

Class Diagram

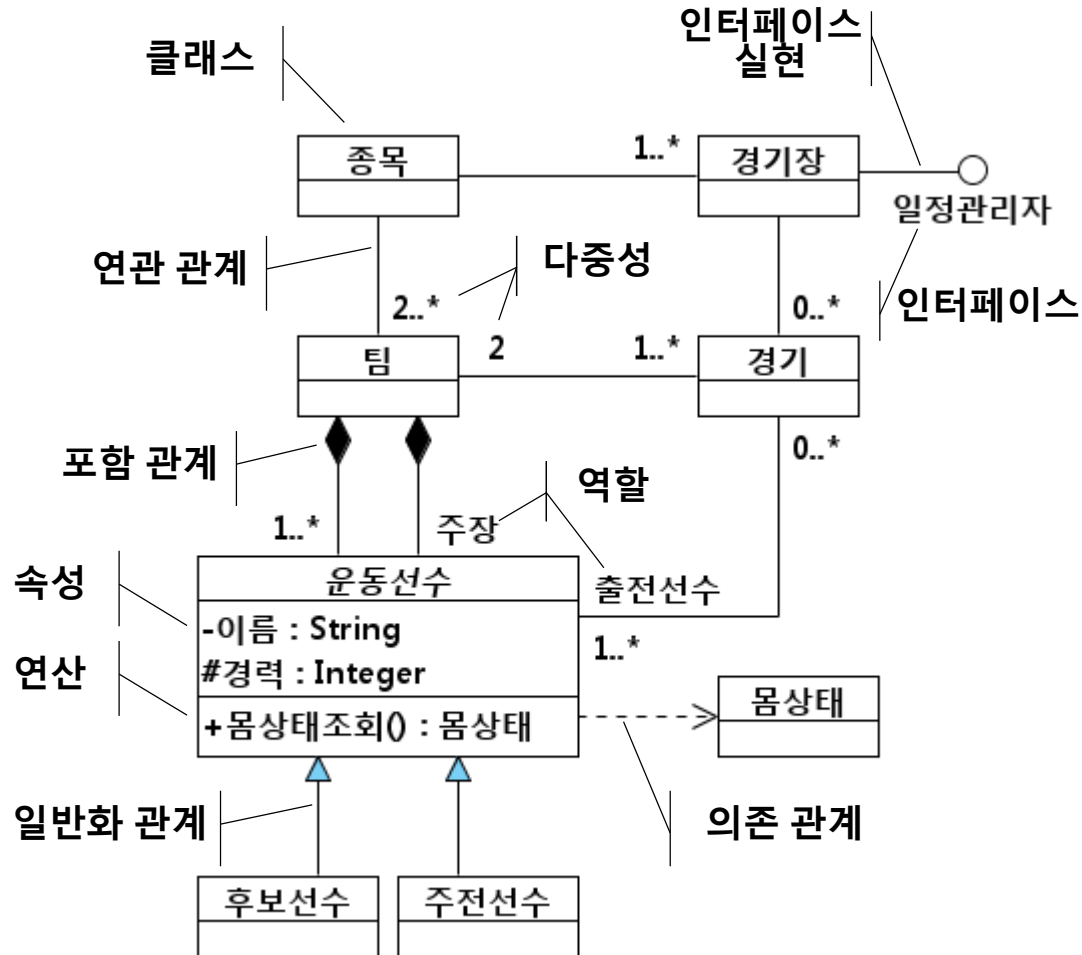
Objects, Classes

**Relationships – Association, Aggregation/Composition,
Generalization, Dependency**

Attribute

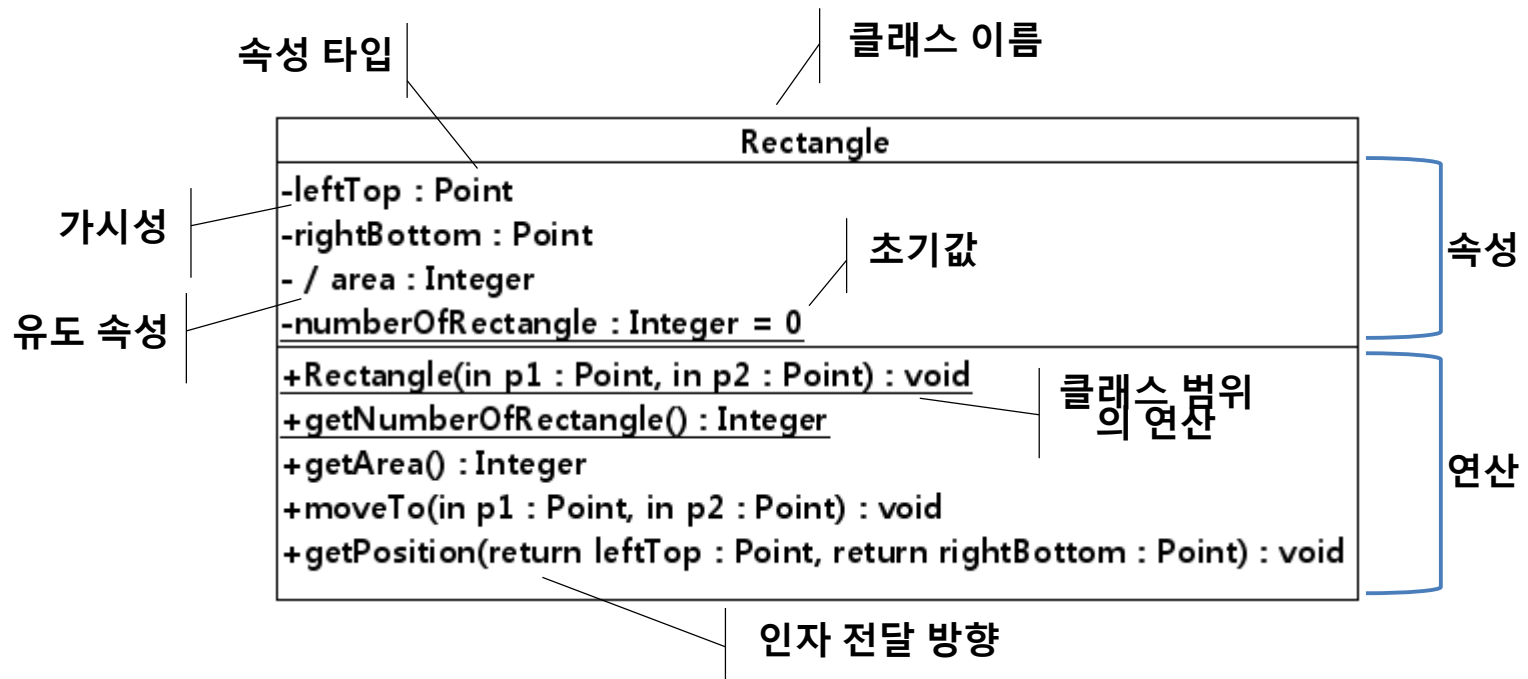
Operation

클래스 다이어그램의 요소



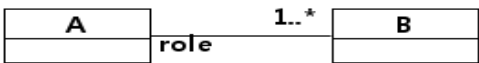
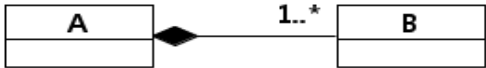

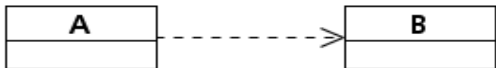
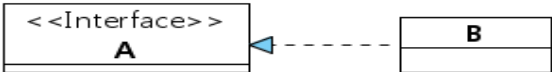
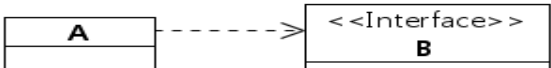
클래스 다이어그램의 요소

❖ 클래스의 표현



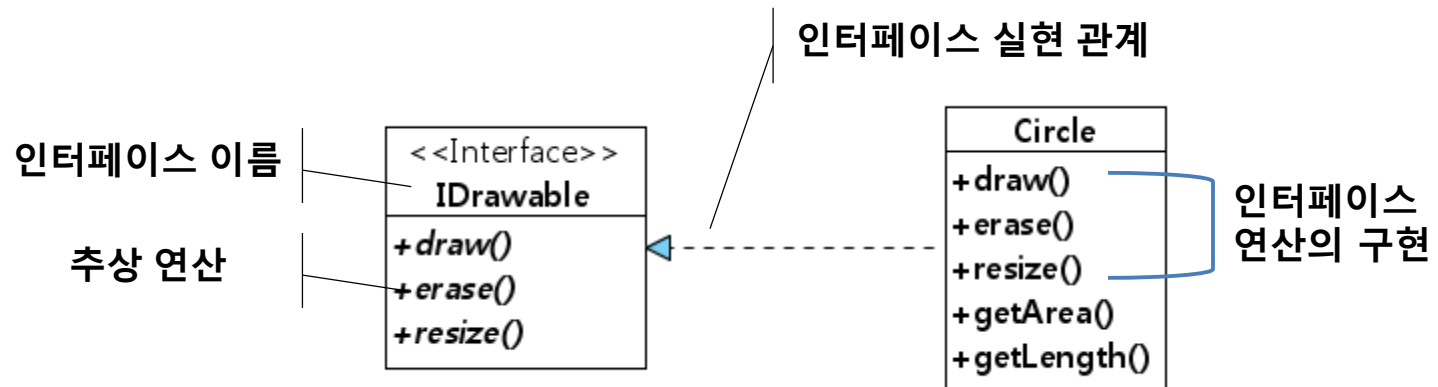
클래스 다이어그램의 요소

❖ 클래스 다이어그램의 관계

관계	표기법	의미
연관 관계		클래스 A와 클래스 B는 연관 관계를 가지고 있다.
포함 관계		클래스 B는 클래스 A의 부분이다.
일반화 관계		클래스 B는 클래스 A의 하위 클래스이다.
의존 관계		클래스 A는 클래스 B에 의존한다.
인터페이스 실현 관계		클래스 B는 인터페이스 A를 실현한다.
인터페이스 의존 관계		클래스 A는 인터페이스 B에 의존한다.

클래스 다이어그램의 요소

❖ 인터페이스



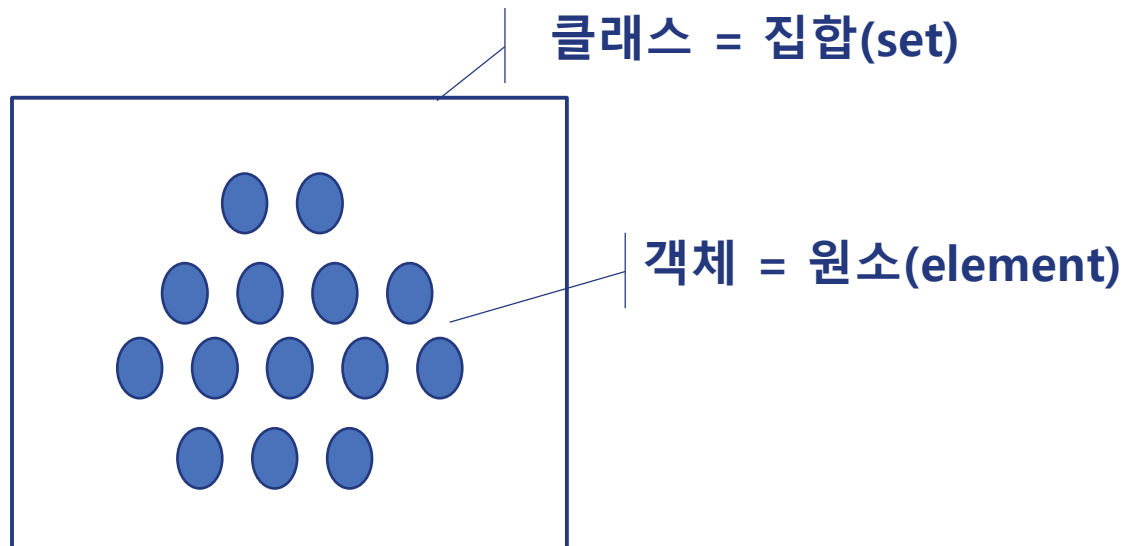
클래스와 객체 - 기본 개념

클래스는 유사한 객체들의 묶음이다

해당 객체	공통점		클래스
	<p>사람이 이동을 위해서 탈 수 있는 교통 수단들</p>		<p>교통수단</p>
	<p>다양한 유형의 사람들</p>		<p>사람</p>
	<p>살아 있는 생물들</p>		<p>생물</p>

클래스는 유사한 객체들의 묶음이다

- ❖ 클래스는 유사한 특성 즉 유사한 상태와 행동을 가지는 객체들을 한꺼번에 부르는 용어이다.
- ❖ 클래스는 마치 집합과 유사하다.

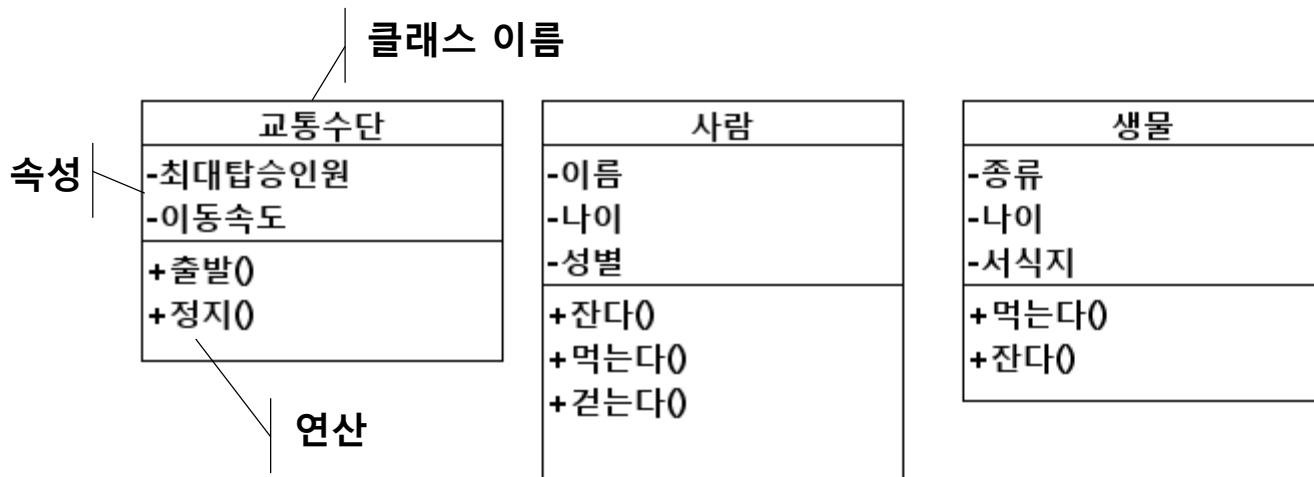


클래스는 유사한 객체들의 묶음이다

객체		UML	Java	C++
유사한 객체들		클래스	클래스	클래스
상태		속성	필드	데이터 멤버
행동		연산	메소드	멤버 함수

클래스는 유사한 객체들의 묶음이다

❖ 클래스의 표현 - 클래스 다이어그램



클래스와 객체 - 기본 원칙

클래스는 오직 한가지 유형의 대상과 개념만을 나타내야 한다

- ❖ 하나의 클래스는 오직 한가지 유형의 대상만을 표현해야 한다.
- ❖ 유사한 속성과 유사한 연산을 가질 수 있는 객체들만을 묶어서 클래스로 정의하도록 한다
- ❖ 만약 유사한 속성과 연산이 없다면 다른 클래스로 정의하도록 한다.

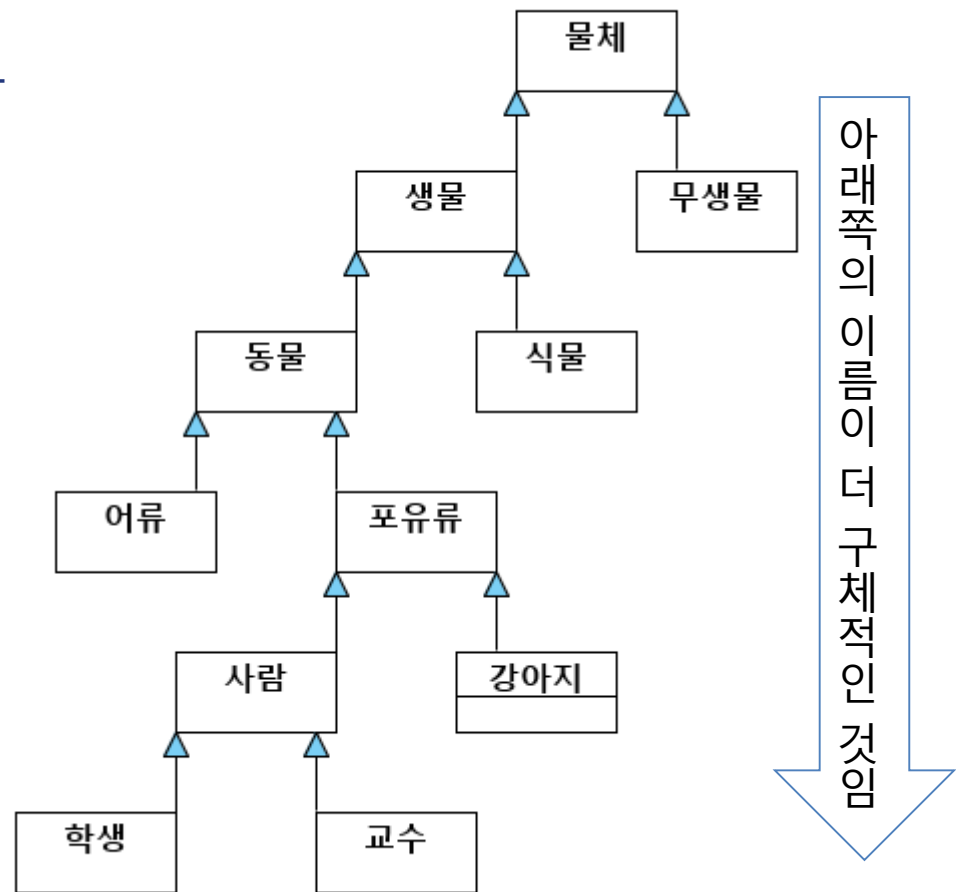
클래스의 이름은 명확하고 구체적이어야 한다.

- ❖ 가장 구체적인 용어를 클래스의 이름으로서 사용해야 한다.

대상 객체들	적합한 이름	부적합한 이름
	어류	생물 동물
	사람	생물 동물 포유류
	컴퓨터	무생물 전자장치

클래스의 이름은 명확하고 구체적이어야 한다.

- ❖ 구체적인 용어를 사용한다는 것은 일반화 계층 구조에서 하단에 위치한 클래스를 뜻한다.



클래스 이름과 속성/연산은 일관성이 있어야 한다.

- ❖ 클래스의 이름으로부터 기대될 수 있는 속성/연산을 가지고 있어야 한다.

클래스	속성	관련 클래스
<div><div>사람</div><div><ul style="list-style-type: none">-위치-출생일-나이-이동속도-임신기간-국적-직업</div></div>	위치	물체
	출생일	생물
	나이	
	이동속도	동물
	임신기간	포유류
	국적	사람
	직업	근로자

클래스 이름과 속성/연산은 일관성이 있어야 한다.

❖ 해당 클래스 고유의 속성/연산이 정의되어 있지 않다면 부적절하다.

학생
-이름 -주소 -나이

교수
-이름 -주소 -나이

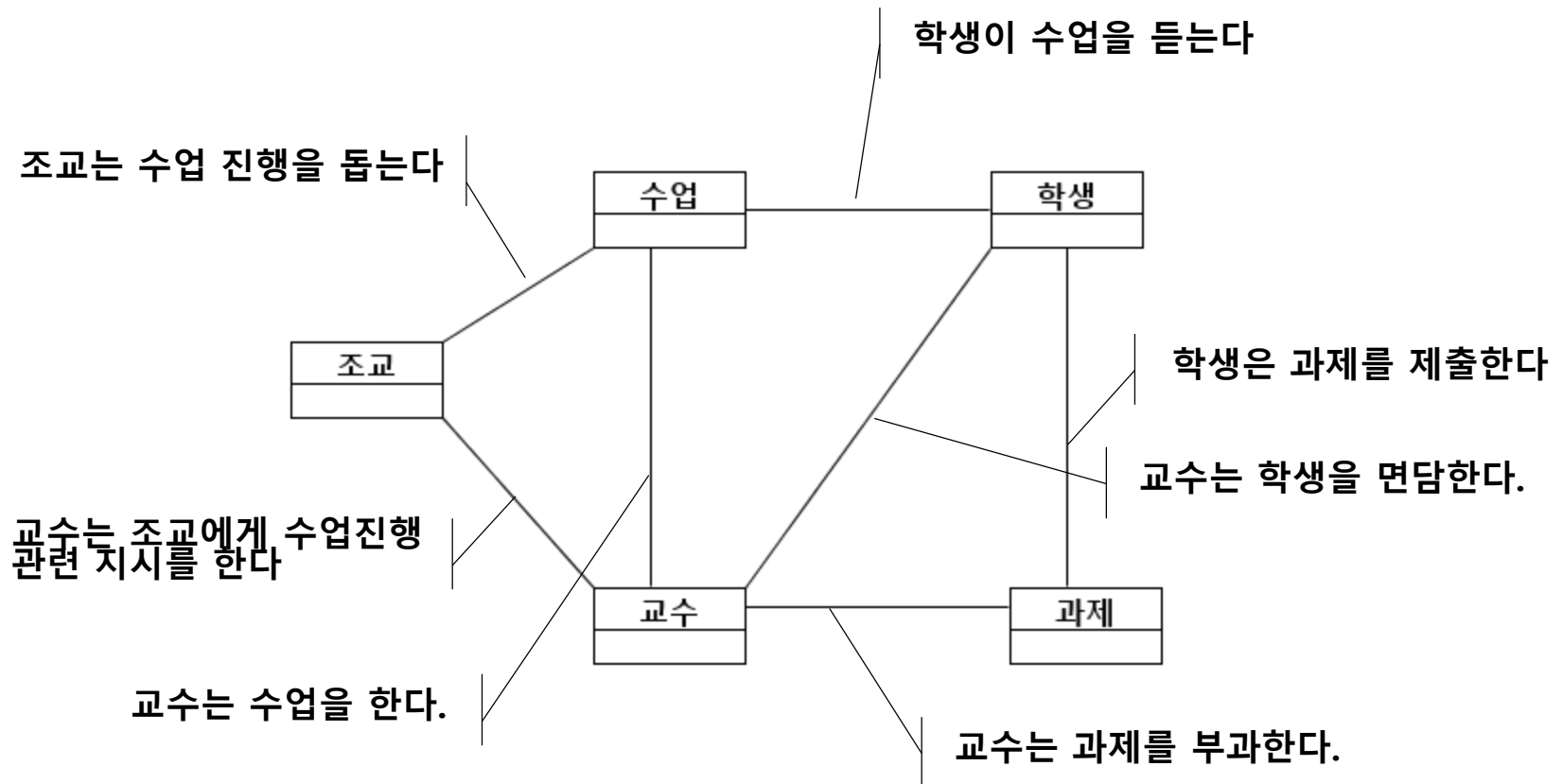
근로자
-이름 -주소 -나이

- 이름, 주소, 나이 속성은 학생, 교수, 근로자의 고유의 정보/속성이 아니므로 이들은 적합한 클래스가 아니다.
- 학생 클래스의 경우에는 소속학과, 학년, 성적 등의 속성이 있어야 하며,
- 교수 클래스의 경우에는 소속학과, 전공, 담당과목수 등의 속성이 정의되어야 한다.
- 근로자 클래스의 경우에는 소속회사, 직급, 담당업무 등의 속성이 정의되어야 만이 클래스의 이름과 속성이 일관된다.

연관 관계 - 기본 개념

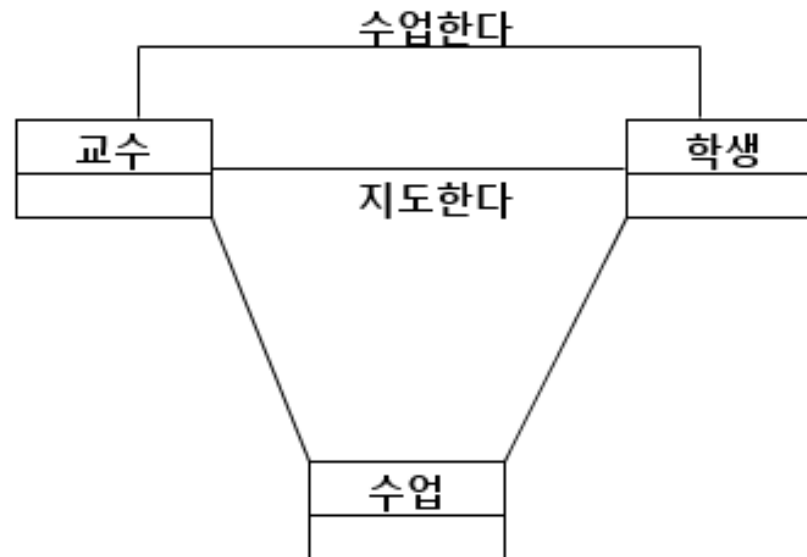
연관 관계는 두 개 이상의 클래스 간의 관련성을 뜻한다

- ❖ 두 클래스 간에 “무엇인가” 관련성이 있음을 뜻한다.
- ❖ 정확한 의미는 관련된 두 클래스에 따라서 달라진다.



