Object Oriented Methodology

Jin Hyun Kim
Fall 2019

Contents

- Programming Problems

- Software Lifecycle **

- Object Oriented Programming**

  - More Concepts of Object-Oriented Programming (OOP)

A (Programming) Problem

- 월드전자는 노트북을 제조하고 판매하는 회사이다. 노트북 판매 대수에 상관없이 매년 임대료, 재산세, 보험료, 급여 등 A만원의 고정 비용이 들며, 한 대의 노트북을 생산하는 데에는 재료비와 인건비 등 총 B만원의 가변 비용이 든다고 한다.

- 예를 들어 A=1,000, B=70이라고 하자. 이 경우 노트북을 한 대 생산하는 데는 총 1,070만원이 들며, 열 대 생산하는 데는 총 1,070만원이 든다.

- 노트북 가격이 C만원으로 책정되었다고 한다. 일반적으로 생산 대수를 늘려 가다 보면 어느 순간 총 수입(판매비용)이 총 비용(=고정비용+가변비용)보다 많아지게 된다. 최초로 총 수입이 총 비용보다 많아져 이익이 발생하는 지점을 손익분기점 (BREAK-EVEN POINT)이라고 한다.

- A, B, C가 주어졌을 때, 손익분기점을 구하는 프로그램을 작성하시오.

Running Problem

- 은행 시뮬레이터

  - 제한된 은행원은 현금 입금 및 출금에 대한 고객 서비스를 제공한다.

  - 고객은 임의의 금액을 입/출금을 요구하면, 은행원은 그의 계좌에서 요구한 금액을 입/출금하고 잔고를 조정한다.

    - 고객이 요구한 입/출금액은 10원 단위로 랜덤하게 결정한다.

    - 입/출금의 요구도 랜덤하게 결정한다.

Questions

- What is process to build a program of the above running problem?

- How to obtain exact and common understanding of a given problem?

- How to specify (model) what to be understood?

- How to build a program based on a specification (model)?

SW Development

- Software lifecycle

  - Models for the development of software

  - Set of activities and their dependency relationships to each other to support the development of a software system

  - Examples:

    - Analysis, Design, Implementation, Testing

- Typical Lifecycle questions:

  - Which activities should I select when I develop software?

  - What are the dependencies between activities?

  - How should I schedule the activities?
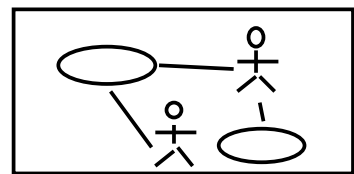
SW Life Cycle

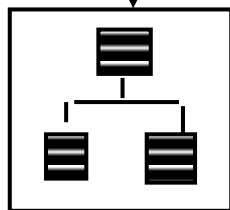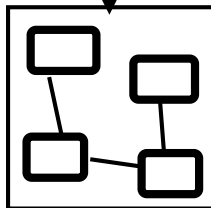| Requirement Elicitation	Analysis	System Design	Detailed Design	Implementation	Testing

Expressed in terms of	Structured by	Realized By	Implemented by	Verified by

Use-Case Model	Application-Domain Model	Sub-system	Solution-Domain Model	Source Code	Test Case
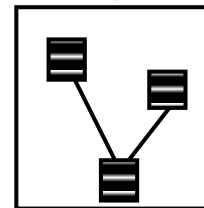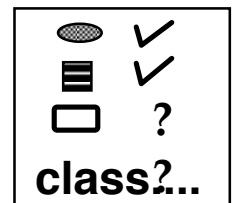
Requirement Identification

- **Two questions need to be answered**:

  - How can we identify the purpose of a system?

  - What is inside, what is outside the system?

- These two questions are answered during requirements elicitation and analysis

  - **Requirements elicitation**: Definition of the system in terms understood by the customer ("Requirements specification")

  - **Analysis**: Definition of the system in terms understood by the developer (Technical specification, "Analysis model")

  - **Requirements Process**: Contains the activities Requirements Elicitation and Analysis

Requirement Elicitation

- Bridging the gap between end user and developer:

  - Questionnaires: Asking the end user a list of pre-selected questions

  - Task Analysis: Observing end users in their operational environment

  - Scenarios: Describe the use of the system as a series of interactions between a concrete end user and the system

  - Use cases: Abstractions that describe a class of scenarios.

Scenario Example: Warehouse on Fire

- Bob, driving down main street in his patrol car notices smoke coming out of a warehouse. His partner, Alice, reports the emergency from her car.

- Alice enters the address of the building into her wearable computer , a brief description of its location (i.e., north west corner), and an emergency level.

- She confirms her input and waits for an acknowledgment.

- John, the dispatcher, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He allocates a fire unit and sends the estimated arrival time (ETA) to Alice.

- Alice received the acknowledgment and the ETA.

Observations about Warehouse on Fire Scenario

- Concrete scenario

  - Describes a single instance of reporting a fire incident.

  - Does not describe all possible situations in which a fire can be reported.

- Participating actors

  - Bob, Alice and John

After the scenarios are formulated

- Find all the use cases in the scenario that specify all instances of how to report a fire

  - Example: "Report Emergency" in the first paragraph of the scenario is a candidate for a use case

- Describe each of these use cases in more detail

  - Participating actors

- Describe the entry condition

- Describe the flow of events

- Describe the exit condition

- Describe exceptions

- Describe nonfunctional requirements

- Functional Modeling (see next lecture)

Requirement Analysis

- How can we identify the purpose of a system?

- What is inside, what is outside the system?

Object-Oriented Programming

Object Oriented Programming

- 기존 C와 같은 구조적 프로그래밍 언어는 동작되는 자료와 처리 동작 자체를 서로 별도로 구분

  - 처리동작과 자료 사이의 관계가 서로 밀접한 연관성을 갖지 못함

  - 프로그램이 커지거나 복잡해지면 프로그램이 혼란스럽게 되어 에러를 찾는 디버깅 및 프로그램의 유지보수가 어려워 짐

```
                    자동차,책,사과,물건,배
         ┌──────┬──────┬──────┬──────┐
      소유하다    타다     먹다     읽다    맛있다
```
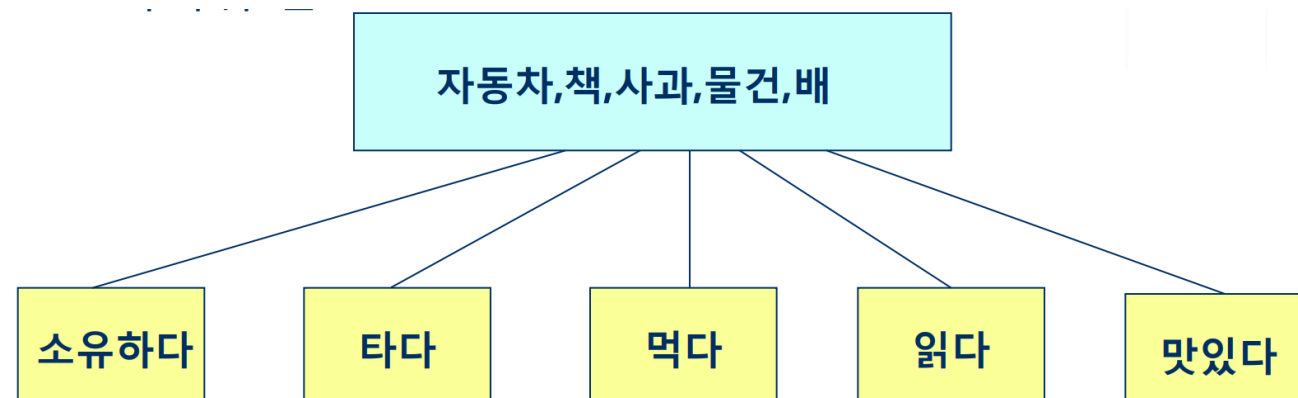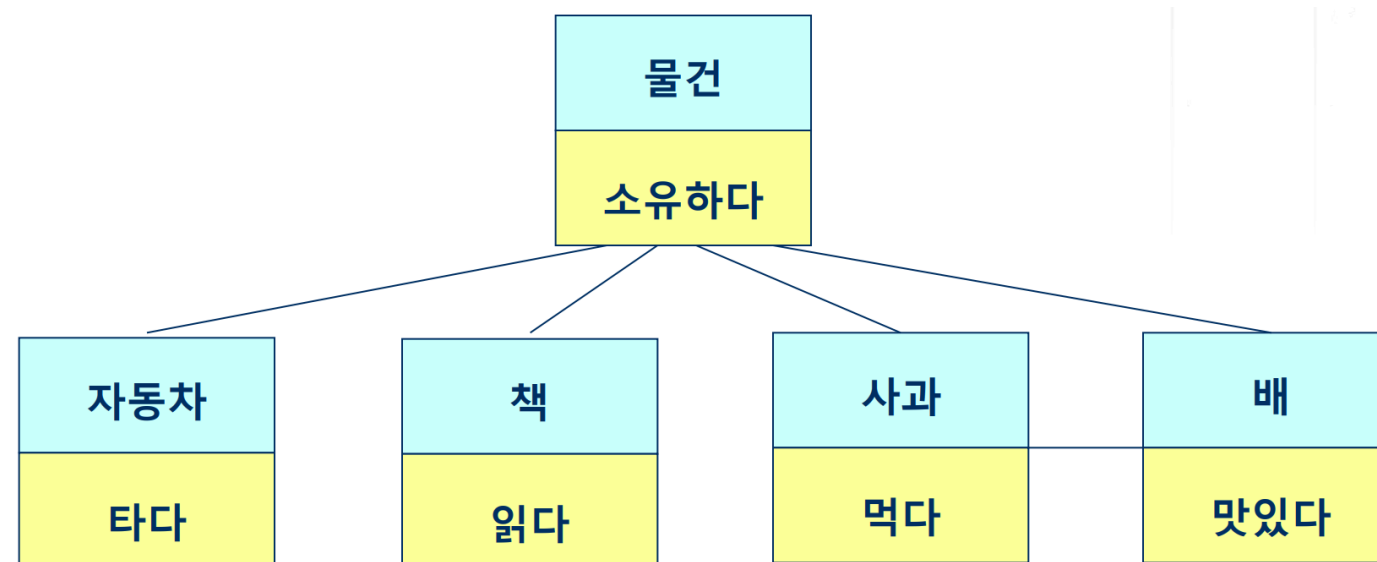
Object (객체)

- 객체지향 프로그래밍에서는 자료와 이의 처리동작을 하나로 묶어서 다룰수 있는 객체(object)라는 개념을 도입

- 프로그램은 처리하는 절차보다도 동작되는 자료에 중점을 둔 객체, 객체 갂의 상호관계로 표현

물건	자동차	사과	책	배
소유하다	타다	먹다	읽다	맛있다

Object (객체)

- 물건의 객체는 다른 객체 보다 상위관계를 갖는 객체이고 이 객체에서 갖는 성질은 다른 객체도 공유

- 비슷한 성질의 사과, 배는 서로 성질을 공유하는 관계

```
                    ┌──────────┐
                    │   물건    │
                    ├──────────┤
                    │  소유하다  │
                    └──────────┘
        ┌──────────┬──────┴──────┬──────────┐
  ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐
  │  자동차   │ │    책     │ │   사과    │ │    배     │
  ├──────────┤ ├──────────┤ ├──────────┤ ├──────────┤
  │   타다    │ │   읽다    │ │   먹다    │ │  맛있다   │
  └──────────┘ └──────────┘ └──────────┘ └──────────┘
```

Object Oriented Programming

- 객체지향 프로그래밍 시각

  - 문제의 영역을 단순한 자료의 처리 흐름으로 보지 않음

    - 구조적 프로그래밍 서로 관련된 자료와 연산(함수)들이 서로 독립적으로 정의되어 취급

    - 문제 영역 내에 존재하는 여러 연관된 객체들을 정의하고 이들 객체들이 서로 정보를 주고 받는다고 보는 시각 (객체 갂의 관계)

- 객체지향 프로그래밍에서는 프로그램은 여러 개의 객체로 구성

  - 객체(object)는 자료(특성(attribute))와 이를 대상으로 처리하는 동작인 연산(함수,메쏘드(method))을 하나로 묶어 만든 요소로 프로그램을 구성하는 실체

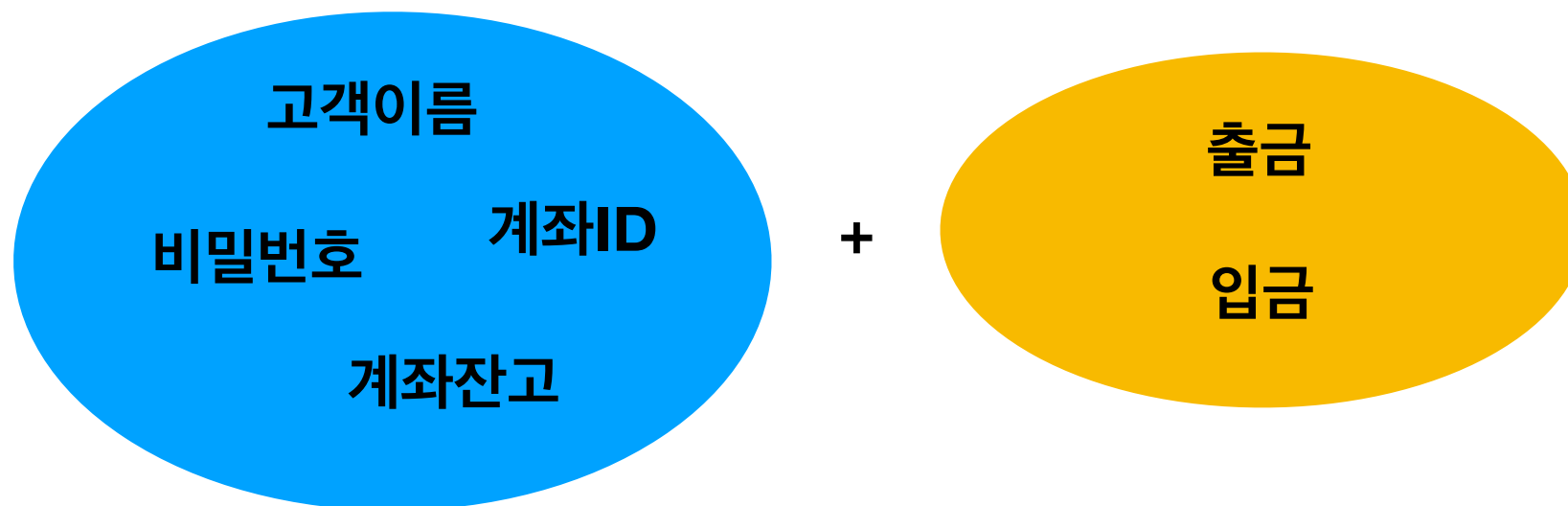 - 객체란 단순히 자료를 표현하는 변수 만을 가지는 것이 아니라 그 객체가 무엇을 할 수 있는가를 정의한 함수(메쏘드)로 구성
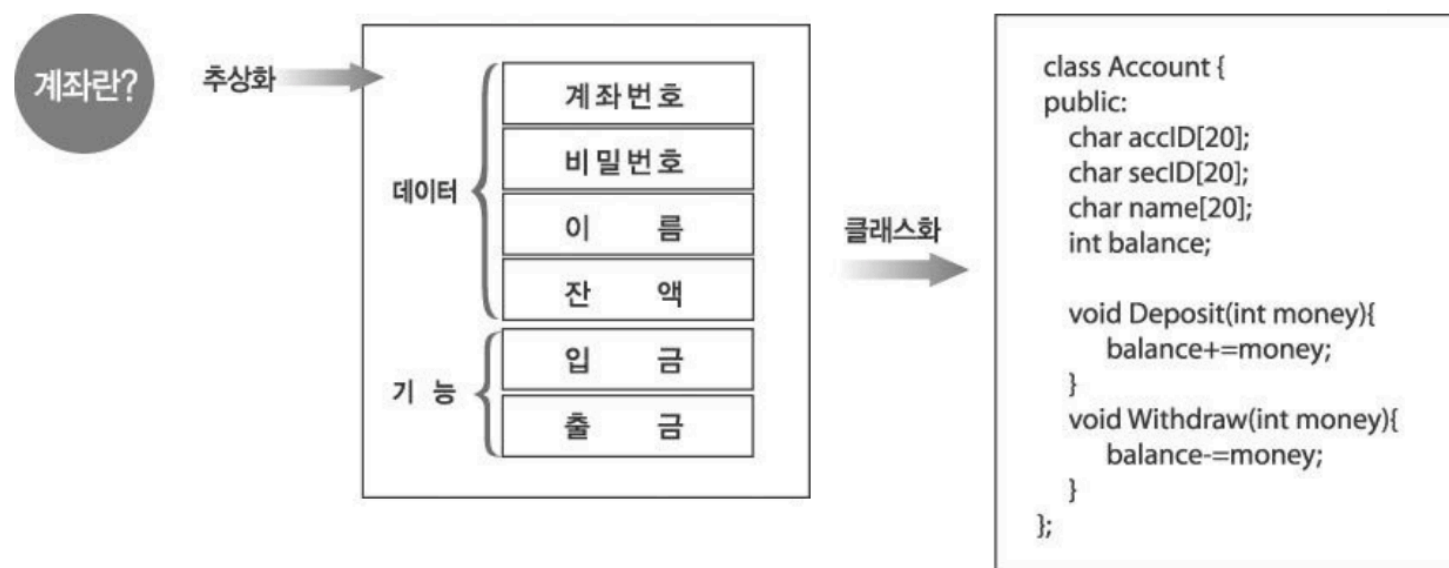
비교

- 구조적 프로그래밍

  자료 + 연산 = 프로그램

- 객체지향 프로그래밍

  자료 + 연산 = 자료 연산
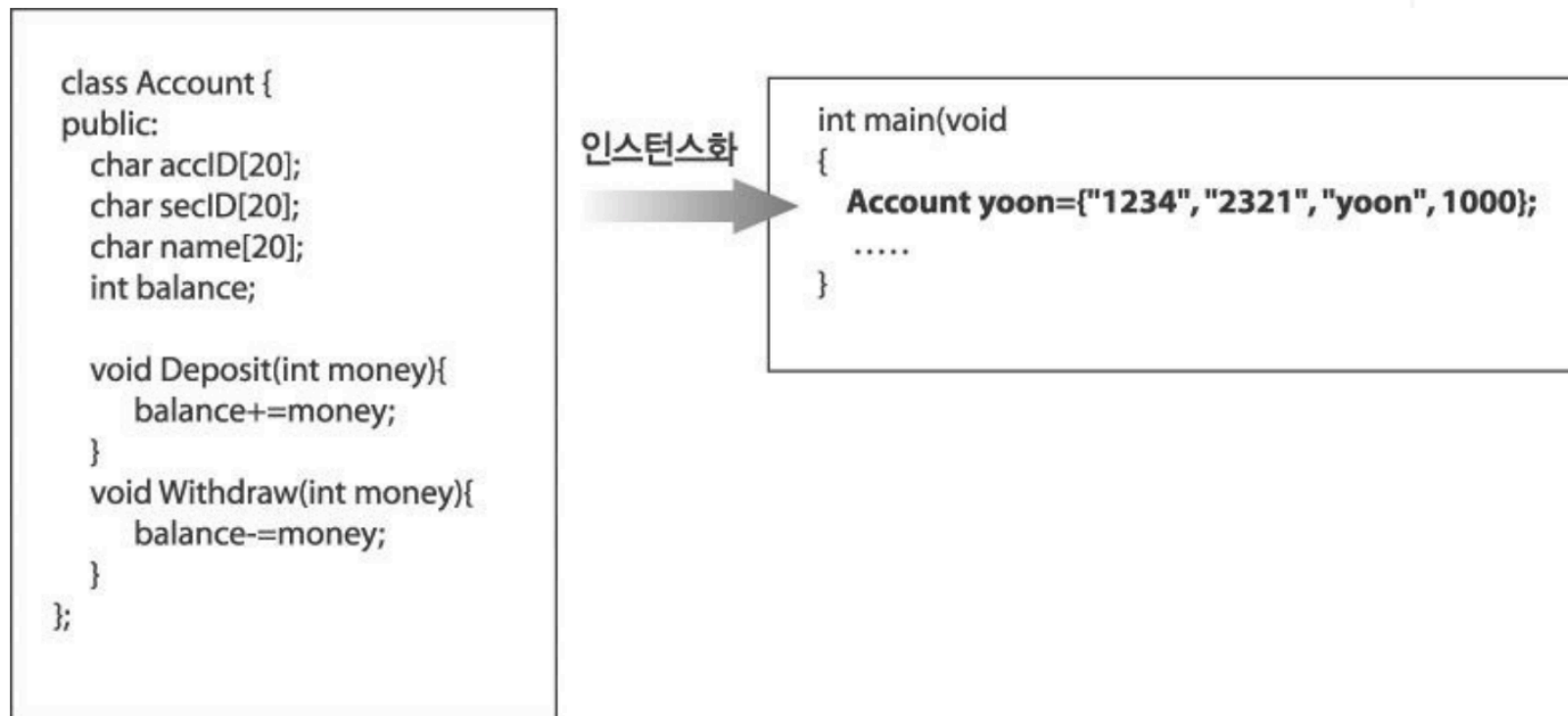
  객체 + … + 객체 = 프로그램

고객이름
비밀번호   계좌ID
계좌잔고

+

출금
입금

데이터 추상화와 클래스

- 사물의 관찰 이후의 데이터 추상화

  - 현실 세계의 사물을 데이터적인 측면과 기능적인 측면을 통해서 정의하는 것

- 데이터 추상화 이후의 클래스화

  - 추상화된 데이터를 가지고 사용자 정의 자료형을 정의하는 것

계좌란? → 추상화 →

데이터 {
  계좌번호
  비밀번호
  이   름
  잔   액
}

기능 {
  입   금
  출   금
}

→ 클래스화 →

```
class Account {
public:
    char accID[20];
    char secID[20];
    char name[20];
    int balance;

    void Deposit(int money){
        balance+=money;
    }
    void Withdraw(int money){
        balance-=money;
    }
};
```

객체 생성

- 선언된 클래스를 사용하여 객체를 생성

  - 클래스화 이후의 인스턴스화 (instantiation)

```
class Account {
public:
    char accID[20];
    char secID[20];
    char name[20];
    int balance;

    void Deposit(int money){
        balance+=money;
    }
    void Withdraw(int money){
        balance-=money;
    }
};
```

인스턴스화 →

```
int main(void
{
    Account yoon={"1234", "2321", "yoon", 1000};
    .....
}
```

More Concepts of OOP

- Abstraction

- Encapsulation

- Inheritance

- Polymorphism

- Class

- Object

- Methods

- Message Passing

- Association

- Aggregation

- Composition

Abstraction

- Data Abstraction is the property by virtue of which only the essential details are displayed to the user.

