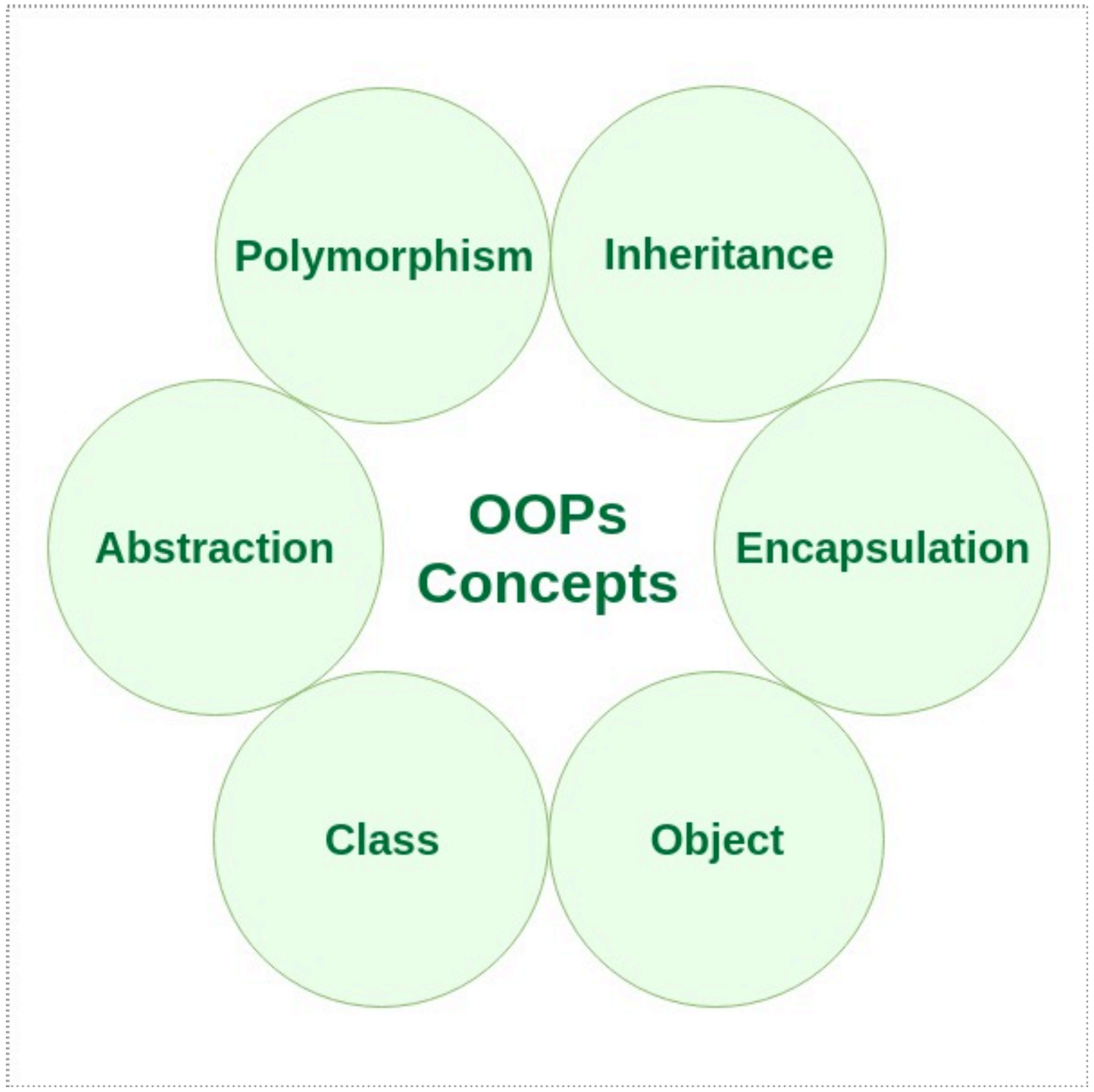


Object Oriented Methodology

Jin Hyun Kim
Fall 2019

Contents

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

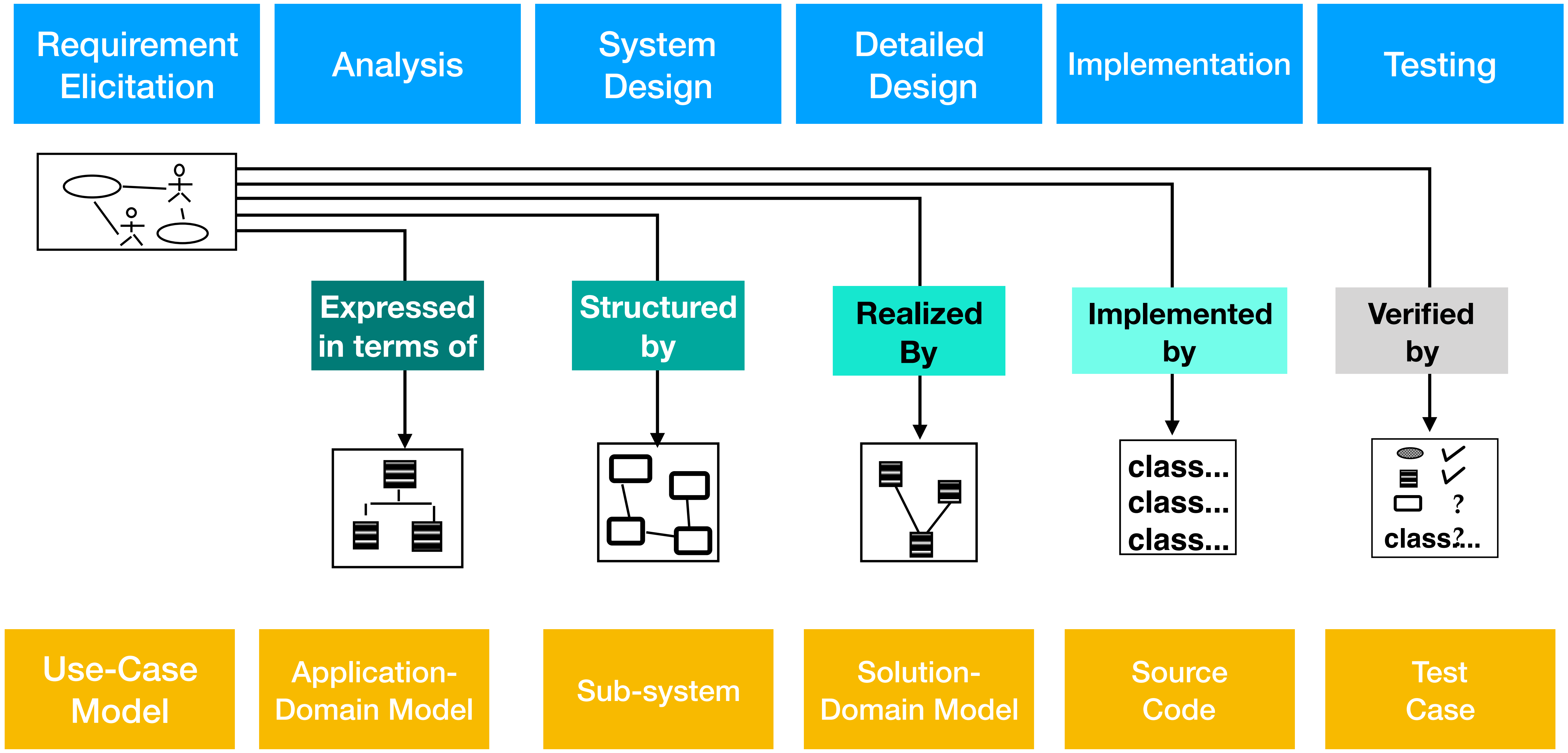
- What is process to build a program of the above running problem?
- How to obtain exact and common understanding of a given problem?
- How to specify (model) what to be understood?
- How to build a program based on a specification (model)?

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 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?

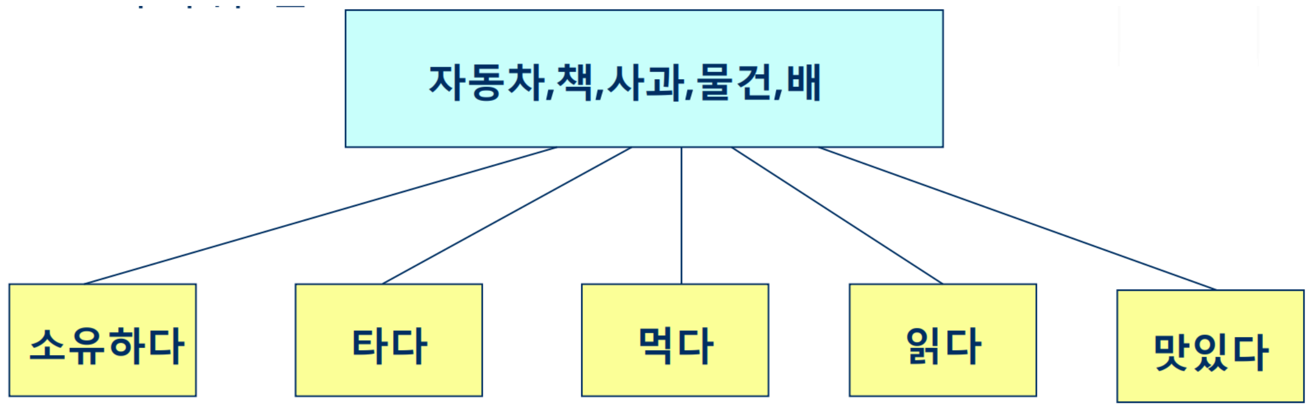
Object-Oriented Programming

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



Object (객체)