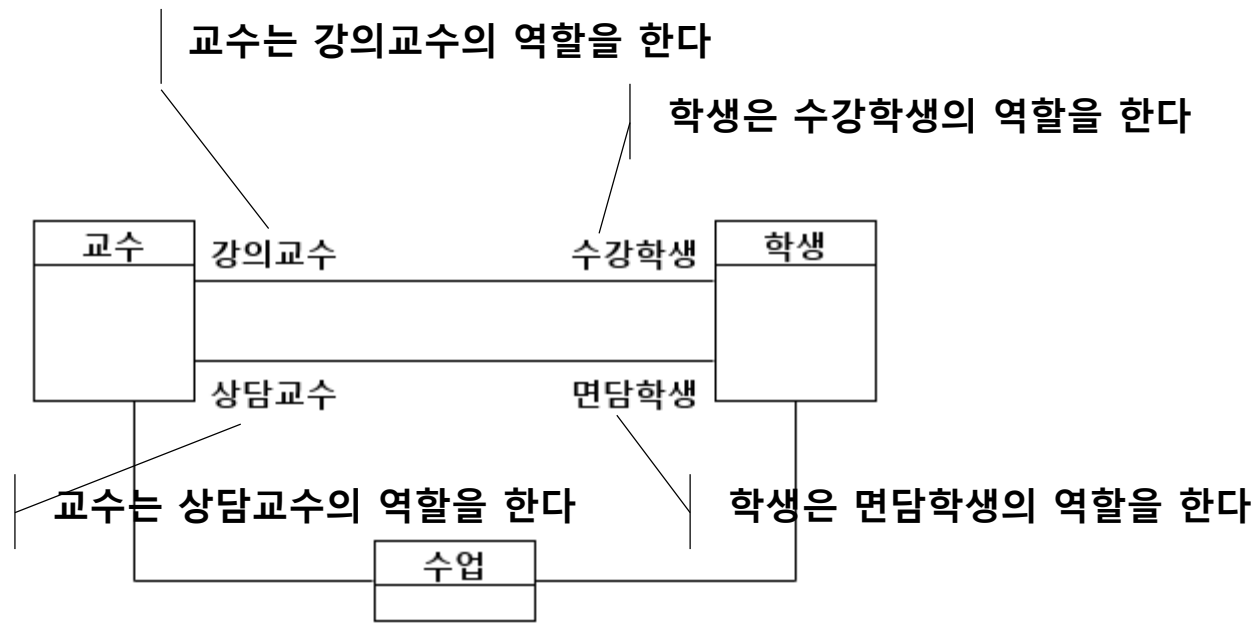
연관의 이름과 역할은 연관 관계의 의미를 명확하게 할 수 있다.

- ❖ 두 클래스 간의 연관 관계의 의미를 연관의 이름(association name)과 역할(role)을 이용해서 구체적으로 표현할 수 있다.
- ❖ 연관의 이름을 이용한 의미 표현



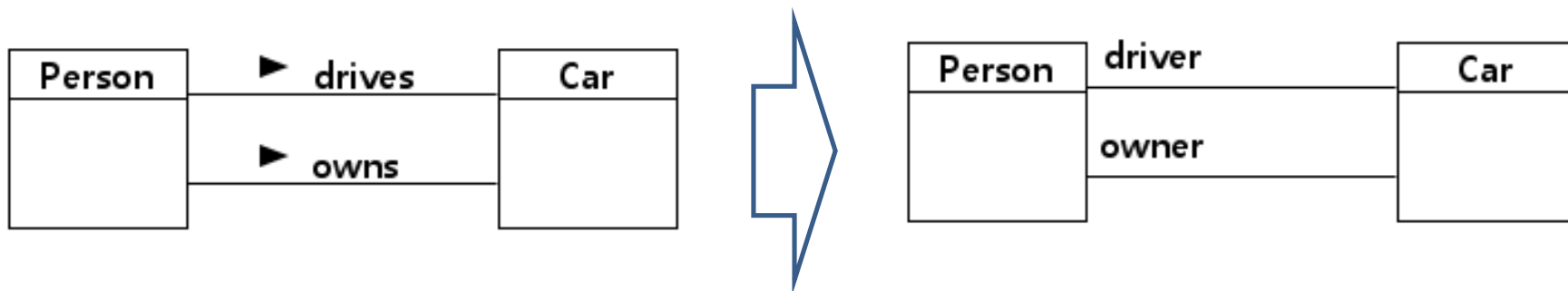
연관의 이름과 역할은 연관 관계의 의미를 명확하게 할 수 있다.

❖ 역할을 이용한 의미 표현



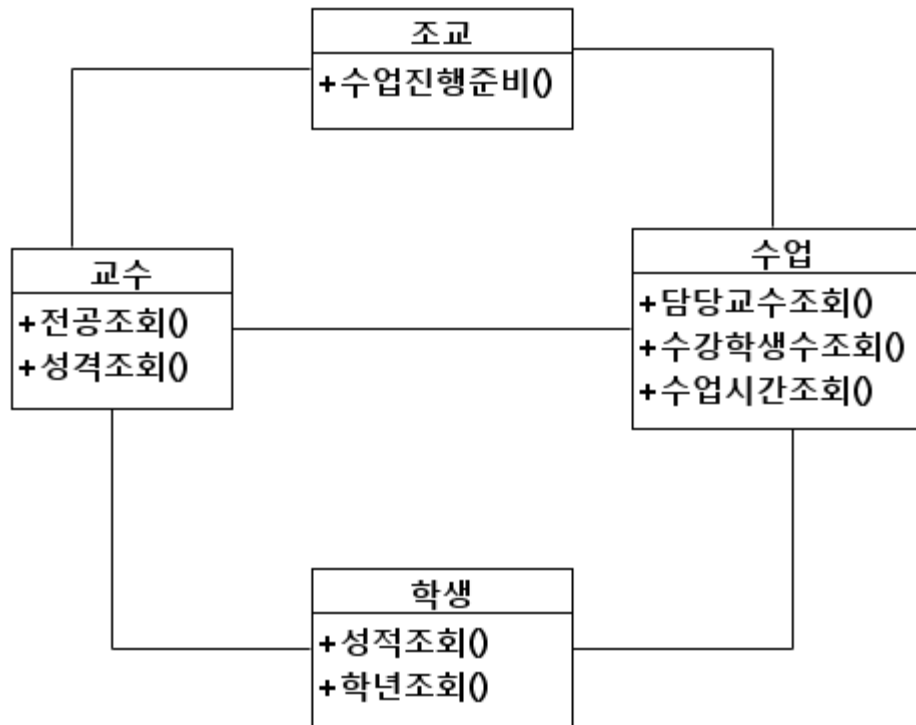
연관의 이름 보다는 역할이 보다 권장된다

- ❖ 연관의 의미를 명확하기 위하여 연관의 이름과 역할이 사용될 수가 있다.
- ❖ **연관의 이름 보다는 역할의 사용이 더 권장된다.**
 - 연관 이름은 일반적으로 동사구이지만, 역할은 명사이므로 이해하기가 더 편하다.
 - 역할은 상대 객체에 대한 참조 변수의 이름으로 사용된다.
 - 일부 UML 모델링 도구에서는 역할의 이름을 지정되지 않으면 연관을 위한 필드를 자동으로 생성하지 않기도 한다

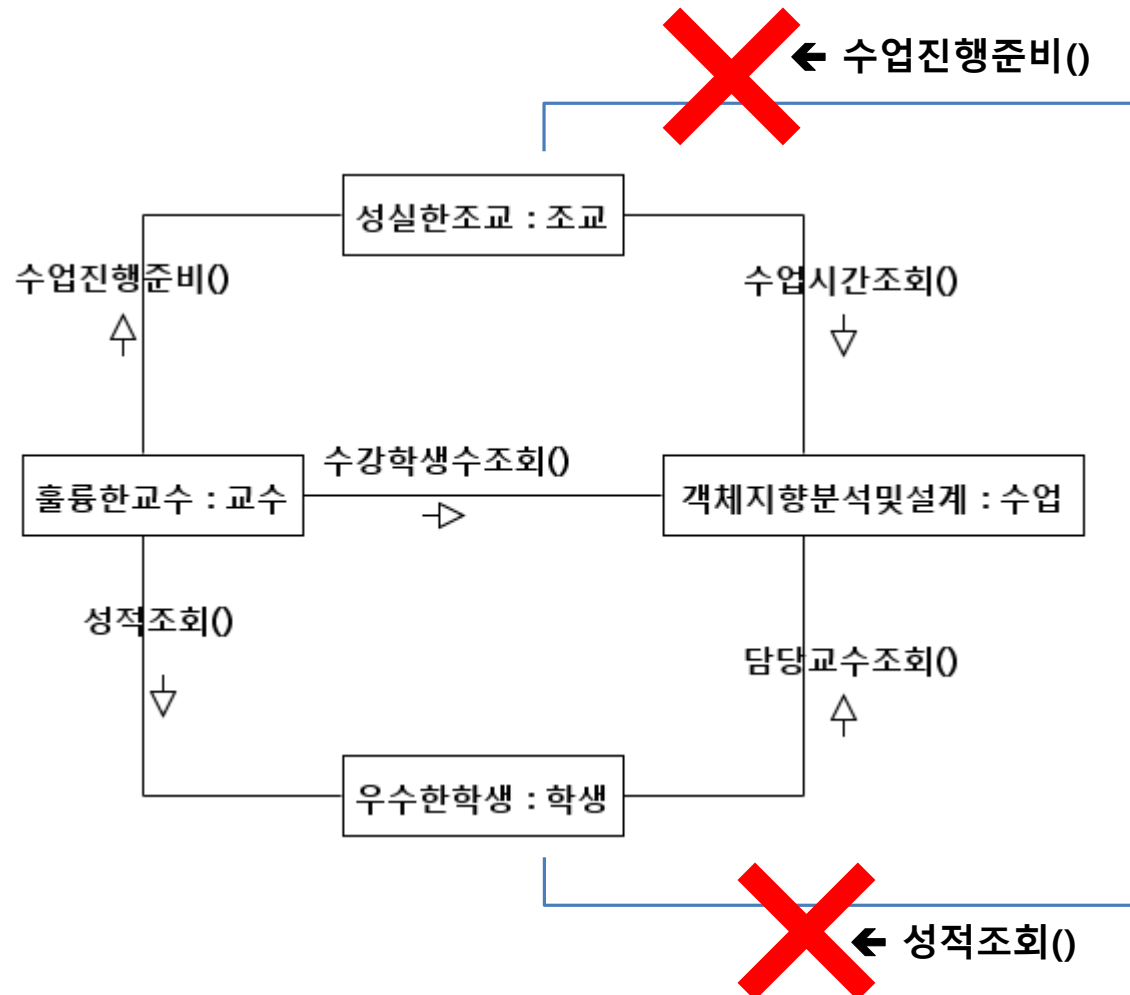


연관 관계는 메시지 전달의 통로 역할을 한다

- ❖ 한 객체는 연관 관계가 있는 다른 객체의 기능을 이용할 수가 있다.
- ❖ 반대로 말하면 연관 관계가 없는 객체의 기능을 이용할 수는 없다.

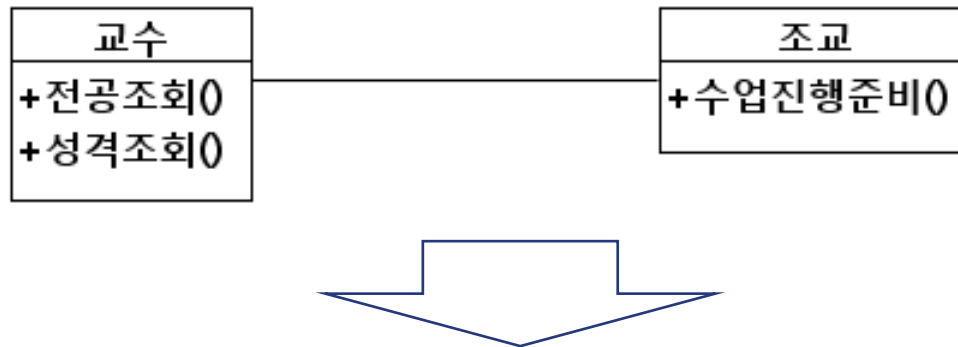


연관 관계는 메시지 전달의 통로 역할을 한다



연관 관계는 메시지 전달의 통로 역할을 한다

❖ 연관 관계의 표현 - C++



```
class 교수 {
    조교* a조교 ;
public: void 전공조회() { a조교....() ; }
public: void 성격조회() { a조교....() ; }
}
```

```
class 조교 {
    교수* a교수 ;
public: void 수업진행준비() { a교수....() ; }
}
```


연관 관계는 메시지 전달의 통로 역할을 한다

```
class 교수 {  
    private 조교* a조교 ;  
    public: void 전공조회() { a조교....() ; }  
    public: void 성격조회() { a조교....() ; }  
    public: void set조교(조교* a조교) {  
        this->a조교 = a조교 ;  
    }  
}
```

```
class 조교 {  
    교수* a교수 ;  
    public: void 수업진행준비() { a교수....() ; }  
    public: void set교수(교수* a교수) {  
        this->a교수 = a교수 ;  
    }  
}
```

p1:교수

t1:조교

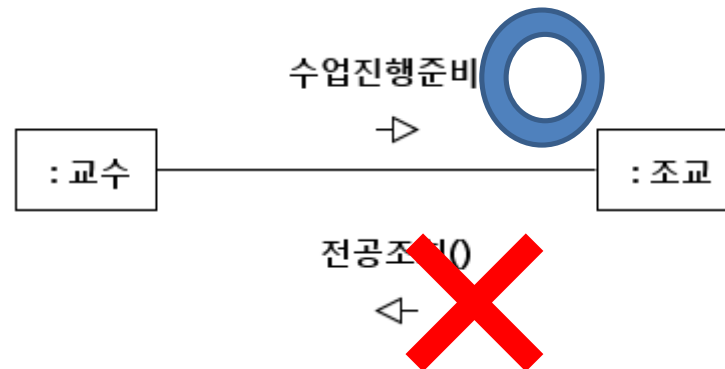
```
void main() {  
    교수* p1 = new 교수(...);  
    조교* t1 = new 조교(...);  
  
    p1->set조교(t1);  
    t1->set교수(p1);  
    ...  
}
```

연관 관계는 메시지 전달의 통로 역할을 한다

- ❖ 연관 관계는 방향성이 지정될 수가 있다.



- ❖ 연관 관계의 방향성은 메시지 전달의 방향을 뜻한다.



연관 관계는 메시지 전달의 통로 역할을 한다

❖ 연관 관계의 방향성 표현 - C++

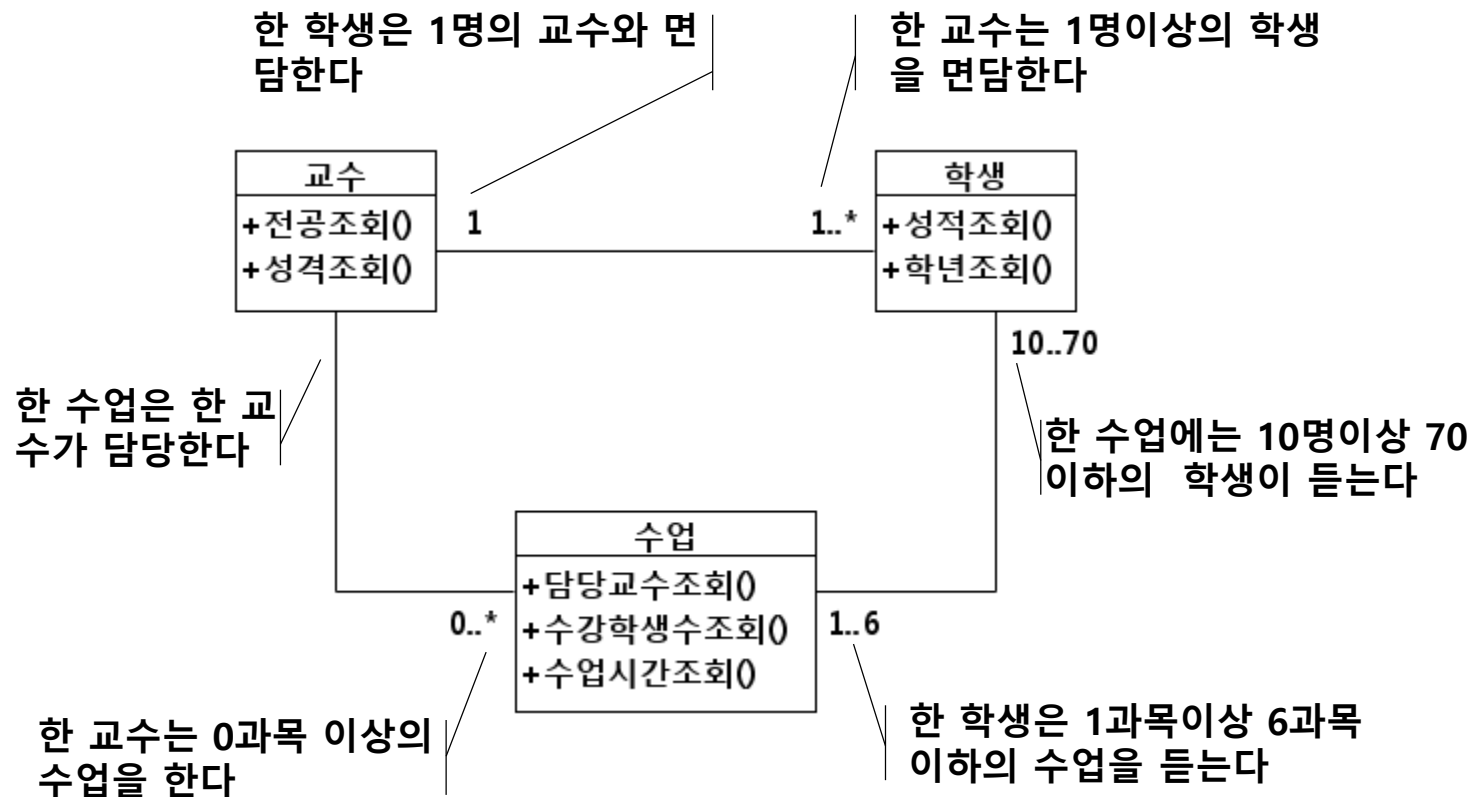


```
class 교수 {
    private: 조교* a조교 ;
    public: void 전공조회() { a조교....() ; }
    public: void 성격조회() { a조교....() ; }
}
```

```
class 조교 {
    private 교수 a교수 ;
    public: void 수업진행준비() { a교수....() ; }
}
```

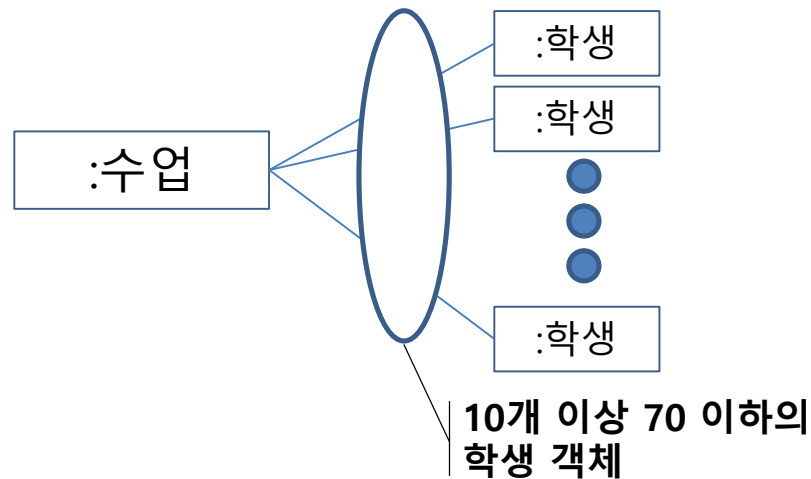
연관 관계의 다중성은 복수 개의 객체와의 관련성을 표현한다

- ❖ 관계를 맺을 수 있는 실제 상대 객체의 수를 다중성(multiplicity)를 통하여 지정할 수 있다



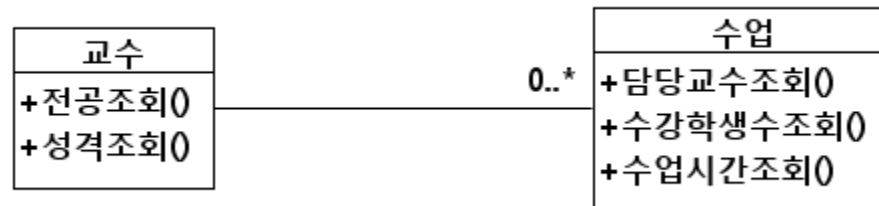
연관 관계의 다중성은 복수 개의 객체와의 관련성을 표현한다

- ❖ 다중성은 클래스 다이어그램에 표시되지만 실제로는 연관되는 상대 객체의 수에 대한 제약이다.



연관 관계의 다중성은 복수 개의 객체와의 관련성을 표현한다

❖ 연관 관계의 다중성 표현 - C++

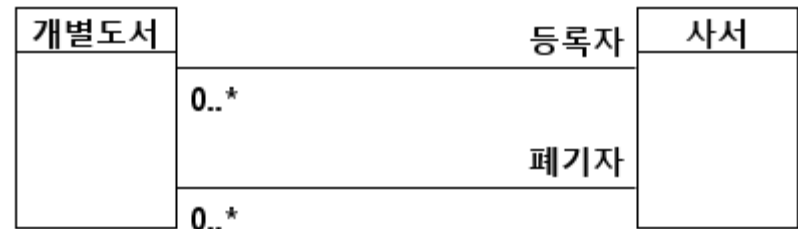
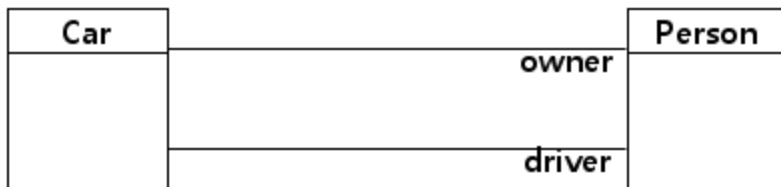


```
class 교수 {  
    private: vector<수업*> a수업 ;  
    public: void 전공조회() { ... }  
    public: void 성격조회() { ...; }  
}
```

```
class 수업 {  
    private: 교수* a교수 ;  
    public: void 담당교수조회() { ... }  
    public: void 수강학생수조회() { ... }  
    public: void 수업시간조회() { ... }  
}
```

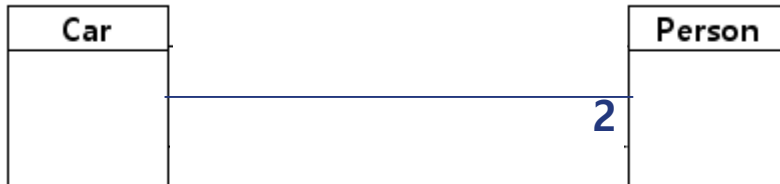
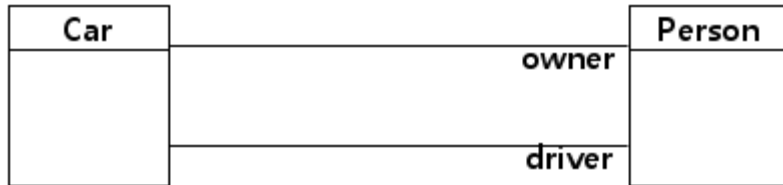
다중 연관은 동일한 클래스 간의 존재하는 복 수개의 연관 관계를 뜻한다.

- ❖ 동일한 두 클래스 간에 2개 이상의 연관 관계가 있는 경우 "다중(multiple) 연관 관계가 있다"라고 말한다.



- ❖ **연관의 다중성**은 동일한 의미/역할의 복 수개의 객체와의 관계를 뜻하는 반면에 **다중 연관**은 다른 의미/역할을 가지는 객체를 뜻한다

다중 연관 vs. 연관의 다중성

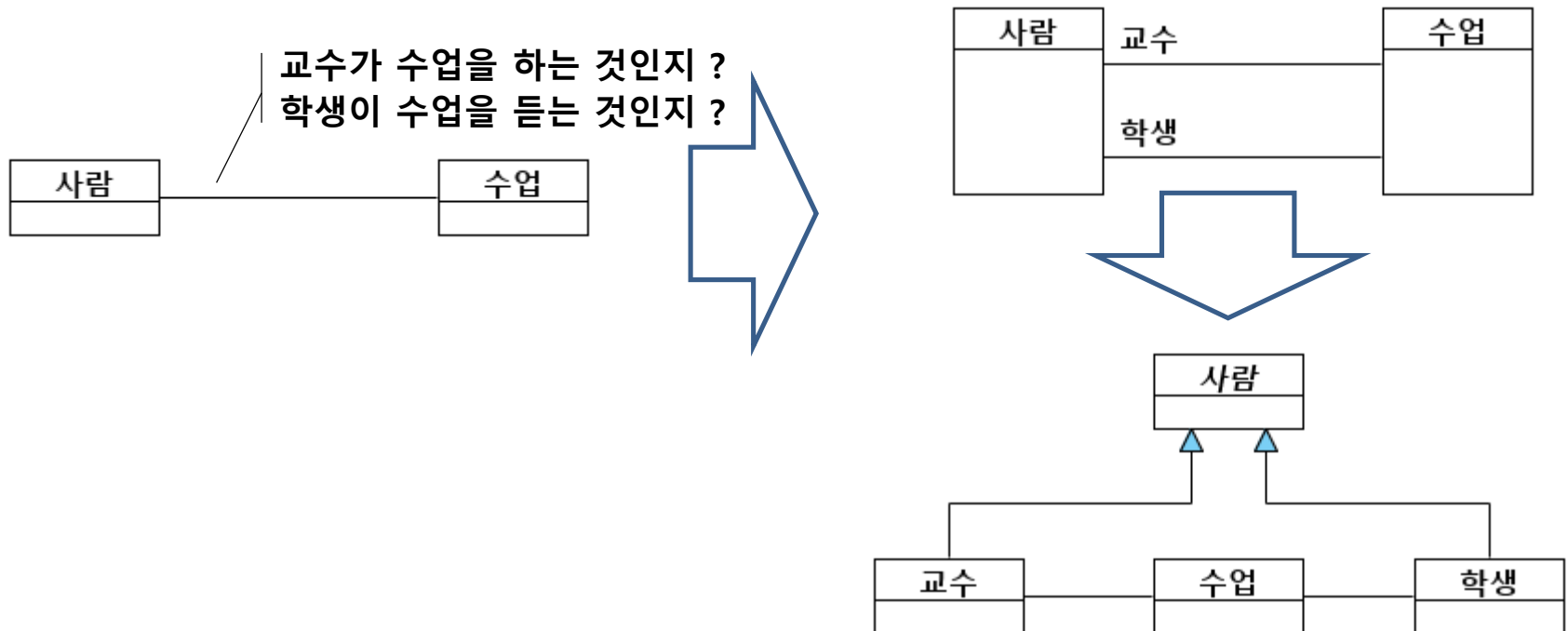


```
class Car {
    Person* owner ;
    Person* driver ;
public:
    void setOwner(Person* p) {
        owner = p ;
    }
    void setDriver(Person* p) {
        driver = p ;
    }
}
```

```
class Car {
    Person* persons[2] ;
public:
    void setPerson(int i, Person* p) {
        persons[i] = p ;
    }
}
```

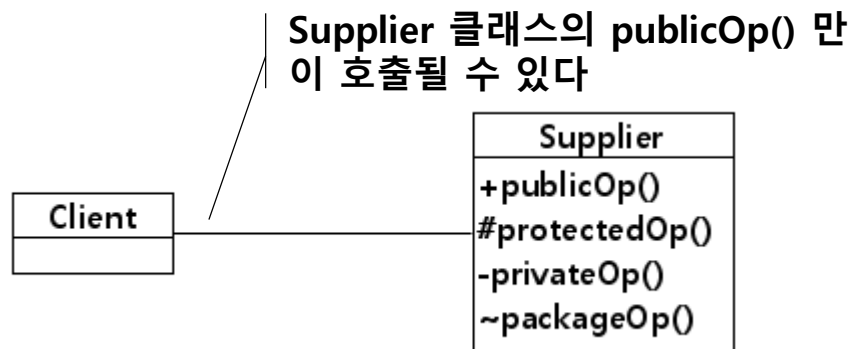

연관 관계의 의미는 관련된 클래스들에 의해서 명확하게 결정되어야 한다

- ❖ 연관 관계의 의미는 관련된 두 클래스로부터 명확하게 결정될 수 있어야 한다.
- ❖ 관련된 클래스만으로 연관의 의미를 결정할 수 없다면 클래스 다이어그램을 수정되어야 한다



연관 관계는 메시지 전달과 일관되어야 한다

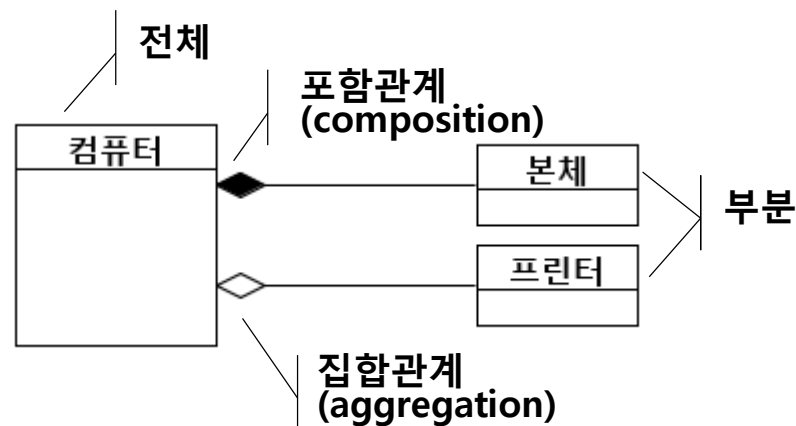
- ❖ 클래스 간의 연관 관계와 객체 간의 메시지 전달은 다음과 같은 규칙을 준수해야 한다
 - 연관 관계가 있는 클래스 간에 메시지가 전달 될 수 있다.
 - 메시지 전달 방향은 방향성과 일치되어야 한다.
 - 메시지에 대응되는 연산이 공용의 가시성으로 정의되어 있어야 한다



집합 관계와 포함 관계 - 기본 개념

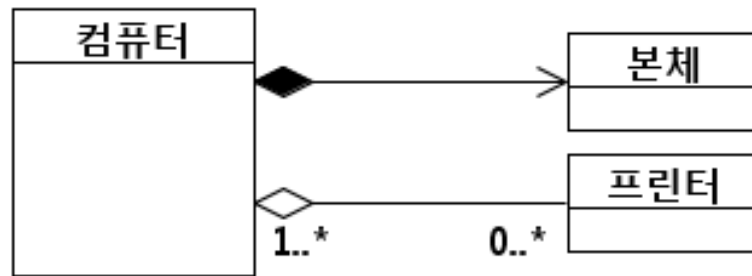
집합/포함 관계는 두 대상 간의 포함/소속의 의미를 표현한다

- ❖ 집합/포함관계는 두 객체간의 포함/소속의 의미를 클래스 수준에서 표현한 것이다
- ❖ 집합/포함 관계는 “전체는 부분으로 구성된다.” 또는 “부분은 전체의 부분이다.”라는 의미로 해석된다.



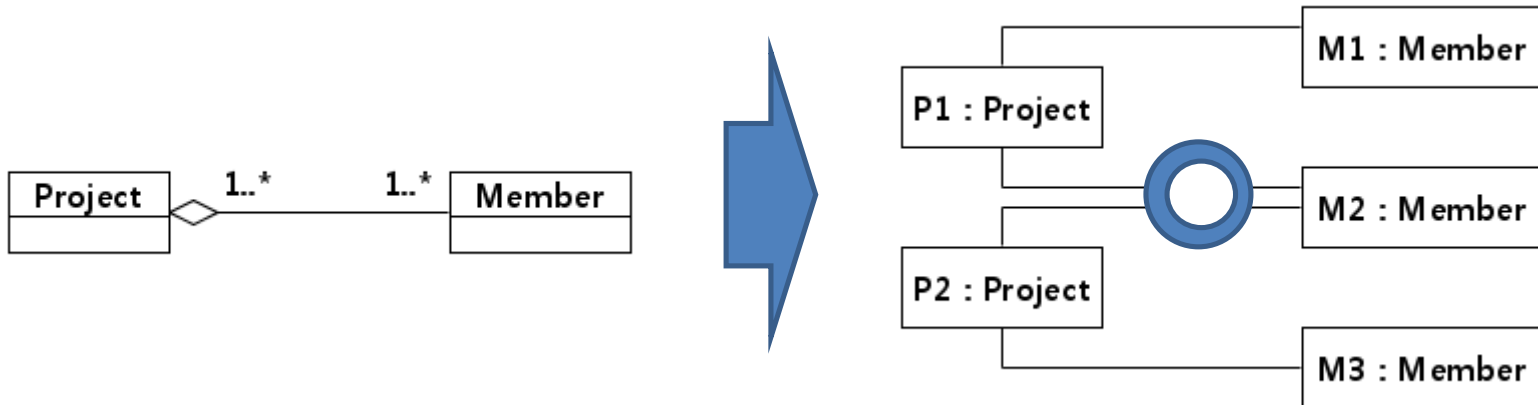
집합/포함 관계는 연관 관계의 일종이다

- ❖ 연관 관계가 가지고 있던 특성 즉 방향성, 다중성, 이름/역할 등을 집합/포함 관계에도 적용할 수가 있다
 - 방향성: 전체 객체와 부분 객체 간의 메시지 전달 방향을 표현할 수가 있다.
 - 다중성: 전체 객체와 부분 객체에 다중성을 기술함으로써 집합/포함 관계에 관련된 객체의 수를 지정할 수가 없다.
 - 이름/역할: 집합/포함 관계는 항상 "has-a" 즉 "구성된다." 또는 "포함된다."의 의미만을 가지기 때문에 이름/연관으로 관계의 의미를 추가적으로 기술할 필요가 없다

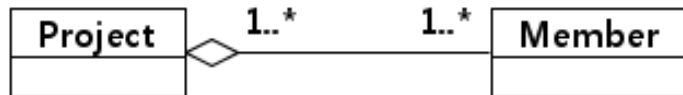


집합 관계에서는 부분 객체가 다수의 전체 객체에 의해
서 공유될 수 있다.

- ❖ 집합(aggregation) 관계는 공유 집합(shared aggregation) 관계라고도 불리며 부분 객체가 여러 개의 전체 객체들 사이에서 공유(shared)될 수 있음을 뜻한다.



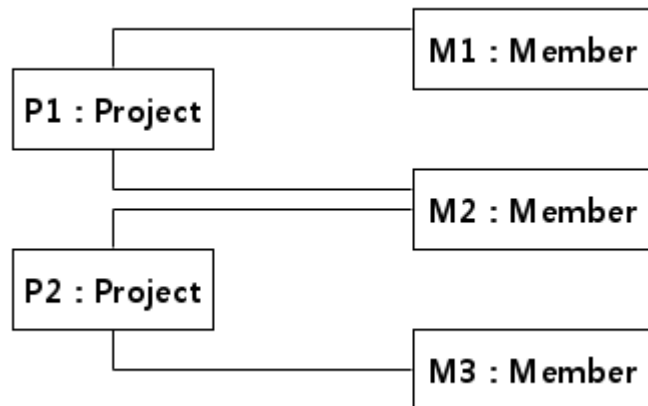
집합 관계에서는 부분 객체가 다수의 전체 객체에 의해 서 공유될 수 있다.



```
class Project {
    vector<Member*> members ;
public:
    void addMember(Member* m) {
        members->push_back(m) ;
        m->addProject(this) ;
    }
    void removeMember(Member* m) {
        members->remove(m) ;
        m->removeProject(this) ;
    }
};
```

```
class Member {
    vector<Project*> projects ;
public:
    void addProject(Project* p) {
        projects->push_back(p) ;
    }
    void removeProject(Project* p) {
        projects->remove(p) ;
    }
};
```

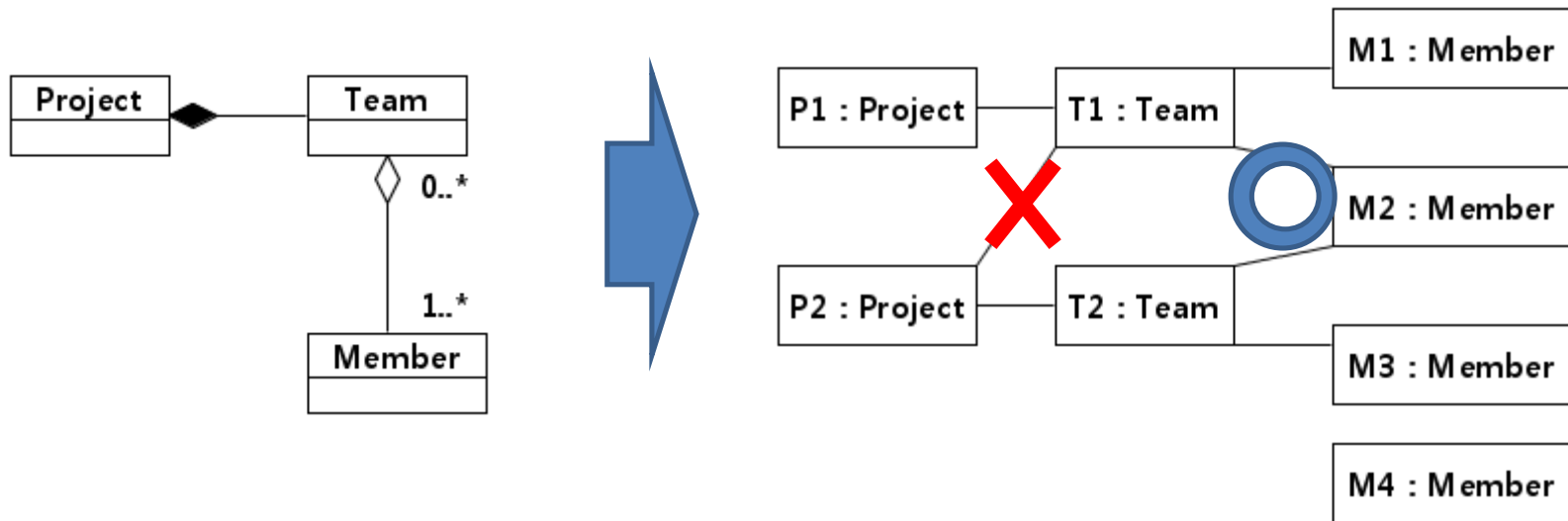
집합 관계에서는 부분 객체가 다수의 전체 객체에 의해 서 공유될 수 있다.



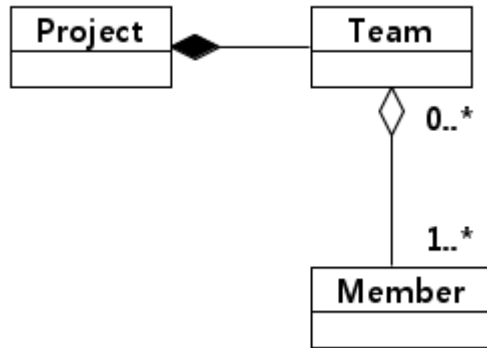
```
int main() {  
    Project p1 ;  
    Project p2 ;  
    Member m1, m2, m3 ;  
  
    p1.addMember(&m1) ;  
    p1.addMember(&m2) ;  
  
    p2.addMember(&m2) ;  
    p2.addMember(&m3) ;  
}
```


포함 관계에서는 부분 객체가 오직 하나의 전체 객체에 소속된다

- ❖ 포함(composition) 관계는 부분 객체가 오직 하나의 전체 객체에 포함될 수 있음을 뜻한다.



포함 관계에서는 부분 객체가 오직 하나의 전체 객체에 속속된다

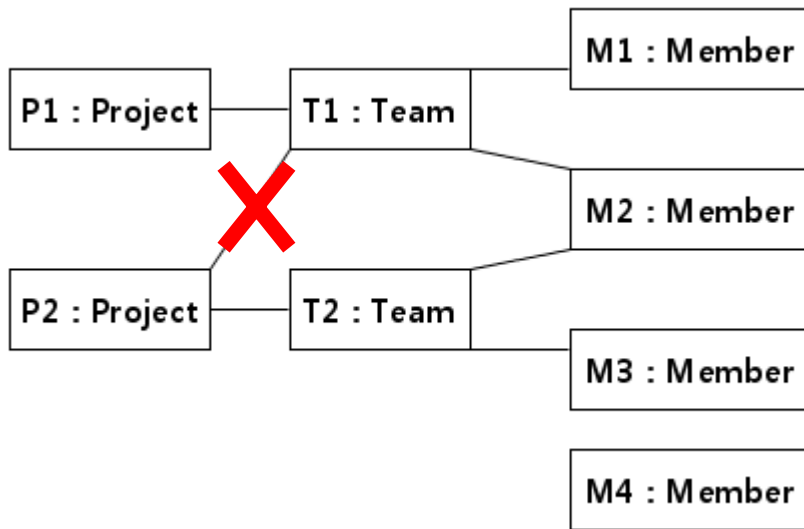


```
class Member {
    vector<Team*> teams ;
public:
    void addTeam(Team* t) {
        teams ->push_back(t) ;
    }
    void removeTeam(Team* t) {
        teams ->remove(t) ;
    }
};
```

```
class Project {
    Team theTeam ;
public:
    Project() : theTeam(*this) {}
    Team& getTeam() { return theTeam ; }
};
```

```
class Team {
    Project& theProject ;
    vector<Member*> members ;
public:
    Team(Project& project) : theProject(project) {}
    void addMember(Member* m) {
        members->push_back(m) ;
        m->addTeam(this) ;
    }
    void removeMember(Member* m) {
        members->remove(m) ;
        m->removeTeam(this) ;
    }
};
```

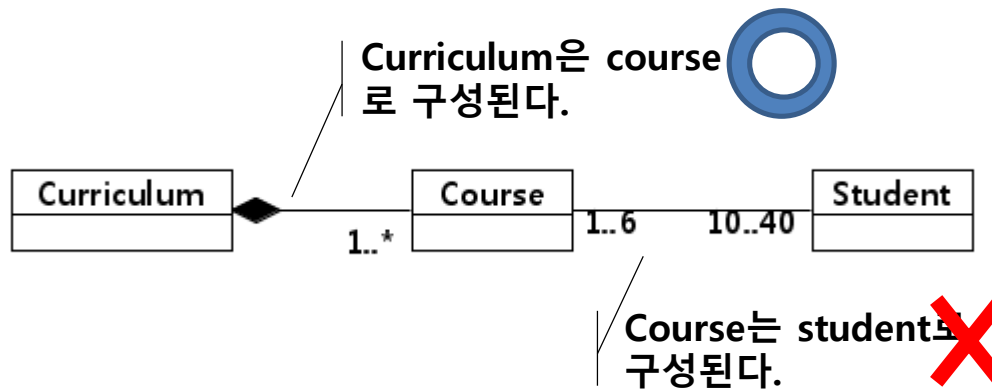
포함 관계에서는 부분 객체가 오직 하나의 전체 객체에 소속된다



```
int main() {  
    Project p1 ;  
    Project p2 ;  
  
    Team& t1 = p1.getTeam() ;  
    Team& t2 = p2.getTeam() ;  
  
    Member m1, m2, m3, m4 ;  
    t1.addMember(&m1) ;  
    t1.addMember(&m2) ;  
  
    t2.addMember(&m2) ;  
    t2.addMember(&m3) ;  
}
```

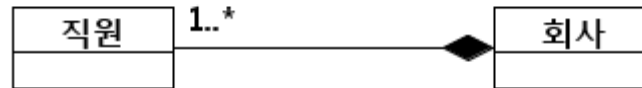
반드시 Has-a의 의미가 성립되어야 한다.

- ❖ 집합/포함 관계는 항상 “전체는 부분으로 구성된다. 또는 “부분은 전체의 부분이다.”라는 명제가 성립될 때만 사용되어야 한다.

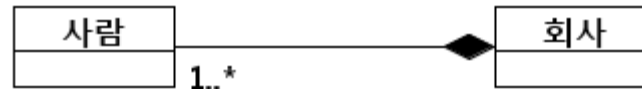


반드시 Has-a의 의미가 성립되어야 한다.

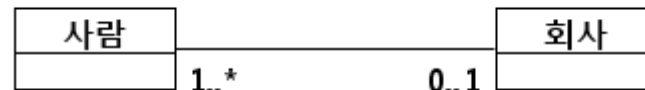
❖ 집합관계의 사용 예



❖ 집합관계의 부적절한 사용 예



❖ 연관 관계를 이용한 올바른 표현



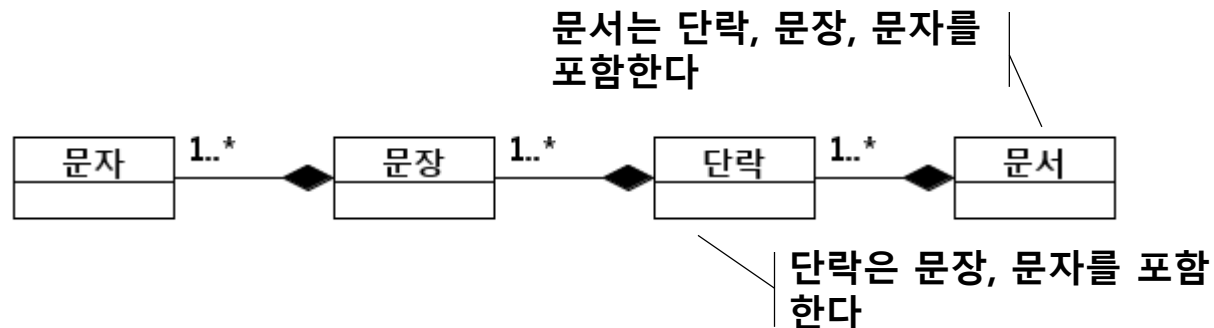
사람이 회사에 속할 수도 있고(1), 그렇지 않을 수도 있다(0).

