"A 'class' is where we teach an 'object' to behave."

-Rich Pattis

Object Oriented Programming

- "Object-oriented programming is a method of programming based on a hierarchy of classes, and well-defined and cooperating objects."
- What is a class?
- "A class is a structure that defines the data and the methods to work on that data. When you write programs in the Java language, all program data is wrapped in a class, whether it is a class you write or a class you use from the Java platform API libraries."
  - a new data type

Object Oriented Programming

- In other words break the problem up based on the things / data types that are part of the problem
- Not the only way
- One of many different kinds of strategies or paradigms for software development
  - functional, procedural, event driven, data flow, formal methods, agile or extreme, ...

Clicker 1

- What kind of assignment handout do you prefer?
  A. A long assignment handout
  B. A short assignment handout
  - Why?
Example - Monopoly

Classes Needed:

If we had to start from scratch what classes would we need to create?

A programming problem

- Given a file of cities’ (x, y) coordinates, which begins with the number of cities:
  
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>50</td>
</tr>
<tr>
<td>90</td>
<td>60</td>
</tr>
<tr>
<td>10</td>
<td>72</td>
</tr>
<tr>
<td>74</td>
<td>98</td>
</tr>
<tr>
<td>5</td>
<td>136</td>
</tr>
<tr>
<td>150</td>
<td>91</td>
</tr>
</tbody>
</table>

- Write a program to draw the cities on a DrawingPanel, then a terrible event occurs (zombie apocalypse, nuclear meltdown) that turns all cities red that are within a given radius:
  
  Ground zero x: 100
  Ground zero y: 100
  Area of effect: 75

A solution

Scanner input

```java
Scanner input = new Scanner(new File("cities.txt"));
int cityCount = input.nextInt();
int[] xCoords = new int[cityCount];
int[] yCoords = new int[cityCount];
for (int i = 0; i < cityCount; i++) {
    xCoords[i] = input.nextInt();
    yCoords[i] = input.nextInt();
}
...```

- **parallel arrays**: 2+ arrays with related data at same indexes.
  - Considered poor style. (Relationship exists in the programmer’s mind, but not explicit in the program.)

Observations

- The data in this problem is a set of points.
- An alternative is to store them as `Point` objects.
  - A `Point` would store a city’s x/y data.
  - We could compare distances between `Points` to see whether the terrible event affects a given city.
  - Each `Point` would know how to draw itself.
- The driver program would be shorter and cleaner.
**Clients of objects**

- **client program**: A program that uses objects.
  - Example: Zombies is a client of DrawingPanel and Graphics.

```java
// Zombie.java (client program)
public class Zombie {
    main(String[] args) {
        new DrawingPanel(...)
        new DrawingPanel(...)
        ...
    }
}
```

```java
// DrawingPanel.java (class)
public 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.

  - **Other classes**: String, Random, Scanner, File, ...

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

**Blueprint analogy**

- **iPod blueprint**
  - **state**: current song, volume, battery life
  - **behavior**: power on/off, change station/song, change volume, choose random song

- **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

**Abstraction**

- **abstraction**: A distancing between ideas and details.
  - We can use objects without knowing how they work.

- **abstraction in an iPhone**:
  - You understand its external behavior (buttons, screen).
  - You may not understand its inner details, and you don't need to if you just want to use it.
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)

```java
Point p1 = new Point(5, -2);
Point p2 = new Point(); // origin, (0, 0)
```

- Data in each `Point` object:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>the point’s x-coordinate</td>
</tr>
<tr>
<td>y</td>
<td>the point’s y-coordinate</td>
</tr>
</tbody>
</table>

- Methods in each `Point` object:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toString()</td>
<td>returns a String representation of this <code>Point</code></td>
</tr>
<tr>
<td>setColor(Color c)</td>
<td>Set this <code>Point</code>'s color</td>
</tr>
<tr>
<td>distance(Point p)</td>
<td>how far away the point is from point p</td>
</tr>
<tr>
<td>draw(Graphics g)</td>
<td>displays the point on a drawing panel</td>
</tr>
</tbody>
</table>

Point class as blueprint

```
Point class
---
state:
  int x, y
behavior:
  setLocation(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.

Clicker 2 What is output by the following code?

