Note #3 – OO Design Concept

Dr. Lily Chang

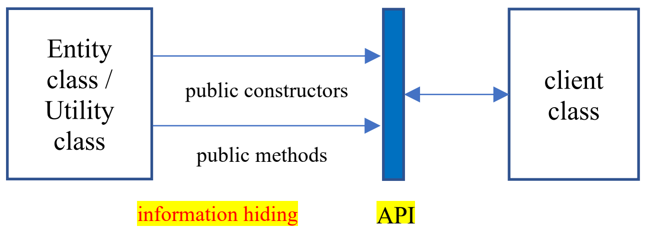
- **Data Abstraction and Encapsulation**

Implementation of a class starts with the **data abstraction**, which is to determine the attributes (data fields) needed in order to meet the requirements. **Data encapsulation** is that the data fields of a class should always be hidden from external classes (no direct data access.) In other words, private data members (instance variables) are accessed and modified through public methods (well-controlled indirect data access.) Once the attributes of a class are determined, the associated operations (methods or services) can be defined and exposed to the client classes. Indirect data access preserves the integrity of the data.

Class encapsulation is the separation of implementation from the use of a class. The details of class implementation are encapsulated and hidden from the client classes. A client class does not need to know the details of how the operations (methods) are implemented, just need to know how to call the public methods for the services provided. For example, you can create a Circle object and find the area of the circle without knowing how the area of a circle is computed. For this reason, a class is also known as an **abstract data type** (ADT).

The exposed public methods of a class are considered as a "contract" between the class providing the services and the client class that is using the services. The "contract" (or "specification") contains the signatures of the public methods that can be utilized by the client classes. The collection of public constructors, methods, and data fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the class's **specification,** also known as Application Programming Interface (API.)

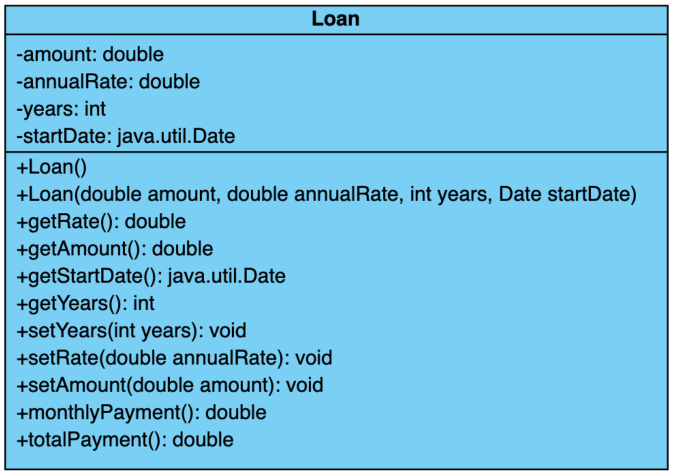
Client classes can use the public methods without knowing how they are implemented. This is also known as **information hiding.** The advantage of encapsulation and information hiding is to minimize the impact on the client classes in case there is a need to change the implementation. If the method signatures remain the same, the client classes will not be impacted at all.



Data abstraction and encapsulation are two sides of the same coin. Many real-life examples illustrate the concept of class abstraction. For example, building a computer system. Your personal computer has many components—a CPU, memory, disk, motherboard, fan, and so on. Each component can be viewed as an object that has properties and methods. To get the components to work together, you need to know only how each component is used and how it interacts with the others. You don't need to know how the components work internally. The internal implementation is encapsulated and hidden from you. You can build a computer without knowing how a component is implemented.

The computer-system analogy precisely mirrors the object-oriented approach. Each component can be viewed as an object of the class for the component. For example, you might have a class that models all kinds of fans for use in a computer, with properties such as fan size and speed and methods such as start and stop. A specific fan is an instance of this class with specific property values.

As another example, consider getting a loan. A specific loan can be viewed as an object of a **Loan** class. In order to compute the monthly and total payments, the software system must keep track of the interest rate, loan amount, and loan period in years. Therefore, the Loan class should include the relevant data fields, such as loan amount, loan period, annual interest rate and the starting date. Let's assume the Loan class is a template for car loans. When you buy a car, a loan object is created by instantiating the class with your loan interest rate, loan amount, and loan period. You can then use the methods to find the monthly payment and total payment of your loan. The client classes of the **Loan** class do not need to know how these methods are implemented.


Loan
-amount: double
-annualRate: double
-years: int
-startDate: java.util.Date
+Loan()
+Loan(double amount, double annualRate, int years, Date startDate)
+getRate(): double
+getAmount(): double
+getStartDate(): java.util.Date
+getYears(): int
+setYears(int years): void
+setRate(double annualRate): void
+setAmount(double amount): void
+monthlyPayment(): double
+totalPayment(): double

From a class developer's perspective, a class is designed for use by many different clients. In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through constructors, properties (instance variables), and methods.

- **Object Oriented Thinking**

The procedural paradigm focuses on designing functions. The programs written with procedural programming paradigm are composed of functions, routines, or procedures; for example, a C program is composed of functions, a COBOL program is composed of paragraphs, and a BASIC program is composed of routines, etc. On the other hand, software design using the object-oriented paradigm focuses on identifying objects and the operations associated with the objects. Data and methods are organized and encapsulated together into a class, which is used as an ADT for creating objects. Objects are entities in a software system representing the instances of real-world and system entities, which could be tangible entities such as tables or buildings, or intangible entities such as events or meetings. Object-oriented software design provides some **key benefits for handling changes** in the future, such as extendibility, maintainability, and reusability, etc. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

- **Stepwise Refinement**

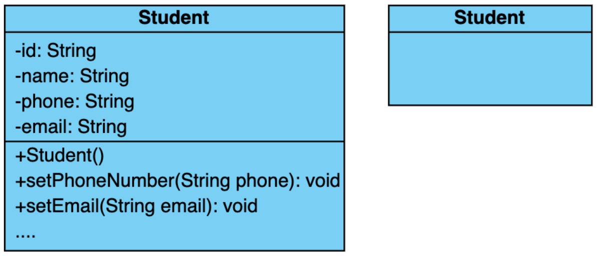
Developing software is essentially a problem-solving process. How would you get started on this process? Would you immediately start coding? Beginning programmers often start by trying to work out the code to every detail. Although details are important in the final program, concern for detail in the early stages may block the problem-solving process. To make problem-solving flows as smoothly as possible, you begin by designing the algorithms for solving the problem and isolating the details from design. Once an algorithm or solution is chosen, then you can implement the details. Stepwise refinement is a skill to deal with a complex problem. You can break a large problem into smaller manageable subproblems or add more details gradually to the solutions. This approach makes the program easier to write, reuse, debug, test, modify, and maintain.

- **Class Relationships**

A software system normally includes more than one class. In particular, a more complex software system contains many interacting classes. At the design phase of a software development process, **class diagrams** are commonly used to visualize the relationships between classes. If there are many tangled lines (relationships) in a class diagram, you would know that it is a bad design! Because this is an indication that future changes will be expensive since the change of one class will have impact on other classes (ripple effect.) Class diagram is one of the diagrams provided in UML (Unified Modeling Language). It is a very useful tool at the design phase to explore the relationships among classes. It provides a static and structural view of the software system. The objective is to create a good design with a balance of cohesion and coupling before the implementation happens.

- **Notations in a Class Diagram**

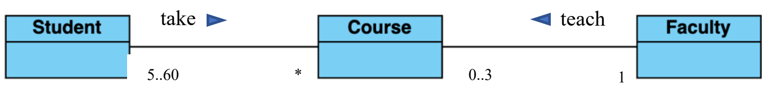
In a class diagram, classes are represented by rectangles, which either bear only the name of the class (in bold) or show attributes and operations as well. Each rectangle is divided into horizontal parts. The top part contains the name of the class. Class names begin with an uppercase letter and are singular nouns. The middle part lists the attributes (instance variables) and the third part contains the operations (methods) of the class. Attributes and operations are listed at least with their names; however, more detailed class diagrams also list the data types of the attributes, and the return types and parameters of the operations. For example, the Student class below. An **abstract class** or abstract method is indicated by the use of *italics* for the name in the class diagram. The attributes and operations are implemented as instance variables and methods in Java classes. The visibility of the attributes and operations can be defined with −, +, ~ or #, representing **private**, **public, package** or **protected**, respectively.



The common relationships among classes in a class diagram are association, dependency, aggregation, composition, and generalization.

(1) Association

Association is a general binary relationship that describes an activity between two classes. An association between two classes means that there is a structural relationship between them. Associations are represented by solid lines with or without labels. For example, a student taking a course is an association between the Student class and the Course class, and a faculty member teaching a course is an association between the Faculty class and the Course class.



In the diagram above, the labels are "take" and "teach". Each relationship may have an optional small black triangle that indicates the direction of the relationship. In this figure, the ► indicates that a student takes a course (as opposed to a course taking a student). Each class involved in the relationship may have a role name that describes the role it plays in the relationship. The role names are omitted in the above diagram.

Each class involved in an association may specify a **multiplicity** value at each end of the association line. Multiplicity specifies how many of the class's objects are involved in the relationship. A multiplicity could be a number or an interval that specifies how many of the class's objects are involved in the relationship. The character * means an unlimited number of objects, and the interval *m..n* indicates that the number of objects is between *m* and *n*, inclusively. In the diagram above, each student may take any number of courses, and each course must have at least 5 and at most 60 students. Each course is taught by only one faculty member, and a faculty member may teach from 0 to 3 courses per semester.

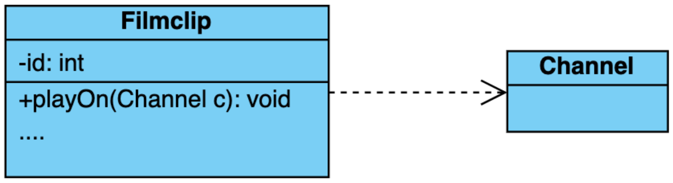
In Java programs, you can implement associations by using instance variables and methods. For example, the relationships in the class diagram above may be implemented using the classes below. The relation "a student takes a course" is implemented using the **addCourse** method in the Student class and the **addStudent** method in the Course class. The relation "a faculty teaches a course" is implemented using the **addCourse** method in the Faculty class and the **setInstructor** method in the Course class. The Student class may use a list to store the courses that the student is taking, the Faculty class may use a list to store the courses that the faculty is teaching, and the Course class may use a list to store students enrolled in the course and a data field to store the instructor who teaches the course.

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

public class Course {
    private Student[] studentlist;
    private Faculty instructor;
      ...
    public void addStudent(Student s) {
    }