집합/포함 관계는 비대칭적이고 이행적이다

- ❖ 집합/포함 관계는 연관 관계와 달리 비대칭적이고 이행적인 특성을 가진다.
- ❖ 비대칭적: A가 전체이고 B가 부분이면, A가 부분이고 B가 전체가 될 수는 없다

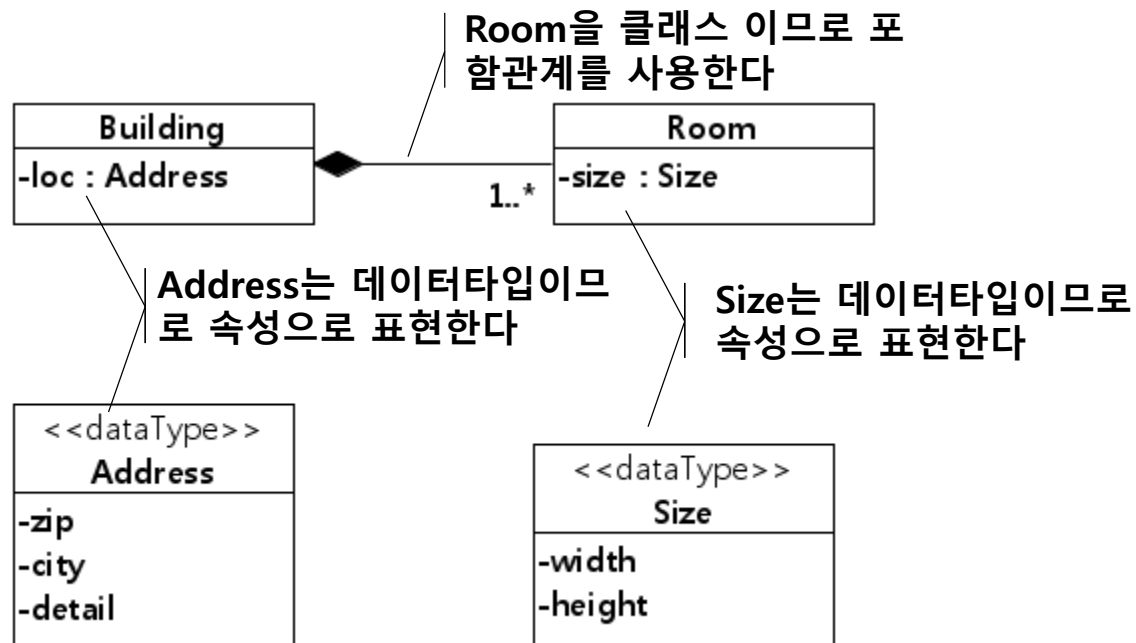


- ❖ 이행적: A가 B를 포함하고, B가 C를 포함하면, A는 C를 포함한다



포함 관계와 속성의 구분

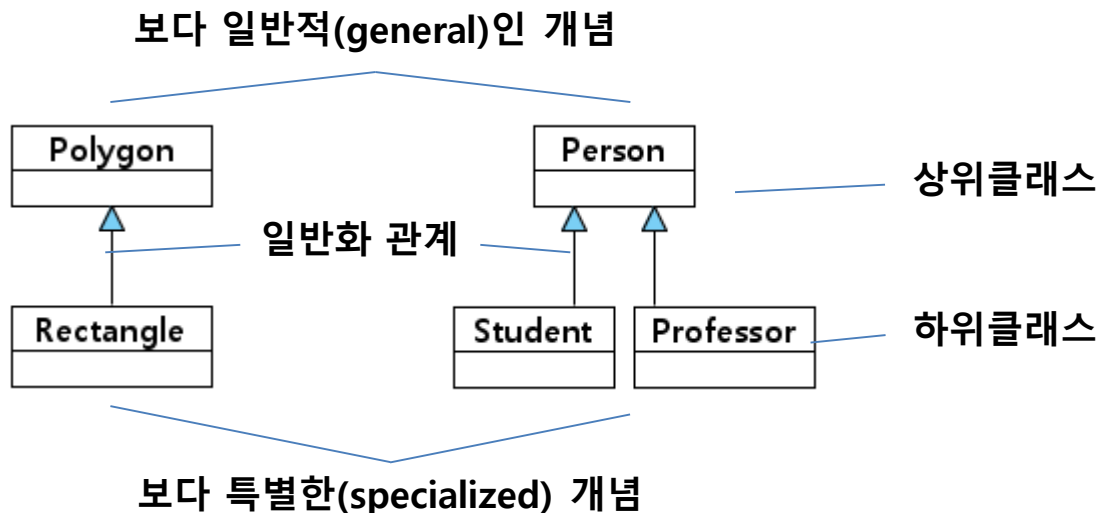
- ❖ 부분의 역할을 하는 쪽이 클래스인 경우에는 포함관계를 이용하고, 부분의 역할을 하는 쪽이 클래스가 아니라 데이터타입인 경우에는 속성으로서 정의하도록 한다



일반화 관계 - 기본 개념

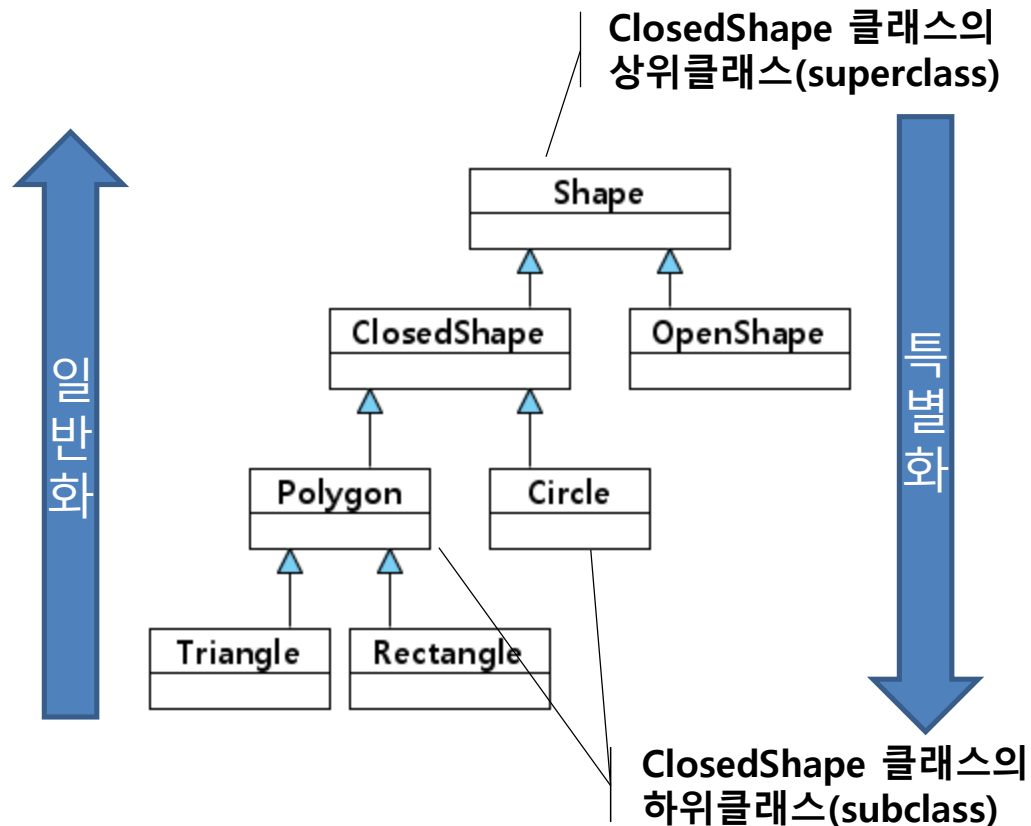
일반화 관계는 보다 일반적인 클래스와 보다 구체적인 클래스 간의 관계를 뜻한다.

- ❖ 일반화(generalization)는 한 클래스(상위 클래스)가 다른 클래스(하위 클래스) 보다 일반적인 개념/대상임을 의미하는 관계이다.



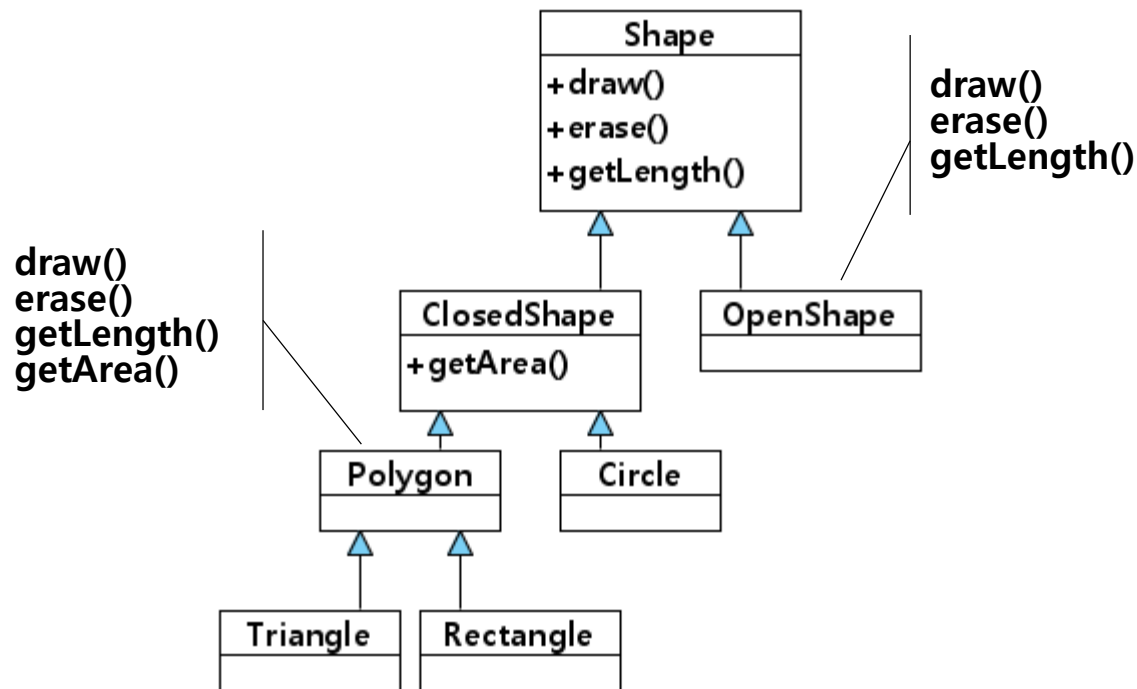
일반화 관계는 보다 일반적인 클래스와 보다 구체적인 클래스 간의 관계를 뜻한다.

- ❖ 많은 클래스 간의 일반화 관계를 전체적으로 정의한 것을 일반화 계층 구조라고 부른다.



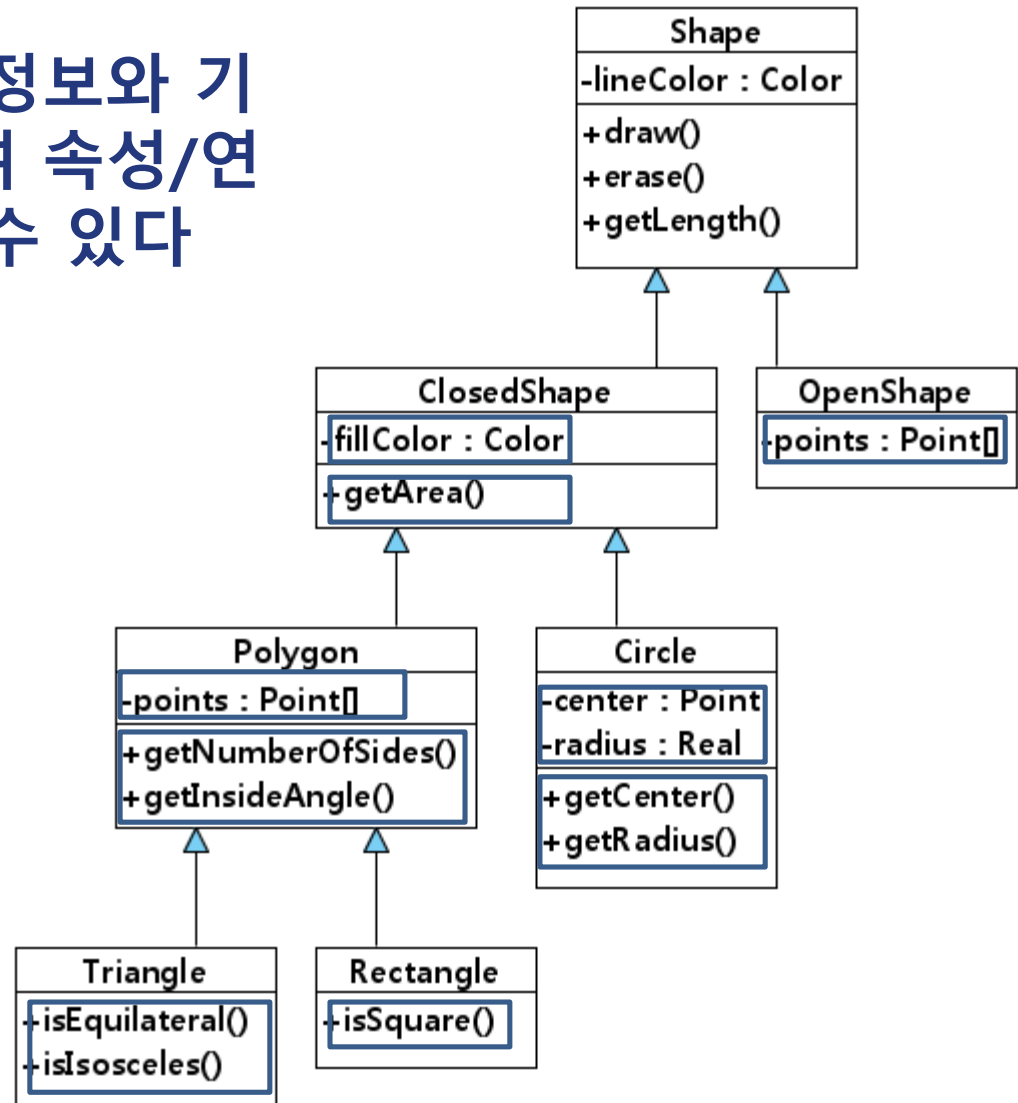
하위 클래스는 상위 클래스의 속성/연산과 관계를 물려받는다.

- ❖ 하위 클래스가 상위 클래스의 특별한 개념이라는 것은 하위 클래스가 상위 클래스의 모든 특성을 보유함을 가정한다.

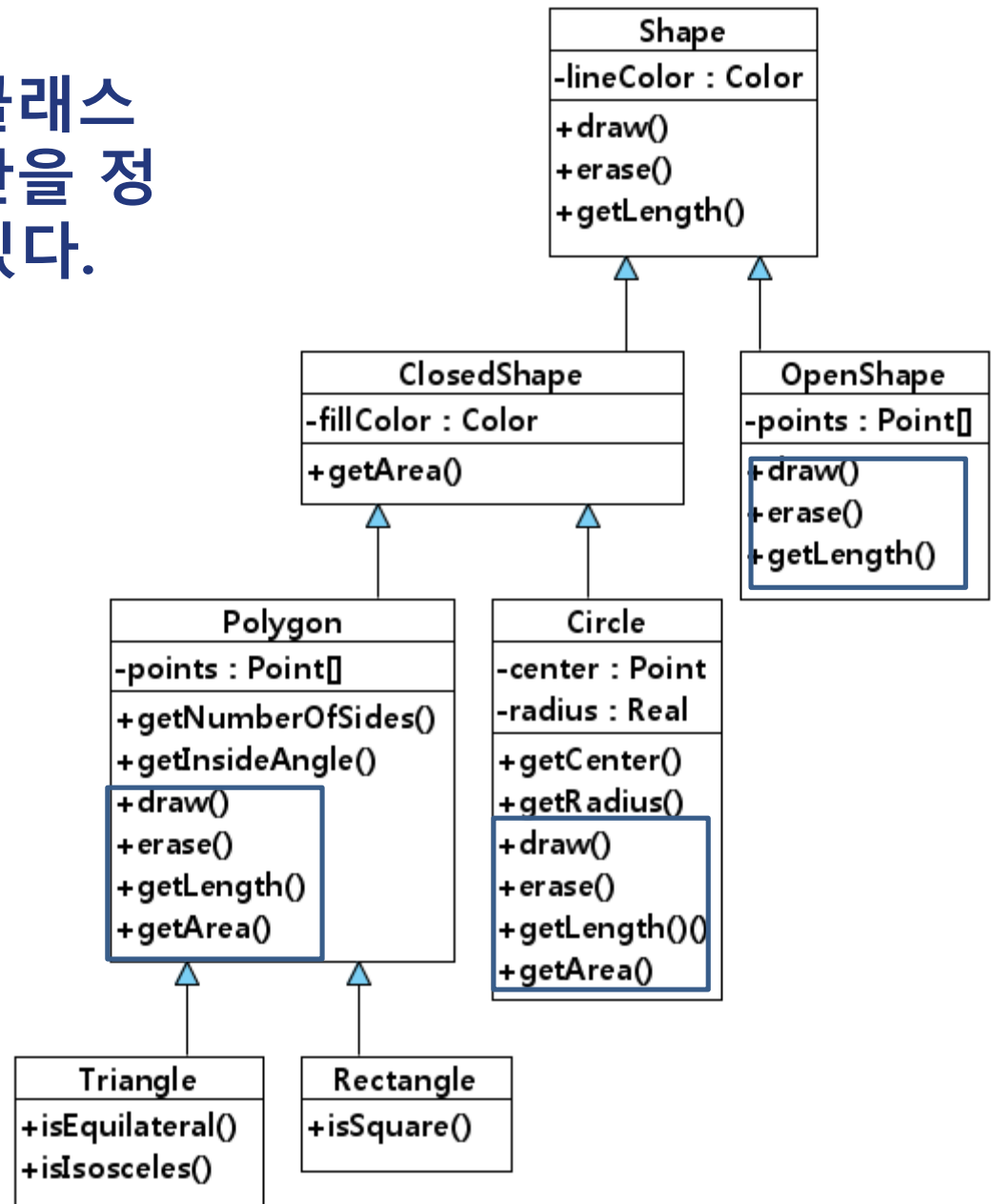


하위 클래스는 고유의 속성/연산과 관계를 추가할 수 있다

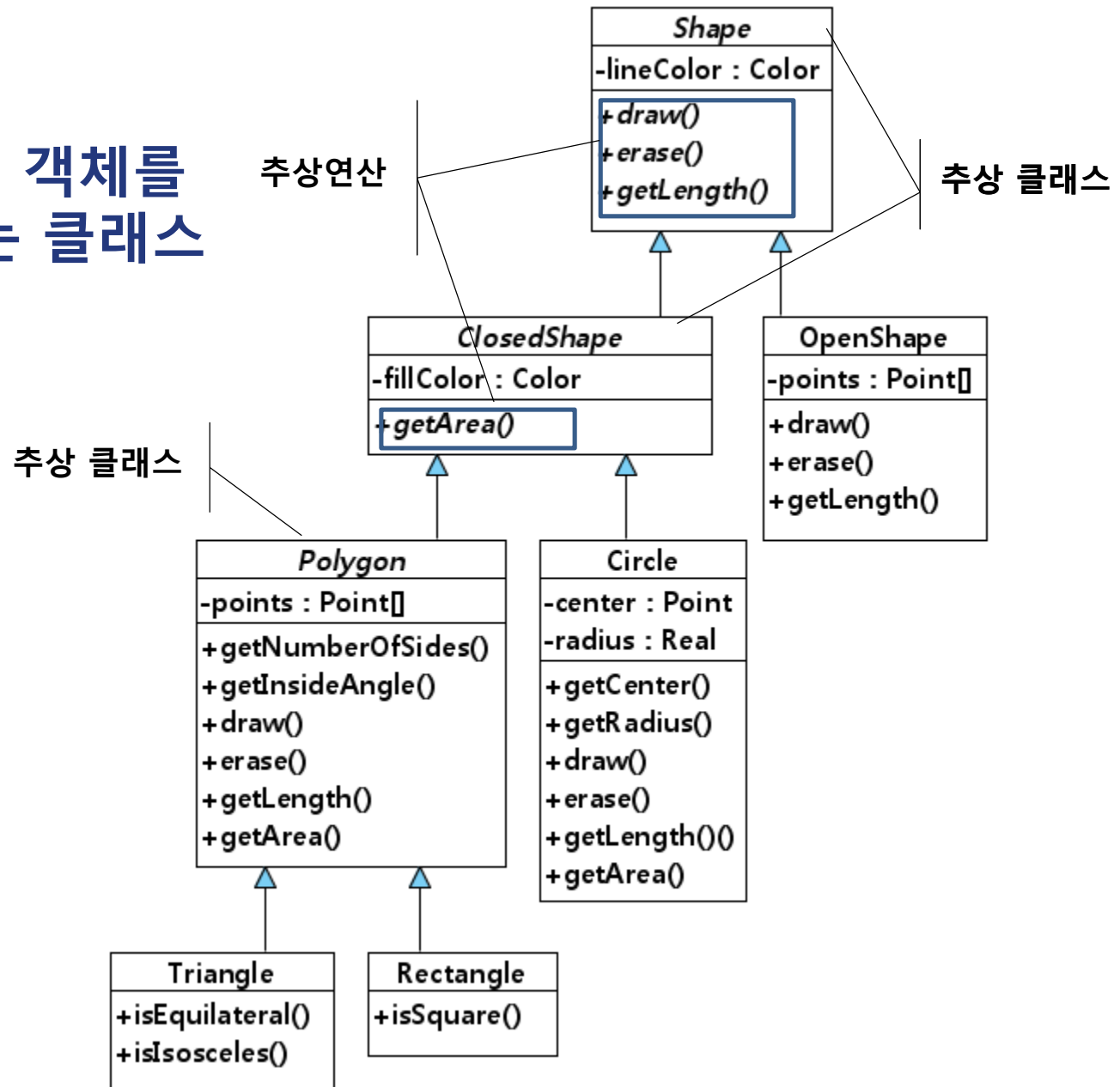
- ❖ 하위 클래스 고유의 정보와 기능을 제공하기 위하여 속성/연산과 관계를 추가할 수 있다



- ❖ 하위 클래스는 상위클래스로부터 물려받은 연산을 정의 즉 구현을 할 수 있다.

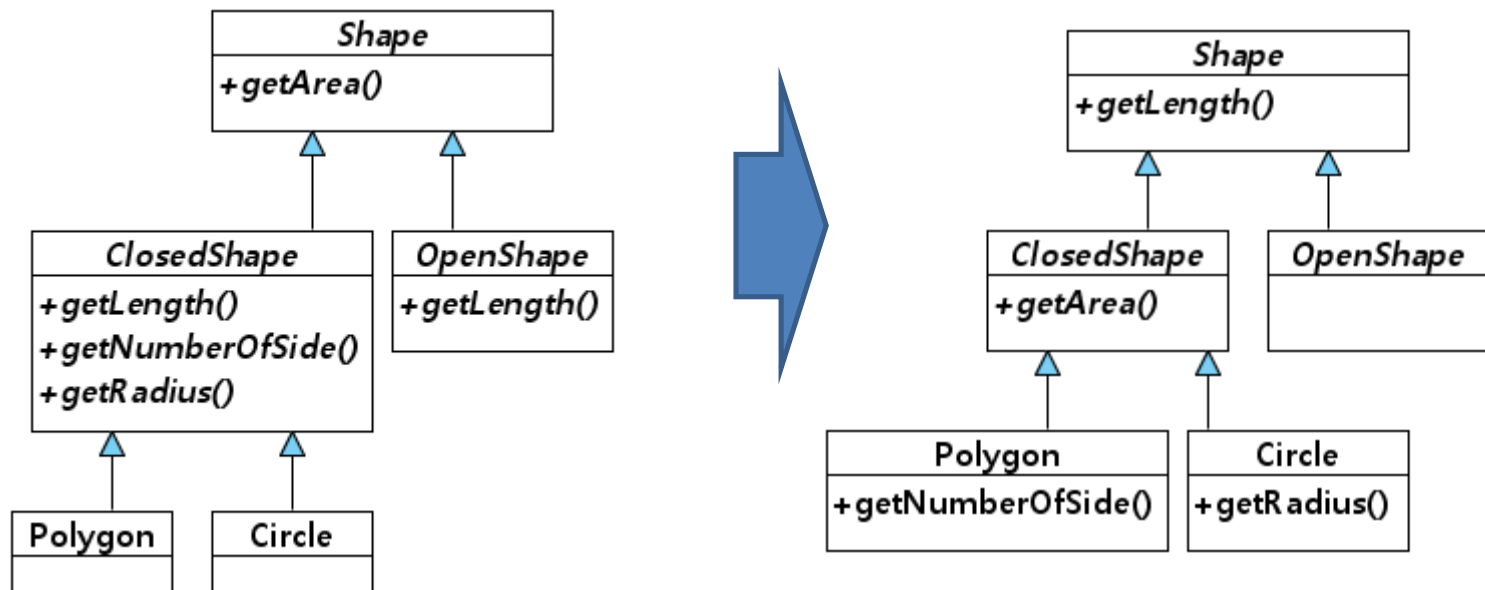


❖ 추상 클래스는 객체를 생성할 수 없는 클래스이다.



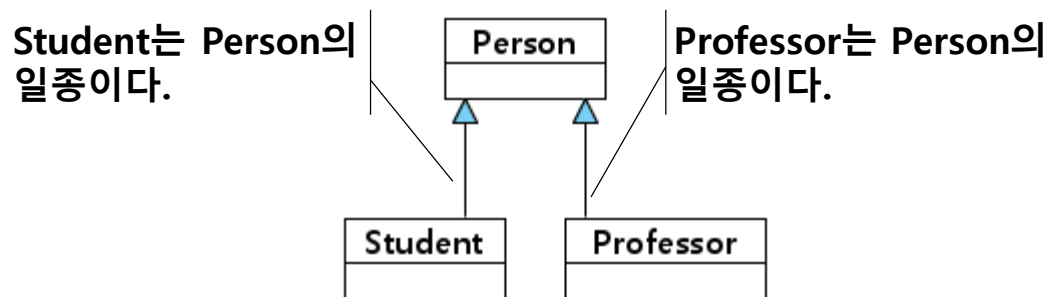
상위 클래스의 모든 멤버는 모든 하위 클래스에게 의미가 있어야 한다

- ❖ 모든 하위 클래스에게 동일하게 상위 클래스의 속성과 연산이 상속된다.
- ❖ 일부의 속성과 연산이 일부의 하위 클래스에게만 의미가 있게 적용된다면 클래스 다이어그램의 수정이 필요하다

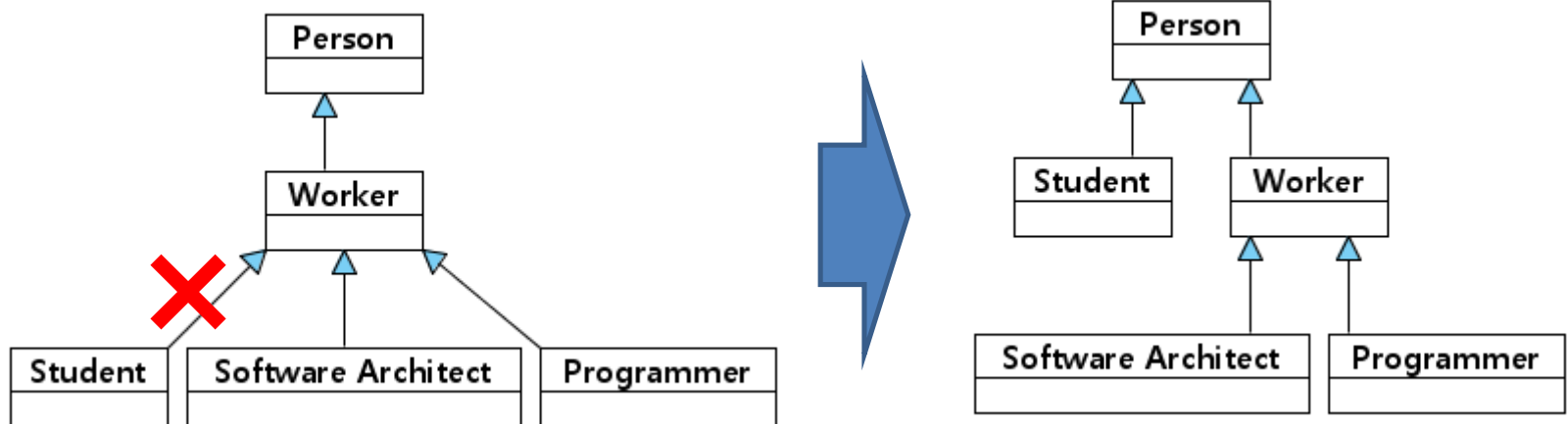


하위 클래스와 상위 클래스는 Is-a가 만족되어야 한다.

- ❖ 다형성 등의 특징을 바탕으로 시스템의 확장성을 제공하기 위해서는 상속 관계는 반드시 일반화 관계를 준수해야 한다.



하위 클래스와 상위 클래스는 Is-a가 만족되어야 한다.

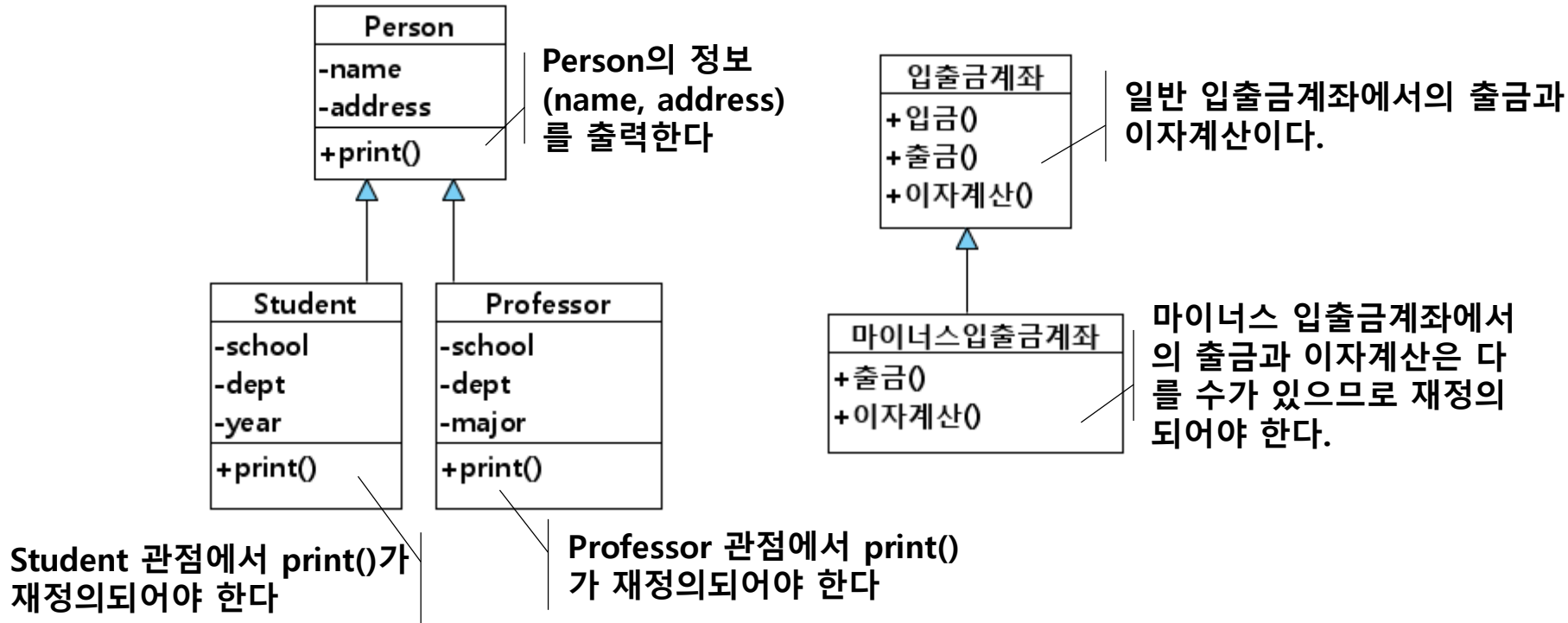


하위 클래스와 상위 클래스는 Is-a가 만족되어야 한다.

❖ Q) Stack is a subclass of List

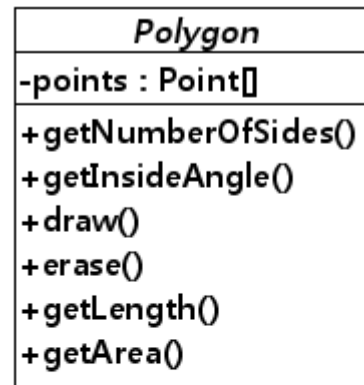
상속 받은 연산은 반드시 하위 클래스의 관점에서 정확해야 한다

- ❖ 기능적 재정의: 상위 클래스로부터 물려 받은 연산이 하위 클래스 관점에서는 기능적으로 부적절할 수가 있다.



상속 받은 연산은 반드시 하위 클래스의 관점에서 정확해야 한다

- ❖ 비기능적 재정의: 연산의 기능적인 측면에서는 동일하지만 수행 속도 또는 메모리 사용 등의 측면에서 상이한 구현이 필요한 경우에는 하위 클래스에서 재정의할 수도 있다



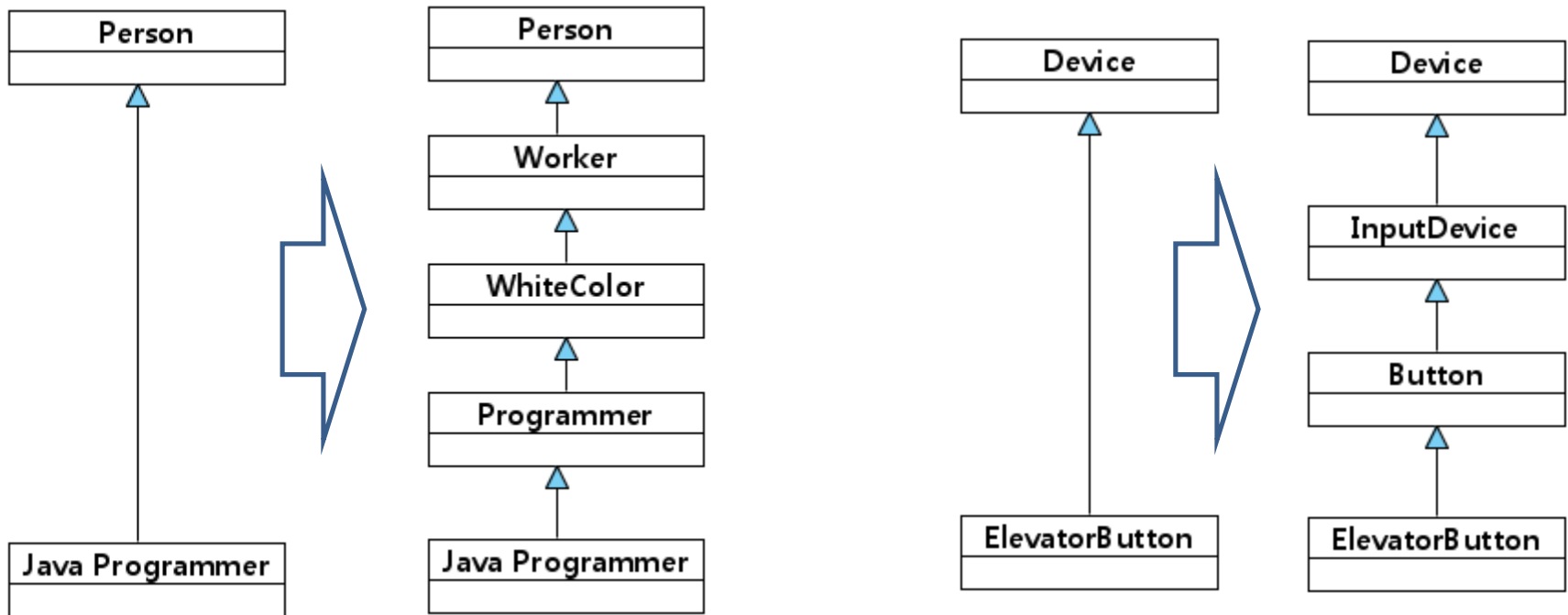
Polygon::getArea() 보다 효율적인 알고리즘을 적용한다

일반화 계층 구조는 균형적이어야 한다

- ❖ 적절한 계층 구조는 재사용성과 확장성이 높은 객체지향 시스템을 개발하는 데 큰 도움을 줄 수 있다
- ❖ 상위 클래스와 하위 클래스 간의 의미적 차이가 적절해야 한다. 상위 클래스와 하위 클래스 간에 의미적 차이가 너무 크면 그 상이에 존재하는 개념/대상에 해당되는 클래스들을 추가할 수 있다.
- ❖ 형제 클래스들은 동등한 수준의 개념을 의미해야 한다. 형제 클래스들은 개념적으로 동등한 수준이어야 한다. 즉 비교가 가능해야 한다.

일반화 계층 구조는 균형적이어야 한다

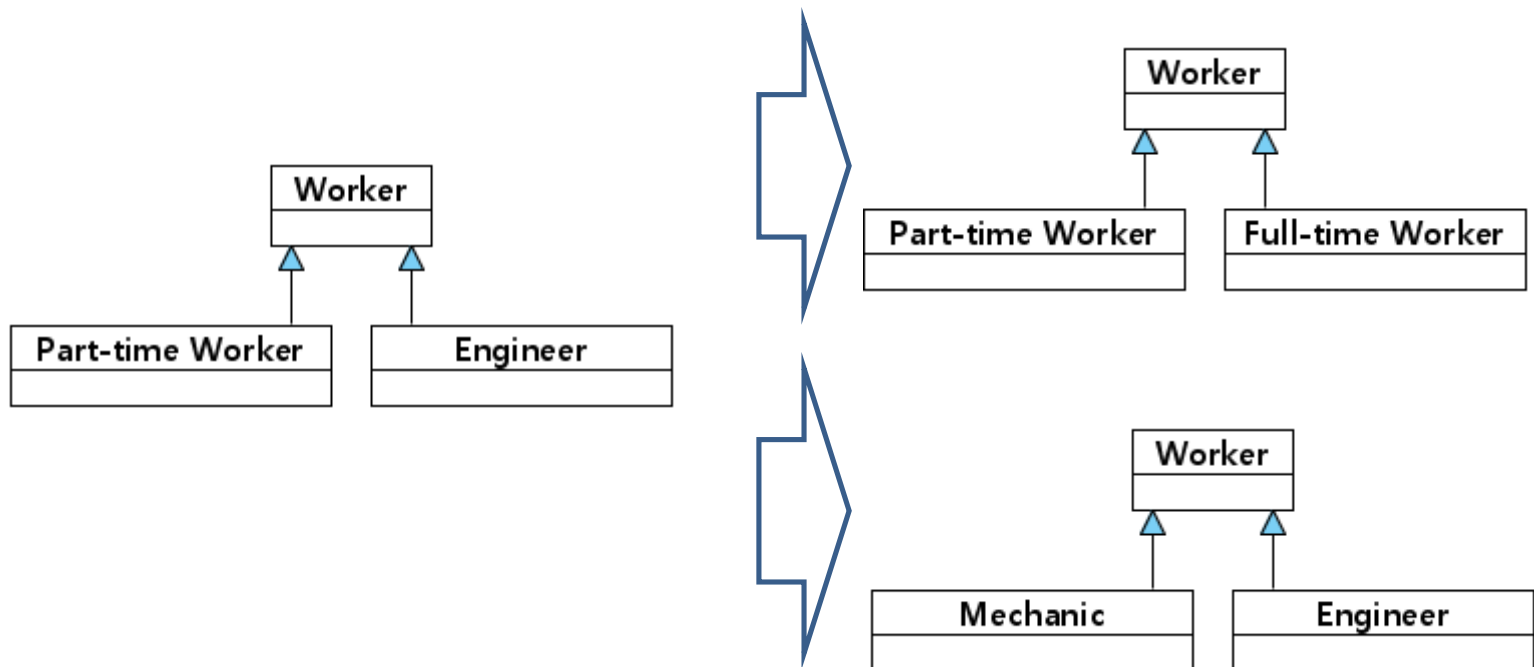
- ❖ 상위 클래스와 하위 클래스 간의 의미적 차이가 적절해야 한다.



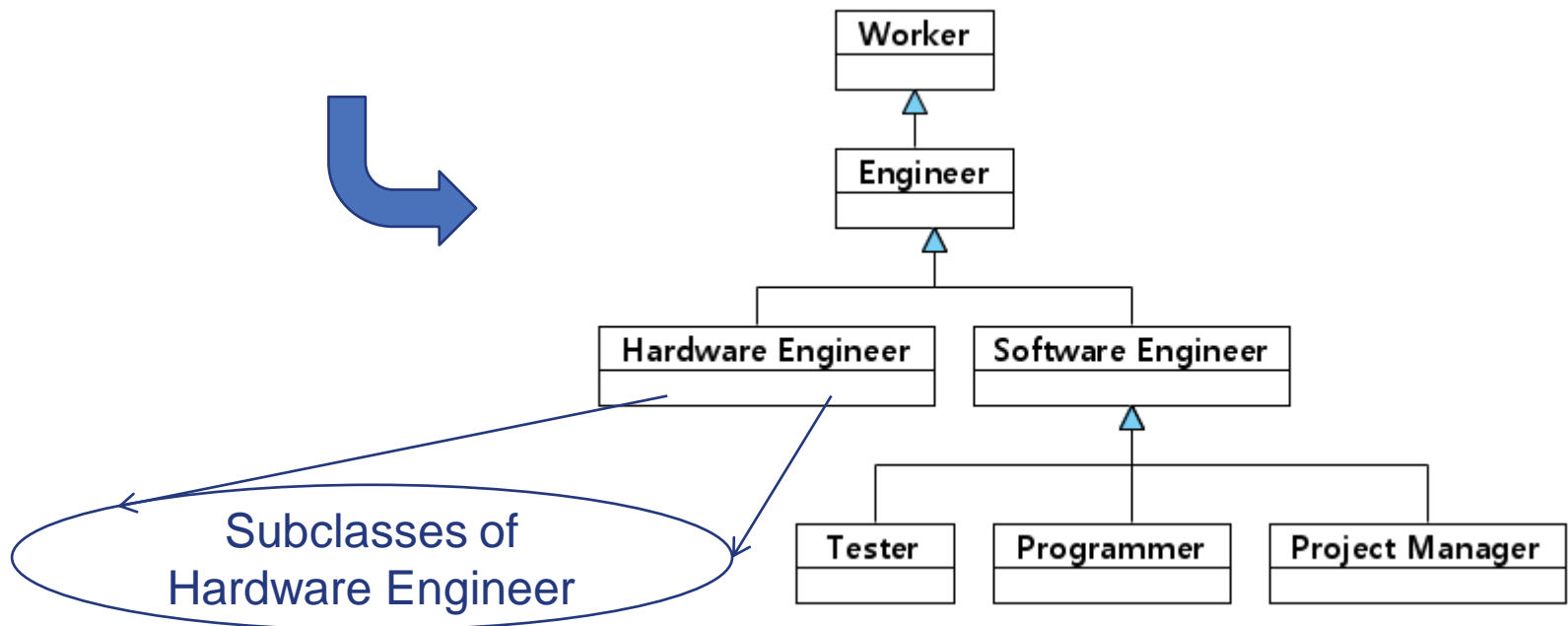
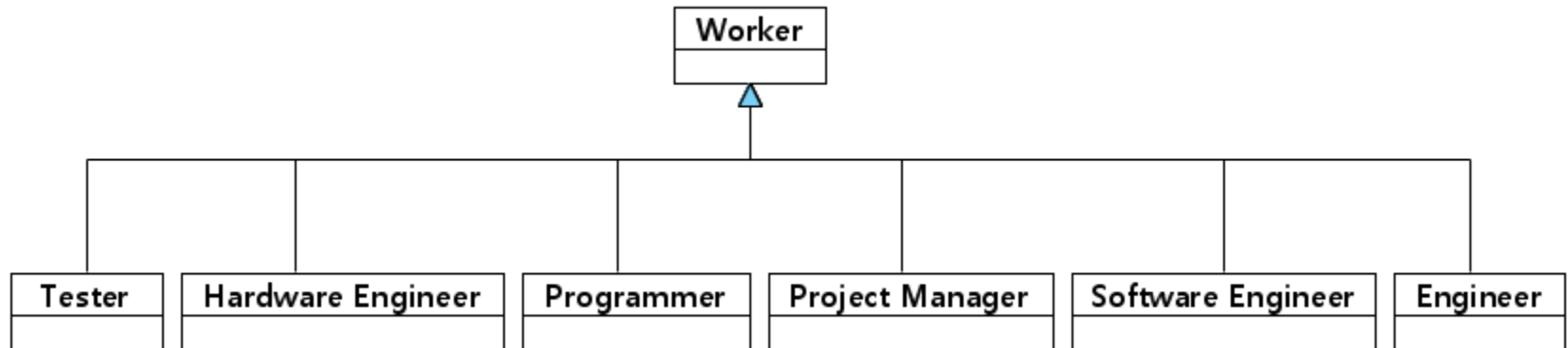
- ❖ DIT(Depth of Inheritance Tree)

일반화 계층 구조는 균형적이어야 한다

- ❖ 형제 클래스들은 동등한 수준의 개념을 의미해야 한다.
- ❖ 하위 클래스들은 상위 클래스의 Partition이 되어야 한다.



일반화 계층 구조는 균형적이어야 한다



추상 클래스와 인터페이스의 차이

❖ 클래스, 추상 클래스/인터페이스의 비교

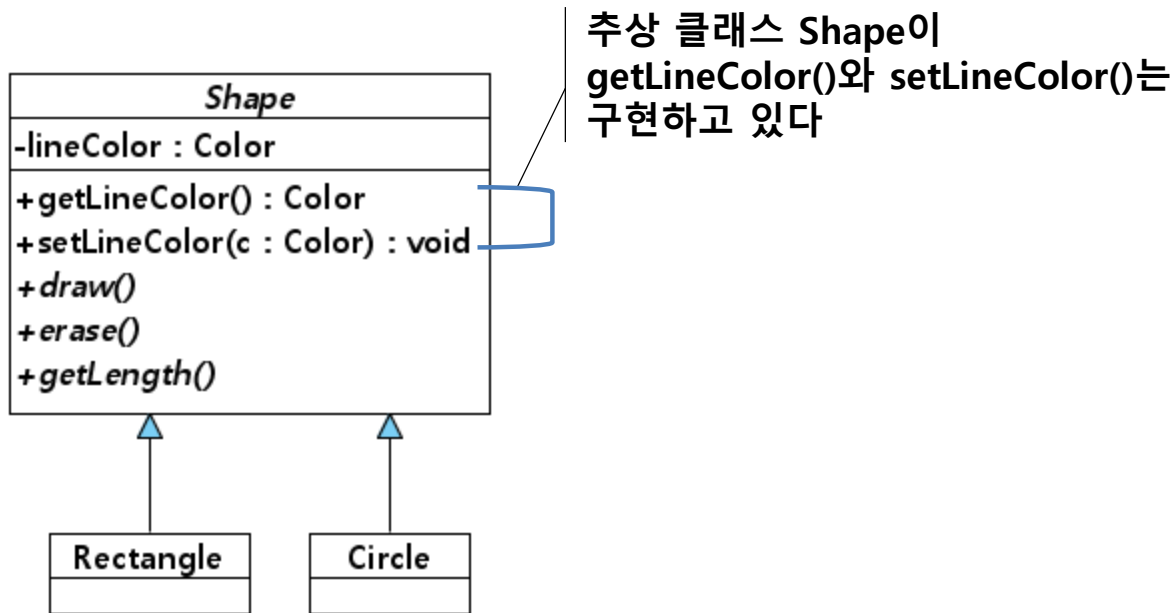
	구체 클래스	추상 클래스	인터페이스
객체의 생성	가능	불가능	
용도	기능의 구현	기능의 명세	

❖ 추상 클래스와 인터페이스의 비교

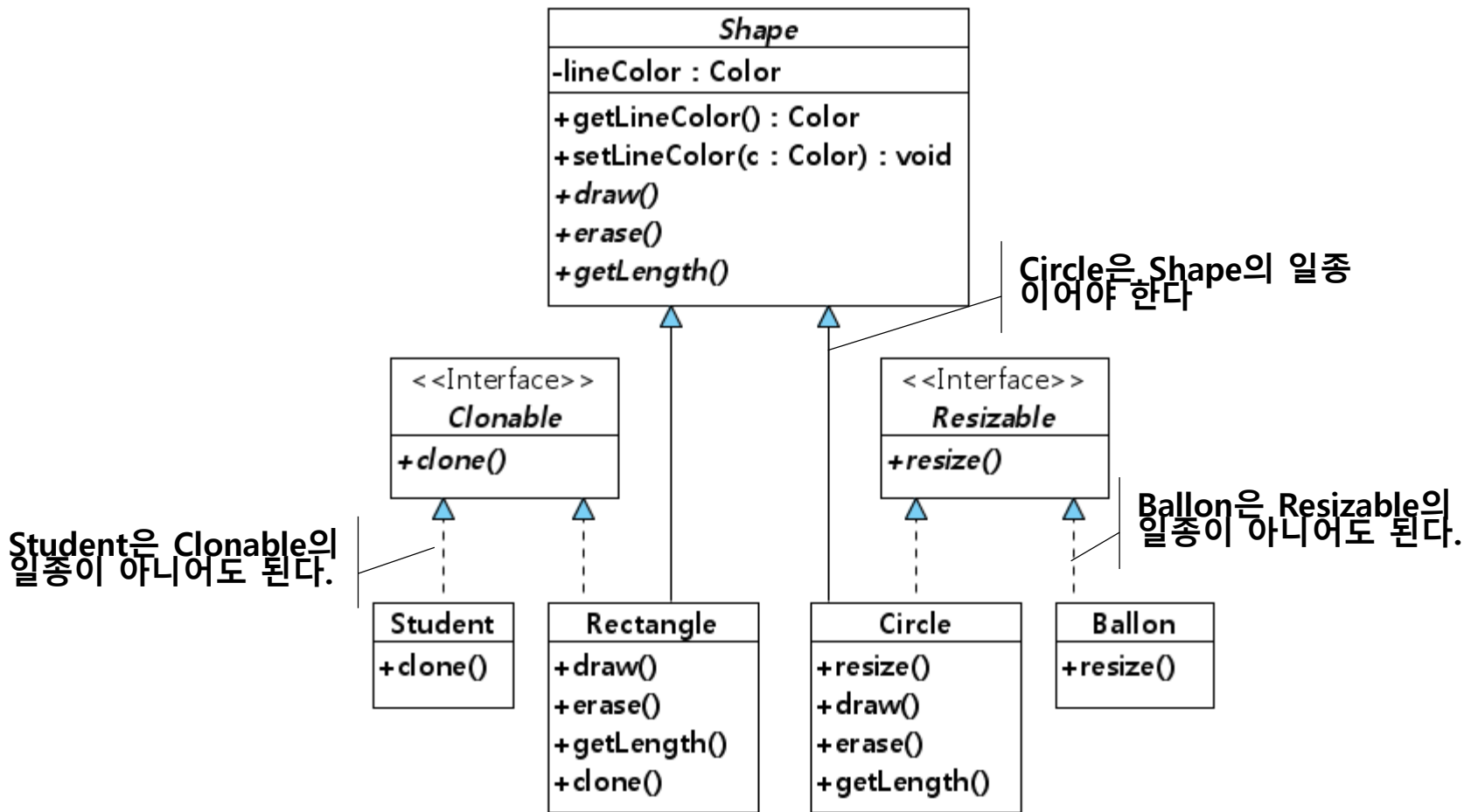
	추상 클래스	인터페이스
구현의 포함 여부	일부 가능	불가능
명세의 영향 대상	자식 클래스 (Is-A 관계 준수)	임의의 구현 클래스