```java
Point p1 = new Point();
Point p2 = new Point();
System.out.print(p1 == p2);
```

A. Syntax error
B. Runtime error
C. false
D. true
E. no output
Object state: Fields

- **Fields**
  - **field**: A variable inside an object that is part of its state.
    - Each object has its own copy of each field.
  - Declaration syntax:
    ```java
    access_modifier type name;
    ```
    - Example:
      ```java
      public class Student {
        // each Student object has a name and gpa field (instance variable)
        private String name;
        private double gpa;
      }
      ```

- **Point class, version 1**
  ```java
  public class Point {
    private int x;
    private int y;
  }
  ```
  - Save this code into a file named `Point.java`.
  - The above code creates a new type named `Point`.
  - Each `Point` object contains two pieces of data:
    - an int named `x`, and
    - an int named `y`.
  - `Point` objects do not contain any behavior (yet).

- **Accessing fields**
  - Other classes can access/modify an object’s fields.
    - depending on the access modifier
    - access: `variable.field`
    - modify: `variable.field = value;`
  - Example:
    ```java
    Point p1 = new Point();
    Point p2 = new Point();
    System.out.println("the x-coord is " + p1.x); // access
    p2.y = 13; // modify
    ```
A class and its client

- **Point.java** is not, by itself, a runnable program.
  - A class can be used by **client** programs.

```java
public class PointMain {
    public static void main(String args) {
        Point p1 = new Point();
        p1.x = 7;
        p1.y = 2;
        Point p2 = new Point();
        p2.x = 4;
        p2.y = 3;
        ...
    }
}
```

### Point.java (class of objects)
```java
public class Point {
    int x;
    int y;
}
```

### PointMain.java (client program)
```java
public class PointMain {
    public static void main(String args) {
        Point p1 = new Point();
        p1.x = 7;
        p1.y = 2;
        Point p2 = new Point();
        p2.x = 4;
        p2.y = 3;
        ...
    }
}
```

Object behavior: Methods

Client code redundancy

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

  ```java
  // draw each city
  Point p1 = new Point();
  p1.x = 15;
  p1.y = 37;
  g.fillOval(p1.x, p1.y, 3, 3);
  g.drawString("(\(p1.x, p1.y\))", p1.x, p1.y);
  ```

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

```
main would call the method as follows:
```
```java
draw(p1, g);
```

Eliminating redundancy, v1

- We can eliminate the redundancy with a static method:

  ```java
  // Draws the given point on the DrawingPanel.
  public static void draw(Point p, Graphics g) {
      g.fillOval(p.x, p.y, 3, 3);
      g.drawString("(\(p.x, p.y\))", p.x, p.y);
  }
  ```

- main would call the method as follows:
```
```java
draw(p1, g);
```
Problems with static solution

- We are missing a major benefit of objects: code reuse.
  - Every program that draws Points would need a draw method.

- The syntax doesn't match how we're used to using objects.
  - `draw(p1, g);` // static (bad)

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

  ```java
  p1.draw(g); // inside the object (better)
  ```

Instance methods

- **instance method** (or **object method**): Exists inside each object of a class and gives behavior to each object.
  ```java
  public type name(parameters) {
    statements;
  }
  ```
  - same syntax as static methods, but without static keyword

Example:
```java
public void shout() {
    System.out.println("HELLO THERE!");
}
```

Point objects w/ method

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

  ```java
  public class Point {
    private int x;
    private int y;
    // Draws this Point object with the given pen.
    public void draw(Graphics g) {
      ...;
    }
  }
  ```

- 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?

```java
Point p1 = new Point(7, 2);
Point p2 = new Point(4, 3);
p1.draw(g);
p2.draw(g);
```
The implicit parameter

- implicit parameter:
  The object on which an instance method is called.
  - During the call `p1.draw(g);`
    the object referred to by `p1` is the implicit parameter.
  - During the call `p2.draw(g);`
    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.

Method questions

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

Point class, version 2

```java
public class Point {
    int x;
    int y;

    // Changes the location of this Point object.
    public void draw(Graphics g) {
        g.fillOval(x, y, 3, 3);
        g.drawString("(" + x + ", " + y + ")", x, y);
    }
}
```

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

Class method answers

```java
public class Point {
    int x;
    int y;

    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }

    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }
}
```