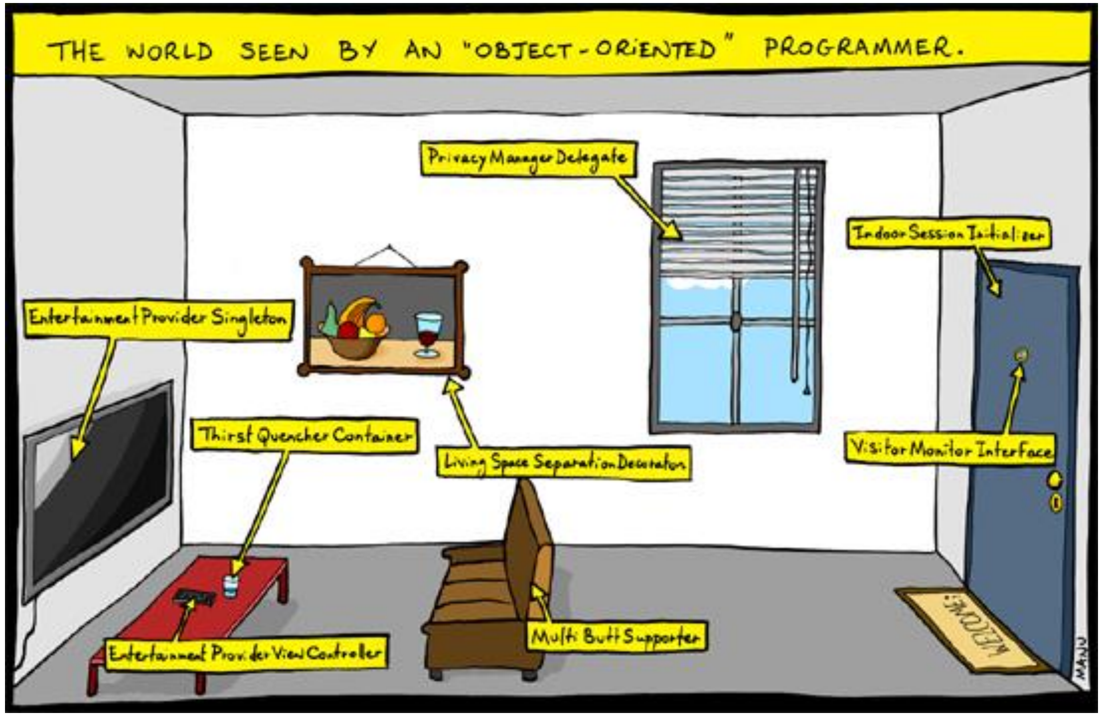
CSc 110, Autumn 2016

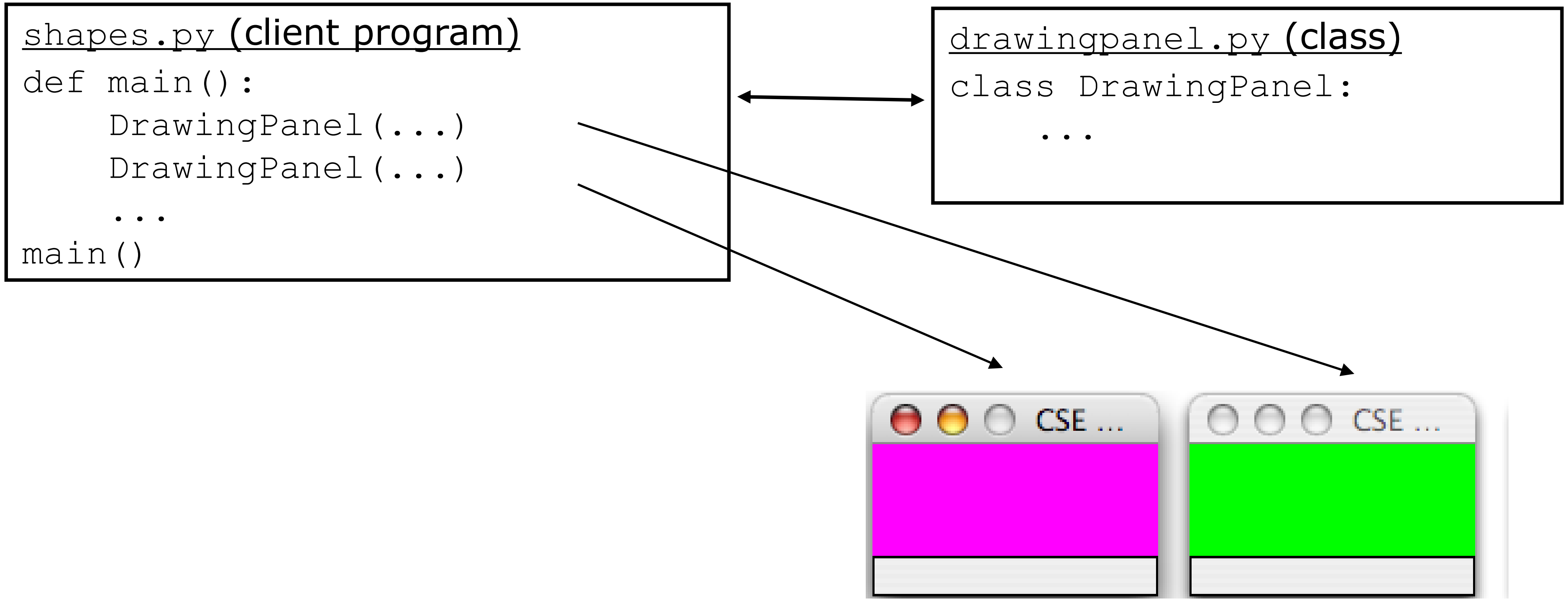
Lecture 29: Objects

Adapted from slides by Marty Stepp and Stuart Reges


THE WORLD SEEN BY AN "OBJECT-ORIENTED" PROGRAMMER.
Privacy Manager Delegate
Indoor Session Initializer
Entertainment Provider Singleton
Thirst Quencher Container
Living Space Separation Decorator
Visitor Monitor Interface
Multi Butt Supporter
Entertainment Provider View Controller
WELCOME
MANU

Clients of objects

- **client program**: A program that uses objects.
    - Example: `shapes` is a client of `DrawingPanel`.



shapes.py (client program)

```
def main():
    DrawingPanel(...)
    DrawingPanel(...)
    ...
main()
```

drawingpanel.py (class)

```
class DrawingPanel:
    ...
```

Classes and objects

- **class**: A program entity that represents either:
    1. A program / module, or
    2. **A template for a new type of objects.**

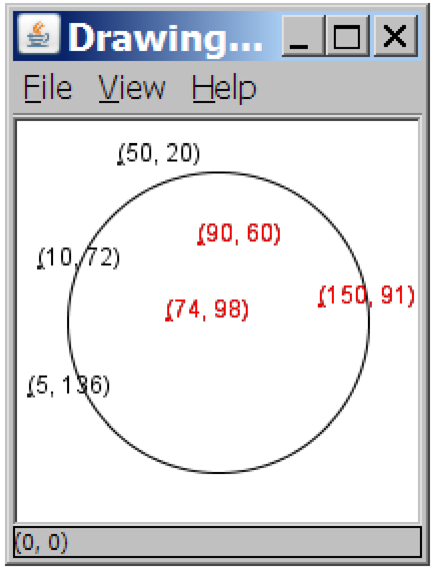
    - The `drawingpanel` class is a template for creating `DrawingPanel` objects.

- **object**: An entity that combines state and behavior.
    - **object-oriented programming (OOP)**: Programs that perform their behavior as interactions between objects.

A programming problem

- Given a file of cities' names and (x, y) coordinates:

```
Winslow 50 20
Tucson 90 60
Phoenix 10 72
Bisbee 74 98
Yuma 5 136
Page 150 91
```

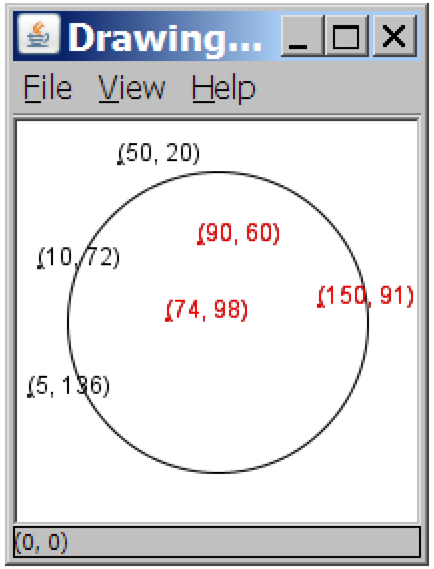


- Write a program to draw the cities on a `DrawingPanel`, then simulates an earthquake that turns all cities red that are within a given radius:

```
Epicenter x? 100
Epicenter y? 100
Affected radius? 75
```

