

# CSE 11

# Accelerated Intro to Programming

## Lecture 6

Greg Miranda, Spring 2021

This lecture is being recorded

# Announcements

- Quiz 6 due Monday @ 8am
- PA2 due Wednesday @ 11:59pm
- Survey 2 due tonight @ 11:59pm

# So far...

- 3 different kinds of data

- int

- • String

- • boolean *true / false*

- Very simple kinds of data

- Tons of information that can't be represented by only these 3 types of data

- Applications keep track of much more than a single piece of informatino

- • Chess game

- • Microsoft Office

- • Google Documents

# Compound Data

- Central concepts in this course

- Methods

- We will keep coming back to this

- Compound Data

- Need to know how to combine simple pieces of data together into more complicated structures
    - Idea

- Take multiple pieces of information
      - Package them together
      - Using that packaged together information

- Going to learn more about what this thing called class is
  - We've been using this idea of this keyword class
    - Writing example classes with everything inside them
  - More about what's going on when we use a class
    - When we make a class
    - How we use them to package up data
- Examples of packing data together
  - Drawing / graphing

- Drawing / Graphing - common piece of compound data
  - Tons of different fields use it
    - Art on paper, physics, engineering
  - Points, coordinates, really matter
- Points are not defined by a single number
  - Crucial to understanding points on a coordinate plane, or points anywhere
    - There are 2 numbers, packaged together that mean something
  - This also gives us a visual, graphical representation of the data
    - We can see that there are 2 numbers involved in defining the point in the upper right
- What is it going to look like to specify the shape of a point in Java?
  - Let's construct some points and do some computation




- At the top of the file, above ExamplesLec class - add class Point

```
class Point {  
    int x;  
    int y;  
}
```

- What's different about these field definitions than ones we've written before?

- What's different about these field definitions than field definitions we've written before?
  - No equals
- Every time we've written something before, we've had something on the right-hand side
  - A number or a calculation
    - `int examplesOfNum = 4 + 5;`
- We don't have that here. Why?
  - This point class is going to describe all possible points
  - Can't just pick and say:
    - "I'm only going to represent the point where x is 4 and y is 5 with this class"



- Point is going to describe the shape of any points that are made up of 2 integers
  - Without specifying any particular point
- In a different part of the program:
  - We will specify particular points
- These are uninitialized field definitions
  - Field definitions without a value
    - No value given to these
  - These are the most common kinds of field definitions that we write

# Constructor

- Next thing we need to write something that looks like a method



```
Point (int x, int y) {  
    this.x = x;  
    this.y = y;  
}
```

- What makes this different than a method?

- In many way, this is not exactly like methods we've written before

- The shape looks kind of like a method

- Main thing different:

- The only thing that appears before the parenthesis: the name of the class

- That's what makes this really special

- This has a name:

- Constructor

- We write constructors by writing the name of the class before the open parenthesis

- Instead of writing some return type and some name


- Special kinds of methods:

- Used for creating new Points in this case

- We use the constructor so we can create many different points using the same class

↓ Point

↓

- For now, ignore the body of the constructor....
  - Treat this as cookie cutter code
  - Always going to do the same pattern when we make a class
- 
  - Name of the class
  - List of parameters that exactly match the fields
  - In the body, one line per parameter
    - Says this.<name> = <name>;
- Future weeks we will get into more detail how constructors work
  - Too many details to cover right now...
  - For now, just use this pattern

# Creating Points

- Let's use the Point class to create several points

- New syntax coming...

- Point fourFivePoint = new Point(4, 5);

- Point negOneThreePoint = new Point(-1, -3);

- What's different here?

*constructor*



- This is a way we can write a program that represents the two points that were drawn earlier
  - This gives us the ability to represent these as values inside Java
- Let's run it...
  - What is this going to print?

- It's useful to think about a picture representation of these Point values
  - Not only do we have the graph
  - But it's possible to draw these Points we created as pictures

↳ memory model

# Objects

- Formal definition in Java:
  - Things that get created with new are called **objects**
  - Each time we use the new keyword with the name of the class and values for each of the fields
    - We say we created a new object
- Something interesting happened
  - Wrote a program that used new
- What did it print out?