추상 클래스와 인터페이스의 차이

❖ 추상 클래스는 구현을 포함할 수 있다



추상 클래스와 인터페이스의 차이



문제

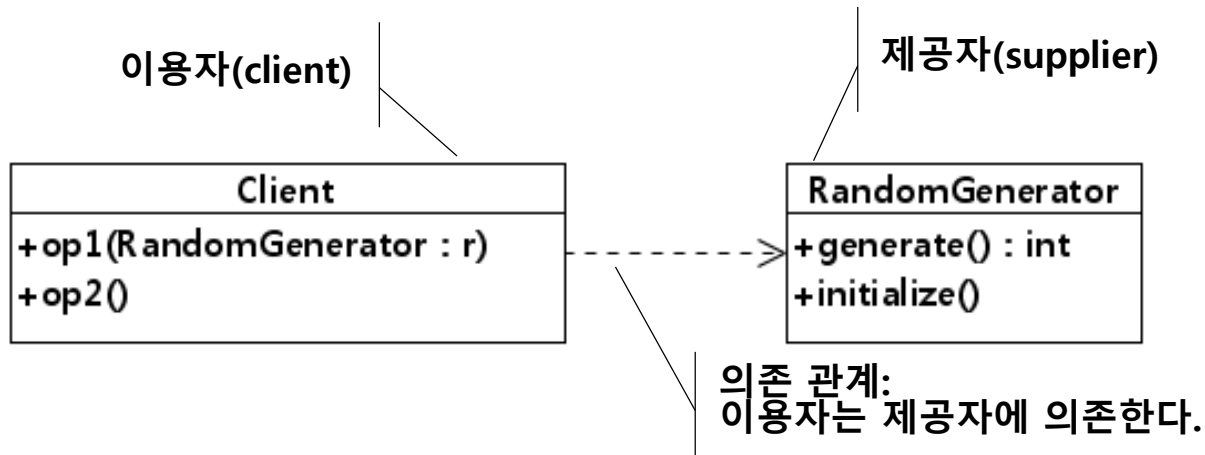
❖ 온도조절시스템

1. 시스템은 주기적으로 온도센서를 통하여 현재 온도를 구한다.
 - * 각 방에는 1개 이상의 온도센서가 있다.
 - * 방의 현재 온도는 온도 센서의 평균 값을 사용한다.
2. 시스템은 현재 온도와 설정된 희망 온도, 그리고 현재 날짜/시간 및 방의 종류(사무실, 강의실, 연구실)를 바탕으로 냉난방제어장치를 제어한다.
 - * 희망 온도는 각 방 별로 설정할 수 있다.
 - * 희망 온도의 기본값은 다음과 같다.
 - * 사무실: 25..28
 - * 강의실: 24..27
 - * 연구실: 25..28

의존 관계 - 기본 개념

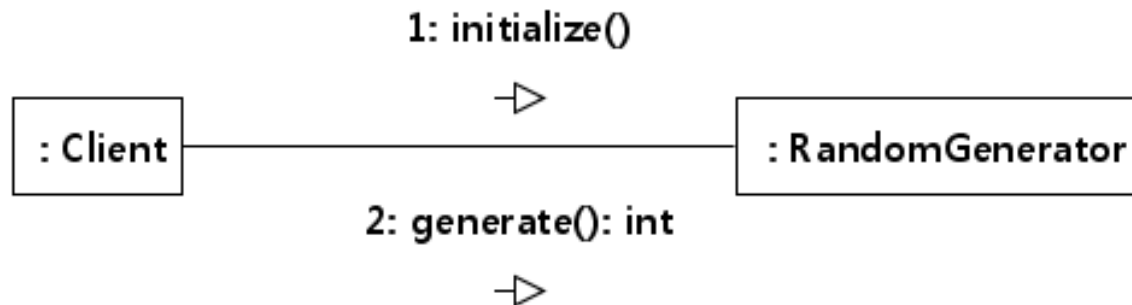
클래스 간의 의존 관계는 두 클래스의 연산 간의 호출 관계를 표현한다

- ❖ 의존 관계는 제공자의 변경이 이용자에 영향을 미칠 수 있음을 뜻한다
- ❖ 의존 관계는 제공자의 변경이 이용자의 변경을 유발함을 의미한다.



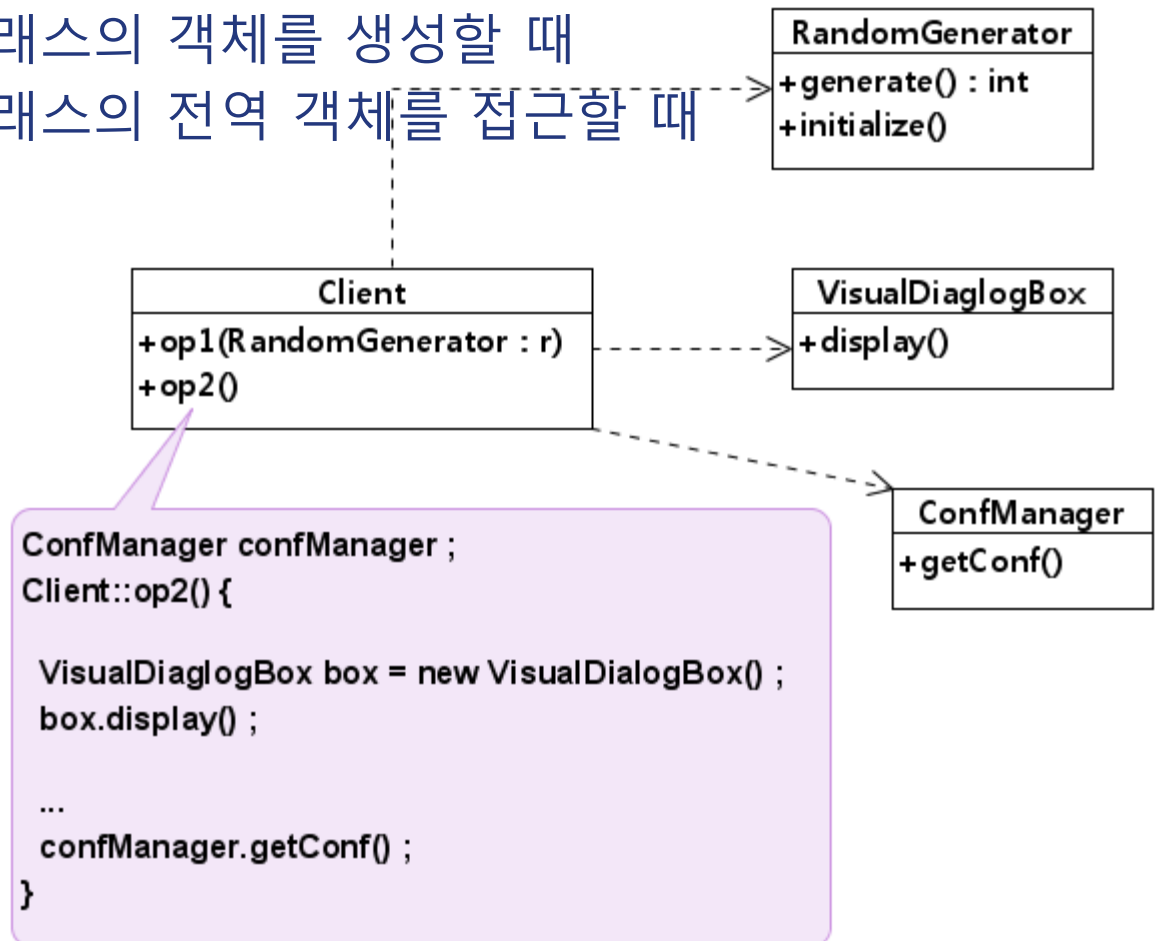
이용자는 의존 관계를 통해서 제공자의 연산을 호출할 수 있다.

- ❖ 연관 관계와 마찬가지로 의존 관계가 있는 클래스 간에 메시지가 전달될 수 있다. 구체적으로 말하면 사용자 클래스는 제공자 클래스의 연산을 호출할 수 있다.



의존 관계는 연산의 인자, 지역 객체, 전역 객체와의 관계를 표현한다.

- ❖ 연산의 인자 타입으로 제공자 클래스의 객체가 사용될 때
- ❖ 연산에서 제공자 클래스의 객체를 생성할 때
- ❖ 연산에서 제공자 클래스의 전역 객체를 접근할 때



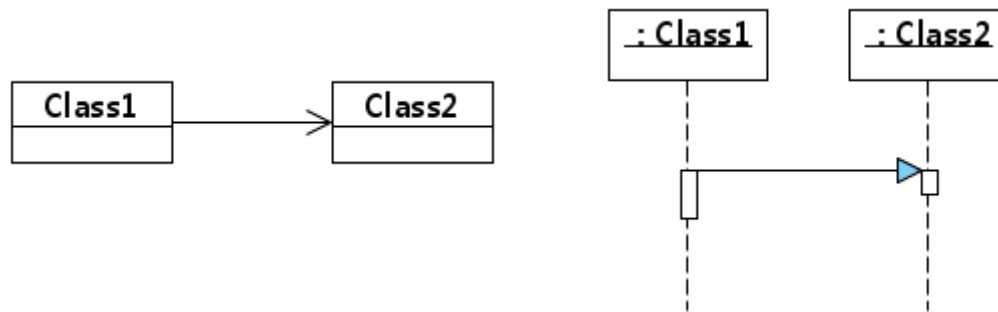
연관 관계와 의존 관계

❖ 연관 관계와 의존 관계의 비교

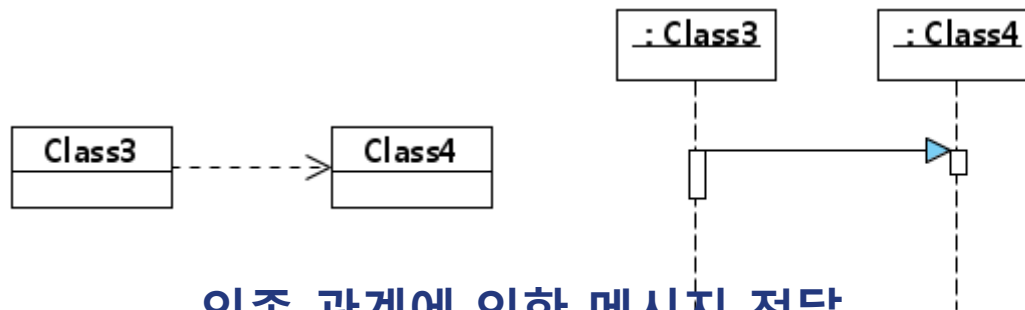
	연관 관계	의존 관계
역할	메시지 전달의 통로	
관계의 발생 형태	상대 클래스의 속성으로 대응	해당 연산의 인자 클래스 해당 연산 내부 객체의 클래스 해당 연산에서 접근하는 전역 객체의 클래스
관계의 지속 범위	해당 객체의 생명주기	해당 연산 내부
방향성	양방향 가능	단방향

역할

- ❖ 역할: 연관 관계와 의존 관계는 관련된 상대 객체에게 메시지를 전달한다는 측면에서는 동일하다.



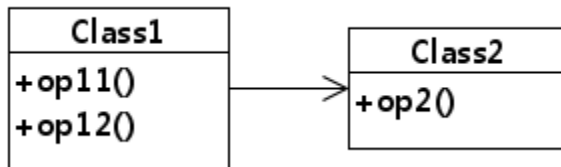
연관 관계에 의한 메시지 전달



의존 관계에 의한 메시지 전달

발생 형태

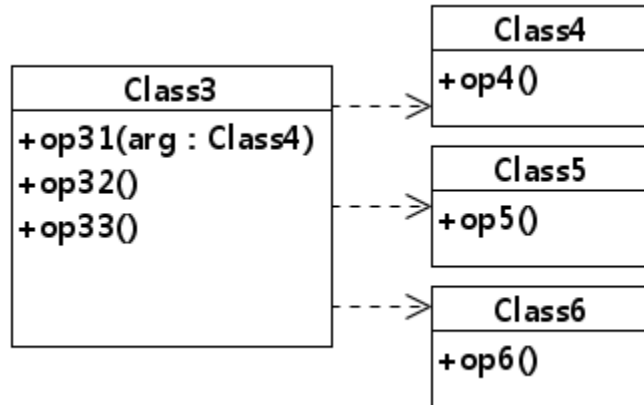
❖ 연관 관계의 발생



```
class Class1 {  
    private Class2 aClass2 ;  
    public void op11() {  
        aClass2.op20 ;  
    }  
    public void op12() {  
        aClass2.OOOOO() ;  
    }  
}
```

발생 형태

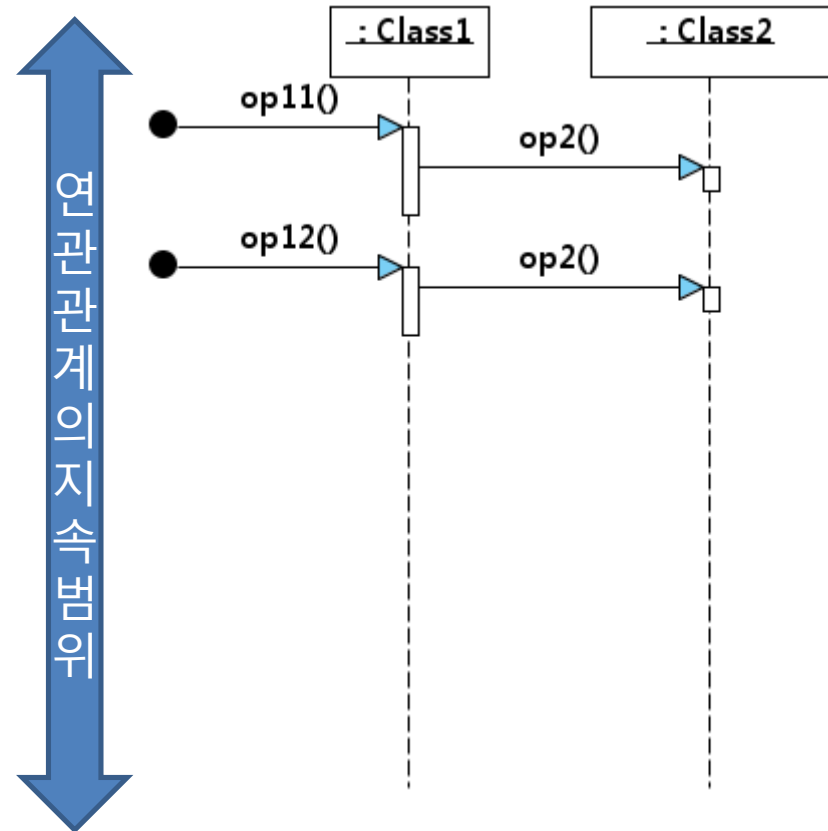
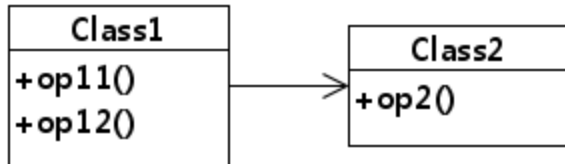
❖ 의존 관계의 발생



```
class Class3 {  
    public void op31(Class4 arg) {  
        arg.op4() ;  
    }  
    public void op32() {  
        Class5 c5 = new Class5() ;  
        c5.op5() ;  
    }  
    public void op33() {  
        Class6::op6() ;  
    }  
}
```

관계의 지속 범위

❖ 연관 관계의 지속 범위



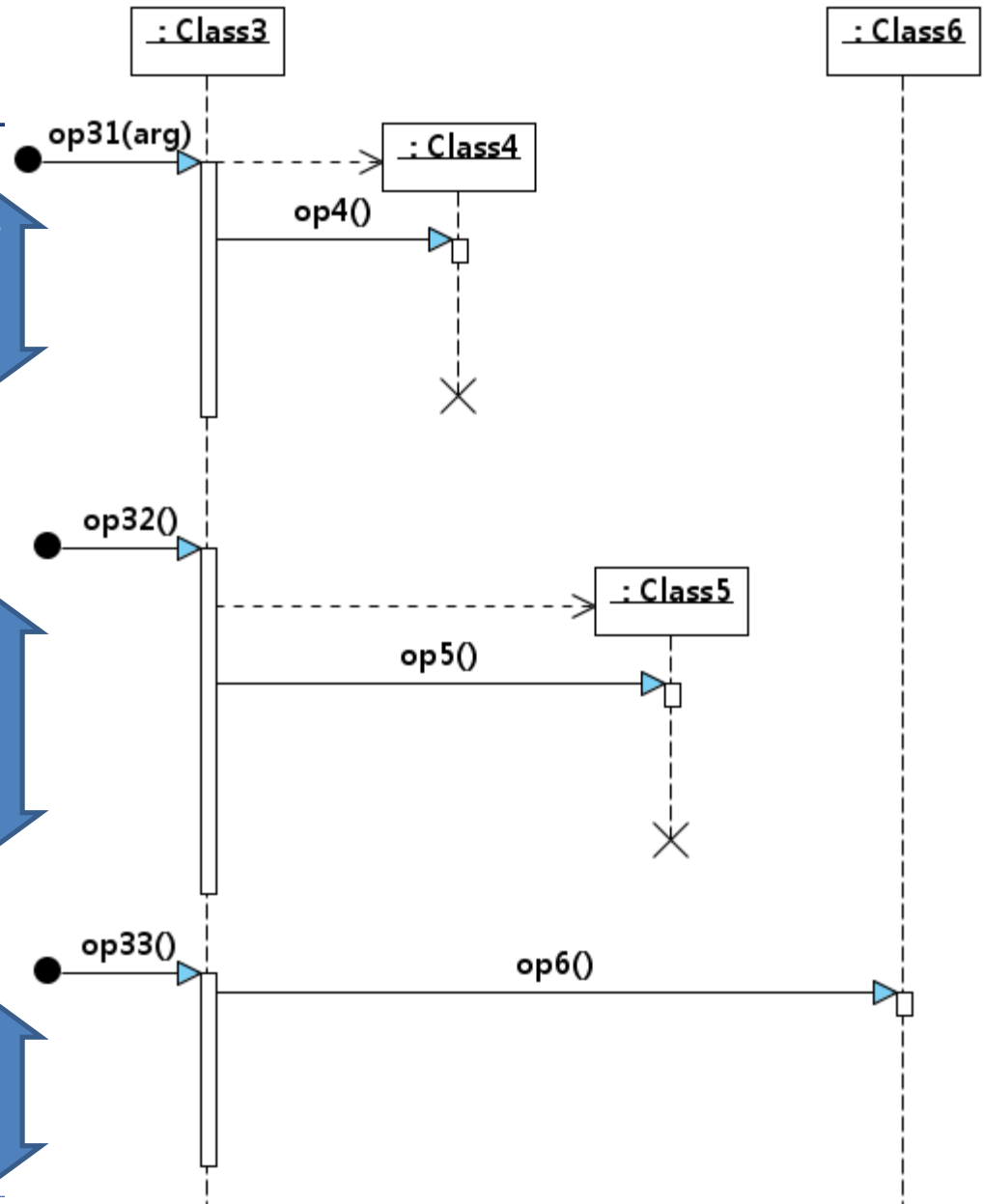
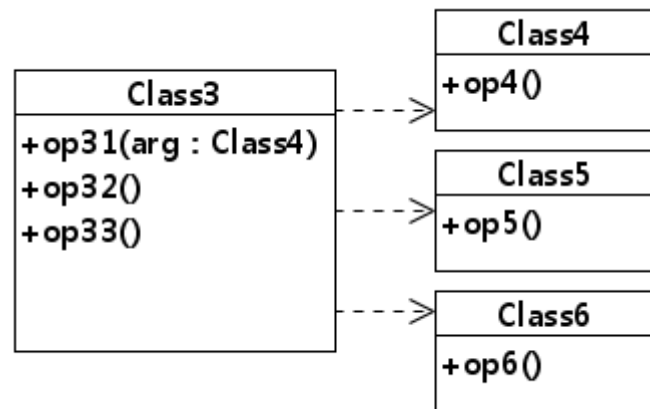
관계의 지속 범위

❖ 의존 관계의 지속 범위

Class4
와의
관계
범위

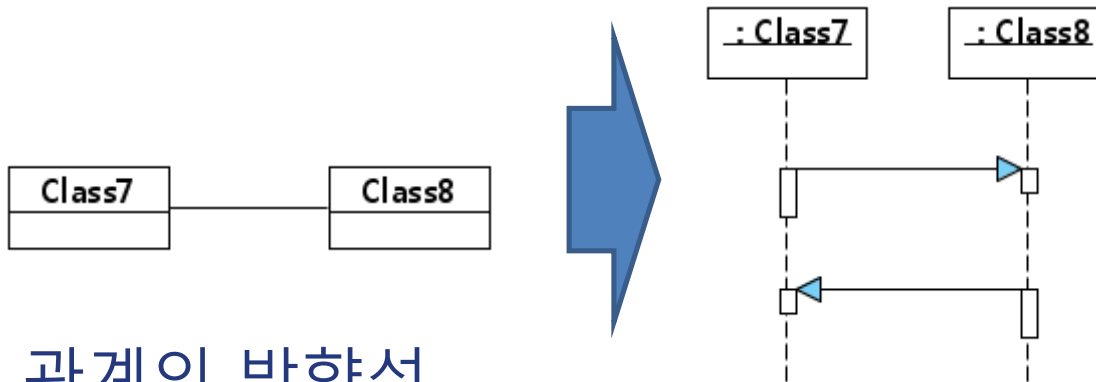
Class5
와의
관계
범위

Class6
와의
관계
범위

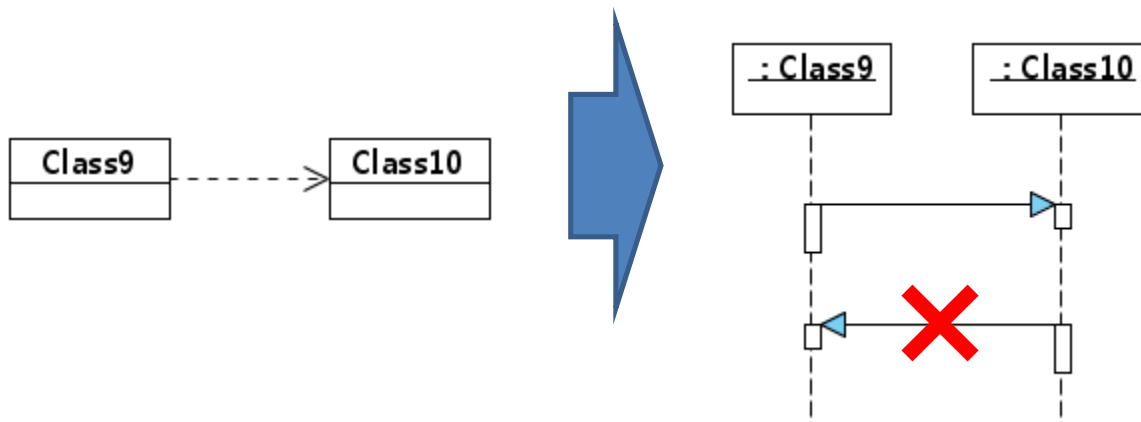


방향성

❖ 연관 관계의 방향성

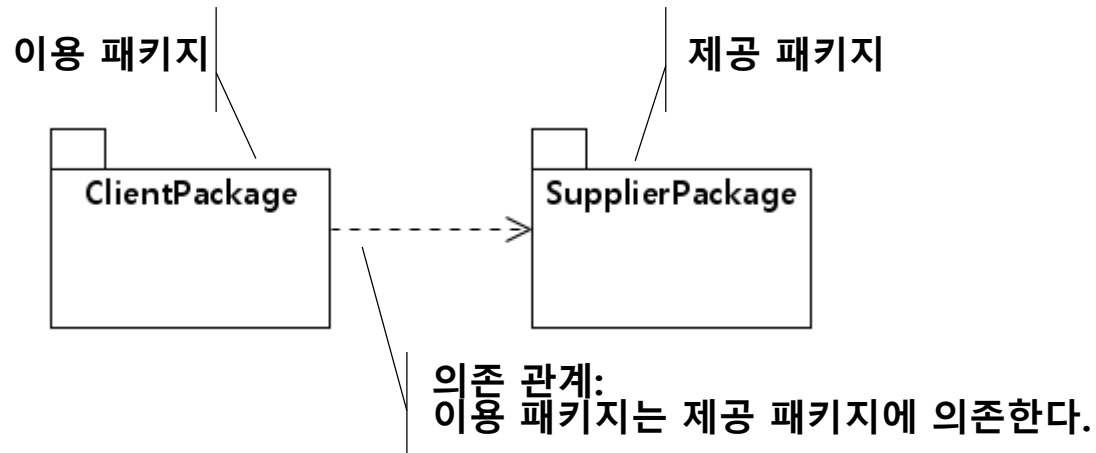


❖ 연관 관계의 방향성



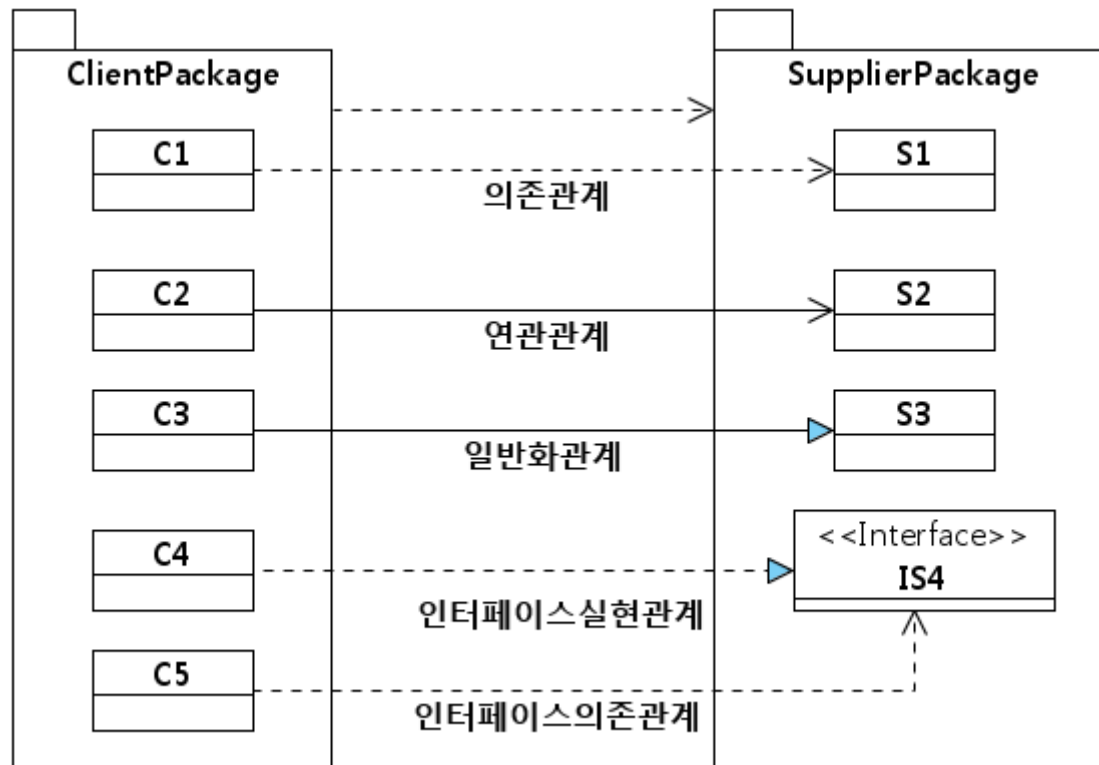
의존 관계: 패키지 다이어그램

❖ 의존 관계는 패키지 사이에 유용하게 사용될 수 있다.



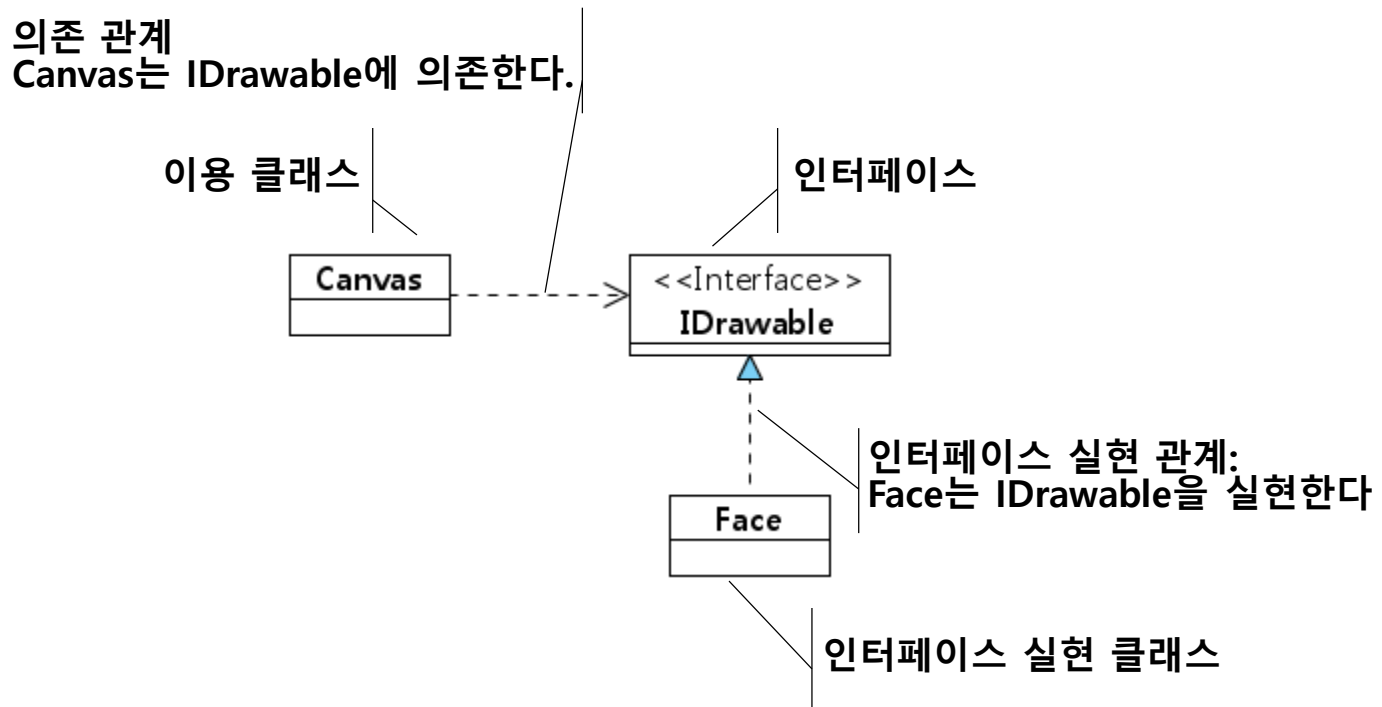
의존 관계: 패키지 다이어그램

❖ 패키지 간의 의존 관계의 의미



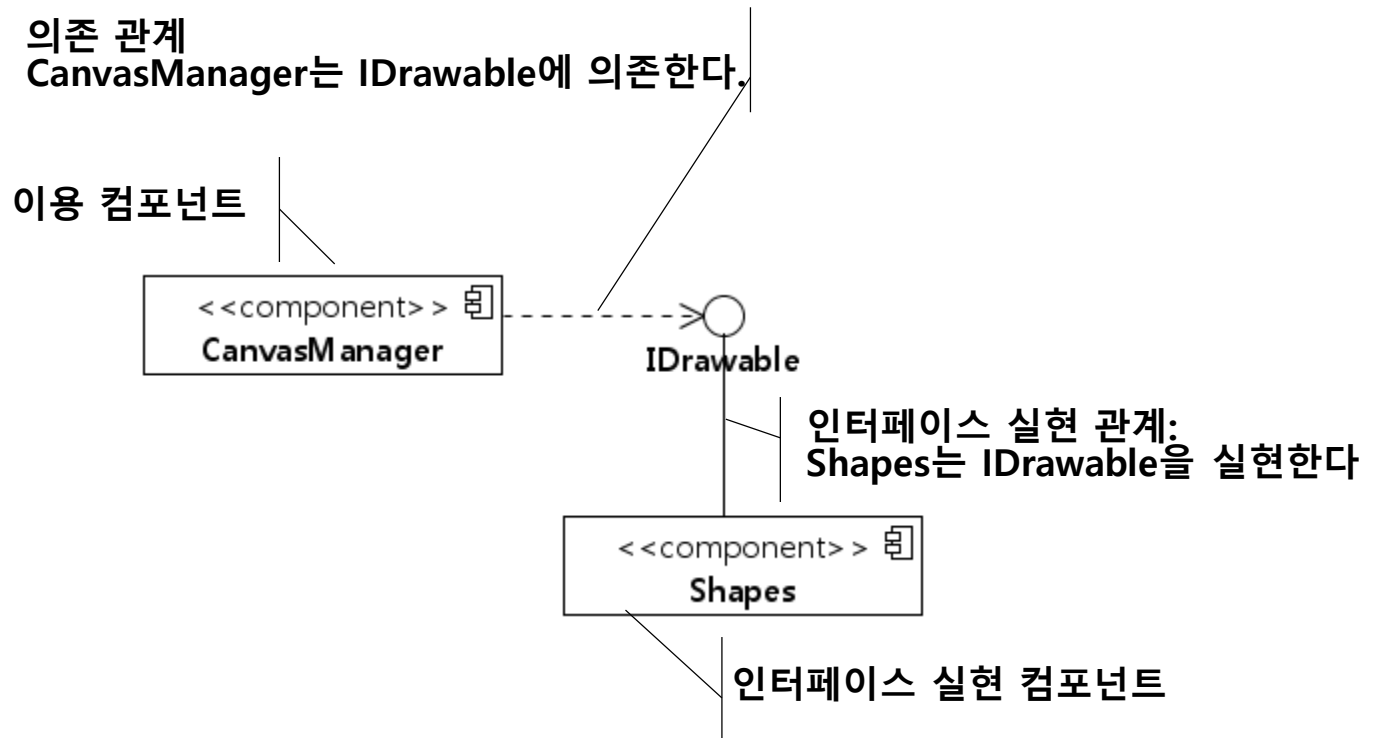
의존 관계: 클래스 다이어그램

- ❖ 인터페이스와 인터페이스 이용자 간의 이용 관계를 표현할 때 사용될 수 있다.
- ❖ 인터페이스와 의존 관계 – 클래스 다이어그램



의존 관계: 컴포넌트 다이어그램

❖ 인터페이스와 의존 관계 – 컴포넌트 다이어그램



속성 - 기본 개념

속성은 대상 객체의 상태/정보를 표현하기 위한 수단이다.

❖ 클래스의 속성은 클래스가 나타내고자 하는 객체들의 정보와 상태를 표현하는 변수들이다.

모니터 객체들의 상태/
정보를 위한 속성

모니터
-유형
-크기
-해상도
-가격
-동작여부

직원 객체들의 상태/
정보를 위한 속성

직원
-이름
-소속회사
-부서
-직급

속성은 가시성, 타입, 이름, 다중성, 초기값으로 정의된다

❖ 속성은 이름뿐만 아니라, 가시성, 타입, 다중성, 초기값이 함께 정의될 수 있다.

- 가시성 이름 : 타입 [다중성] = 초기값

사람
-이름 : String -주소 : String -전화번호 : String[2] -생년월일 : Date -나이 : UnlimitedNatural = 1 -성별 : Gender = MALE

속성은 가시성, 타입, 이름, 다중성, 초기값으로 정의된다

❖ 가시성: 클래스 외부로부터의 속성에 대한 접근의 허용 여부를 지정한다.

가시성의 유형	표현 방법	설명
공용(public)	'+'	클래스 외부에서도 접근이 된다.
보호(protected)	'#'	하위 클래스로부터의 접근이 가능하다.
전용(private)	'-'	클래스 내부에서만 접근이 가능하다.
패키지(package)	'~'	소속된 동일 패키지로부터 접근이 가능하다.

UML 기본 타입, 사용자 정의 타입, 대상 프로그래밍 언어의 타입을 타입으로서 사용할 수 있다

타입의 종류	타입
UML 기본 타입	Integer UnlimitedNatural Boolean String Real
사용자 정의 타입	<<dataType>> <<enumeration>>
대상 프로그래밍 언어의 타입	Java 언어의 int, float, String, Integer 등
	C++ 언어의 int, float, string, vector<> 등

UML 기본 타입, 사용자 정의 타입, 대상 프로그래밍 언어의 타입을 타입으로서 사용할 수 있다

- ❖ <<dataType>>을 이용한 사용자 정의 타입 Color 정의

Car
-maker : String -bodyColor : Color -price : UnlimitedNatural

<<dataType>> Color
-Rvalue : UnlimitedNatural -Gvalue : UnlimitedNatural -Bvalue : UnlimitedNatural

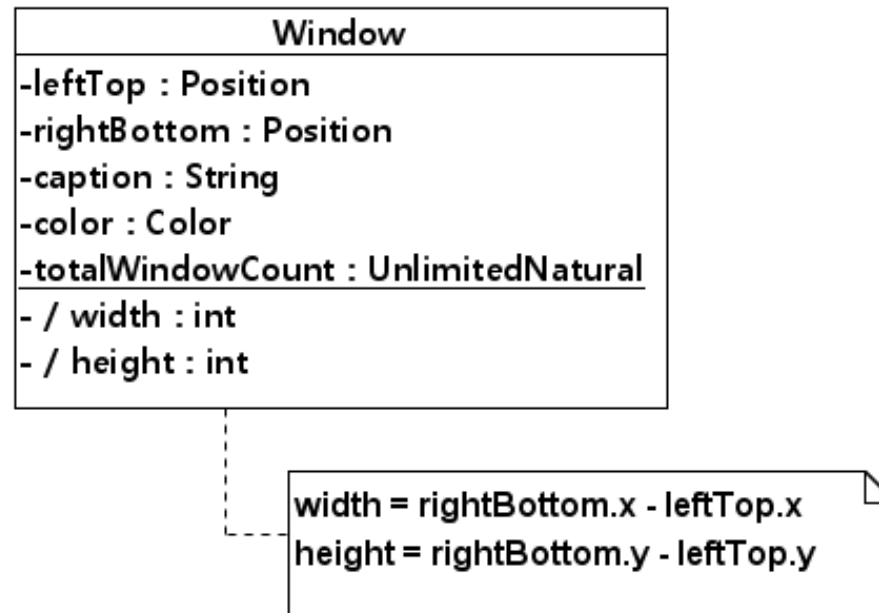
- ❖ <<enumeration>>을 이용한 사용자 정의 타입 Month 정의

<<dataType>> Date
-year : UnlimitedNatural -month : Month -day : UnlimitedNatural {day >=1 and day <= 31}

<<enumeration>> Month
JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC

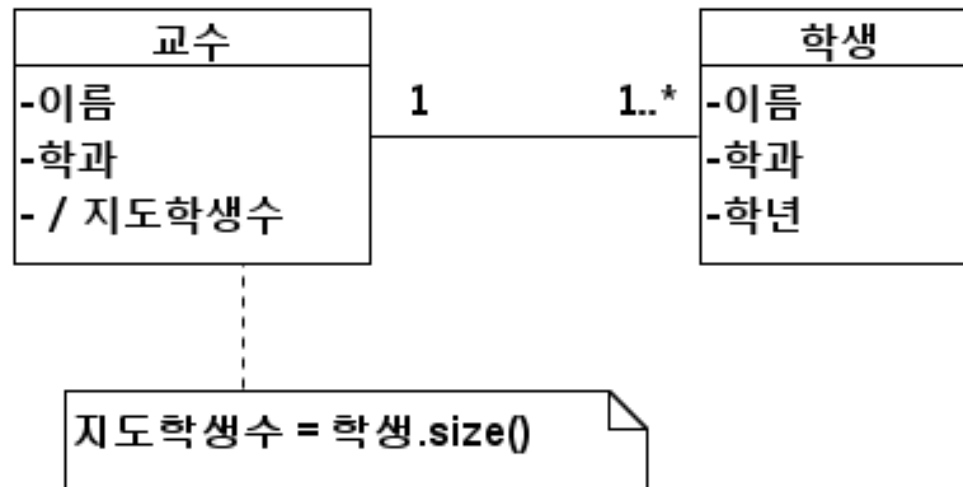
유도 속성의 사용

- ❖ 그 값이 다른 속성에 의해서 결정될 수 있는 속성을 유도(derived) 속성이라고 부른다
- ❖ 유도 속성은 클래스 다이어그램에서 가시성과 속성의 이름 사이에 "/"을 이용하여 표시한다.
- ❖ 유도 속성의 값이 어떻게 다른 속성으로부터 계산될 수 있는 지를 명시적으로 기술하는 것이 바람직하다.




유도 속성의 사용

- ❖ 유도 속성의 값은 자신의 다른 속성뿐만 아니라 다른 클래스와의 관계로부터 결정될 수도 있다.

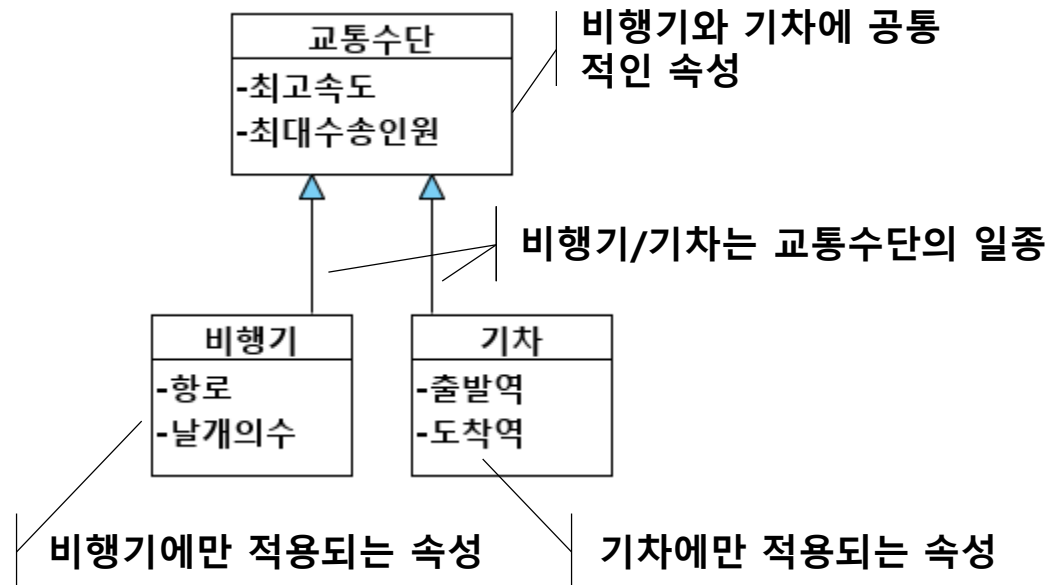


속성은 오직 하나의 구체적인 정보를 표현해야 한다.

부적절한 예	설명		적절한 예
연료	연료의 유형인지 연료의 양인지 연비 인지가 명확하지가 않다.		연료유형 현재연료의양 연비
탑승인원	현재 탑승인원인지 최대탑승가 능인원인지 평균탑승인원인지가 명확하지가 않다.		현재탑승인원 최대탑승인원 평균탑승인원
해상도	최大海상도인지 표현이 가능한 모든 해상도 인지가 명확하지가 않다.		최大海상도 표현가능해상도[]
연락처	전화번호(유선 또는 무선)인지 전자우편 주소인지 명확하지가 않다.		유선전화번호 무선전화번호 전자우편주소

속성은 해당되는 전체 객체에 공통적이어야 한다.

- ❖ 일부 객체에만 의미가 있는 속성들이 있는 경우에는 하위 클래스를 생성하여 일반화 관계를 맺도록 클래스 다이어그램을 재구성한다.



속성은 다른 클래스와 독립적인 클래스 고유의 정보를 표현해야 한다.

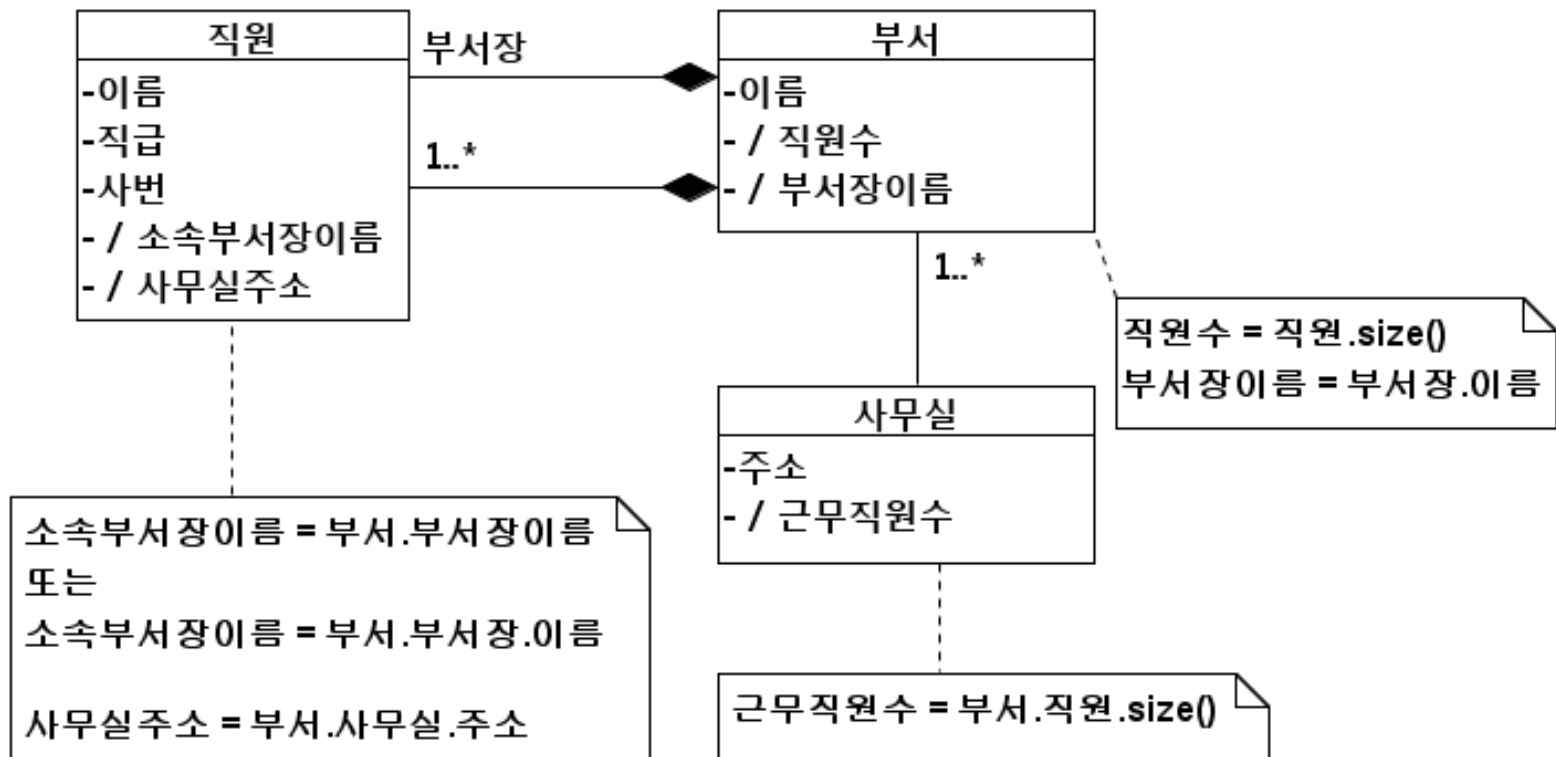
- ❖ 속성은 클래스가 나타내고자 하는 객체들의 고유의 독립적인 정보를 표현해야 하며, 다른 클래스/객체에 의존적인 정보를 속성으로 표현해서는 안 된다.

직원	
-이름	적절한 속성
-직급	
-사번	
-소속부서이름	부적절한 속성
-소속부서직원수	
-소속부서장이름	
-사무실주소	
-사무실근무직원수	

속성	설명
이름 직급 사번	직원에 의해서 결정되는 정보이므로 속성으로서 적절하다.
소속부서이름 소속부서직원수 소속부서장이름	이 값들은 직원 자체의 정보라기 보다는 직원이 소속한 부서의 정보이다.
사무실주소 사무실근무직원수	이 값들은 지원 자체의 정보라기 보다는 직원이 소속된 부서가 사용하는 사무실의 정보이다.

속성은 다른 클래스와 독립적인 클래스 고유의 정보를 표현해야 한다.

❖ 수정된 클래스 다이어그램



속성은 값 자체가 중요하며 객체로서 독립적으로 존재하는 것은 아니다.

❖ 클래스/객체는

- 독립적으로 존재하며 실체로서 각 객체의 식별성이 중요하다.
- 즉 동일한 클래스로부터 생성된 객체라 하더라도 각 객체를 구분할 필요가 있다.

사서
-이름 : Name
-연락처 : Contact

도서
-이름 : String
-식별자 : ISBN

엘리베이터
-위치 : Position = 1

문
-색상 : Color

❖ 속성

- 객체를 설명(description)하기 위한 정보로서 값 자체가 의미가 있다.
- 즉 동일한 값을 가지는 속성이 따로 구분될 필요가 없다.

<<dataType>> Name

<<dataType>> Contact

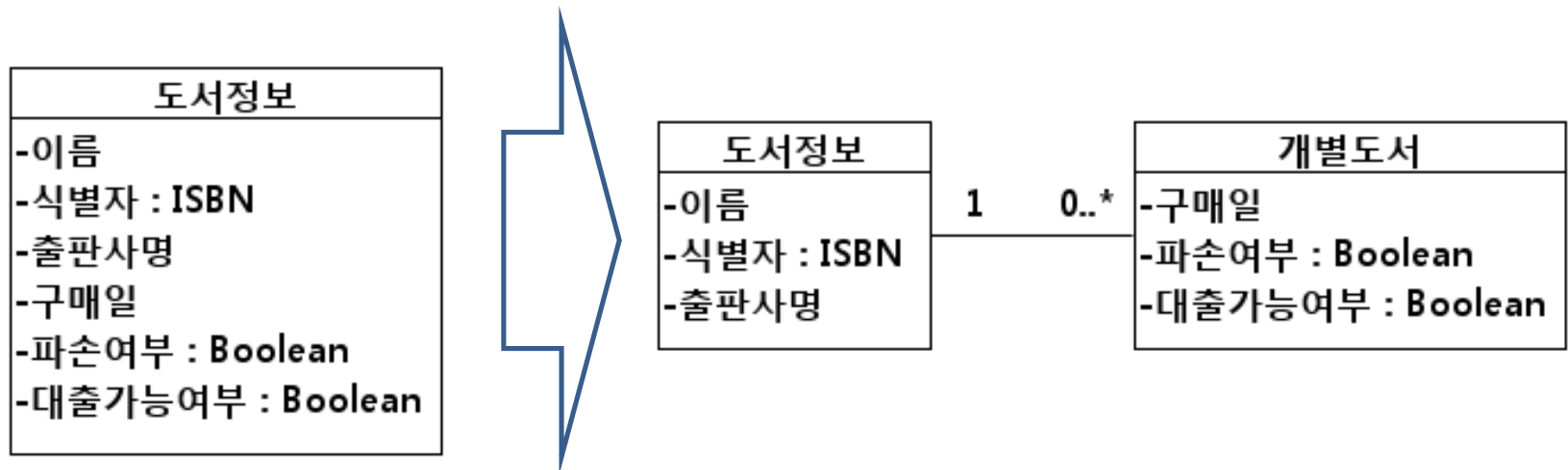
<<dataType>> ISBN

<<dataType>> Position

<<dataType>> Color

클래스의 속성들은 밀접한 관련성이 있어야 한다.

- ❖ 속성들 간의 관련성이 낮을 때 즉 클래스의 응집도(cohesion)이 낮을 때는 관련된 속성들끼리 묶어서 클래스를 분할 하는 것이 바람직하다.



연산 - 기본 개념

연산은 객체가 제공하는 행동을 표현한다.

- ❖ 객체의 행동을 클래스 수준에서는 연산(operation)으로 표현한다

학생
+공부한다()
+보고서를제출한다()
+시험을본다()

삼각형
+넓이를구한다()
+길이를구한다()
+확대한다()
+이동한다()

연산은 가시성, 이름, 인자, 반환타입으로 정의된다

- ❖ 연산은 이름을 포함하여 가시성, 인자, 반환 타입을 통하여 구체적으로 정의된다.
 - 가시성 이름 (인자방향 인자이름1 : 타입, ..., 인자 방향 인자이름 n : 타입) : 반환타입

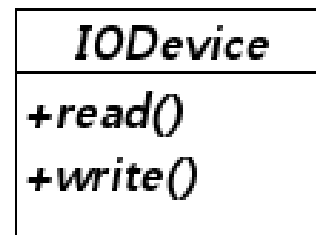
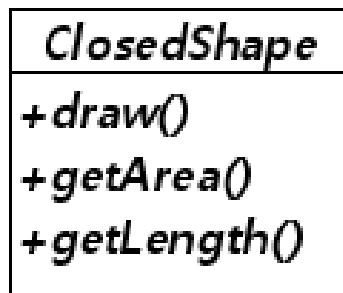
Triangle
+getArea() : Real +getLength() : Real +zoom(in scale : Real) : void +zoom(in scaleX : Real, in scaleY : Real) : void +move(in dx : Integer, in dy : Integer) : void +getColor(out lineColor : Color, out fillColor : Color) : void

연산은 가시성, 이름, 인자, 반환타입으로 정의된다

- ❖ 가시성: 클래스 외부로부터의 연산에 대한 접근의 허용 여부를 지정한다.
- ❖ 이름: 연산을 구분하기 위한 이름으로서 overloading이 가능하다. 즉 인자의 개수와 타입이 다르면 동일한 이름의 연산을 정의하는 것이 허용된다.
- ❖ 인자방향: 인자방향은 호출자와 피호출자 간에 인자 값의 전달 방향을 뜻한다. UML에서는 in, out, inout의 세가지 유형을 정의하고 있다.
- ❖ 인자 이름: 인자 이름은 인자들 중에서 고유한 이름을 가지며 전달되는 값의 의미를 표현하는 용어를 사용하는 것이 바람직하다.
- ❖ 타입: 전달되는 인자 값의 타입을 뜻한다. UML 기본 타입, 사용자 정의 타입, 프로그래밍 언어의 타입을 사용할 수가 있다.
- ❖ 반환타입: 연산의 수행 후에 반환되는 값의 타입을 지정한다.

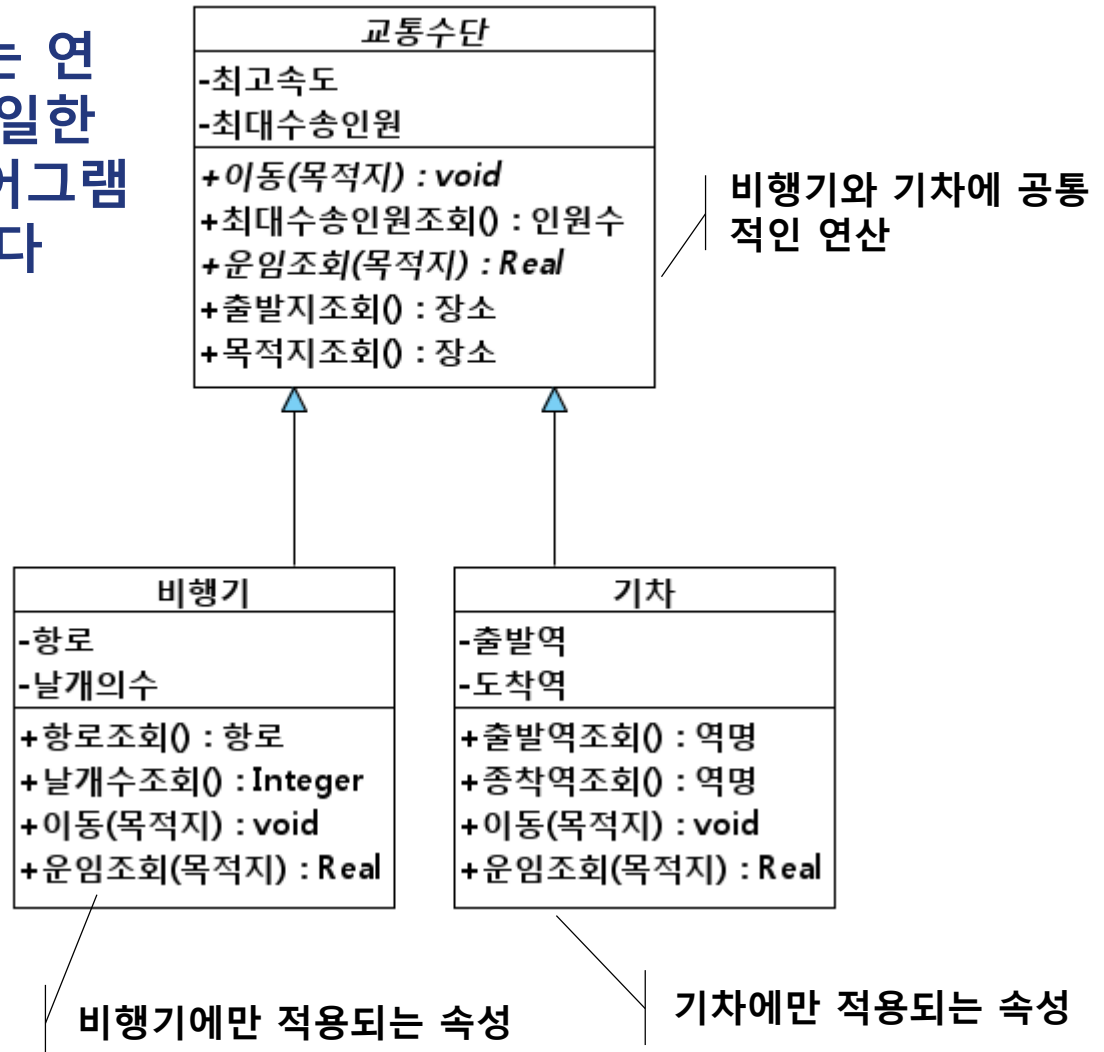
추상 연산

- ❖ 구현이 불가능한 연산을 추상(abstract) 연산이라고 부른다
- ❖ 객체를 실체화하지 못하는 클래스를 추상(abstract) 클래스라고 부른다.
- ❖ 추상 연산과 추상 클래스의 표현 - 클래스 다이어그램
 - 이탤릭체로 표시된다.



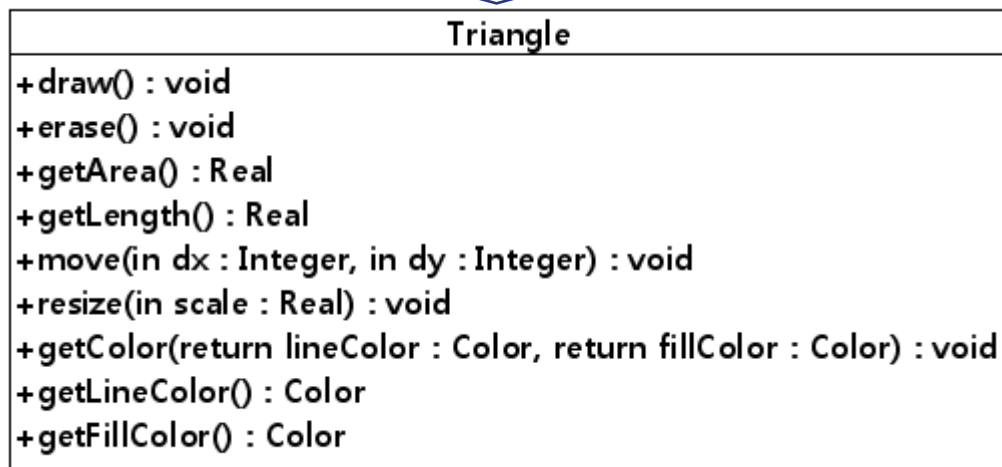
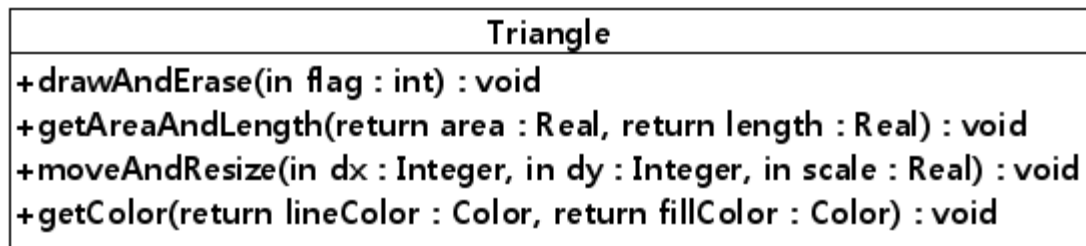
연산은 해당되는 전체 객체에 공통적이어야 한다

- ❖ 일부 객체에만 적용되는 연산이 있는 경우에도 동일한 방법으로 클래스 다이어그램을 재구성할 필요가 있다



연산은 오직 하나의 기능만을 제공해야 한다

- ❖ 연산은 클래스가 제공할 수 있는 단위 기능이 되어야 한다
- ❖ 여러 개의 기능을 제공하는 연산은 각 기능 별로 연산을 정의하도록 해야 한다



연산의 구현 방법이 노출시키지 않아야 한다

- ❖ 연산 내부의 구현 방법 즉 알고리즘이 외부에 노출되면 연산 구현 방법의 변경이 많은 호출자의 연쇄적인 수정을 유발할 수가 있다.
- ❖ 따라서 연산 내부의 구현 방법이 연산의 인터페이스 즉 연산의 이름, 인자, 반환타입으로서 노출되지 않도록 해야 한다.

```
void sort(  
    int sortMethod, int data[], int size)  
{  
    if ( sortMethod == 0 ) {  
        // bubble sort  
    } else if ( sortMethod == 1 ) {  
        // quick sort  
    }  
}
```



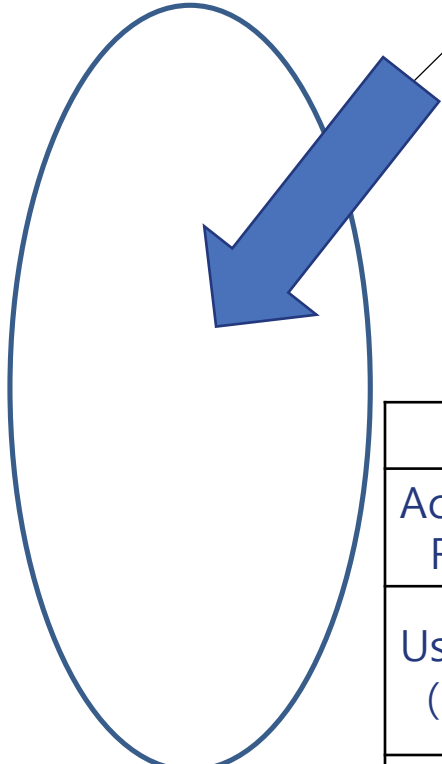
```
void sort(int data[], int size) {  
    // 더 좋은 정렬 방법 결정  
    // 결정된 방법에 따라서  
    // bubblesort() 또는 quickSort() 호출  
}  
void bubbleSort(int data[], int size) {  
    // bubble sort  
}  
void quickSort(int data[], int size) {  
    // quick sort  
}
```

연산은 클래스의 관점에서 바라본 기능의 전체적인 결과를 의미하는 이름을 가져야 한다


- ❖ 연산의 이름이 연산의 구현 방법을 뜻해서는 안 된다
- ❖ 연산이 제공하는 기능의 전체를 뜻하는 용어가 연산의 이름으로 사용되어야 한다
 - 연산이 제공하는 유일한 기능의 결과를 뜻하는 용어가 연산의 이름으로 사용되어야 한다.
- ❖ 연산이 정의된 클래스를 수행 주체로 하여 연산의 이름을 결정해야 한다.

연산은 클래스의 관점에서 바라본 기능의 전체적인 결과를 의미하는 이름을 가져야 한다

- ❖ 연산의 이름이 연산의 구현 방법을 뜻해서는 안 된다



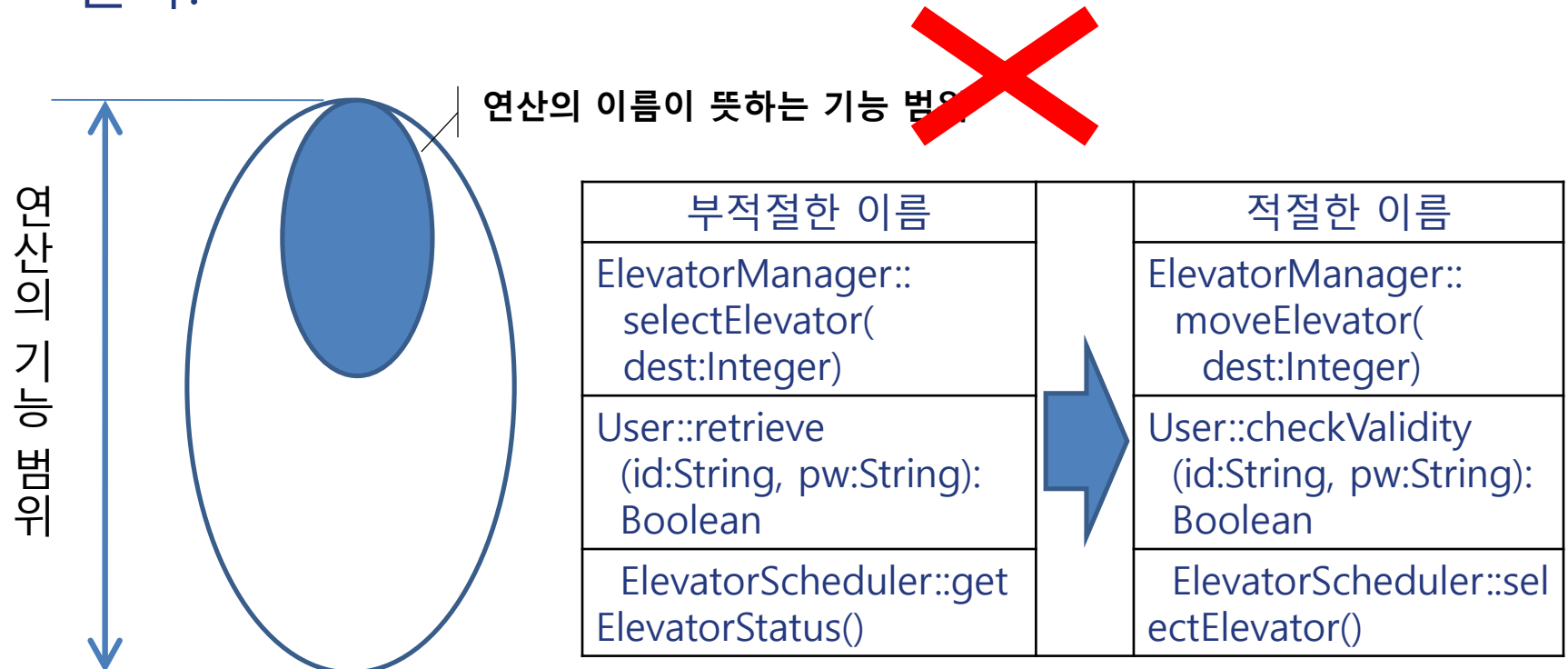
연산의 이름이 연산 내부의 구현 방법을 뜻한다.



부적절한 이름		적절한 이름
Account::calculateBalance(): Real		Account::getBalance():Real
User:: Compare (id:String, pw:String):Boolean		User:: checkValidity (id:String, pw:String): Boolean
ElevatorScheduler::selectNear Elevator()		ElevatorScheduler::selectElevator()


연산은 클래스의 관점에서 바라본 기능의 전체적인 결과를 의미하는 이름을 가져야 한다

- ❖ 연산이 제공하는 기능의 전체를 뜻하는 용어가 연산의 이름으로 사용되어야 한다. 즉 전체 결과가 아닌 연산이 제공하는 기능의 부분만을 뜻하는 이름을 사용해서는 안 된다.



연산은 클래스의 관점에서 바라본 기능의 전체적인 결과를 의미하는 이름을 가져야 한다

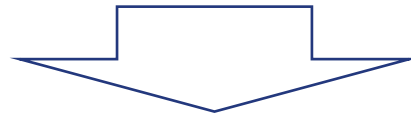
- ❖ 연산이 정의된 클래스를 수행 주체로 하여 연산의 이름을 결정해야 한다.
- ❖ 클래스는 연산의 수행 주체가 되어야 하므로 클래스를 주어로 하고 연산을 술어로 한 문장이 성립되어야 한다

부적절한 이름		적절한 이름
Pump::receiveGas(): Gas		Pump::giveGas(): Gas
Reviewer::submitPaper()		Reviewer::reviewPaper()
ElevatorManager::requestElevator()		ElevatorManager::moveElevator()

관련된 복 수개의 인자/반환 값들을 데이터 타입으로 정의할 수 있다

- ❖ 연산의 인자 또는 반환 값으로서 여러 개의 변수가 항상 함께 사용된다면 대응되는 복합 타입을 정의할 수 있다.
- ❖ 복합 데이터 타입의 정의 예 1

Employee
+setHireDate(in day : Integer, in month : Integer, in year : Integer) : void +getHireDate(return day : Integer, return month : Integer, return year : Integer) : void +getWorkingDay(in day : Integer, in month : Integer, in year : Integer) : Integer

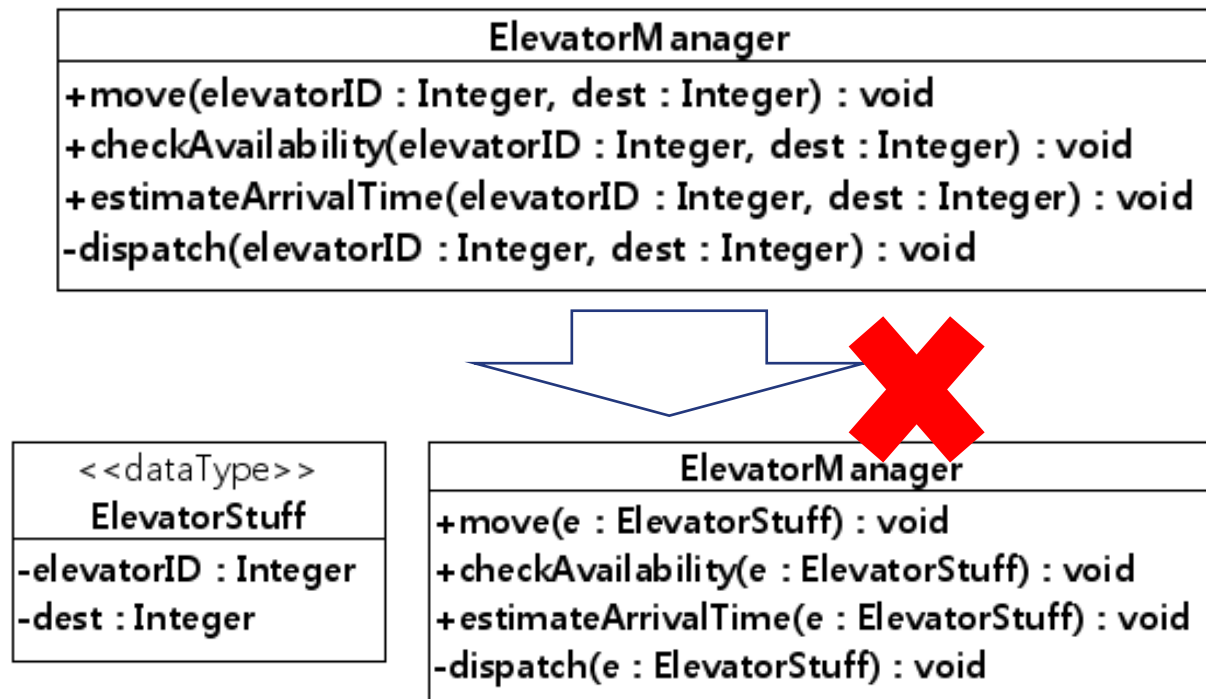


<<dataType>> Date
-day : Integer -month : Integer -year : Integer

Employee
+setHireDate(return date : Date) : void +getHireDate() : Date +getWorkingDay(in startingDate : Date) : Integer

관련된 복 수개의 인자/반환 값들을 데이터 타입으로 정의할 수 있다

- ❖ 인자/반환들을 묶는 의미 있는 개념이 존재할 때만 복합 타입을 사용해야 한다.
- ❖ 묶어서 한번에 부를 수 있는 개념/용어가 없다면 묶지 않고 그대로 개별적으로 사용해야 한다.



선행/후행 조건의 사용

- ❖ 신뢰도 높은 소프트웨어를 개발하기 위해서는 각 연산의 선행/후행 조건을 명확하게 표현하고 이를 프로그램에서 구현하는 것이 바람직하다

Account
-balance : Integer
+open() : void
+deposit(amount : Integer) : void
+withdraw(amount : Integer) : void
+close() : void
+getBalance() : Integer

```
context Account::open():void  
post: balance = 0
```

```
context Account::deposit(amount:Integer):void  
pre: amount > 0  
post: balance = balance@pre + amount
```

```
context Account::withdraw(amount:Integer):void  
pre: amount > 0  
pre: balance >= amount  
post: balance = balance@pre - amount
```

```
context Account::close():void  
pre: balance = 0
```


Q&A