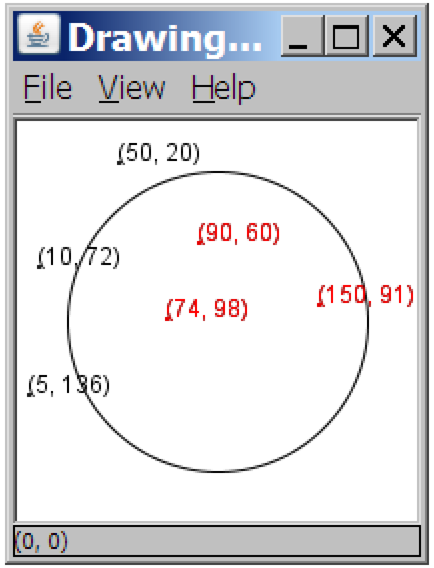
Observations

- The data in this problem is a set of points.
- It would be better stored together



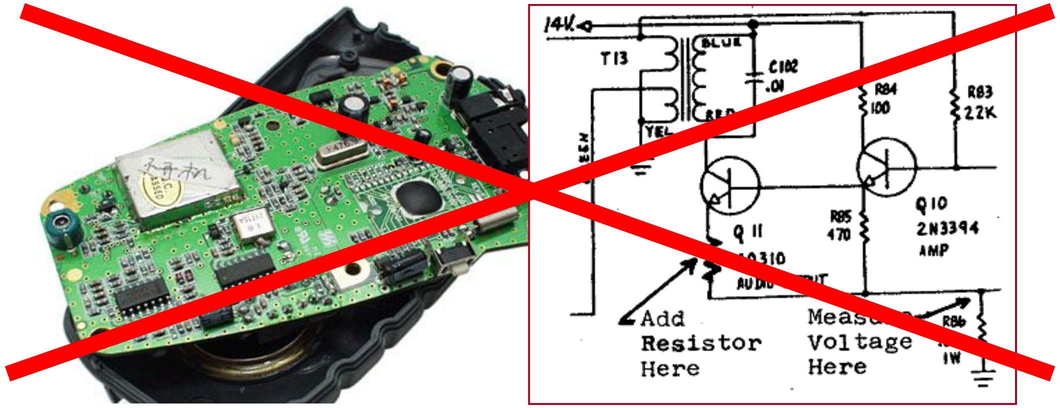
Observations

- The data in this problem is a set of points.
- It would be better stored as `Point` objects.
  - A `Point` would store a city's x/y data.
  - We could compare distances between `Points` to see whether the earthquake hit a given city.
  - Each `Point` would know how to draw itself.
  - The overall program would be shorter and cleaner.