    public void setInstructor(Faculty f) {
    }
}

public class Faculty {
    private Course[] courselist;
    ...
    public void addCourse(Course c) {
    }
}
```

The above example is one of the ways to implement the associations. There are many possible ways to implement associations. For example, the student and faculty information in the Course class can be omitted, since they are already in the Student and Faculty class.

(2) Dependency

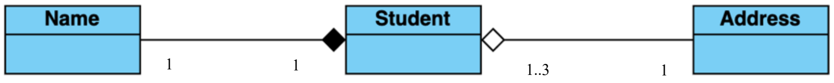
A dependency relationship represents another connection between classes and is indicated by a dashed line, with optional arrows at the ends and with optional labels. One class depends on another if changes to the second class might require changes to the first class. An association from one class to another automatically indicates a dependency. No dashed line is needed between classes if there is already an association between them. However, for a transient relationship, i.e., a class that does not maintain any long-term connection to another class but does use that class occasionally, we should draw a dashed line from the first class to the second. For example, the Filmclip class "uses" the Channel class as a parameter in one of the method signatures, but does not use it as the data type of some instance variable.



(3) Aggregation and Composition

Aggregation is a special form of association that represents an ownership relationship between two objects. Aggregation models "has-a" relationships. The owner object is called an aggregating object, and its class is called an aggregating class. The subject object is called an aggregated object, and its class is called an aggregated class. The aggregation relationship is indicated by a hollow diamond on the owner object's end.

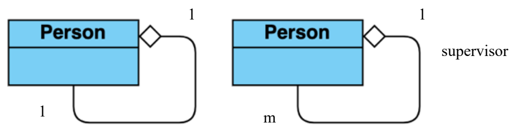
We refer the aggregation between two objects as **composition** if the existence of the aggregated object is dependent on the aggregating object. In other words, if a relationship is a composition, the aggregated object cannot exist on its own. For example, "a student has a name" is a composition relationship between the Student class and the Name class because Name is dependent on Student, whereas "a student has an address" is an aggregation relationship between the Student class and the Address class because an address can exist by itself. Composition implies exclusive ownership, i.e., an object owns another object. The parts live and die with the owner because they have no role in the software system independent of the owner. In UML, a filled diamond is attached to the aggregating class (in this case, Student) to denote the composition relationship with an aggregated class (Name), and an empty diamond is attached to an aggregating class (Student) to denote the aggregation relationship with an aggregated class (Address), as shown below.



In the diagram above, each student has only one address and each address can be shared by up to 3 students. Each student has one name, and the name is unique for each student. An aggregation relationship is usually implemented as a data field in the aggregating class. For example, the relationships in the diagram above may be implemented using the Student class, Name class and Address class. The relation "a student has a name" and "a student has an address" are implemented in the data field name and address in the Student class.

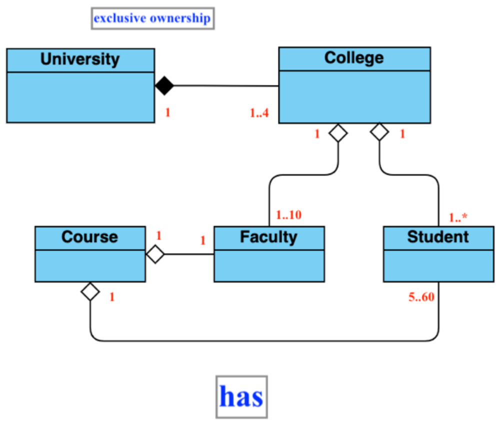
```
public class Student {
    private Name name;
    private Address address;
    ...
}
```

Aggregation may exist between objects of the same class. For example, a person may have a supervisor. This is illustrated in the figure below. In the relationship "a person has a supervisor," a supervisor can be represented as an instance variable in the Person class. If a person can have several supervisors, you may use an array to store the supervisors.



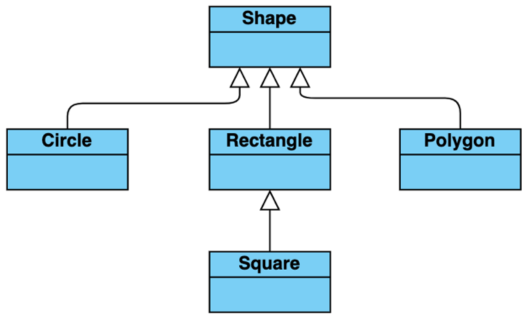
```
public class Person {
    private Person supervisor;
    ...
}
```