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

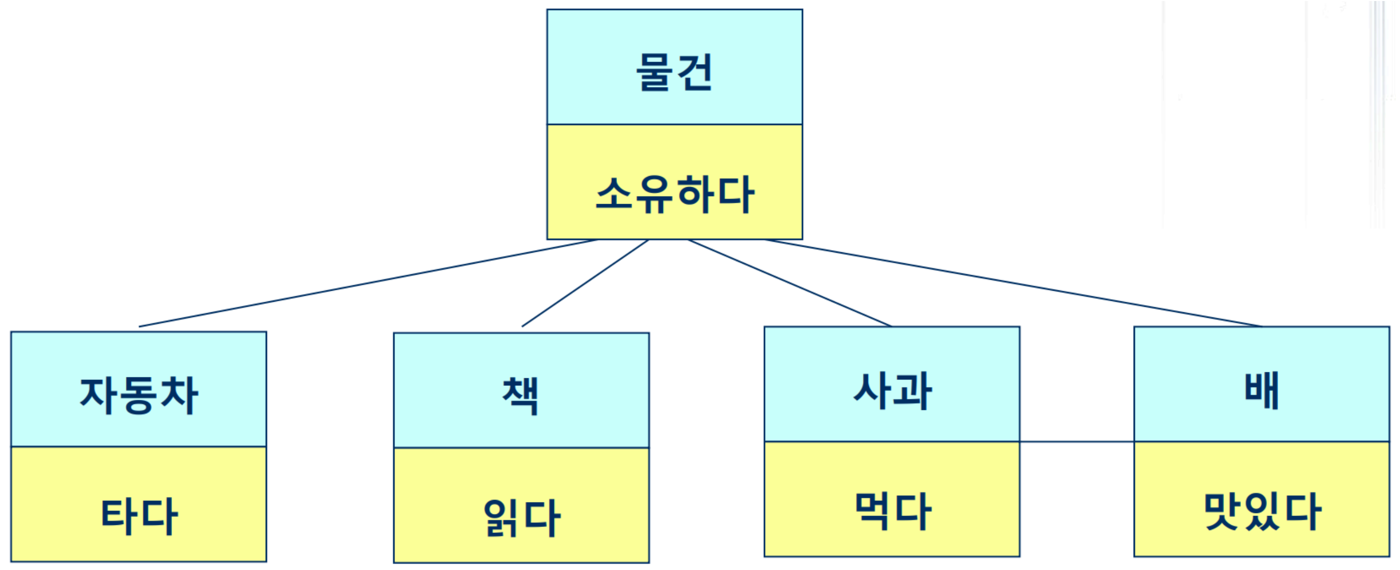
- 프로그램은 처리하는 절차보다도 동작되는 자료에 중점을 둔 객체, 객체 각의 상호관계로 표현

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



Object Oriented Programming

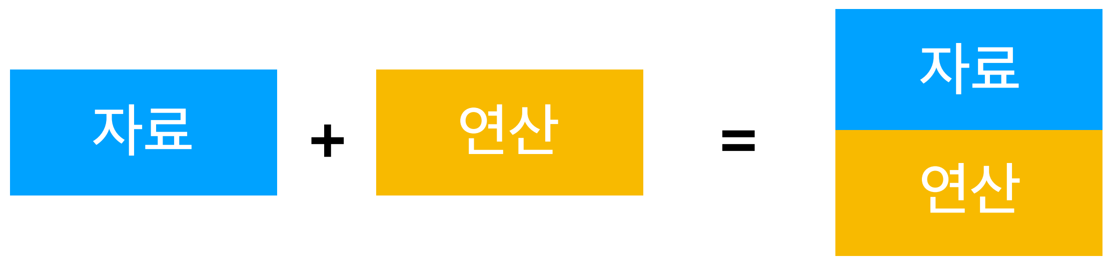
- 객체지향 프로그래밍 시각
  - 문제의 영역을 단순한 자료의 처리 흐름으로 보지 않음
    - 구조적 프로그래밍 서로 관련된 자료와 연산(함수)들이 서로 독립적으로 정의되어 취급
    - 문제 영역 내에 존재하는 여러 연관된 객체들을 정의하고 이들 객체들이 서로 정보를 주고 받는다고 보는 시각 (객체 갂의 관계)
- 객체지향 프로그래밍에서는 프로그램은 여러 개의 객체로 구성
  - 객체(object)는 자료(특성(attribute))와 이를 대상으로 처리하는 동작인 연산(함수,메쏘드(method))을 하나로 묶어 만든 요소로 프로그램을 구성하는 실체
  - 객체란 단순히 자료를 표현하는 변수 만을 가지는 것이 아니라 그 객체가 무엇을 할 수 있는가를 정의한 함수(메쏘드)로 구성

