LEC 10

# CSE 122

# Introduction to Objects

Questions during Class?

Raise hand or send here

sli.do #cse122

BEFORE WE START

***Talk to your neighbors:***

*What did you do last weekend?*

Instructor Melissa Lin

TAs Poojitha Arangam, Darel Gunawan, Colton Harris, Atharva Kashyap, Eesha Kunisetty, Audrey Lin, Di Mao, Steven Nguyen, Ben Wang, Jaylyn Zhang

# Lecture Outline

- Announcements
- OOP Review
- Example
- Abstraction
- Encapsulation

# Announcements

- Quiz 1 was on Monday
  - grades released next week
- Resub 3 form due last night
- P2 Absurdle due tomorrow
- C2 released Friday

# Lecture Outline

- Announcements
- **OOP Review**
- Example
- Abstraction
- Encapsulation

# (PCM) Object Oriented Programming (OOP)

- **procedural programming**: Program that does things by carrying out a series of steps
  - Classes that *do* things

- **object-oriented programming (OOP):** Program that does things using interactions between *things* (ie. objects)
  - Classes that *represent* things

# (PCM) Classes & Objects

- Classes can define the *template* for an object
  -  Like the blueprint for a house!

- Objects are the actual *instances* of the class
  -  Like the actual house built from the blueprint!

We create a new instance of a class with the `new` keyword

e.g., `Scanner console = new Scanner(System.in);`

# (PCM) State & Behavior

- Objects can tie related *state* and *behavior* together
- *State* is defined by the object's *fields* or *instance variables*
  - *Scanner's state may include what it's scanning, where it is in the input, etc.*
- *Behavior* is defined by the object's *instance methods*
  - *Scanner's behavior includes "getting the next token and returning it as an `int`", "returning whether there is a next token or not", etc.*

# (PCM) Syntax

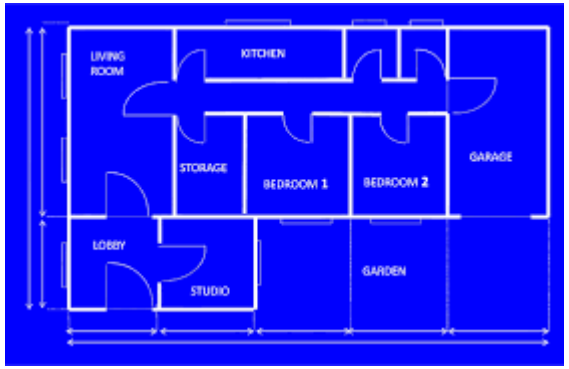
```
public class MyObject {
    // fields
    type1 fieldName1;
    type2 fieldName2;
    ...

    // instance methods
    public returnType methodName(...) {
        ...
    }
}
```

# (PCM) Instance Variables

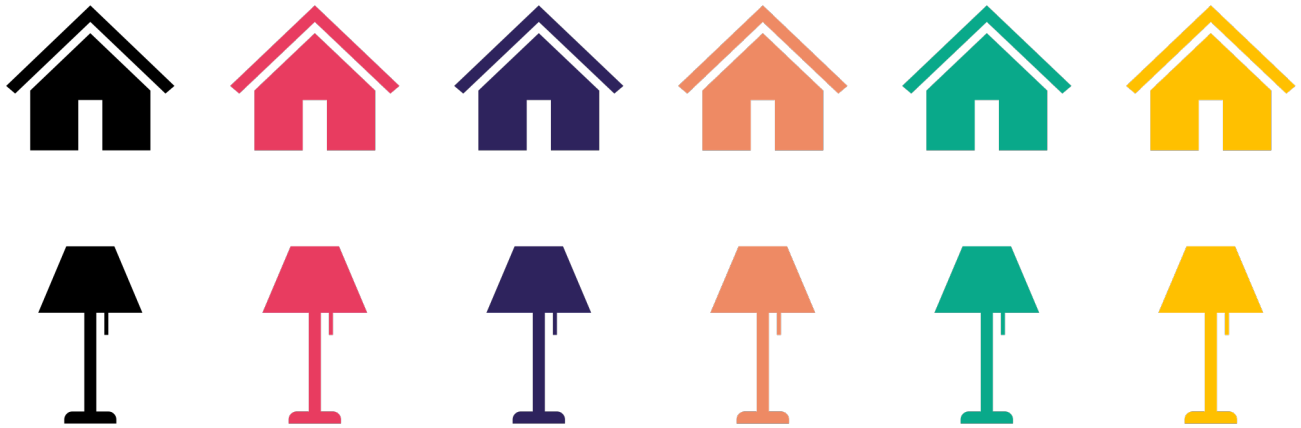
- Fields are also referred to as *instance variables*

- Fields are defined in a class
- Each *instance* of the class has their own copy of the fields
  - Hence *instance* variable! It's a variable tied to a specific instance of the class!




# (PCM) Instance Methods

- Instance methods are defined in a class
- Calling an instance method on a particular *instance* of the class will have effects on *that* instance



# Lecture Outline

- Announcements
- OOP Review
- **Example** 
- Abstraction
- Encapsulation

# Representing a point

How would we do this given what we knew last week?

Maybe `int x, int y`?

Maybe `int[]`?

# Representing a point

`int x, int y`

- Easy to mix up x, y
- Just two random `int`s floating around – easy to make mistakes!

`int[]`

- Not really what an array is for
- Again, just two `int`s – just have to "trust" that we'll remember to treat it like a point

Let's make a class instead!

# Lecture Outline

- Announcements
- OOP Review
- Example
- **Abstraction** 
- Encapsulation

# (PCM) Abstraction

The separation of ideas from details, meaning that we can *use* something without knowing exactly *how* it works.

You could use the `Scanner` class without understanding how it worked internally!

# (PCM) Client v. Implementor

We have been the *clients* of many objects this quarter!

Now we will become the *implementors* of our own objects!

# Lecture Outline

- Announcements
- OOP Review
- Example
- Abstraction
- **Encapsulation** 

# Encapsulation

Objects *encapsulate* state and expose behavior.

Encapsulation is hiding implementation details of an object from its clients.

Encapsulation provides *abstraction*.

# private

The `private` keyword is an *access modifier* (like `public`)

Fields declared `private` cannot be accessed by any code outside of the object.

We **always** want to encapsulate our objects' fields by declaring them `private`.

# Accessors and Mutators

Declaring fields as private removes all access from the user.

If we want to give some back, we can define instance methods.

| Accessors ("getters") | Mutators ("setters") |
|---|---|
| `getX()` | `setX(int newX)` |
| `getY()` | `setY(int newY)` |
| | `setLocation(int newX, int newY)` |