  - Ex: A car is viewed as a car rather than its individual components.

  - Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details.

  - The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects

Abstraction

- The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

Encapsulation

- Encapsulation is defined as the wrapping up of data under a single unit.

  - The mechanism that binds together code and the data it manipulates.

  - A protective shield that prevents the data from being accessed by the code outside this shield.

Encapsulation

Variables
Data
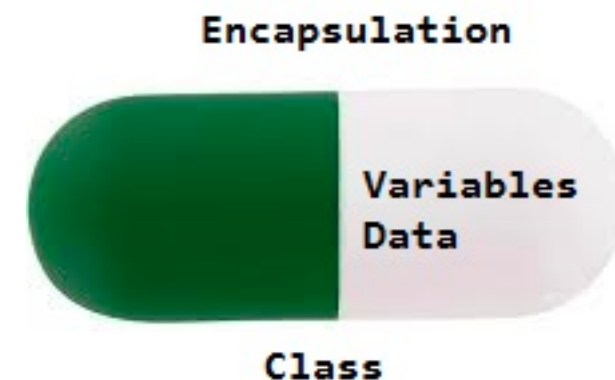
Class

Inheritance

- Inheritance is an important pillar of OOP(Object Oriented Programming).

  - One class is allow to inherit the features(fields and methods) of another class

  - Super Class: The class whose features are inherited is known as superclass(or a base class or a parent class).

  - Sub Class: The class that inherits the other class is known as subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

  - Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Inheritance

- The keyword used for inheritance is extends.

```
class derived-class extends base-class
{
    //methods and fields

}
```

Polymorphism

- Polymorphism refers to the ability of OOPs programming languages to differentiate between entities with the same name efficiently.

```java
1  // Java program to demonstrate Polymorphism
2
3  // This class will contain
4  // 3 methods with same name,
5  // yet the program will
6  // compile & run successfully
7  public class Sum {
8
9      // Overloaded sum().
10     // This sum takes two int parameters
11     public int sum(int x, int y)
12     {
13         return (x + y);
14     }
15
16     // Overloaded sum().
17     // This sum takes three int parameters
18     public int sum(int x, int y, int z)
19     {
20         return (x + y + z);
21     }
22
23     // Overloaded sum().
24     // This sum takes two double parameters
25     public double sum(double x, double y)
26     {
27         return (x + y);
28     }
29
30     // Driver code
31     public static void main(String args[])
32     {
33         Sum s = new Sum();
34         System.out.println(s.sum(10, 20));
35         System.out.println(s.sum(10, 20, 30));
36         System.out.println(s.sum(10.5, 20.5));
37     }
38 }
39
```

Class

- A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. I

Class

- Modifiers: A class can be public or has default access (Refer this for details).

- Class name: The name should begin with an initial letter (capitalized by convention).

- Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.

- Interfaces(if any): A class can implement more than one interface.

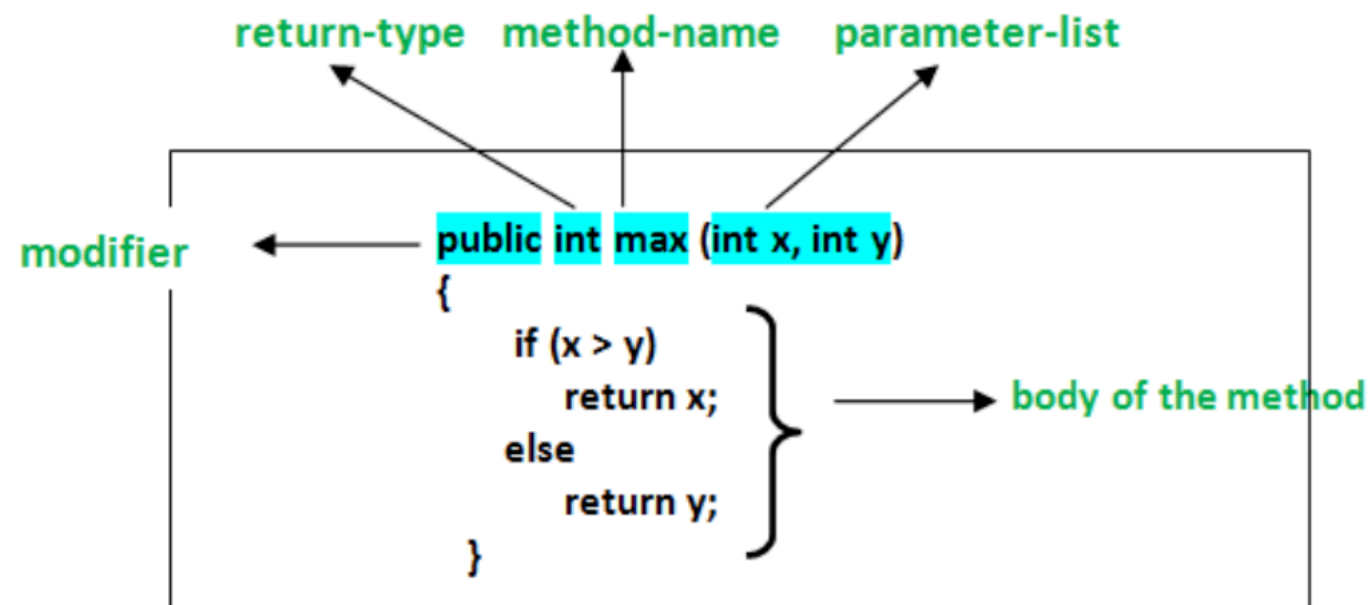- Body: The class body surrounded by braces.