비교

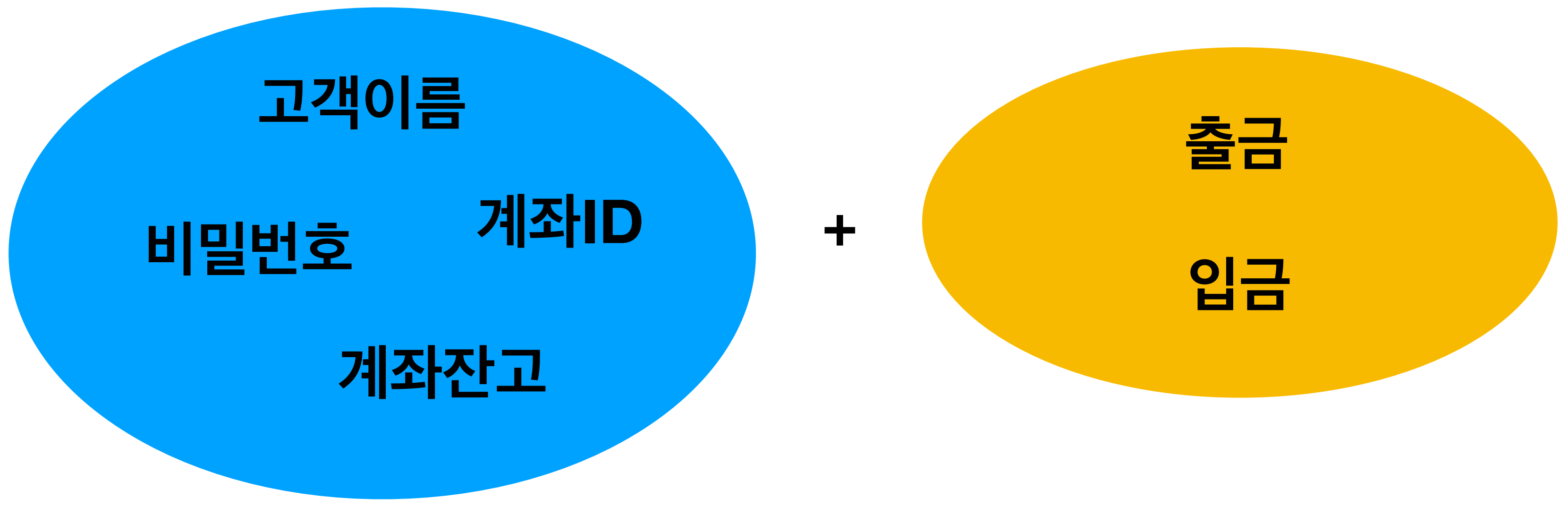
- 구조적 프로그래밍

자료 + 연산 = 프로그램

- 객체지향 프로그래밍

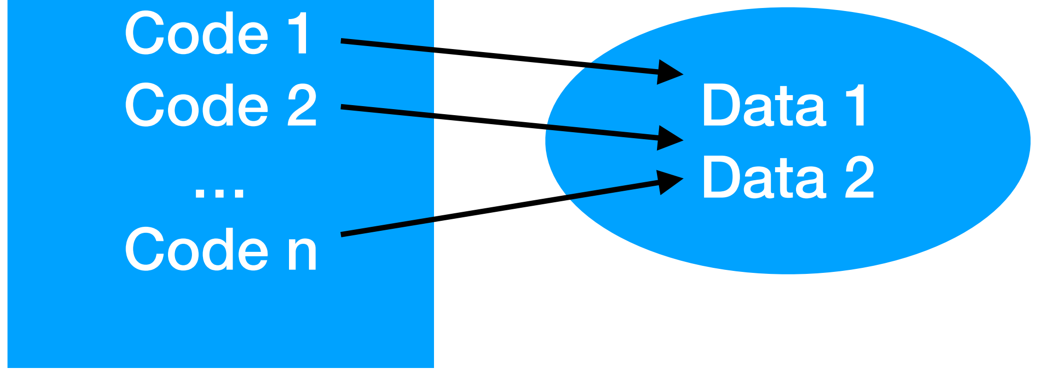


객체 + ... + 객체 = 프로그램

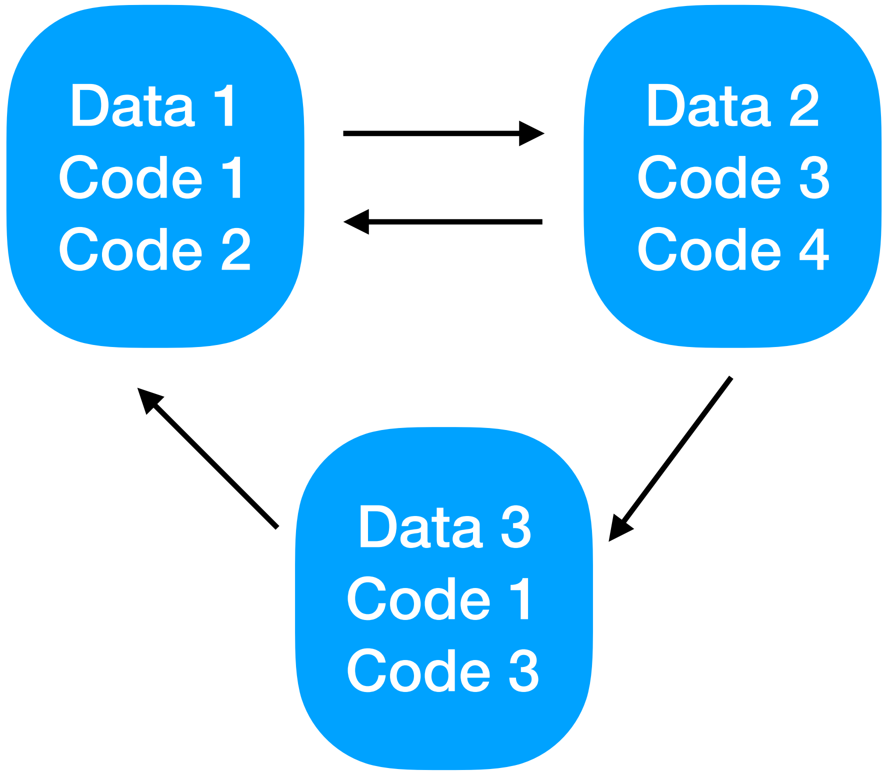


비교

Procedural Applications

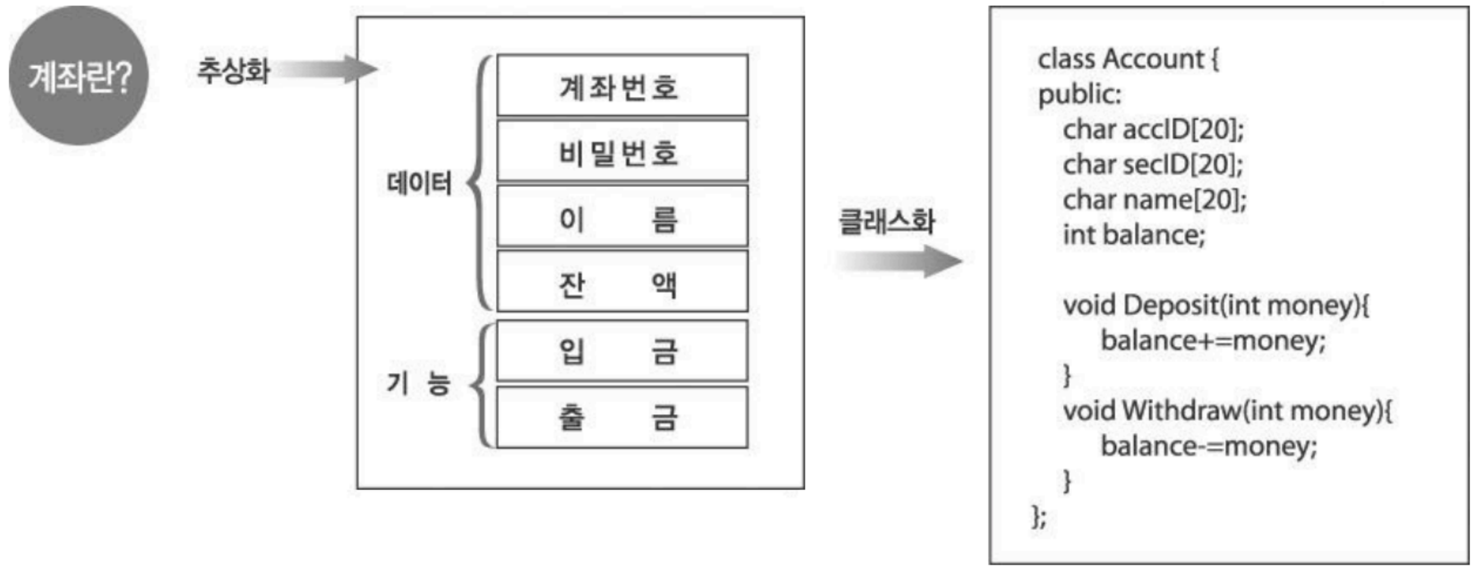


OOP Applications



데이터 추상화와 클래스

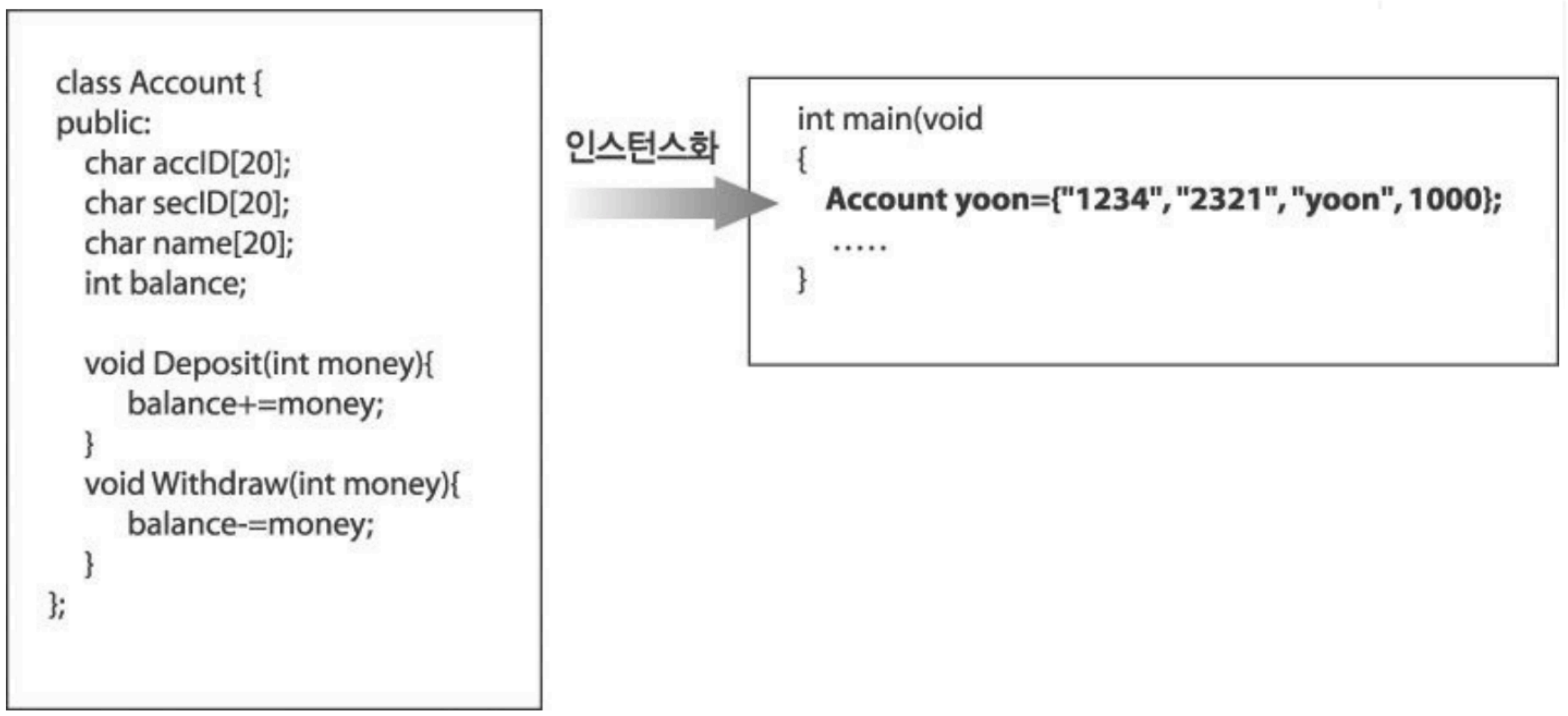
- 사물의 관찰 이후의 데이터 추상화
  - 현실 세계의 사물을 데이터적인 측면과 기능적인 측면을 통해서 정의하는 것
- 데이터 추상화 이후의 클래스화
  - 추상화된 데이터를 가지고 사용자 정의 자료형을 정의하는 것



```
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;
   }
};
```

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

More Concepts of OOP

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

- Class
- Object
- Methods
- Interfaces

- Association
- Aggregation
- Composition

Abstraction

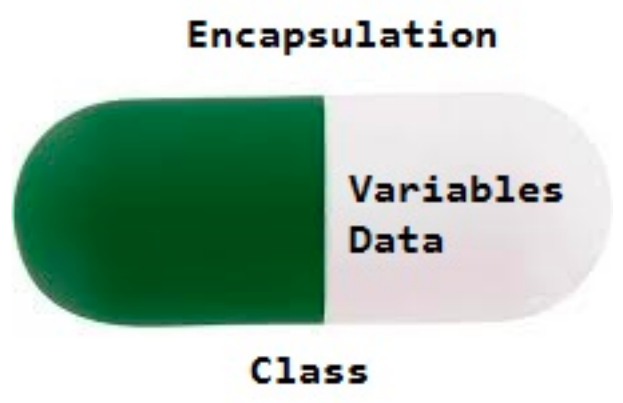
- Data Abstraction is the property by virtue of which only the essential details are displayed to the user.
  - Ex: A car is viewed as a car rather than its individual components.
  - Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring 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.



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.