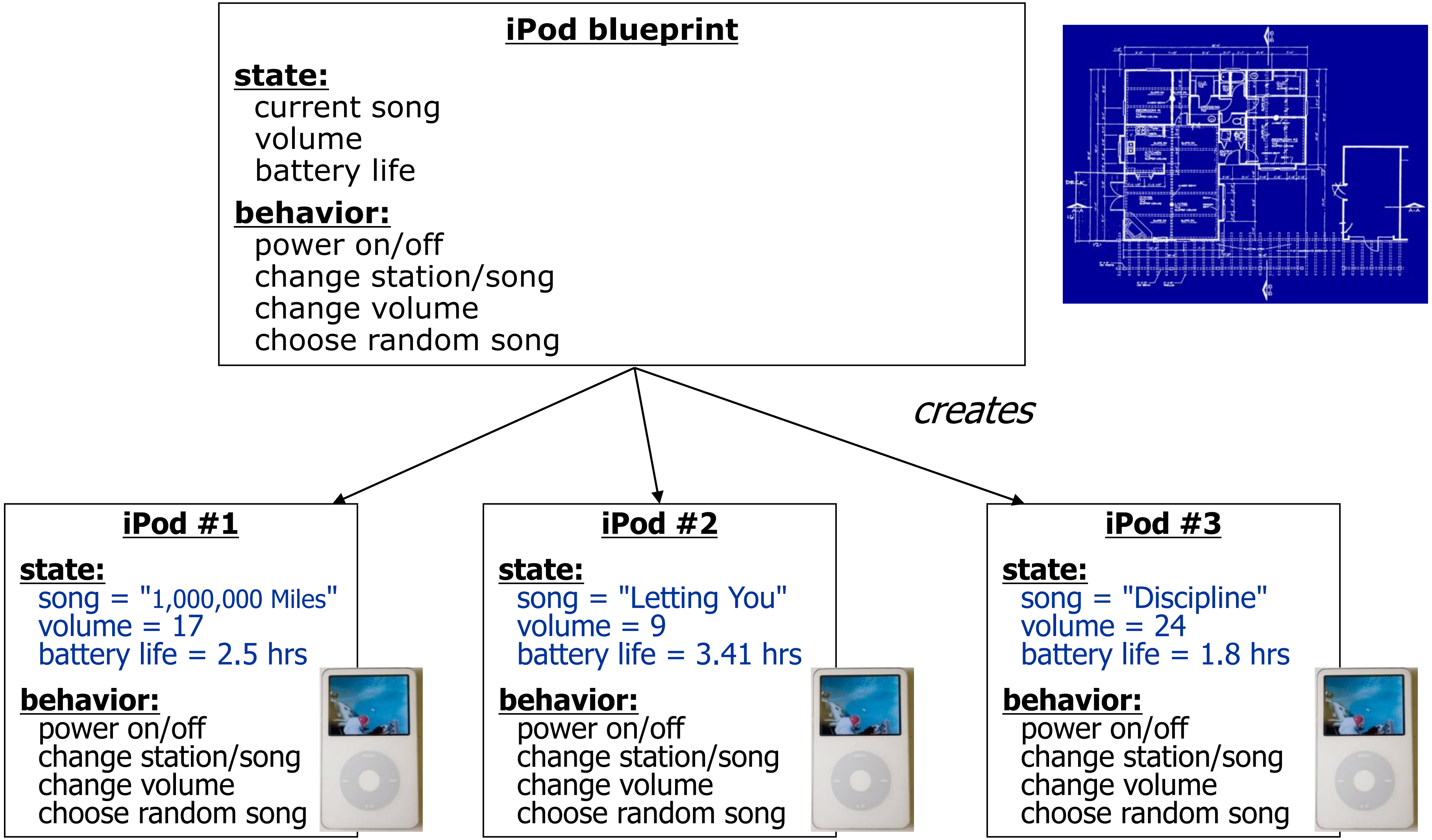


Abstraction

- **abstraction**: A distancing between ideas and details.
  - We can use objects without knowing how they work.
- abstraction in an iPod:
  - You understand its external behavior (buttons, screen).
  - You don't understand its inner details, and you don't need to.



Blueprint analogy



iPod blueprint

state:
current song
volume
battery life

behavior:
power on/off
change station/song
change volume
choose random song

creates

iPod #1

state:
song = "1,000,000 Miles"
volume = 17
battery life = 2.5 hrs

behavior:
power on/off
change station/song
change volume
choose random song

iPod #2

state:
song = "Letting You"
volume = 9
battery life = 3.41 hrs

behavior:
power on/off
change station/song
change volume
choose random song

iPod #3

state:
song = "Discipline"
volume = 24
battery life = 1.8 hrs

behavior:
power on/off
change station/song
change volume
choose random song

Our task

- In the following slides, we will implement a `Point` class as a way of learning about defining classes.
    - We will define a type of objects named `Point`.
    - Each `Point` object will contain x/y data called **fields**.
    - Each `Point` object will contain behavior called **methods**.
    - **Client programs** will use the `Point` objects.

Point objects (desired)

```
p1 = Point(5, -2)
p2 = Point()              # origin, (0, 0)
```

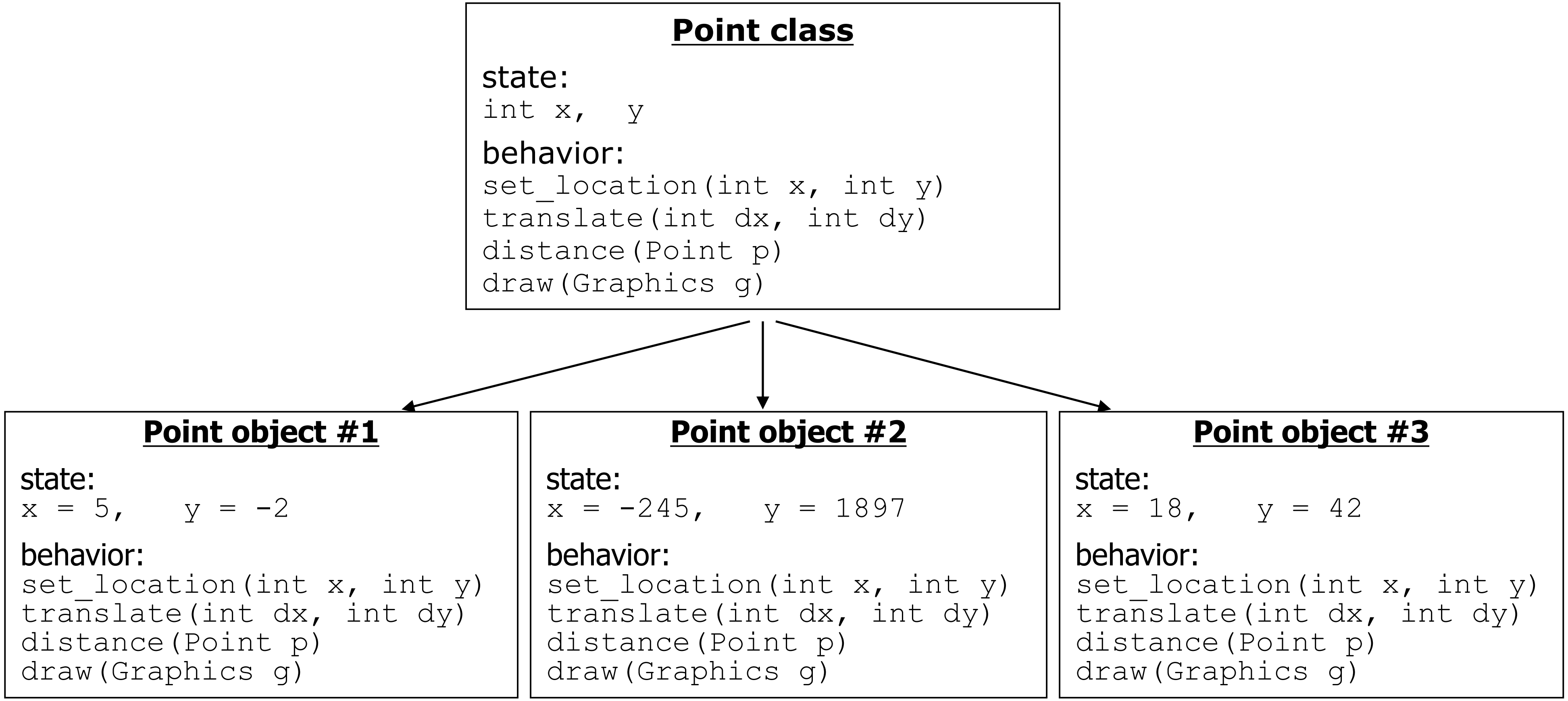
- Data in each `Point` object:

Field name	Description
`x`	the point's x-coordinate
`y`	the point's y-coordinate

- Methods in each `Point` object:

Method name	Description
`setLocation(`**`x`**`, `**`y`**`)`	sets the point's x and y to the given values
`translate(`**`dx`**`, `**`dy`**`)`	adjusts the point's x and y by the given amounts
`distance(`**`p`**`)`	how far away the point is from point *p*
`draw(`**`g`**`)`	displays the point on a drawing panel

`Point` class as blueprint



Point class

state:

```
int x,  y
```

behavior:

```
set_location(int x, int y)
translate(int dx, int dy)
distance(Point p)
draw(Graphics g)
```

Point object #1

state:

```
x = 5,   y = -2
```

behavior:

```
set_location(int x, int y)
translate(int dx, int dy)
distance(Point p)
draw(Graphics g)
```

Point object #2

state:

```
x = -245,   y = 1897
```

behavior:

```
set_location(int x, int y)
translate(int dx, int dy)
distance(Point p)
draw(Graphics g)
```

Point object #3

state:

```
x = 18,   y = 42
```

behavior:

```
set_location(int x, int y)
translate(int dx, int dy)
distance(Point p)
draw(Graphics g)
```

- The class (blueprint) will describe how to create objects.
- Each object will contain its own data and methods.