Object

- An instance of a class consists of

  - State : It is represented by attributes of an object. It also reflects the properties of an object.

  - Behavior : It is represented by methods of an object. It also reflects the response of an object with other objects.

  - Identity : It gives a unique name to an object and enables one object to interact with other objects.

Identity	State/Attributes	Behaviors
Name of dog	Breed Age Color	Bark Sleep Eat
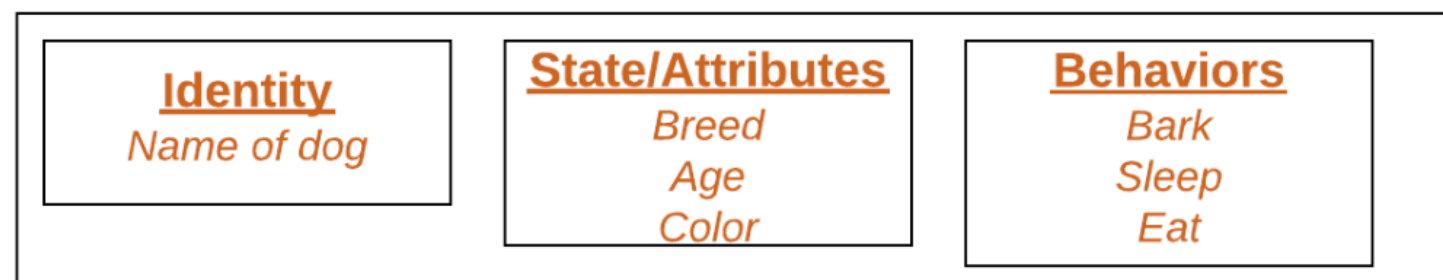
Methods

- A collection of statements that perform some specific task and return result to the caller.

- Can perform some specific task without returning anything.

- Allow us to reuse the code without retyping the code.

return-type method-name parameter-list

modifier ← public int max (int x, int y)
{
  if (x > y)
      return x;
  else
      return y;
}
→ body of the method

**Exception list**: The exceptions you expect by the method can throw, you can specify these exception(s).

Message Passing

- Objects communicate with one another by sending and receiving information to each other.

- A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results.

- Message passing involves specifying the name of the object, the name of the function and the information to be sent.
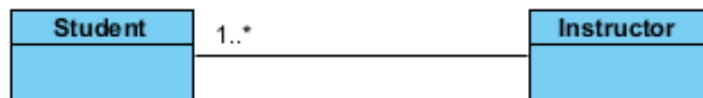
Association

- Aggregation is a special form of association, simply meaning the act of establishing a relationship between two unrelated classes.

A single student can associate with multiple teachers:

Student		Instructor
	1..*	
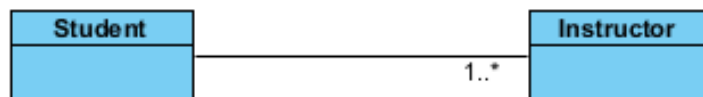
The example indicates that every Instructor has one or more Students:

Student		Instructor
	1..*	

We can also indicate the behavior of an object in an association (i.e., the role of an object) using role names.

Student	1..*	learns from	Instructor
	teaches	1..*	

Association

- It occurs when there's a one-way (HAS-A) relationship between the two classes you associate through their objects.

- One-directional association.

- Represents a HAS-A relationship between two classes.

- Only one class is dependent on the other.

Association

- For example, every Passenger has a Car but a Car doesn't necessarily have a Passenger.

  - When you declare the Passenger class, you can create a field of the Car type that shows which car the passenger belongs to.

  - Then, when you instantiate a new Passenger object, you can access the data stored in the related Car as well.

Compositions

- A stricter form of aggregation.

- It occurs when the two classes you associate are mutually dependent on each other and can't exist without each other.

- This kind of relationship between objects is also called a PART-OF relationship.