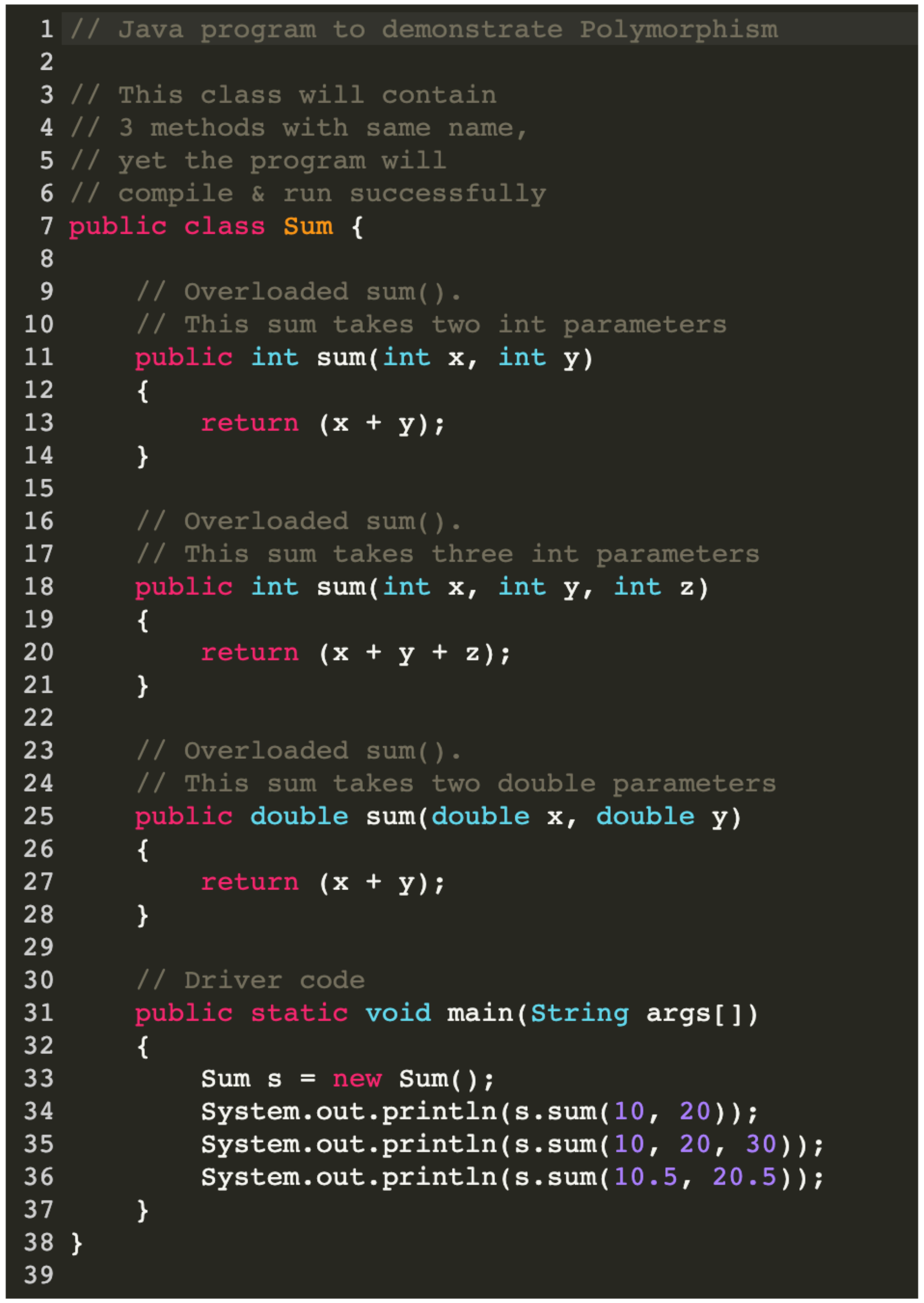
Inheritance

- 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
// Java program to demonstrate Polymorphism

// This class will contain
// 3 methods with same name,
// yet the program will
// compile & run successfully
public class Sum {

    // Overloaded sum().
    // This sum takes two int parameters
    public int sum(int x, int y)
    {
        return (x + y);
    }

    // Overloaded sum().
    // This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }

    // Overloaded sum().
    // This sum takes two double parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }

    // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Class

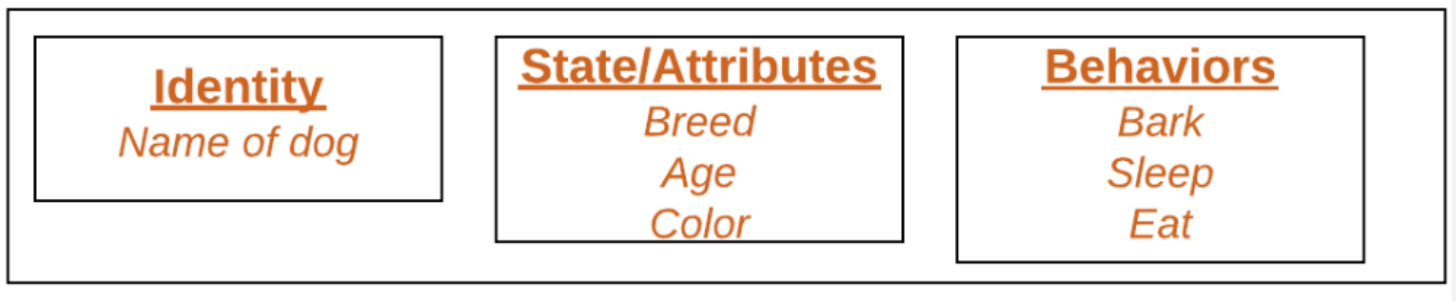
- A class is a user defined blueprint or prototype from which objects are created.
- It represents the set of properties or methods that are common to all objects of one type. I

Class

- Modifiers: (Accessibility) 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.

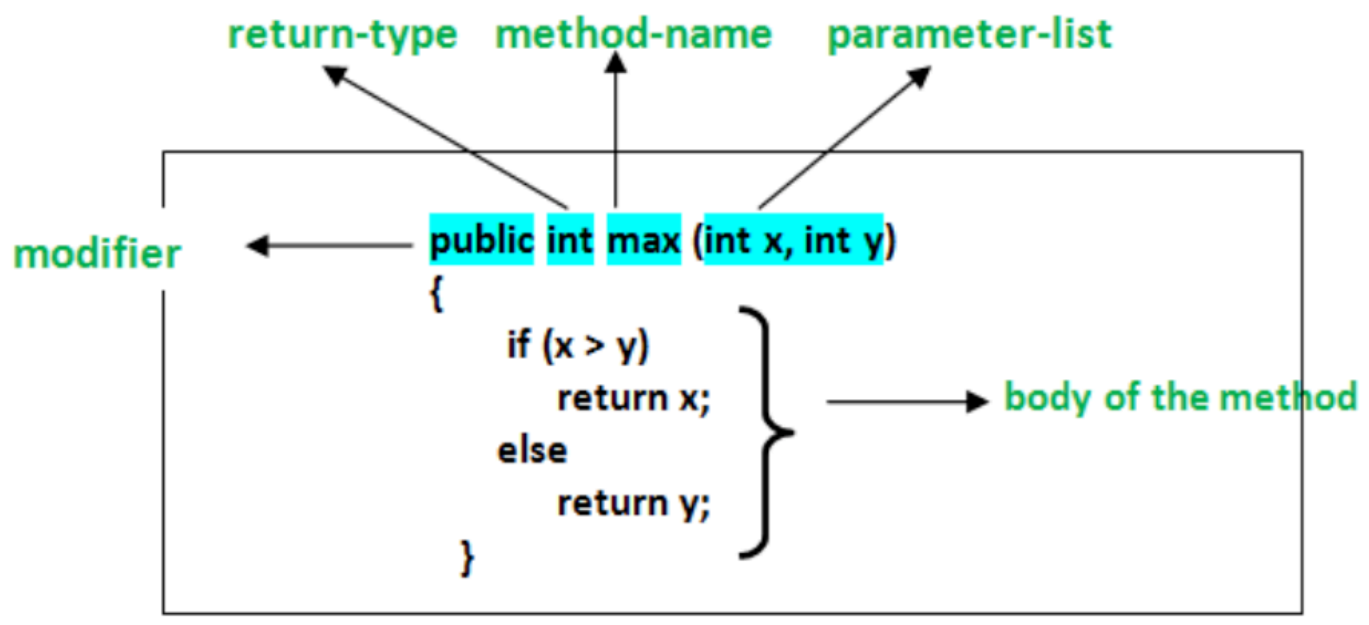
Object

- 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.



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.



```
public int max (int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

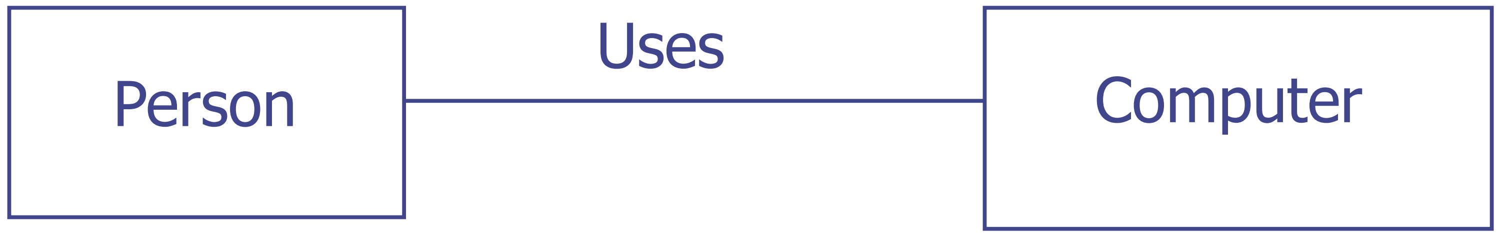
Exception list: The exceptions you expect by the method can throw, you can specify these exception(s).

Interface

- An interface is a description of the actions that an object can do
- For example when you flip a light switch, the light goes on, you don't care how, just that it does.
- In Object Oriented Programming, an Interface is a description of all functions that an object must have in order to be an "X".
- Again, as an example, anything that "ACTS LIKE" a light, should have a turn_on() method and a turn_off() method.
- The purpose of interfaces is to allow the computer to enforce these properties and to know that an object of TYPE T (whatever the interface is) must have functions called X,Y,Z, etc.

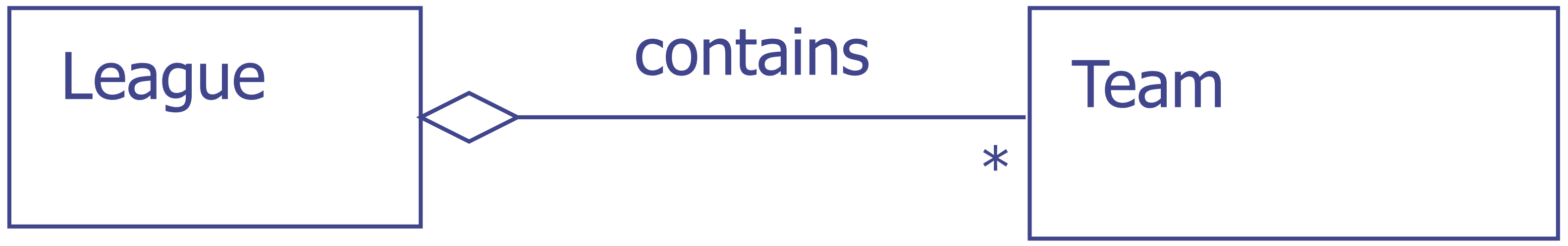
Association

- An association is a connection between classes, a semantic connection between objects of classes involved in the association.
- Association typically represents “has a” or “uses” relationships.
- Could be as simple as one object is calling a method of another object.



Aggregation

- 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.

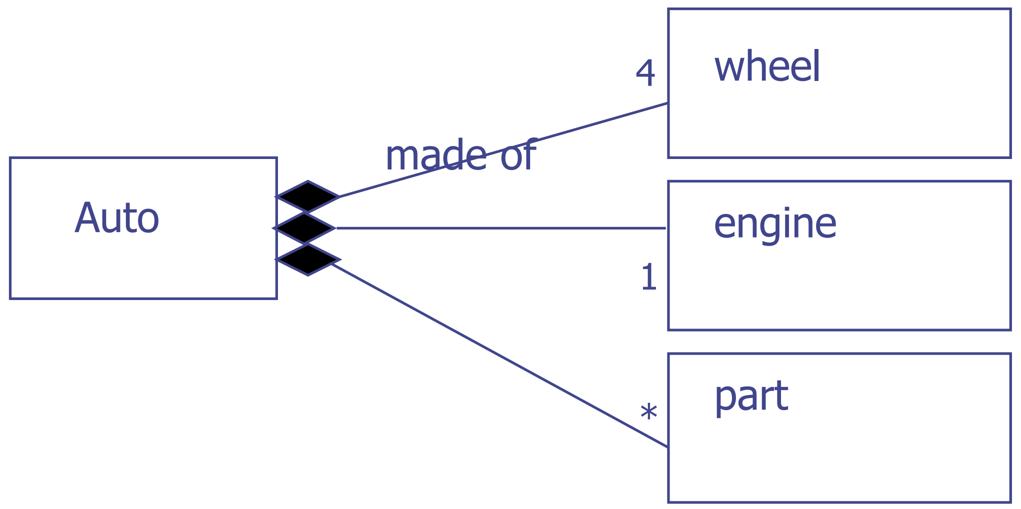


Aggregation

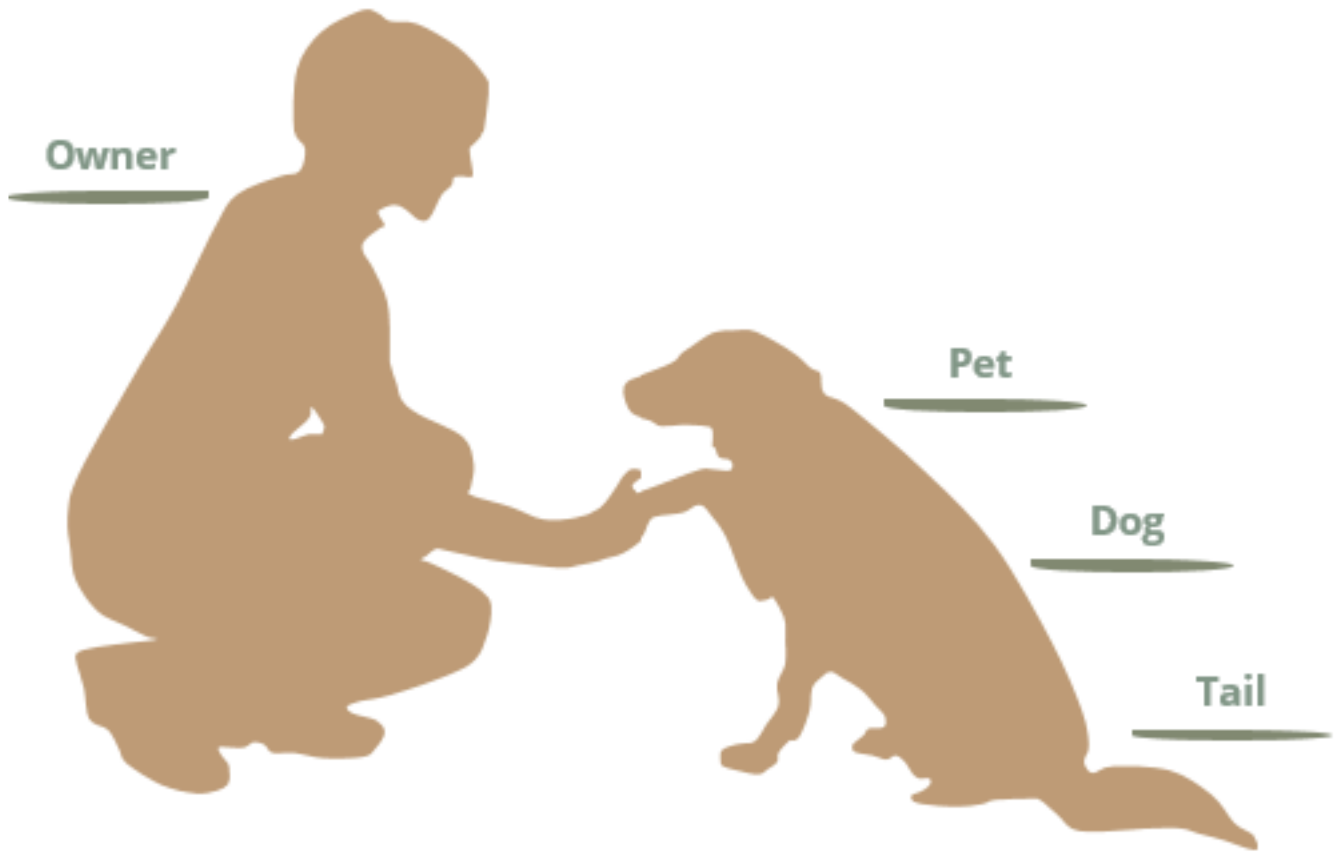