Fields

- **field**: A variable inside an object that is part of its state.
    - Each object has *its own copy* of each field.

- Declaration syntax:

  self.**name = value**

    - Example:

```
class Student:
    def __init__(self):
        self.name = ""          # each Student object has a
        self.gpa = 0.0          # name and gpa field
```

Client code redundancy

- Suppose our client program wants to draw `Point` objects:

```
# draw each city
p1 = Point()
p1.x = 15
p1.y = 37
p.canvas.create_oval(p1.x, p1.y, p1.x + 3, p2.x + 3);
p.canvas.create_string(p1.x, p1.y, "(" + str(p1.x) + ", " + str(p1.y) + ")")
```

- To draw other points, the same code must be repeated.
    - We can remove this redundancy using a method.

Eliminating redundancy, v1

- We can eliminate the redundancy with a function:

```
# Draws the given point on the DrawingPanel.
def draw(p, panel):
    panel.canvas.create_oval(p1.x, p1.y, p1.x + 3, p2.x + 3);
    panel.canvas.create_string(p1.x, p1.y, "(" + str(p1.x) + ", " + str(p1.y) + ")")
```

- `main` would call the method as follows:

```
draw(p1, panel)
```

Problems with function solution

- We are missing a major benefit of objects: code reuse.
  - Every program that draws `Point`s would need a `draw` function.

- The syntax doesn't match how we're used to using objects.

```
draw(p1, panel)      # function (bad)
```

- The point of classes is to combine state and behavior.
  - The `draw` behavior is closely related to a `Point`'s data.
  - The function belongs *inside* each `Point` object.

```
p1.draw(panel)       # inside the object (better)
```

Instance methods

- **method** (or **object function**): Exists inside each object of a class and gives behavior to each object.

```
def name(self, parameters):
    statements
```

  - same syntax as functions, but with an extra `self` parameter

Example:

```
def shout(self):
    print("HELLO THERE!")
```

Instance method example

```
class Point:
    def __init__(self):
        self.x = 0
        self.y = 0

    # Draws this Point object on the given panel
    def draw(self, panel):
        ...
```

- The `draw` method no longer has a `Point p` parameter.
- How will the method know which point to draw?
  - How will the method access that point's x/y data?