Additional examples are shown below. The exclusive ownership between the University class and the College class means, an instance of University has at least one and at most four instances of College. If the instance of University is destroyed, all instances of College associated with the instance of University will be destroyed together. On the other hand, the ownership between the Course and Student or Faculty is not exclusive. That is, an instance of Student or Faculty can exist in the system independent to a specific instance of Course.

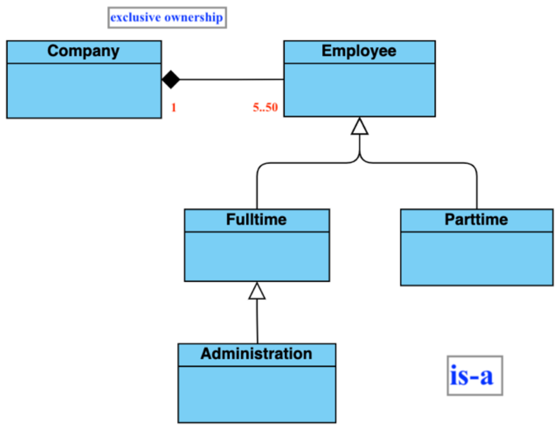


(4) Generalization

Object-oriented approach promotes reuse. Generalization relationship models the common "features" among classes, including attributes and operations. You can consider the common features as the "reusable part", which is modeled as a "superclass" or "parent class" of the "subclasses" or "child classes". In other words, a superclass is a generalization of subclasses, and a subclass is an extension of the superclass. We can also say the generalization models the "is-a-kind-of" relationship; for example, "rectangle" is a kind of "shape" as shown below. Generalization also promotes extendibility since you can always define new subclasses if additional or specific features are needed. "Generalization" is a term used at the design phase; however, it is implemented using the "inheritance" in object-oriented programming languages, such as the inheritance in Java. In a class diagram, a class that is a subclass of another class is connected to it by an arrow with a solid line for its shaft and with a triangular hollow arrowhead. The arrow points from the subclass to the superclass.



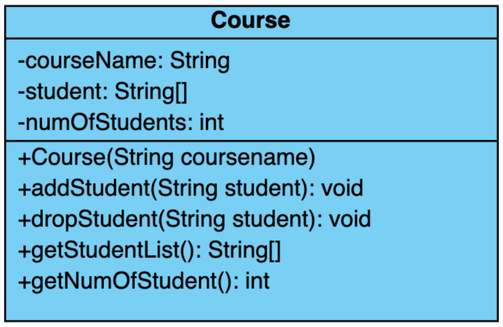
Additional examples are shown below. An instance of Fulltime or Parttime "is a kind of" Employee. An instance of Administration "is a kind of" Fulltime.



- **Example 1 – Designing the Course Class**

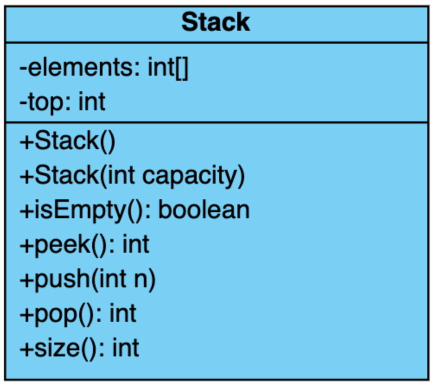
Suppose you need to keep track of the course enrollment. You start with the data abstraction and decide what attributes you need to meet the requirement. In this case, you decide a Course class is needed and must include the attributes that store the course identity associated with a list of students currently enrolled. Next, you should determine the operations needed to manage the attributes. Since there are 2 attributes, you should provide the operations that can be used to create a course and add/drop a student to/from the course. Below is a sample design for a Course class for this purpose. A Course object can be created using the constructor **Course(String name)** by passing a course name. You can add students to the course using the **addStudent(String student)** method, drop a student from the course using the

dropStudent(String student) method, and return all the students in the course using the **getStudentList()** method.


Course
-courseName: String
-student: String[]
-numOfStudents: int
+Course(String coursename)
+addStudent(String student): void
+dropStudent(String student): void
+getStudentList(): String[]
+getNumOfStudent(): int

- **Example 2 – Designing the Stack Class**

Recall that a **stack** is a data structure that holds data in a last-in, first-out fashion. Stacks have many applications. For example, the compiler uses a stack to process method invocations. When a method is invoked, its parameters and local variables are pushed into a stack. When a method calls another method, the new method's parameters and local variables are pushed into the stack. When a method finishes its work and returns to its caller, its associated space is released from the stack. For simplicity, assume the stack holds the **int** values. In this case, you need to keep track of the list of integers that are on the stack and keep track of the top position where the integer to be removed next. For the stack operations, add (push) and remove(pop) are essential. You can also define common operations on a list of integers, such as isEmpty(), size(), etc. A sample Stack class is defined below.


Stack
-elements: int[]
-top: int
+Stack()
+Stack(int capacity)
+isEmpty(): boolean
+peek(): int
+push(int n)
+pop(): int
+size(): int