- For example, every Passenger has a Car but
  a Car doesn’t necessarily have a Passenger.
  - When you declare the Passenger class, you can create a field of the Car type that shows which car the passenger belongs to.
  - Then, when you instantiate a new Passenger object, you can access the data stored 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.

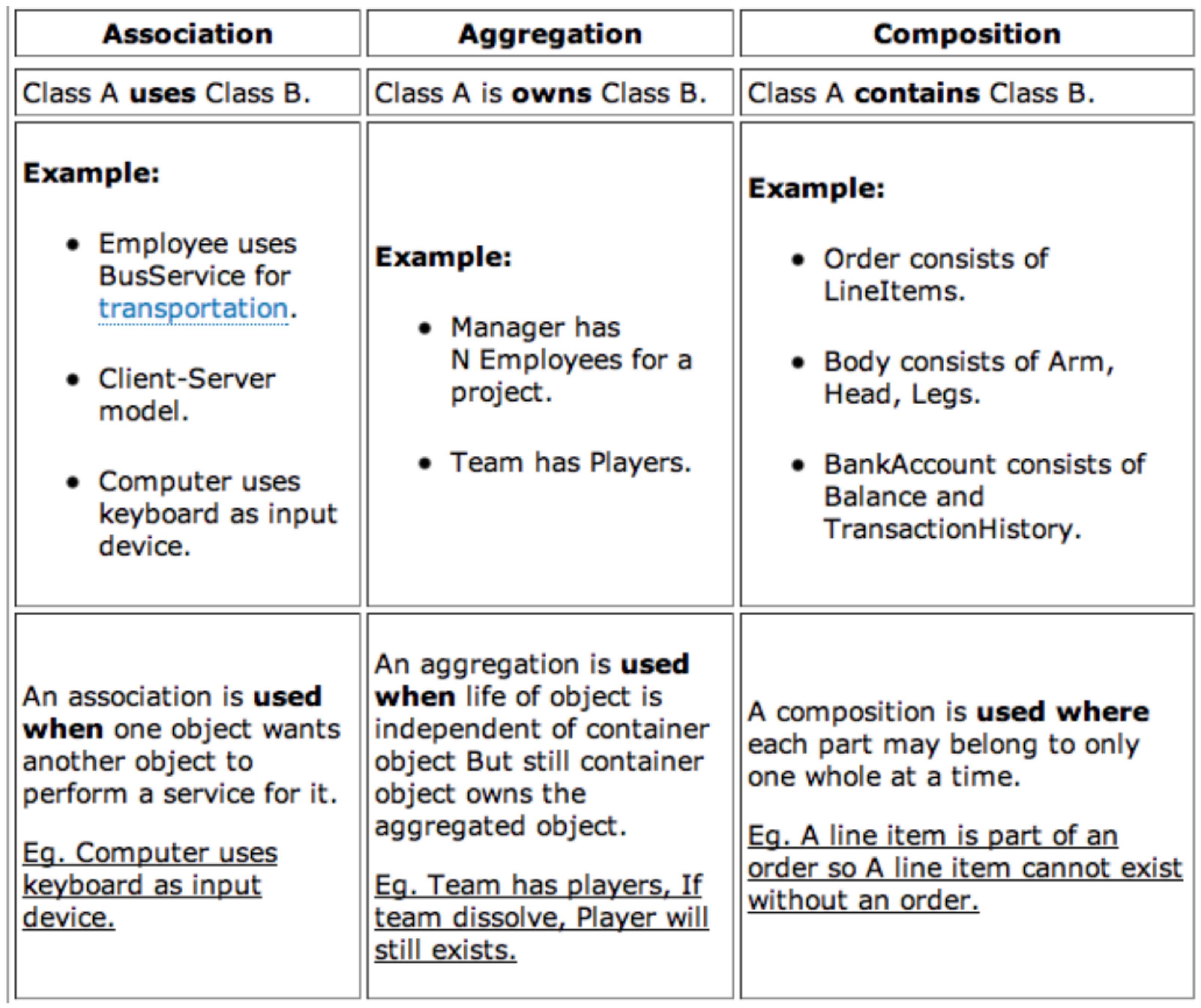


Association vs Aggregation vs Composition



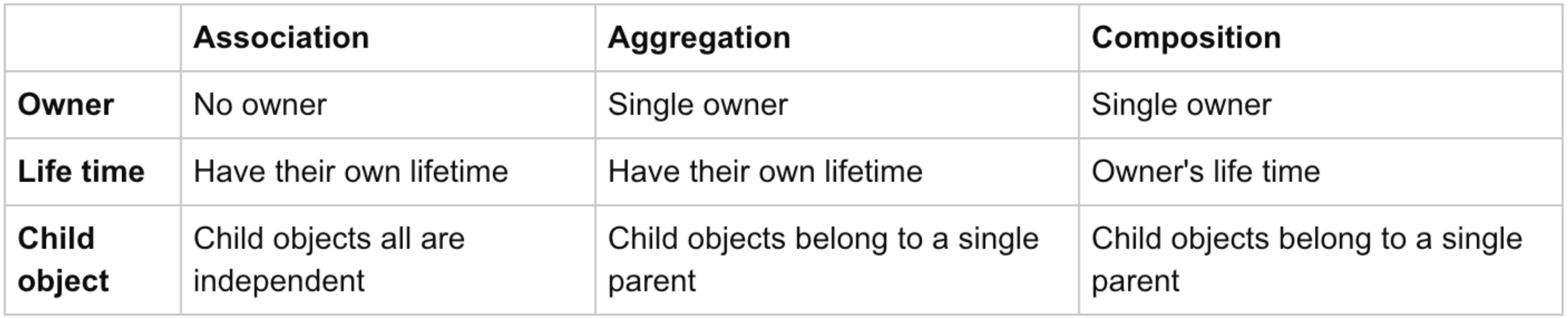
- An owner feeds pets, pets please owners (association)
- A tail is a part of both dogs and cats (aggregation / composi tion)
- A cat is a kind of pet (inheritance / generaliz ation)

Association vs Aggregation vs Composition



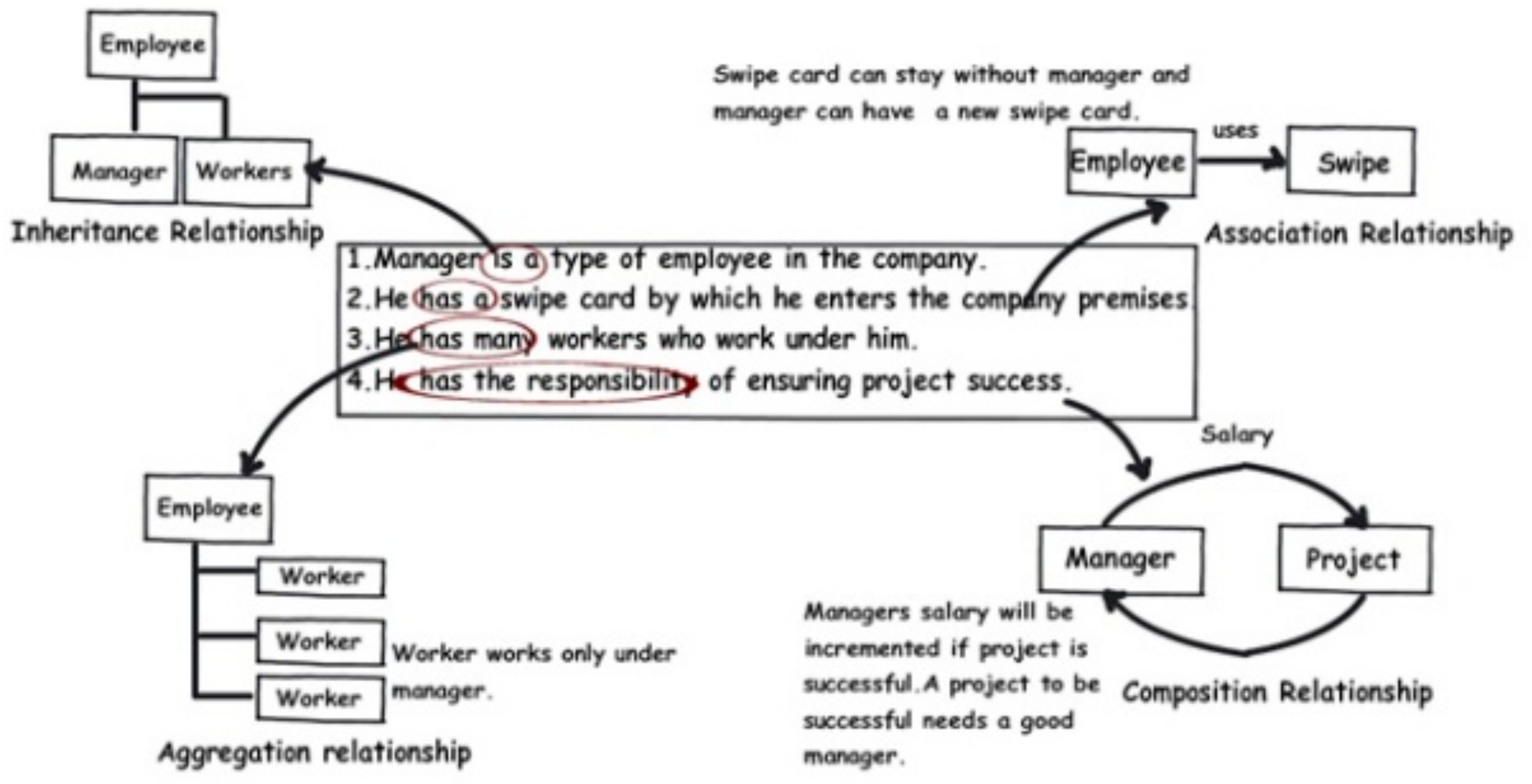
Association	**Aggregation**	**Composition**
Class A **uses** Class B.	Class A is **owns** Class B.	Class A **contains** Class B.
Example: • Employee uses BusService for transportation. • Client-Server model. • Computer uses keyboard as input device.	**Example:** • Manager has N Employees for a project. • 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.

Association vs Aggregation vs Composition

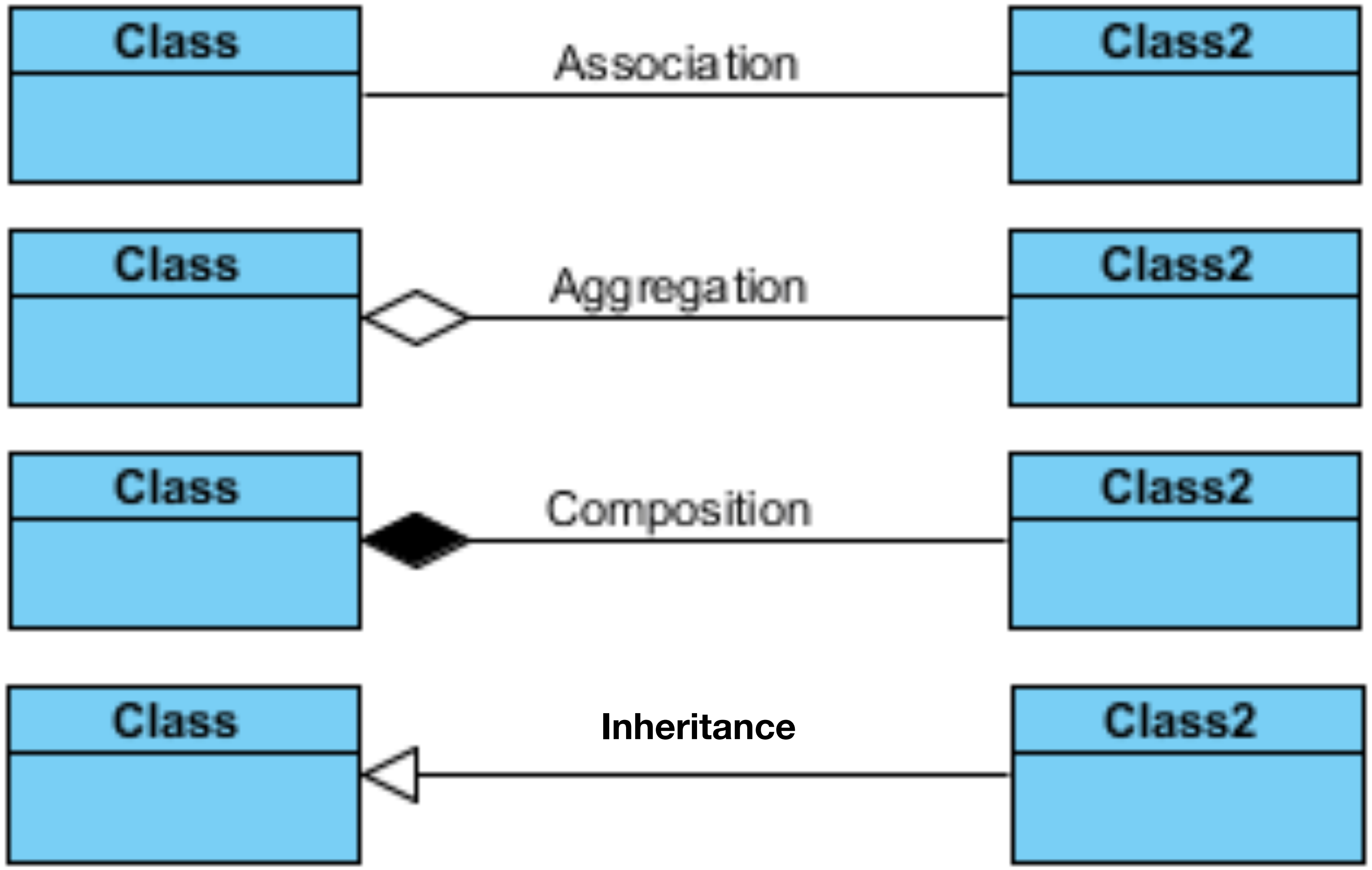


	Association	Aggregation	Composition
Owner	No owner	Single owner	Single owner
Life time	Have their own lifetime	Have their own lifetime	Owner's life time