  - For example, take a Car and an Engine class.
    A Car cannot run without an Engine, while an Engine also can't function without being built into a Car.
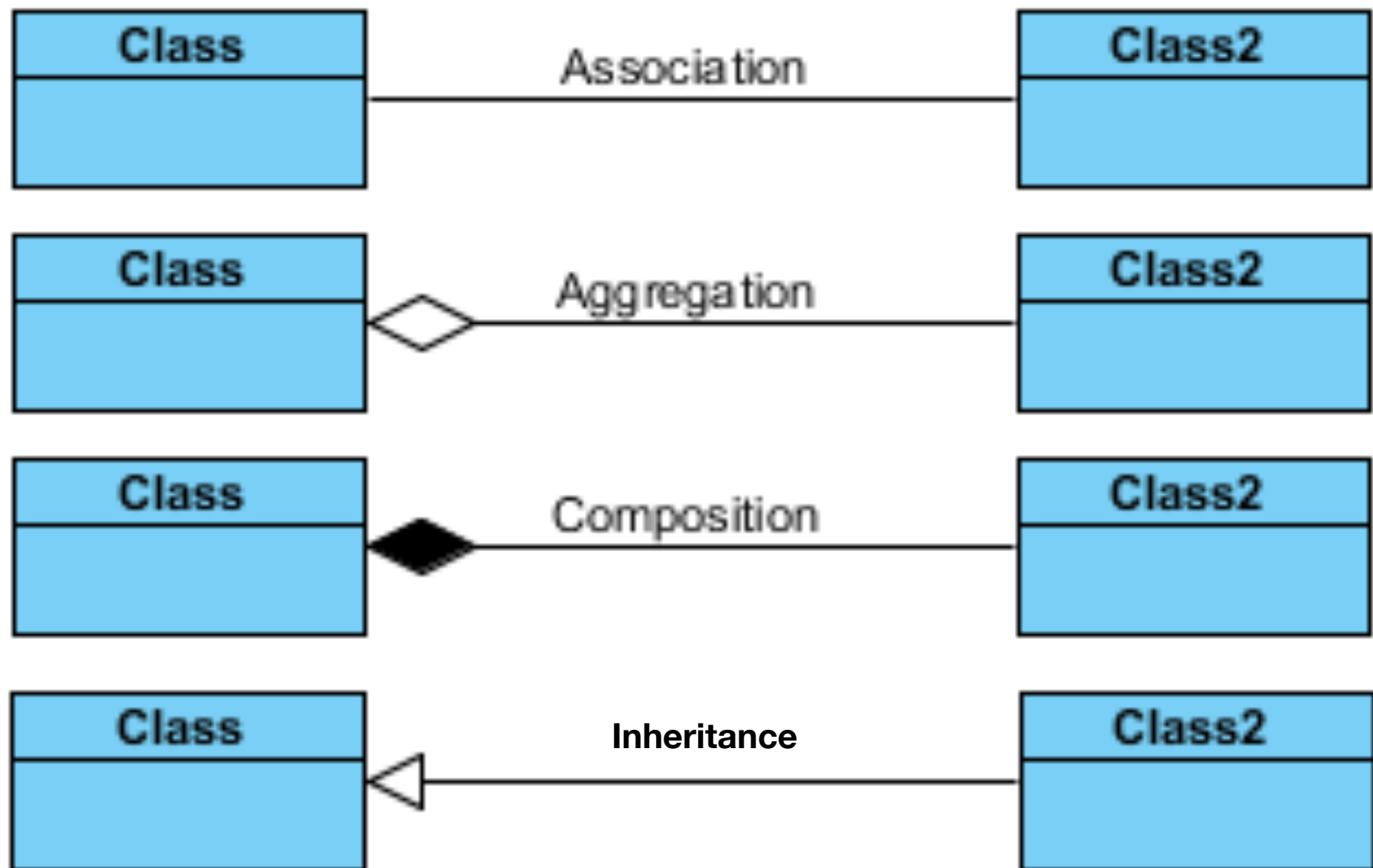
Association vs Aggregation vs Composition
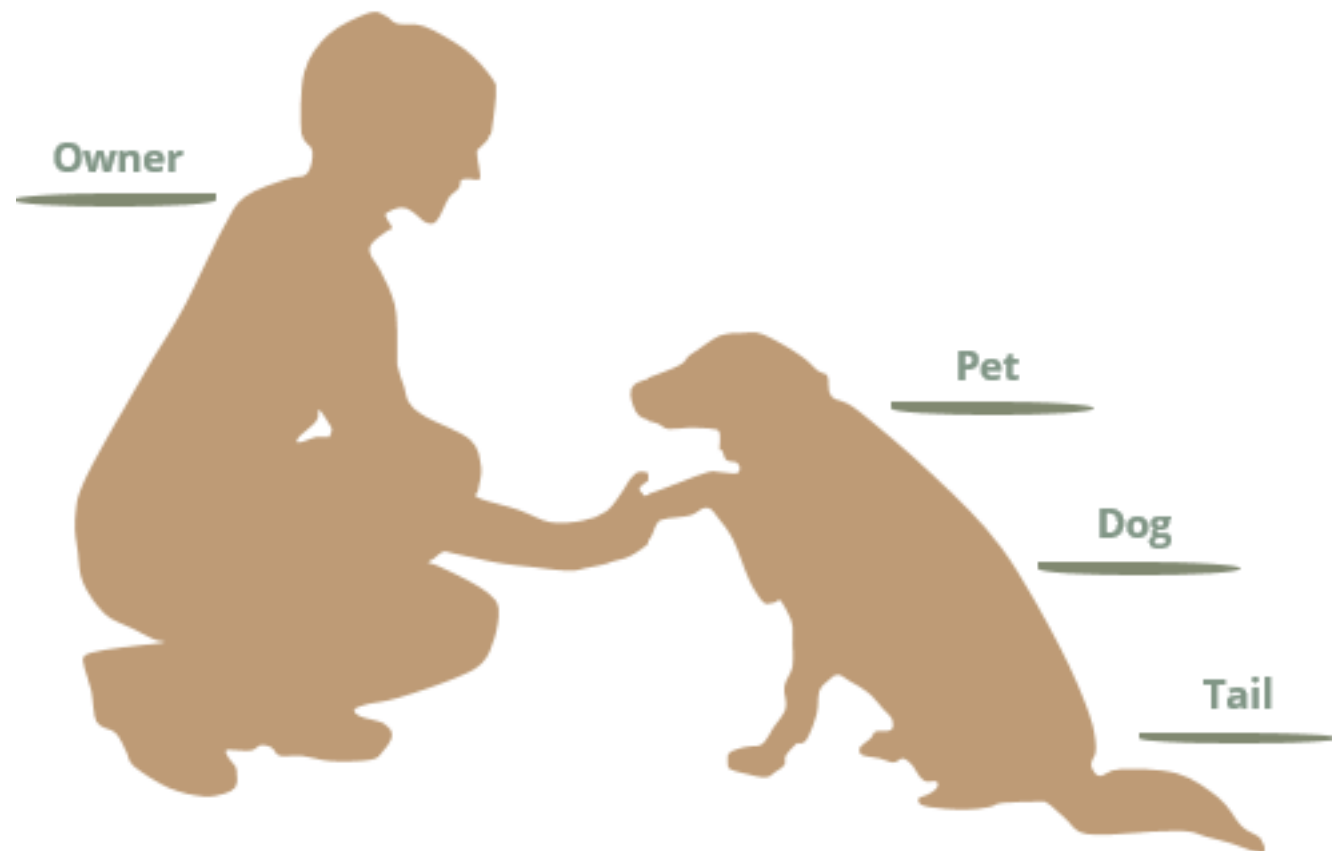
- Owners feed pets, pets please owners (association)

- a tail is a part of both dogs and cats (aggregation / composition)

- a cat is a kind of pet (inheritance / generalization)

# Association vs Aggregation vs Composition

Association	Aggregation	Composition
Class A **uses** Class B.	Class A is **owns** Class B.	Class A **contains** Class B.
Example: • Employee uses BusService for transportation. • Client-Server model. • Computer uses keyboard as input device.	**Example:** • Manager has N Employees for a project. • Team has Players.	**Example:** • Order consists of LineItems. • Body consists of Arm, Head, Legs. • BankAccount consists of Balance and TransactionHistory.
An association is **used when** one object wants another object to perform a service for it. Eg. Computer uses keyboard as input device.	An aggregation is **used when** life of object is independent of container object But still container object owns the aggregated object. Eg. Team has players, If team dissolve, Player will still exists.	A composition is **used where** each part may belong to only one whole at a time. Eg. A line item is part of an order so A line item cannot exist without an order.

UML Notation

Composition vs Aggregation

- **Aggregation** implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child).

  - Delete the Class and the Students still exist.

- **Composition** implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.
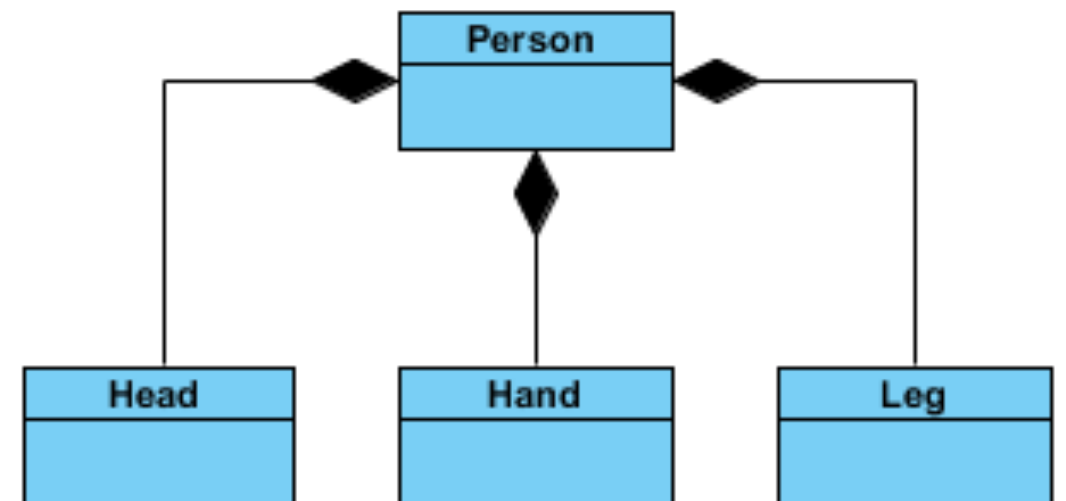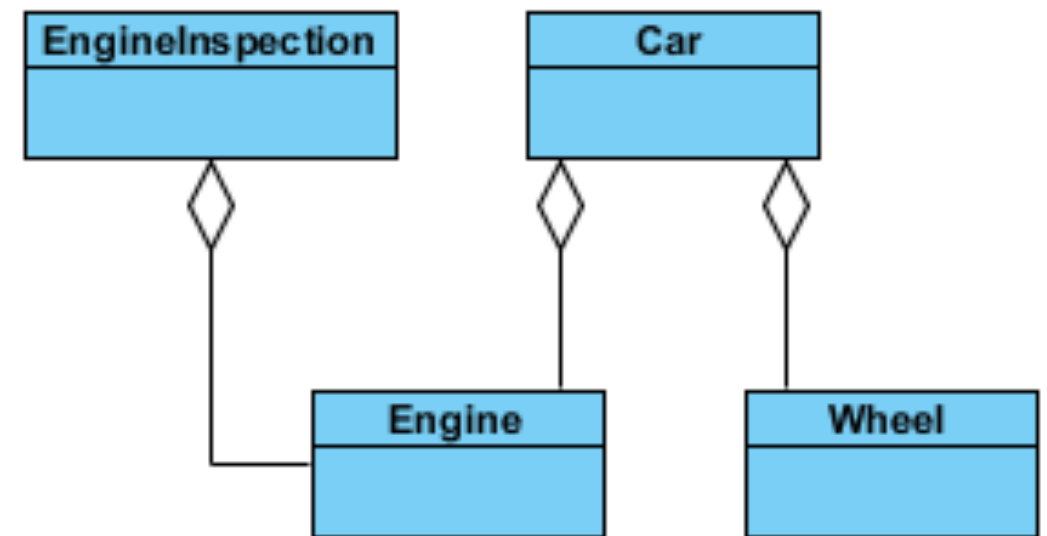
Next Class

- Object-Oriented Design using Use-case Diagram, Sequence Diagram, and Class Diagram