Lecture 17

Object-oriented thinking
Why Objected-Oriented Programming?

- Abstraction and modularity
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, 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
Practice: Write a test class for this Robot object

Robot

- leftMotorPin: int
- rightMotorPin: int
- servoPin: int
- leftLED: int
- rightLED: int

+ Robot()
+ moveForward(forHowManyMillis:int): void
+ moveBackward(forHowManyMillis: int): void
+ turnLeft(degrees: double): void
+ turnRight(degrees: double): void
+ wave(): void
+ wink(whichSide: String): void
+ blink(): void
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
You implement these associations using data fields and methods.

```java
public class Student {
    private Course[] courseList;
    
    public void addCourse(Course s) { ... }
}

public class Course {
    private Student[] classList;
    private Faculty faculty;
    
    public void addStudent(Student s) { ... }
    
    public void setFaculty(Faculty faculty) { ... }
}

public class Faculty {
    private Course[] courseList;
    
    public void addCourse(Course c) { ... }
}
```
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 {
    ...
}
```
```
public class Student {
    private Name name;
    private Address address;
    ...
}
```
```
public class Address {
    ...
}
```

Aggregated class  Aggregating class  Aggregated class
Aggregation and Composition

• What about “a person has a supervisor”?

![Diagram showing a relationship between Person and Supervisor]
Designing a class example

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

```java
class Course {
    private String courseName;
    private String[] students;
    private int numberOfStudents;

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

    public String getCourseName() {
        return courseName;
    }

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

    public void dropStudent(String student) {
        for (int i = 0; i < numberOfStudents; i++) {
            if (students[i].equals(student)) {
                for (int j = i; j < numberOfStudents - 1; j++) {
                    students[j] = students[j + 1];
                }
                numberOfStudents--;
                break;
            }
        }
    }

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

    public int getNumberOfStudents() {
        return numberOfStudents;
    }
}
```
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;
    }
}
```