Child object	Child objects all are independent	Child objects belong to a single parent	Child objects belong to a single parent

Association vs Aggregation vs Composition

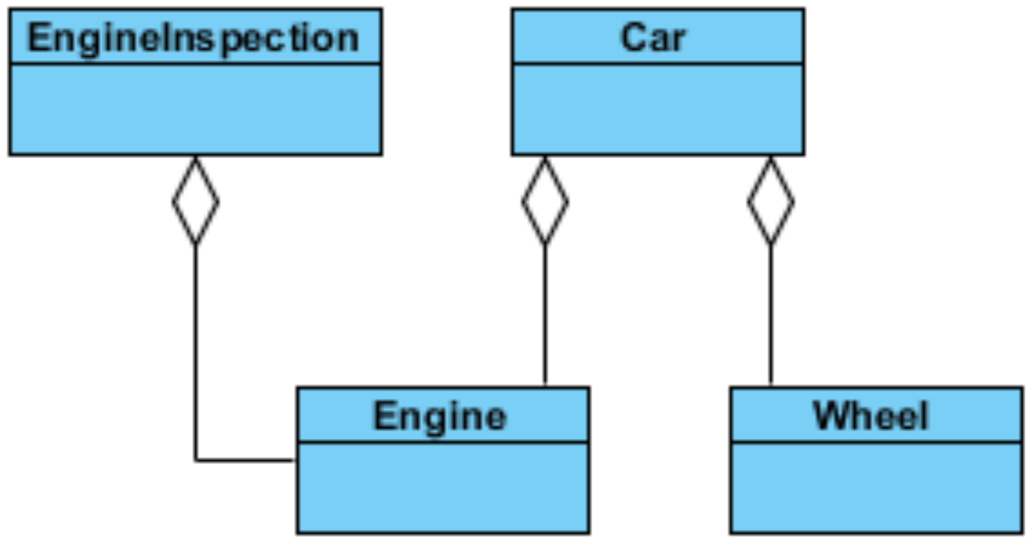

Employee
Manager
Workers
Inheritance Relationship
Swipe card can stay without manager and manager can have a new swipe card.
Employee
uses
Swipe
Association Relationship
1.Manager is a type of employee in the company.
2.He has a swipe card by which he enters the company premises.
3.He has many workers who work under him.
4.He has the responsibility of ensuring project success.
Salary
Manager
Project
Managers salary will be incremented if project is successful.A project to be successful needs a good manager.
Composition Relationship
Employee
Worker
Worker
Worker
Worker works only under manager.
Aggregation relationship

UML Notation

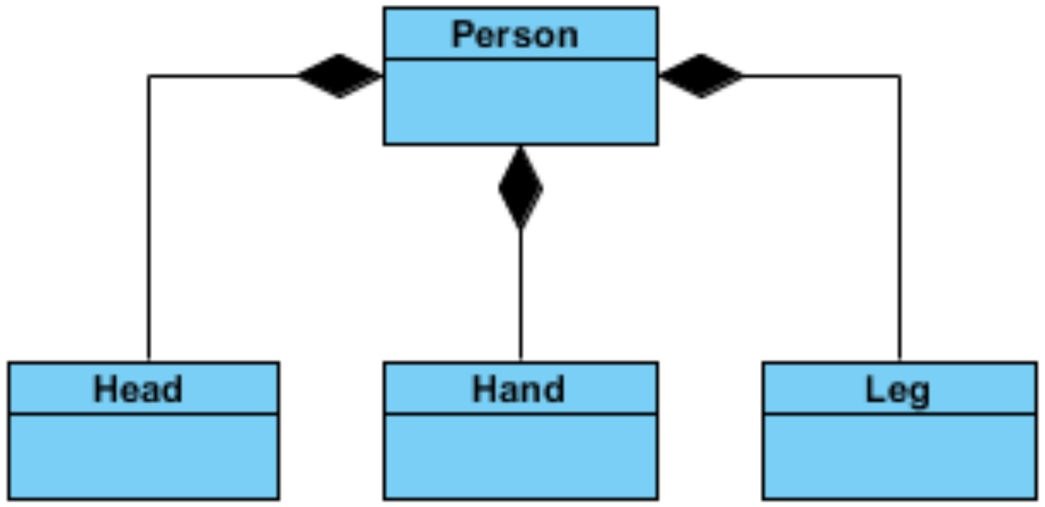


Composition vs Aggregation

- **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.



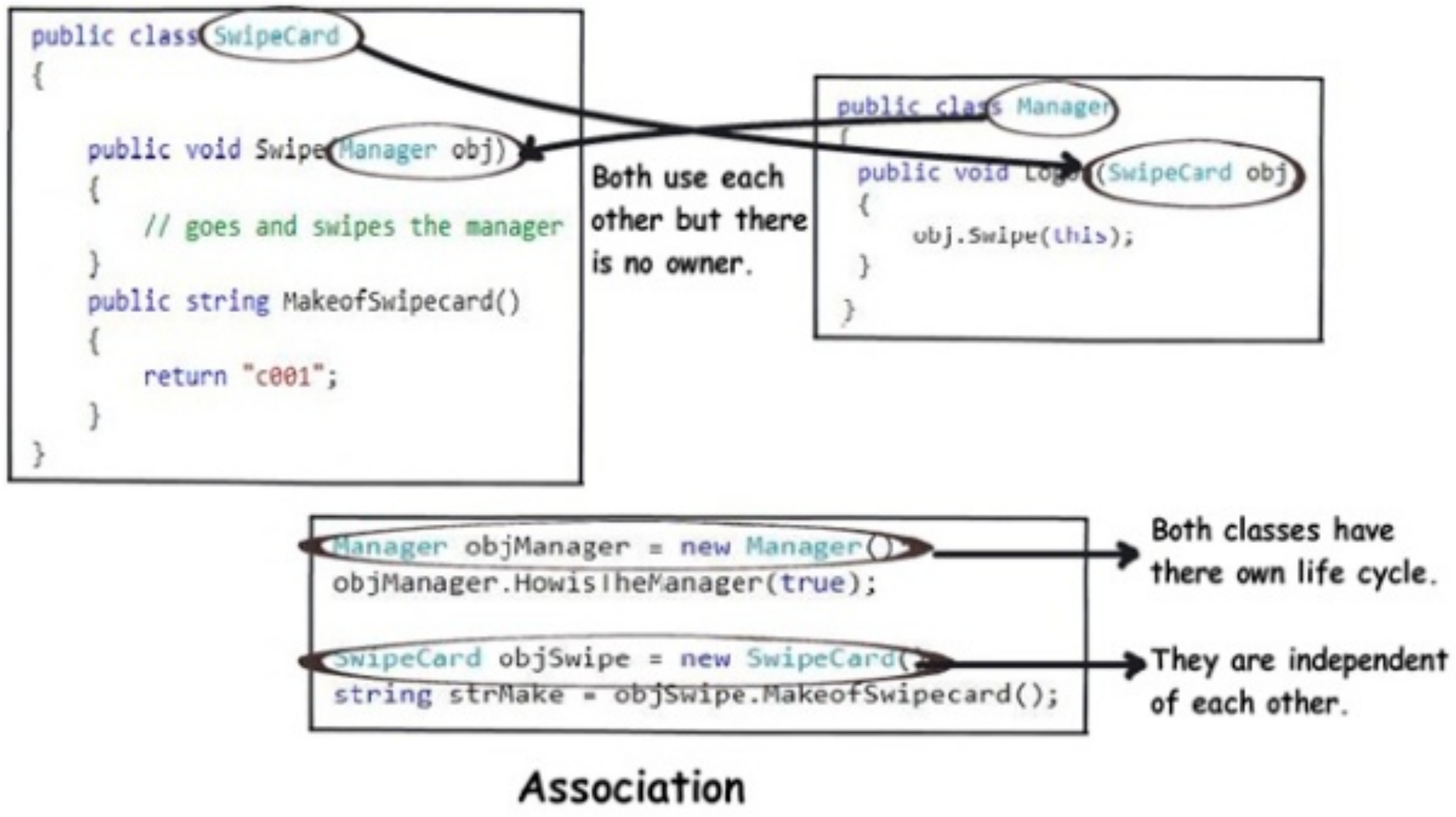
- **Composition** implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.



Exercise

- Find 5 use-cases or examples of association, aggregation and composition relationships from our real-life
- Explain to your partner why they are in such relationships
    - The partner tries to find why they might not in such relationships

Implementation: Association