- Print out the same kind of things as the example classes we've used all along
  - • Says **new**
  - • The name of that class
    - Inside it lists the fields and their values
- The same process is happening for the Points we created with new as the ExamplesLec class
  - This tells us something about what's going on behind the scenes
    - Whenever we use run, something is happening
      - It's printing out a whole bunch of stuff

- Now we see exactly what is happening behind the scenes
  - When we do **run**
    - It's using new to create a single ExamplesLec object
      - Just makes one of them
    - Then it print it out
  - It's as if it said:
    - **new** ExamplesLec **()**;
    - And then printed out all the fields inside ExamplesLec
  - There really is an ExamplesLec object that got created
    - Just like the two Point objects that were created

- So what is that going to look like in terms of this picture?
  - What should we draw to capture the idea that there is an ExamplesLec object that got created?

- Something we could do to see the difference
  - Make another field:
    - Not going to create a Point here
    - Point fourFivePointAgain = this.fourFivePoint;
      - Use one that already exists
  - What is going to print out now?
  - What is the contents of ExamplesLec going to be now?
  - What are possible things this could look like next?

- Objects are created
  - Whenever we have a field that is referring to an object
    - It just stores a reference to the object
  - The object is created and just sits there
    - And many fields can reference the same object

# Class Methods

- Previously we talked about methods
  - How to write them
  - How to call them
- How do we write methods that work with this compound data?
  - Like Points
- We should be able to write methods that do things with points

- Let's look at a simple method first
  - quadrant
    - takes no parameters
    - returns a string of which quadrant the points is in *String* ✓
  - What is the method header going to look like for this?
- Examples:
  - Make sure we understand what it should return for a few different cases
  - String quadA =
    - What should we write to call quadrant()?
    - What did we write before?
      - this.quadrant();
        - Does this still work?

- There is a rule based on how we call methods based on the classes of the objects that we are using
  - We have to use an object of the class that contains the method we want to call in order to call it
- To call quadrant()
  - We can't use this
    - this is referring to the ExamplesLec object
    - We have to use one of the Points
- The thing before the dot in the method call
  - Has to be a reference to an object of the class that contains the method



- Since `quadrant()` is defined inside the Point class
  - The object here has to be a reference to an object of the Point class
- `String quadA = this.fourFivePoint.quadrant();`
  - This is what we need to use to call the quadrant method on the fourFivePoint object
- “We call the method on a reference to an object”
  - this.fourFivePoints - the reference to the object we are using
  - quadrant() - the method we are calling

- String quadA = this.fourFivePoint.quadrant();

- What should this produce?

"1"

"Quadrant 1"

- Another example:

- String quadB = this.negOneThreePoint.quadrant();

- What does this produce?

"3"

- The same method, `quadrant()`, is called
  - But it changes its answer depending on which reference to an object we use to call it
- That means that whatever we write in the body of quadrant
  - Better depend on the values of the reference we used to call this method
    - Better depend on those values because we need to get different answers for these 2 lines
- The way this will work in the body of quadrant
  - We are going to use the keyword **this** again
    - The way we have used the keyword this before
- Introduce a new rule for this

↓  
-3  
• if (this.x > 0 && this.y > 0) { ... }

• When we say this.x, the value that we get when we look up this.x ↓

• Is the value of x on the object we used to call the method

• In the first case:

• We used the object that has this.fourFivePoint to call the method

• In the second case:

• We used the object this.negOneThreePoint to call the method

→ • In the first case, this.x will be 4

• In the second case, this.x will be -1

→ • In the first case, this.y will be 5

• In the second case, this.y will be -3

• The reference we get for this inside a method is always related to the reference that appeared before the method call

- `if (this.x > 0 && this.y > 0) { ... }`
  - How do we finish this?

# Weekly Pay Problem

- `weeklyPay`: takes a number of hours worked and an hourly rate, and returns the pay with overtime (over 40 hours) counting as double the rate
- Examples:

