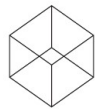


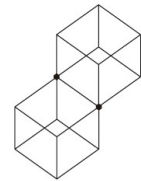
# 1. 객체지향 모델링

---



## JAVA 객체 지향 디자인 패턴

UML과 GoF 디자인 패턴 핵심 10가지로 배우는



# 서론

❖ 재사용성

❖ 확장성

장	패턴	예제 문제	연습문제
5장	스트래티지 패턴	로봇	도서 판매 공의 이동/출력 사람의 이동 가짜(Fake) 프린터
6장	싱글턴 패턴	프린터 관리자	티켓 발행기 서버 팩토리
7장	스테이트 패턴	형광등	도서 대출 음료 자동 판매기
8장	커맨드 패턴	만능 버튼	TV 리모컨 엘리베이터 버튼
9장	옵서버 패턴	성적 출력	배터리 관리 엘리베이터 위치 표시
10장	데커레이터 패턴	도로 표시	이메일 자동차 옵션
11장	템플릿 메서드 패턴	엘리베이터 모터	고객 보고서
12장	팩토리 메서드 패턴	엘리베이터 스케줄링	엘리베이터 모터 자동차 운행
13장	추상 팩토리 패턴	엘리베이터 부품	내비게이션
14장	컴퍼지트 패턴	컴퓨터 부품	디렉터리 관리

# 학습목표

---

## 학습목표

- 모델링 이해하기
- UML 다이어그램 이해하기
- 클래스 다이어그램 이해하기

# 1.1 모델링

---

## ❖ 모델(Model)의 역할

- 서로의 해석을 공유해 합의를 이루거나 해석의 타당성을 검토
- 현재 시스템 또는 앞으로 개발할 시스템의 원하는 모습을 가시화
- 시스템의 구조와 행위를 명세
- 시스템을 구축하는 틀 제공

그림 1-1 자동차의 전체적 구조를 파악할 수 있는 자동차 모델

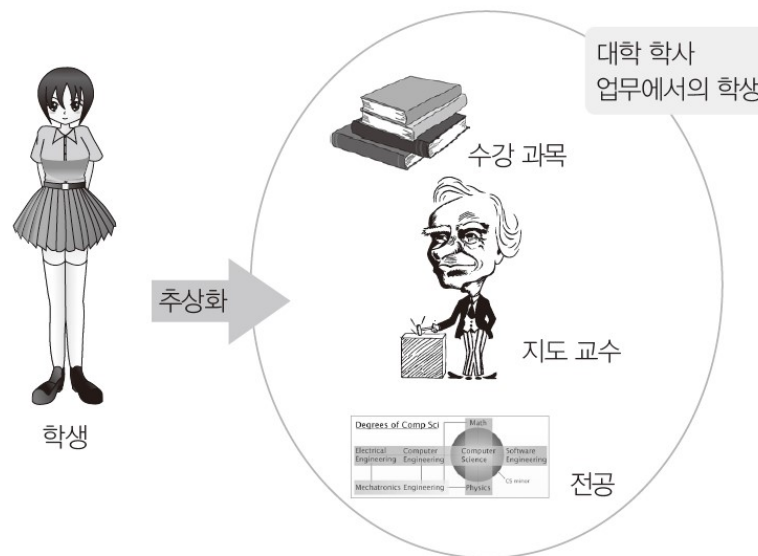


# 모델링

## ❖ 추상화 [abstraction]

- 모델은 추상화에 바탕을 두고 있음
- 특정 관점에서 관련이 있는 점은 부각
- 관련이 없는 것은 무시 [예. 학생의 머리카락 수...]

그림 1-2 대학 학사 업무의 추상화



## 1.2 UML

---

### ❖ UML(unified Modeling Language]

- 대표적인 시스템 모델링 언어
- 요구분석, 시스템설계, 시스템 구현 등의 시스템 개발 과정에서 개발자 사이의 의사 소통이 원활하게 이루어지도록 표준화한 통합 모델링 언어
- 제임스 러버 OMT + 야콥슨 OOSE + 부치 OOAD
- 1997년 UML 1.0
- 2012년 UML 2.5

표 1-1 UML 다이어그램의 종류

분류	다이어그램 유형	목적	
구조 다이어그램 (structure diagram)	클래스 다이어그램 (class diagram)	시스템을 구성하는 클래스들 사이의 관계를 표현한다.	
	객체 다이어그램 (object diagram)	객체 정보를 보여준다.	
	복합체 구조 다이어그램 (composite structure diagram)	복합 구조의 클래스와 컴포넌트 내부 구조를 표현한다.	
	배치 다이어그램 (deployment diagram)	소프트웨어, 하드웨어, 네트워크를 포함한 실행 시스템의 물리 구조를 표현한다.	
	컴포넌트 다이어그램 (component diagram)	컴포넌트 구조 사이의 관계를 표현한다.	
	패키지 다이어그램 (package diagram)	클래스나 유즈 케이스 등을 포함한 여러 모델 요소들을 그룹화해 패키지를 구성하고 패키지들 사이의 관계를 표현한다.	
행위 다이어그램 (behavior diagram)	활동 다이어그램 (activity diagram)	업무 처리 과정이나 연산이 수행되는 과정을 표현한다.	
	상태 머신 다이어그램 (state machine diagram)	객체의 생명주기를 표현한다.	
	유즈 케이스 다이어그램 (use case diagram)	사용자 관점에서 시스템 행위를 표현한다.	
	상호작용 다이어그램 (interaction diagram)	순차 다이어그램 (sequence diagram)	시간 흐름에 따른 객체 사이의 상호작용을 표현한다.
		상호작용 개요 다이어그램 (interaction overview diagram)	여러 상호작용 다이어그램 사이의 제어 흐름을 표현한다.
		통신 다이어그램 (communication diagram)	객체 사이의 관계를 중심으로 상호작용을 표현한다.
		타이밍 다이어그램 (timing diagram)	객체 상태 변화와 시간 제약을 명시적으로 표현한다.

## 1.3 클래스 다이어그램

---

### ❖ 클래스 다이어그램

- 시스템의 **정적인 구조** 표현
- 시간에 따라 변하지 않음
- 클래스와 또다른 클래스 들간의 관계 표현

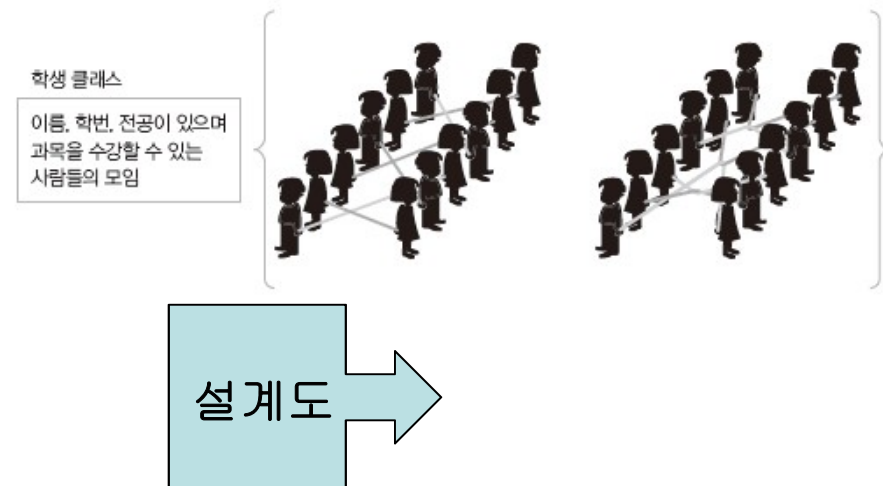


## 1.3.1 클래스

### ❖ 클래스

- 동일한 **속성**을 가지고 있고 동일한 **행위**를 수행하는 객체의 집합
- 객체를 생성하는 설계도

그림 1-3 클래스와 객체



코드 1-1

```
public class Cat {  
    private String name;  
  
    public void meow() {  
        System.out.println(name + "~~~~~" + "웁니다");  
    }  
  
    public Cat(String name) {  
        this.name = name;  
    }  
}
```

그림 1-4 Cat 클래스에서 생성된 고양이 객체

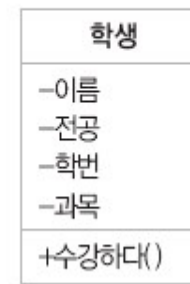
```
Cat cat2 = new Cat("냥냥이");  
Cat cat1 = new Cat("야옹이");
```

# 클래스

## ❖ UML의 클래스 표현

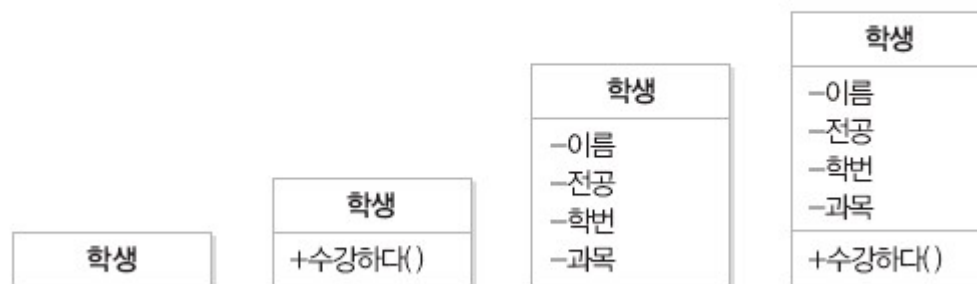
- 세 부분으로 나누어진 박스로 표현
- 가장 윗부분: 클래스 이름
- 중간 부분: 속성
- 마지막 부분: 연산 (operation)

그림 1-5 UML 클래스의 표현 예



## ❖ 여러 가지 클래스 표현 예

그림 1-6 여러 가지 클래스의 표현 예



# 가시화(visibility)

## ❖ 접근제어자

표 1-2 접근 제어자

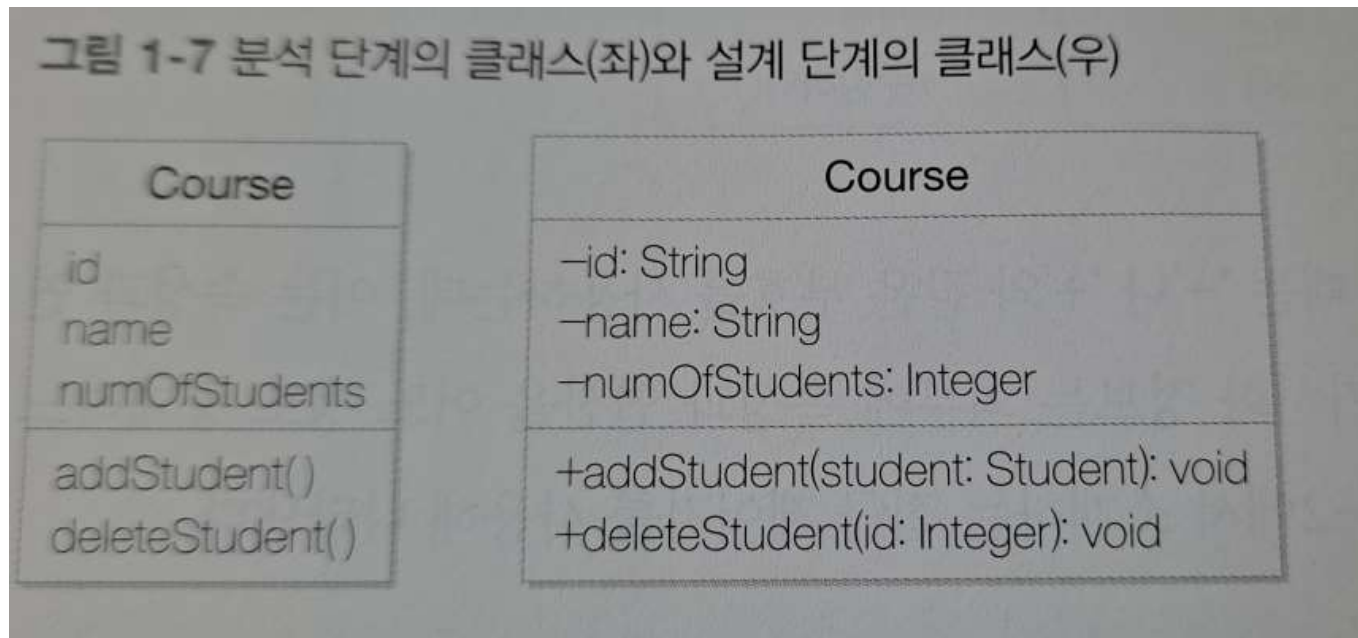
접근 제어자	표시	설명
public	+	어떤 클래스의 객체에서든 접근 가능
private	-	이 클래스에서 생성된 객체들만 접근 가능
protected	#	이 클래스와 동일 패키지에 있거나 상속 관계에 있는 하위 클래스의 객체들만 접근 가능
package	~	동일 패키지에 있는 클래스의 객체들만 접근 가능

## - 생략가능 : [ ]

표 1-3 속성과 연산 표기

	표기 방법
속성	[+ - # ~]이름: 타입[다중성 정보] [=초기값]
연산	[+ - # ~]이름(인자1: 타입1, ..., 인자n:타입n): 반환 타입

## ❖ 분석단계와 설계단계



---

체크포인트\_ 다음 클래스를 코드로 작성하라.

Course
-id: String -name: String -numOfStudents: Integer = 0
+addStudent() +deleteStudent()

---

```
public class Course {  
    String id;  
    String name;  
    int numOfStudents = 0;  
  
    void addStudent() {  
        // 여기에 무엇을 표현해야 할지는 아직 모른다  
    }  
  
    void deleteStudent() {  
        // 여기에 무엇을 표현해야 할지는 아직 모른다  
    }  
}
```

## 1.3.2 관계

### ❖ 객체지향 시스템은 상호관계를 맺는 여러 클래스에서 생성된 객체들이 기능을 수행

표 1-4 관계

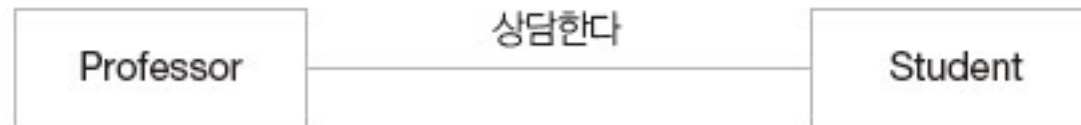
관계	설명
연관 관계 (association)	클래스들이 개념상 서로 연결되었음을 나타낸다. 실선이나 화살표로 표시하며 보통은 한 클래스가 다른 클래스에서 제공하는 기능을 사용하는 상황일 때 표시한다.
일반화 관계 (generalization)	객체지향 개념에서는 상속 관계라고 한다. 한 클래스가 다른 클래스를 포함하는 상위 개념일 때 이를 IS-A 관계라고 하며 UML에서는 일반화 관계로 모델링한다. 속이 빈 화살표를 사용해 표시한다.
집합 관계 (composition, aggregation)	클래스들 사이의 전체 또는 부분 같은 관계를 나타낸다. 집약 <sup>aggregation</sup> 관계와 합성 <sup>composition</sup> 관계가 존재한다.
의존 관계 (dependency)	연관 관계와 같이 한 클래스가 다른 클래스에서 제공하는 기능을 사용할 때를 나타낸다. 차이점은 두 클래스의 관계가 한 메서드를 실행하는 동안과 같은, 매우 짧은 시간만 유지된다는 점이다. 점선 화살표를 사용해 표시한다.
실체화 관계 (realization)	책임들의 집합인 인터페이스와 이 책임들을 실제로 실현한 클래스들 사이의 관계를 나타낸다. 상속과 유사하게 빈 삼각형을 사용하며 머리에 있는 실선 대신 점선을 사용해 표시한다.

# 연관 관계(association)

---

- ❖ 연관된 클래스 상에 실선을 그어 표시
- ❖ 두 클래스 상이의 관계가 명확한 경우에 이름을 사용하지 않아도 됨
- ❖ 그림 1-8. “교수가 학생을 상담한다 “

그림 1-8 Professor 클래스와 Student 클래스의 연관 관계





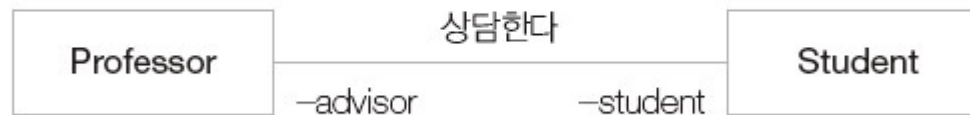
# 연관 관계에서의 역할

---

## ❖ 역할

- 연관 관계에서 각 클래스 객체의 역할은 클래스 바로 옆 연관 관계를 나타내는 선 가까이 기술
- 역할 이름은 연관된 클래스의 객체들이 서로를 참조할 수 있는 **[추후 구현될 클래스에서의) 속성의 이름**으로 활용

그림 1-9 연관 관계에서의 역할



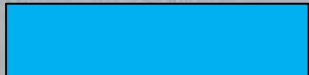

# 체크포인트

---

## ❖ 아래 Person Class를 구현하라

체크포인트\_ 다음 클래스 다이어그램을 코드로 작성하라.

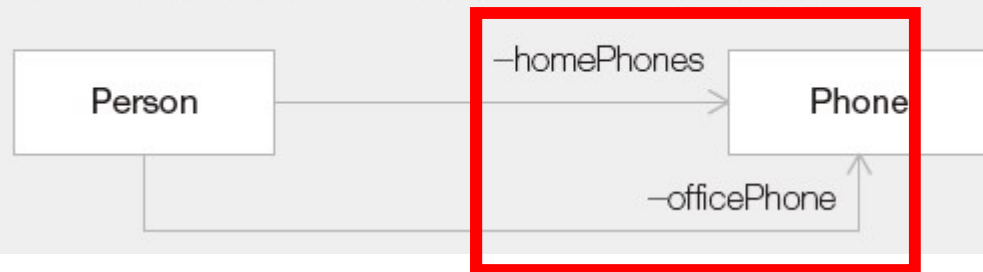


```
public class Person {  
    private   
  
    public Person() {  
        phones =   
    }  
  
    public Phone getPhone(int i) {  
        if (i == 0 || i == 1)  
            return phones[i]; // i = 0이면 집 전화, i = 1이면 사무실 전화  
  
        return null;  
    }  
}
```

이 코드는 집 전화와 사무실 전화를 배열의 인덱스를 통해 구분해야 하므로 매우 불편하다. 이런 경우는 전화기의 역할을 구분해서 사용하면 해결할 수 있다.

---

체크포인트\_ 다음 클래스 다이어그램을 코드로 작성하라.



```
public class Person {  
    private                       
    private                       
  
    public void setHomePhone(PHONE phone) {  
        this.homePhone = phone;  
    }  
  
    public void setOfficePhone(PHONE phone) {  
        this.officePhone = phone;  
    }  
}
```

```
public PHONE getHomePhone() {  
    return homePhone;  
}  
  
public PHONE getOfficePhone() {  
    return officePhone;  
}  
}
```

이 코드는 집 전화와 사무실 전화 각각에 참조가 이루어지므로 setter와 getter 메서드로 상황에 맞게 원하는 해당 전화기를 사용할 수 있다.

---

## ❖ 양방향(bidirectional) 연관 관계

- 두 클래스를 연결한 선에 화살표를 사용하지 않음
- 서로의 존재를 인지
- 그림 1-9. 교수 1인에 학생 1인이 연결

## ❖ 아래 1-9를 코드로 작성하라

그림 1-9 연관 관계에서의 역할



```
public class Professor {  
    private [REDACTED]  
  
    public void setStudent(Student student) {  
        this.student = student;  
        student.setAdvisor(this);  
    }  
  
    public void advise() {  
        student.advise("상담 내용은 여기에...");  
    }  
}  
  
public class Student {  
    private [REDACTED]  
  
    public void setAdvisor(Professor advisor) {  
        this.advisor = advisor;  
    }  
  
    public void advise(String msg) {  
        System.out.println(msg);  
    }  
}
```

```
}  
  
public class Main {  
    public static void main(String[] args) {  
        Professor hongGilDong = new Professor();  
        Student manSup = new Student();  
        hongGilDong.setStudent(manSup);  
        hongGilDong.advise();  
    }  
}
```

이 코드의 연관 관계는 양방향 연관 관계이므로 Professor 클래스 객체에서 Student 클래스 객체를 참조할 수 있는 속성(student)이 있고 Student 클래스 객체에서 Professor 클래스 객체를 참조할 수 있는 속성(advisor)이 있다. 또한 이 속성의 이름이 역할 이름을 활용한 것임을 알 수 있다.



# 다중성(multiplicity)

---

## ❖ 다중성(multiplicity) 의 필요성

- 1명의 교수는 (보통) 많은 학생들을 상담함

표 1-5 다중성 표시

다중성 표기	의미
1	엄밀하게 1
*	0 또는 그 이상
0..*	0 또는 그 이상
1..*	1 이상
0..1	0 또는 1
2..5	2 또는 3 또는 4 또는 5
1, 2, 6	1 또는 2 또는 6
1, 3..5	1 또는 3 또는 4 또는 5

# 양방향, 단방향 연관 관계

## ❖ 양방향 연관 관계

- 두 클래스를 연결한 선에 화살표를 사용하지 않음
- 서로의 존재를 인지

그림 1-10 다중성 예



## ❖ 단방향 연관 관계 (unidirectional association)


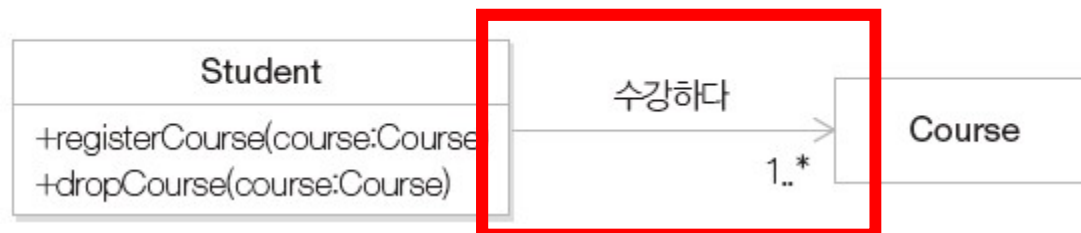
- 한쪽으로만 방향성이 있는 관계 (화살표 : )
- 한 쪽은 알지만 다른 쪽은 상대방의 존재를 모른다는 의미
- Student Class에는 Course 정보를 저장하는 attribute가 존재
- Course Class 에는 Student 정보를 저장하는 attribute가 없음
- 아래 그림 1-11을 코드(Student Class, Course Class) 로 작성하라

그림 1-11 단방향 연관 관계



```
import java.util.Vector;

public class Student {
    private String name;
    private         

    public Student(String name) {
        this.name = name;
        courses = new Vector<Course>();
    }

    public void registerCourse(Course course) {
        courses.add(course);
    }

    public void dropCourse(Course course) {
        if (courses.contains(course)) {
            courses.remove(course);
        }
    }
}
```

```
public Vector<Course> getCourses() {  
    return courses;  
}  
  
public class Course {  
    private String name;  
    public Course(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

이 코드는 Student 객체 하나에 하나 이상의 Course 객체가 연관되어 있기 때문에 다중성을 구현했으며, Student 클래스에 대표적 컬렉션 자료구조인 Vector를 이용해 여러 개의 Course 클래스 객체를 참조할 수 있게 했다. 컬렉션 자료구조에는 Vector 외에도 Set, Map, ArrayList 등 여러 가지가 있으므로 상황에 맞는 적절한 자료구조를 선택해 사용하면 된다. 또한 이 코드는 Student 클래스에서 Course 클래스로 향하는 단방향 연관 관계이기 때문에 Course 클래스에는 Student 객체를 참조하는 속성이 정의되어 있지 않다.