```
public class SwipeCard
{

    public void Swipe(Manager obj)
    {
        // goes and swipes the manager
    }
    public string MakeofSwipecard()
    {
        return "c001";
    }
}
```

Both use each other but there is no owner.

```
public class Manager
{
  public void Logon(SwipeCard obj)
  {
      obj.Swipe(this);
  }
}
```

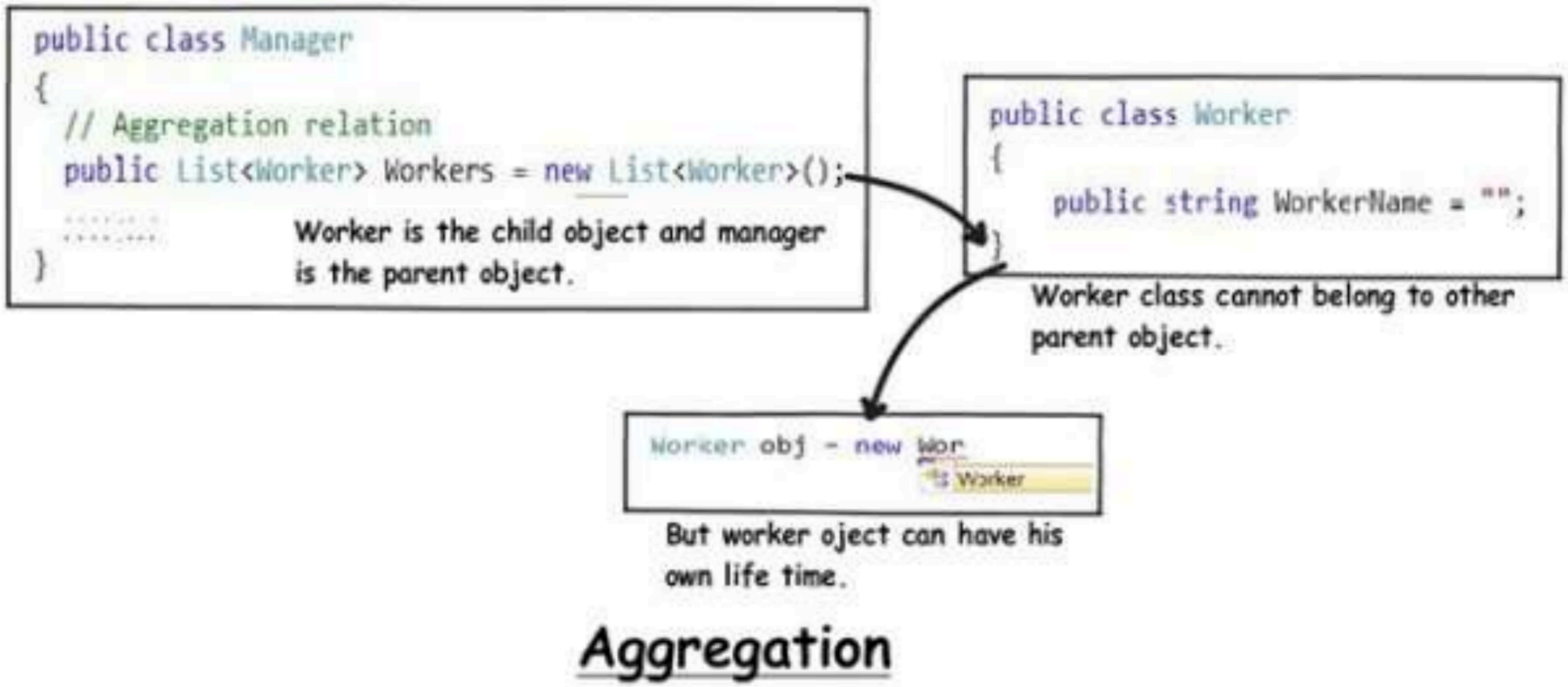
```
Manager objManager = new Manager();
objManager.HowisTheManager(true);

SwipeCard objSwipe = new SwipeCard();
string strMake = objSwipe.MakeofSwipecard();
```

Both classes have there own life cycle.

They are independent of each other.

Association

Implementation: Aggregation



```
public class Manager
{
  // Aggregation relation
  public List<Worker> Workers = new List<Worker>();
  .......
  .......
}
```

Worker is the child object and manager is the parent object.

```
public class Worker
{
    public string WorkerName = "";
}
```

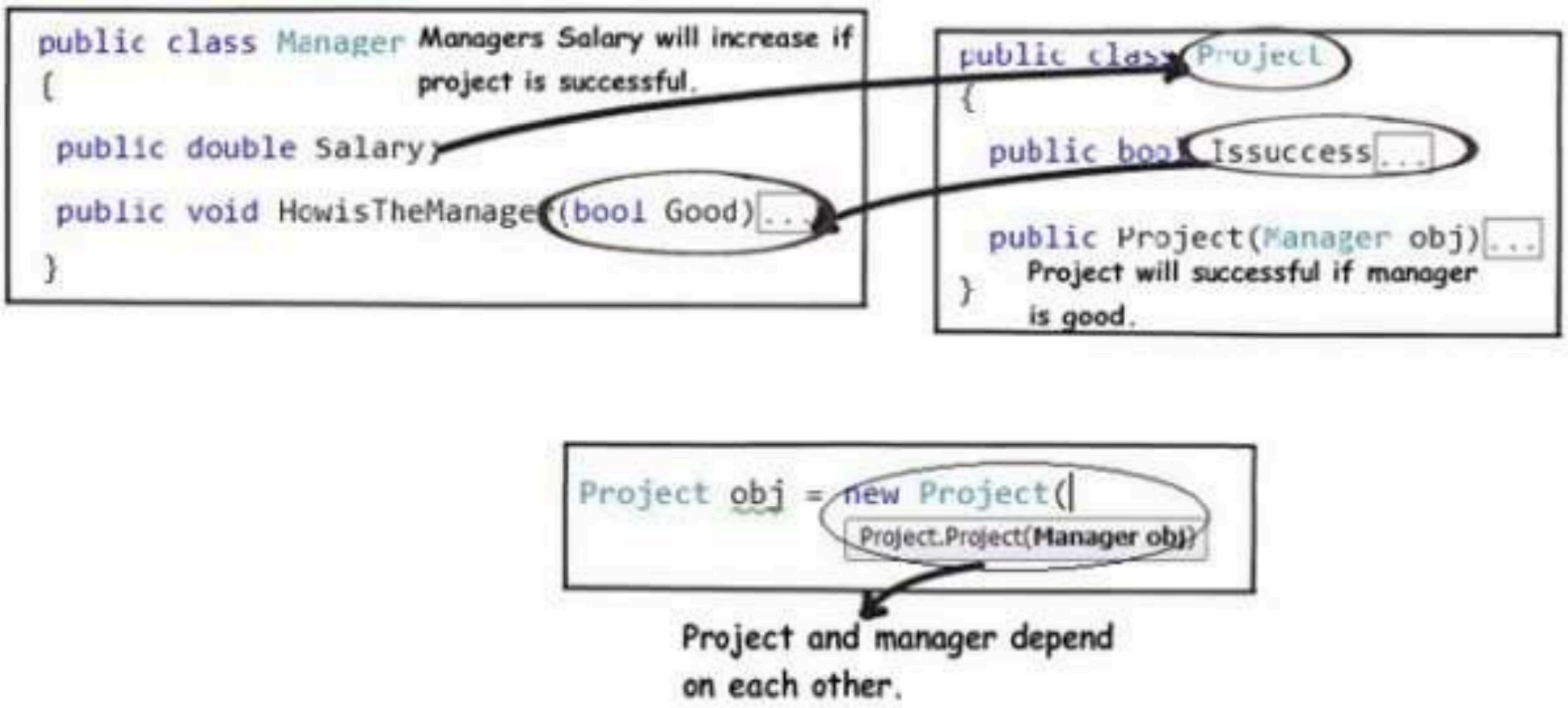
Worker class cannot belong to other parent object.

```
Worker obj = new Wor
```

But worker oject can have his own life time.

Aggregation

Implementation: Composition



```
public class Manager
{
 public double Salary;
 public void HowisTheManager(bool Good)...
}
```

Managers Salary will increase if project is successful.

```
public class Project
{
  public bool Issuccess...

  public Project(Manager obj)...
}
```

Project will successful if manager is good.

```
Project obj = new Project(
                 Project.Project(Manager obj)
```

Project and manager depend on each other.

Next Class

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