# Lec 4: Objects and Classes

Sam McCauley

February 17, 2026

# Admin

- Lab 2 out—start early!
- First quiz this week (details next slide)
- Today: objects and classes

# Quiz this week

- First 22 minutes of Friday will be a quiz
  - Goal: quiz “takes” 15 minutes
  - Have something to do if you finish early. (Maybe read notes for Friday lecture?)
- A practice quiz is available
- Idea: I will make 3 versions of the quiz; one for each lecture section and one practice. Each question will have a similar type and topic on all three quizzes
- The practice quiz will be available in TA hours and office hours. I’m trying to encourage you to go to these sessions, to get help from the staff and also from each other!

# Classes and Objects

# Classes and Objects

- Java is *object-oriented*, meaning that all code is organized into classes and objects
- Hopefully you've seen classes and/or objects before, but let's go through it from scratch regardless
- I'll go pretty quickly. If you're new to classes, it's particularly important that you read the lecture notes.

# What does a class do?

- Organize code into self-contained pieces
- Allows us to work with them at a higher level
- May be unnecessarily unwieldy for small pieces of code

# Classes and Objects Purpose

- Classes and objects are only used for *organization*
- You could write essentially any piece of code using what you know so far, without really taking advantage of classes/objects
- This class is all about getting comfortable with object-oriented programming—we'll be using classes and objects extensively

# What is a class?

- Recall: a function (or method) is a way to group together code that does a similar task, improving organization and allowing us to easily run that code repeatedly
- A class groups together code and data, improving organization and allowing us to use it repeatedly
- In VSCode: let's say I want to keep track of the students in a school. Each student has a name, a graduation year, and a course schedule (consisting of four courses)
  - Let's look at how we can do this without classes in `BasicStudent.java`
  - Let's discuss: what are some problems with this approach?
  - Let's look at how this works *with* classes in `Student.java`

# Instance Variables

- Classes group data together into objects.
  - `Student` is a class. A single instance of the `Student` class is an object.

- The data is stored using *instance variables*. Each is listed in the class with its name and type (and—for now—the `public` keyword)

- Every object of the class will have its own copy of each instance variable

- Can access (for now) using `.` operator

# Arrays of Objects

- In Java, each object must be *instantiated individually*
    - I believe the same is true in Python.
- We need to do the following to get an array of 20 `Student` objects:

```java
Student[] course = new Student[20]; //creates the array
for(int index = 0; index < 20; index++) {
    course[index] = new Student(); //creates a Student
}
```

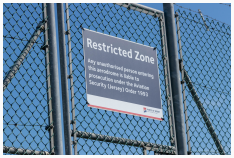
# Methods

- A `class` does not just group together data, but also groups *methods* for the data
- Methods of a class can access the instance variables directly; don't need to be passed as parameters
- We can define our own methods in each class; access using `.` operator
- Declare methods using the notation we've already seen—but no `static` keyword
  - We'll talk about `static` in a couple lectures
- Let's add a method to the `Student` class to find the year of the student

# Accessing Instance Variables Using Methods

- Instance variables should *only ever* be accessed through class methods
- Should never write something like `System.out.println(s1.name);`
- Instead, use *getter* and *setter* methods
- A getter method retrieves the value of a variable
- A setter method sets it
- Let's refactor our `Student` class to use getter and setter methods

# `public` and `private` Instance Variables




- `private` variables can *only* be accessed by methods in the same class
- `public` variables can be accessed directly, even by other classes
- We'll talk about this in more detail in a little bit

## In Pairs: Why Use Getters and Setters?

# Why use Getters and Setters?

- Answer 1: can control what data in your class can be accessed
  - If there's no getter method, other classes cannot access the data
  - If there's no setter method, other classes cannot change the data
  - Basically: hides data from either yourself, or other programmers, as they work on another part of the project
  - Not a matter of *security* (can always just write a getter method)
  - Can help when access is "dangerous"—imagine a sorted array as an instance variable
    - If someone else modifies it, they may accidentally make it so it's no longer sorted!


KEEP
OUT
OR ENTER.
I'M A SIGN,
NOT A COP

# Why use Getters and Setters?

- Answer 2: can control *how* data is accessed
  - Getters and setters can do checks before accessing data
  - Example: If your are accessing a particular element of an array instance variable, can first check that it is in bounds

```
1 public void setCourse(int index, String newCourse) {
2     if(index < 0 || index >= courses.length) {
3         System.out.println("Error: out of bounds!");
4     } else {
5         courses[index] = newCourse;
6     }
7 }
```

# Getters and Setters in this course

- All of your instance variables should be `private`
- Access them only using getters and setters
- (This is a general Java rule, not just in this course.)

# Constructors

- Special methods that call every time an object is instantiated
- (Rough) equivalent of `__init__` in Python
- Do work to “set up” the class
- Always have the same name as the class; no return value (not `void`—no return value at all)
- May have parameters; can have different constructors with different parameter types
- Let’s write two simple constructors for our `Student` class; one that sets `name` and one that sets all values

# Revisiting Object Instantiation

```
1 public Student(String inName) {
2     name = inName;
3 }
```

```
1 Student s1 = new Student("Sam")
```

When an object is instantiated, we call its constructor. That is why the notation for instantiation uses parentheses.

# What goes in a constructor

- Anything required to set up the object for the first time

- If computation is required, should be split off into a *helper method*

# Access Modifiers

# Access Modifiers for Methods

- We can also write `public` or `private` for methods
- All methods we've seen so far are `public`
- `private` is for methods that others shouldn't use
  - Helper methods that no one else has a reason to access
  - Methods that do "dangerous" things (Example: moving a value in what is supposed to be a sorted array)
  - Helper methods are usually private; signal that they are just internal helper methods
- No hard rules: the programmer needs to decide if a method is `public` or `private`

## Access Modifiers

Java has the following access modifiers:

- `public` : the method or variable is accessible from anywhere
- `private` : the method or variable can only be accessed from within the class itself
- `protected` : the method or variable can only be accessed from within the class, or from classes that *inherit* from it [We'll see inheritance right after spring break]
- *default* [meaning no access modifier specified] : the method or variable can only be accessed from a class within the same "package" [We won't cover packages in this course]

# Access Modifiers

Java has the following access modifiers:

- `publ`
- `priv` hin the class itself
- `prot` within the class, ght after spring
- *defau* ariable can only b n't cover packa

**Takeaway:**

All instance variables should be `private` or `protected`.

Methods should be `private`, `protected`, or `public`.

You should only use `private` or `public` for now.