Lecture 17
Object-oriented thinking
Why Objected-Oriented Programming?

- Abstraction and modularity
Class Abstraction

- Class abstraction is simply separating the *implementation* (the code of the class) from code to actually *use* the class.
Class Abstraction

• In order to use a class, you don’t need to know how it was written, you may not even need to look at the actual class.

• Everything you need to be able to use it will be public (sometimes called the class’s contract), everything else is hidden away in the class.

• A class can also be referred to as an abstract data type (ADT).
Abstraction and Modularity

• For example, USB devices

• You don’t need to know how each work internally, just how to talk to each of them

• You can plug them into many different computers and expect them to work the same
**Abstraction and Modularity**

- Our classes are the same - reusable, if you design them correctly and know how to interface with them.

- This is a strength of object-oriented programming – you can create objects that can be used in a variety of ways.
Abstraction and Modularity

• Think of one of our homework assignments – random password generator

• Can be used exactly 1 way, not modular

• But we could turn that into an object and then use it in different programs, change the algorithm, etc
Practice: Write a test class for this Robot object

- Without knowing anything more about a class than it’s public methods and data fields, you can create instances and use those objects in your programs!

- For example, given the following methods available for a Robot class, let’s write a program that creates an instance of a robot and have it move around the screen

  - constructor that accepts a PApplet instance (I’ll explain)
  - moveUp()
  - moveDown()
  - draw()
Object-oriented programming couples data with actions
Class Relationships

- When we’re designing classes, we have to start thinking about the relationships between our classes.

- Association, aggregation, composition
Association

- Just means a general relationship between 2 classes
• in UML, an association is a solid line

• Take and teach are *labels* for the relationship, and the arrows show the direction

• Teacher is the *role*

• *multiplicity* describes how many of the class’s objects are involved. * is unlimited, m...n means $m$ to $n$ inclusive
Association

- You implement these associations using data fields and methods
Aggregation

- *Aggregation* is an association that represents ownership
- “*has-a*” relationship. As in, a Student *has a Name*
Aggregation and Composition

• *Owner* object is the *aggregating object*, and it’s class is the *aggregating class*. The subject is the *aggregated object or class*.

• An object can be owned by several aggregating objects, but if it is exclusively owned, the relationship is called a *composition*.

• For example, “a student has a name” is a composition.

• What about “a student has an address”? 
In UML, a filled diamond is attached to aggregating class in a composition, empty diamond is attached to aggregating class in aggregation.
Aggregation and Composition

- Tip: An aggregation relationship is usually represented by a data field in the *aggregating* class

```java
public class Name {
    // ...
}
```

```java
public class Student {
    private Name name;
    private Address address;
    // ...
}
```

```java
public class Address {
    // ...
}
```

*Aggregated class*  
*Aggregating class*  
*Aggregated class*
Aggregation and Composition

- What about “a person has a supervisor”?
Designing a class example

- Let’s design a Course class
Designing a class example

```
Course

- courseName: String
- students: String[]
- numberOfStudents: int

+ Course(courseName: String)
+ getCourseName(): String
+ addStudent(student: String): void
+ dropStudent(student: String): void
+ getStudents(): String[]
+ getNumberOfStudents(): int
```
Designing a class example

```java
public class Course {
    private String courseName;
    private String[] students = new String[100];
    private int numberOfStudents;

    public Course(String courseName) {
        this.courseName = courseName;
    }

    public void addStudent(String student) {
        students[numberOfStudents] = student;
        numberOfStudents++;
    }

    public String[] getStudents() {
        return students;
    }

    public int getNumberOfStudents() {
        return numberOfStudents;
    }

    public String getCourseName() {
        return courseName;
    }
}
```