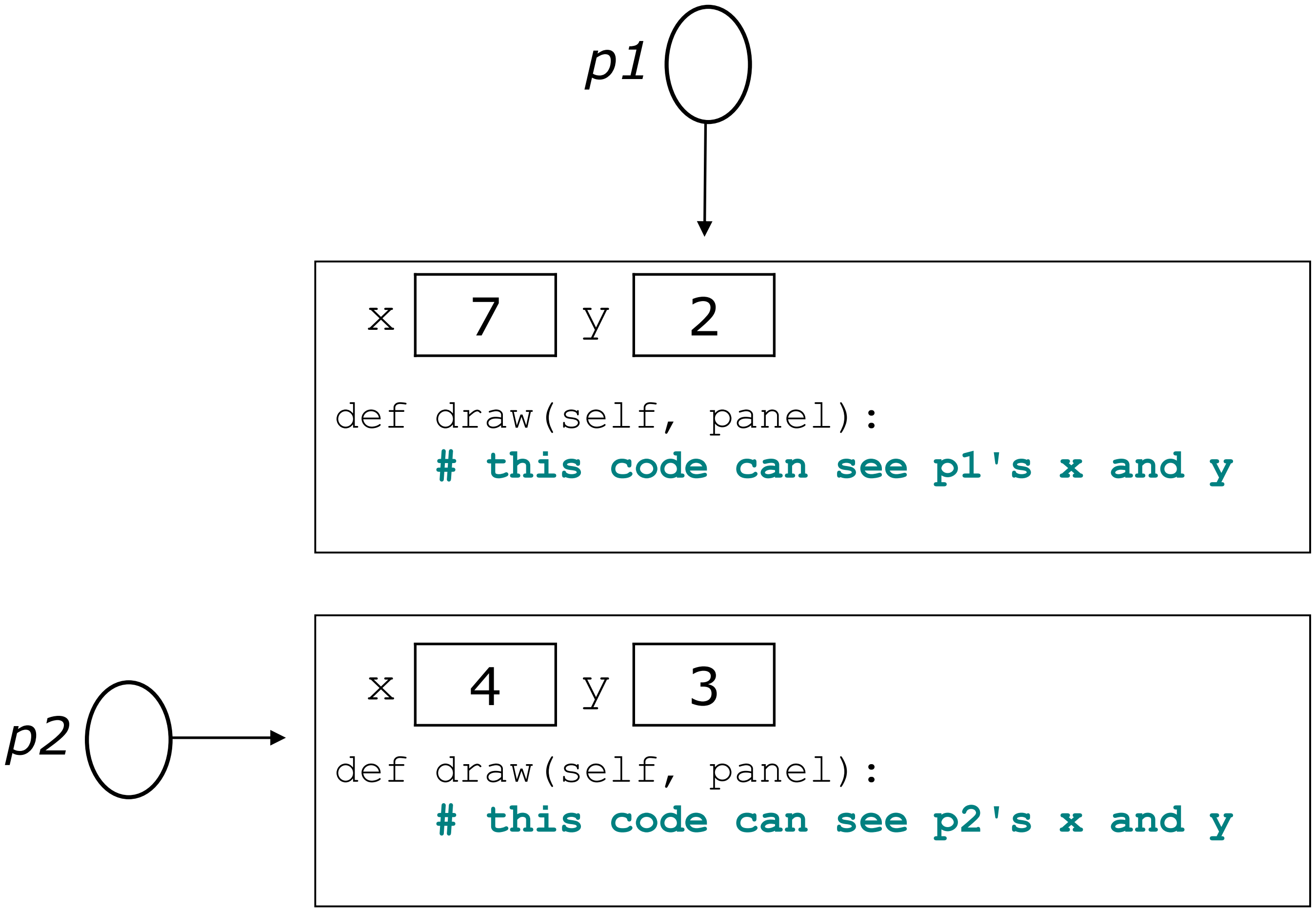
Point objects w/ method

- Each Point object has its own copy of the draw method, which operates on that object's state:

```
p1 = Point()
p1.x = 7
p1.y = 2

p2 = Point()
p2.x = 4
p2.y = 3

p1.draw(panel)
p2.draw(panel)
```



```
def draw(self, panel):
    # this code can see p1's x and y
```

```
def draw(self, panel):
    # this code can see p2's x and y
```

The implicit parameter

- **implicit parameter**:
  The object on which an instance method is called.
  - During the call `p1.draw(panel)`
    the object referred to by `p1` is the implicit parameter.
  - During the call `p2.draw(panel)`
    the object referred to by `p2` is the implicit parameter.
  - The instance method can refer to that object's fields.
    - We say that it executes in the *context* of a particular object.
    - `draw` can refer to the `x` and `y` of the object it was called on.

`Point` class, version 2

```
class Point:
    def __init__(self):
    self.x = 0
    self.y = 0

    # Changes the location of this Point object.
    def draw(self, panel):
        panel.canvas.create_rectangle(x, y, x + 3, y + 3)
        panel.canvas.create_string("(" + str(x) + ", " +
                                   str(y) + ")", x, y)
```

- Each `Point` object contains a `draw` method that draws that point at its current `x`/`y` position.

Class method questions

- Write a method `translate` that changes a `Point`'s location by a given *dx*, *dy* amount.

- Write a method `distance_from_origin` that returns the distance between a `Point` and the origin, (0, 0).

  Use the formula:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

  - Modify the `Point` and client code to use these methods.

Class method answers

```
class Point:
    def __init__(self):
        self.x
        self.y

    def translate(self, dx, dy):
        x = x + dx
        y = y + dy

    def distance_from_origin(self):
        return sqrt(x * x + y * y)
```