### **Reading**

Please read Chapter 10 of book.

### **Abstraction and Encapsulation**

**Abstraction** refers to the separation of implementation from use.

We already know of abstraction provided by methods. Method separate implementation (the body) from the how they are used (the signature).

**Class Abstraction** separates class implementation from how the class is used.

Class's contract is the set of signature of the public methods and public variables.

Details of the implementation of the methods is hidden from the user. This is called **Class Encapsulation**.

Eg. you can create a Circle object and compute it's area without knowing *how* the area is computed.

A class defined in this way is called **Abstract Data Type** (ADT).

Question: What are some other examples of ADT from CS1A?

### **Object Oriented Thinking**

In CS 1A, we were programming by designing methods.

In Object-Oriented Thinking, you couple data and methods into Objects.

**Possible Case Studies:** Using ADT principles, design a class for implementing a Stack or a Queue.

### **Class Relationships**

There four types of relationships between classes

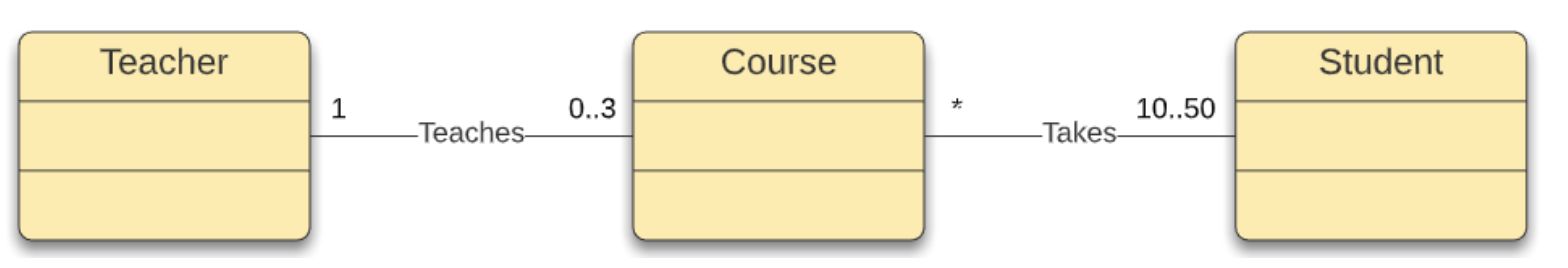
* Association - general binary relationship
* Aggregation - ownership relationship ("has a")
* Composition - exclusive ownership relationship
* Inheritance (covered later) ("is a")

### **Association**

Describes an activity between two classes.

Eg. Teacher teaches Course or Student takes Course

UML notation shows this relationship below.



Association represented by a solid line.

Optional label (eg. Teaches) describes the relationship.

Use multiplicity to specify how many objects are involved in each side of the relationship. It can be:

* a number for the exact number of objects
* \* to mean unlimited number of object
* an interval m..n for a number from m to n

So above:

* 1 Teacher can teach 0 to 3 classes per quarter
* A Course must have at least 10 and at most 50 Students; Students can take any number of courses.

In Java code, this will be represented as:

public class Teacher {

**private Course[] courses;**

public void addCourse(Course c) {

...

}

}

public class Course {

**private Student[] students;**

**private Teacher teacher;**

public void addStudent(Student s) {

...

}

public void setTeacher(Teacher t) {

...

}

}

public class Student {

**private Course[] courses;**

public void addCourse(Course c) {

...

}

}

### **Aggregation and Composition**

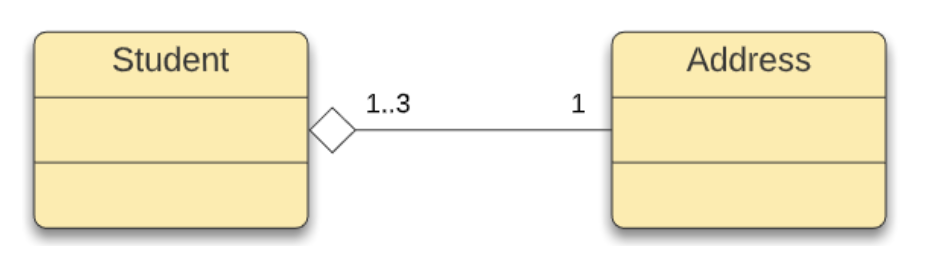
**Aggregation** is a special form of association that represents an ownership relationship.

Models **has-a** relationship.

Here, an aggregating object owns an aggregated object; an object can be owned by multiple other aggregating objects.

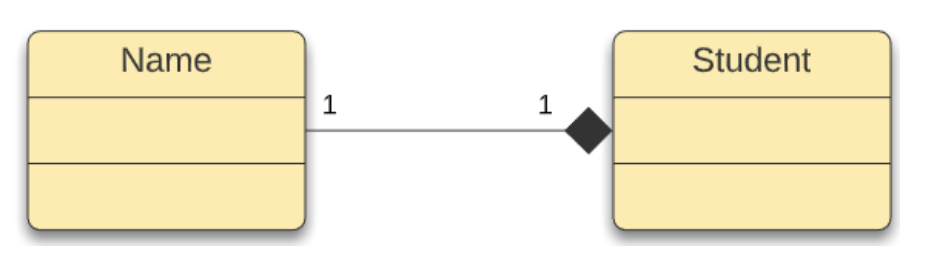
Eg. Student has an Address; the Address can belong to multiple Students

This is represented in the UML below.



When an object is owned exclusively owned by an aggregating object, the relationship is called Composition.

Eg. Student has a Name; Name can not exist with the Student

Next to the aggregating class, empty diamond represents aggregation relationship whereas a filled diamond represents a composition relationship.

Multiplicities are used as before.

In Java code, aggregating relationships are represented as variables in the aggregating class.

public class Student {

**private Name name;**

**private Address address;**

...

}

public class Name {

...

}

public class Address {

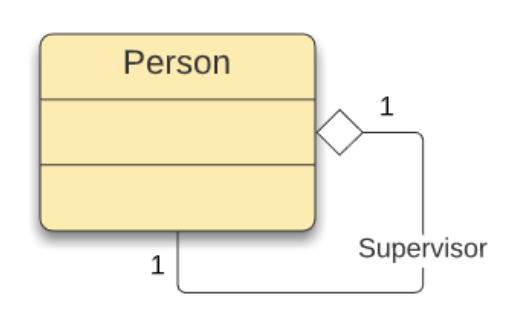
...

}

Aggregation can exist with objects of the same class.

Eg. Person can have Boss who is also a Person

This is represented in UML as below.



What would this look like in Java?

If a Person had multiple bosses, how would the UML and Java code change?